Synchronized quantum network emulator using discrete event simulation

by

Leon Wubben

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 29, 2019 at 9:00.

Student number:	4201388	
Thesis committee:	Prof. dr. S. Wehner,	TU Delft, supervisor
	Dr. D. Elkouss,	TU Delft
	Dr. P. Pawełczak	TU Delft
Daily supervisor:	A. Dahlberg	TU Delft
Master program:	Computer Science	
Track:	Software Technology	
Specialization:	Quantum information & Computing	

An electronic version of this thesis is available at
`http://repository.tudelft.nl/`.

Preface

The thesis currently in front of you is written as part of my MSc degree Computer Science at Delft University of Technology, The Netherlands. The project was done at and for Qutech, a research center for Quantum Computing and Internet and collaboration between Delft University of Technology (TU Delft) and the Netherlands Organization for Applied Scientific Research (TNO).
In this thesis I have build a piece of software that makes emulation of quantum networks possible.
Within the backend of the emulation the NetSquid simulator runs. NetSquid is a discrete event simulator for quantum networks, developed and maintained by QuTech. This simulator receives messages from 'the real world' in the so called CQC format. This format is designed and developed by Axel Dahlberg and Stephanie Wehner at QuTech.
I have build an interface that synchronizes the times in the real world and the simulation world, so that those two worlds can interact with each other.

The thesis is to be defended on Thursday 29 August 2019 at 9:00 at the building of Applied Physics at TU Delft.
The members of the thesis committee are:

Prof. dr. Stephanie Wehner,	QuTech, TU Delft, supervisor
Dr. David Elkouss,	QuTech, TU Delft
Dr. Przemysław Pawełczak	Embedded and Networked Systems, TU Delft

And my daily PhD student supervisor, Axel Dahlberg.

I thank Axel and Stephanie for the guidance throughout the thesis. You are great supervisors and I could not have asked for anyone better. The discussions we had were insightful and kept me on the right track.
I would personally like to thank Kanvi and Constantijn for the support we gave each other during our master theses.
Rob and Loek, for the help with NetSquid and the opportunity for me to continue working with you on NetSquid after my masters.
Thank you to Stephanie, Przemysław and David for being my committee and the advice and support you have given me.
And everyone else within the group I have worked with during my thesis. The list is

long and the fear of forgetting to mention someone is high, I truly have had a wonderful time with all of you.
Mam, pap, Dennis, Nicole en Stefan. Het heeft even geduurd, maar het is toch eindelijk af. Dank jullie voor mij blijven steunen toen ik het wat lastiger had. Ook al zag ik af en toe door de bomen het bos niet meer, de steun had ik nodig om door te blijven werken.
Amy, jouw komst heeft me denk ik de grootste motivatieboost gegeven om eindelijk door te schrijven. Ik kijk er naar uit om je te zien opgroeien tot een nog lievere meid dan dat je momenteel al bent. Wie weet groei je op in te tijd dat het kwantuminternet wereldwijd de standaard wordt, en dat jij trots bent dat je peetoom een kleine bijdrage heeft geleverd om dat een werkelijkheid te maken.

Leon Wubben
12 July 2019

Contents

1. Introduction

In order for remote parties to communicate they need to exchange some information between them. In a perfect world the communication is fast, secure and correct. Unfortunately classically secureness is not guaranteed, others can still read the communication. Even when the messages are encrypted using the best classical cryptographic protocols, they can still be cracked given enough computing power and time. Communication over a quantum internet or quantum network can be information theoretic secure. Many applications of a quantum internet are already known. The most famous is probably the quantum key distribution (QKD) protocol to generate secure encryption keys that can then be used in classical communication. [4, 11]. Other applications are secure identification [10] and other two-party cryptographic tasks [16], clock synchronization [15], secure delegated quantum computation [7], and extending the baseline of telescopes [17].
Many protocols for a quantum internet only need a handful qubits to be feasible, whereas quantum computers need many logical qubits (over 70) before they can realistically solve problems classical computers can not do. This makes a quantum internet an attractive research topic as the quantum processors needed for the communication can be relatively simple.
A problem with quantum communication and quantum computing in general is that the qubits used are very sensitive to noise and errors. This is currently the biggest bottleneck when establishing the quantum internet. Research is done to prevent errors by improving the hardware or correct them with error correction. These problems have to be solved first before we can feasibly use a quantum internet.

We use quantum (network) simulations to simulate the quantum network instead. This enables research about quantum communication to continue when a quantum network is not yet available. These simulators are programs that run on a classical computer and simulate a quantum network or computer. Within this thesis we make use of two of such simulators; SimulaQron and NetSquid. Anything a quantum computer can do can be done by a classical computer, except that, to the best of our knowledge, a quantum computer can be exponentially faster in certain scenarios. We can also simulate quantum communication with classical communication,

except we lose the security aspect.
The simulators are built to test and benchmark certain protocols, so we do not care that our simulated data is not secure from the world outside the simulation. The exponential overhead can be a nuisance when simulating large networks or quantum states, but some protocols can still be tested on smaller datasets and networks.

The classical internet as we know it now has been in development since the 1960s. Many techniques, aspects and protocols of the classical internet can be reused when developing the quantum internet.
Therefore it is useful to design our quantum internet similarly as the classical internet for some things. For accurate simulation we need to keep this design in mind when creating and using the quantum network simulators.
In classical networks information is passed through multiple layers in packages. Multiple networking models exist with different amount of layers. Some well known protocols that use this layered model is the OSI model. On each layer a protocol runs that handles the packages passed from higher levels. The layers are usually unaware of what the other layers have done or will do with the package. This makes it possible for each layer to be developed independent on the others. Changes or optimizations made in one layer should not affect or require additional changes in other layers.
Some common essential layers:

- **The application layer** is the highest layer. It creates the information to send, usually with a user friendly user interface. The data is encoded so it can be handled by the transport layer. HTTP is probably the most famous example as an application layer.
- **The transport layer** provides communication between hosts within the network. On this layer, protocols run to handle requests from the application layer. The transportation layer makes sure the information packets are send in the correct order and those received in the network layer are handled in the correct order.
- **The network layer** routes packages through the network. It finds a path through the network to the destination. It might have to route it through multiple nodes if there is no direct connection.
- **The link layer** transfers the packages between adjacent nodes in the network.
- **The physical layer** is the lowest layer. It consist of the hardware that is responsible for creating and manipulating physical bits of which your information packages consists.

Error correcting protocols can run on each layer to correct errors introduced by previous layers.

The layered structure or stack can also be used to develop a quantum network in a quantum network stack [21]. Each layer works similar as its classical counterpart. The biggest difference is that we now also send quantum information next to classical information. We can create entanglement between nodes in a network using the quantum channels.
The quantum link layer is responsible for generating and keeping track of this entanglement between two nodes [9]. Entanglement is fundamental for a quantum network. However entanglement over long distances is subject to qubit losses. In order to strengthen the entanglement of a link quantum repeaters are used [5].
Since each layer sees the other layers as a black box, we can already develop some of the higher layers that are independent of the lower level hardware layers.

A low-level classical controller within the hardware manages the quantum hardware. It receives commands from a higher level in the network stack. These commands can contain instructions about for example creating entanglement or sending qubits. The quantum hardware together with the low-level classical controller from the backend of platform dependent quantum processing system. See also figure 2.1.

The backend is platform independent, this means that we can substitute the backend with any other backend, be it another simulation or a real quantum hardware, without any change needed to the higher classical application layer. This means the backend should be able to parse the messages that it receives from the higher level.
For this there is an interface between the application and backend. Since this interface combines classical commands to quantum instructions it is called a classical quantum combiner, CQC for short. The classical controller within a backend receives messages or packages from the classical side and translates it to instructions for the hardware [8].

CQC is not only used for messages within a network stack to send and receive qubits. The same format is used to instruct local quantum hardware or simulators with low-level instructions, such as simple qubit gates such as the Hadamard and CNOT gates. These CQC messages are designed to be hardware agnostic. The instructions made by higher level applications should not have to change when we change the hardware or simulator. All that is needed is that the classical system within the backend can receive, create and parse these CQC commands and instruct the quantum hardware based on the commands.

To test the CQC messages, a simulator is available to simulate a simple link and physical layer; SimulaQron. This makes it possible to already build protocols for the application and transport layers without having access to quantum hardware. Since SimulaQron executes the instructions perfectly and instantly, the simulation is not accurate when it comes to timings, noise and losses that happen in a real scenario [8]. CQC and SimulaQron are discussed more in detail in chapter 2.

We need another simulator to test how robust protocols are to errors and delays. For this NetSquid is available [19]. NetSquid is developed by QuTech.
This simulator simulates the hardware components in quantum computers and networks, including noises in qubit operations, decoherence and time delays and losses in channels between nodes. Decoherence and delays are all time dependent. For this NetSquids keeps track of an internal simulation time (see definition 2) to correctly apply noise and schedule events for when qubits are send and received. Within the simulation events are scheduled at certain times on a timeline. NetSquid is a discrete event simulator, when the simulation runs it will jump from event to event on this timeline, discretely increasing its internal simulation time.
NetSquid can be used to benchmark quantum network protocols and test hardware for potential improvements. NetSquid is discussed more in detail in chapter 3.

Definition 1. *Real time*. The time that passes in the real world outside the simulation running on a computer.

Definition 2. *Simulation time*. The time that passes in a simulation on a computer. This time is stored as a variable and is increased discretely within a discrete event simulator.

So CQC helps to build and test the application layers in the network stack and NetSquid is perfect to test and simulate the quantum layers in the stack. Together they are are strong in each layer. NetSquid for accurately simulating the hardware, and CQC to help create protocols and interfaces for the application and transport layer.
In this thesis we have done exactly that by making NetSquid compatible with CQC. We create an interface that reads and sends CQC messages and instructs the NetSquid simulator based on the commands in the messages.
The CQC messages are hardware agnostic, so applications that currently run on SimulaQron through the CQC interface should be able to run on NetSquid without any change to the applications. Similar on how those applications will also run on real quantum network when they become available.
Figure 2.1 shows an overview on how this will look like. The application layers send messages to the backend. This backend can be a real quantum hardware or simulated ones. Communication between the application layer and the backend is done with those CQC messages.

1.1. Summary

In this thesis we make NetSquid compatible with CQC. On top of the NetSquid simulator we build a piece of software that receives incoming classical messages from a higher level classical application and inputs them as instructions to be simulated on NetSquid. The application runs in real time (definition 1) whereas NetSquid has its own different simulation time (definition 2). NetSquid uses this time as an internal clock to schedule new events on its timeline. It is a discrete event simulator, so it discretely increases this simulation time to jump from event to event.

The simulation time in NetSquid can run as fast or slow as the machine the simulation runs on allows. However the application that sends the messages to the backend sends them in real time to NetSquid that simulates it in simulation time. It is not desired that the simulation time runs faster than real time as it might miss messages from the outside world.
This is the challenge that is faced in this thesis. For this we created a simulation manager that synchronizes the two times, turning the simulator into an emulator. If we let the simulation run as fast as possible it would receive some messages too late. One instruction might cause the simulation to advance by one hour *simulation time* in only one millsecond *real time*. Meanwhile the qubits of another application decohered by one hour the next time it tries to access them. To prevent this we need to delay the simulation. This is done by waiting in real time with executing the instructions on the simulator.
There are multiple challenges to overcome. The simulation should stay consistent; instructions should still be handled in the same order they would have in a real quantum network. The simulation should also not be delayed too much, otherwise running the simulation would take an unnecessary amount of real time.

1.2. Overview

- In chapter 3.3 we have a look how network emulation is done classically.

- Chapter 2 describes how to program quantum networks; 2.1 talks about how to send commands from the higher level application (2.2) to a low-level classical controller in the backend. SimulaQron is used as one of those backends and is described in 2.3.

- Chapter 3 describes the NetSquid simulator for more realistic simulations with noise and delays.

- In chapter 4 we describe the problems we face when trying to synchronize *real time* with *simulation time*. Since NetSquid keeps track of an internal simulation time we need to synchronize this time with the real time. Network emulation has been studied in earlier works. Some of these techniques are discussed in 4.3.

- The implementation to solve the challenges is discussed in section 5. A simulation manager is created to stop and start the simulation at appropriate times. And finally chapter 6 shows how well the interface performs under different scenarios. The results are compared to how other simulators performed with the same scenarios.

2. Quantum network programming

A quantum network or quantum internet is build as a network of nodes that can communicate with each other by sending quantum bits (qubits) over a channel and generating entanglement between nodes. Such a network can be represented by a graph, where two nodes are connected by an edge if they are able to communicate with each other.
Within each node there is classical logic that instructs its quantum hardware. The main components in a quantum hardware are its processor. There are different ways to establish qubits in the quantum processors. Two popular implementations to use for quantum networking are NV centers in diamond [13] and ion traps [14]. Creating, modifying, sending and measuring these qubits differ for the different implementations.
The quantum processors are instructed by higher level classical programs. When programming the processors we want the program to be agnostic of the underlying implementation. Running the same program on NV center based processors should yield the same results as running it on ion-trap based processors. However the underlying architecture of these pieces of hardware can be vastly different.
For this we need an interface on top of the processors, or the backend, that can read and translate commands received from higher level programs and instruct the components of its backend accordingly to the command. This interface should also be able to send messages back in a standard format, for example when sending back a measurement result.
Figure 2.1 gives an overview of how the classical and quantum levels communicate in a network with two nodes.

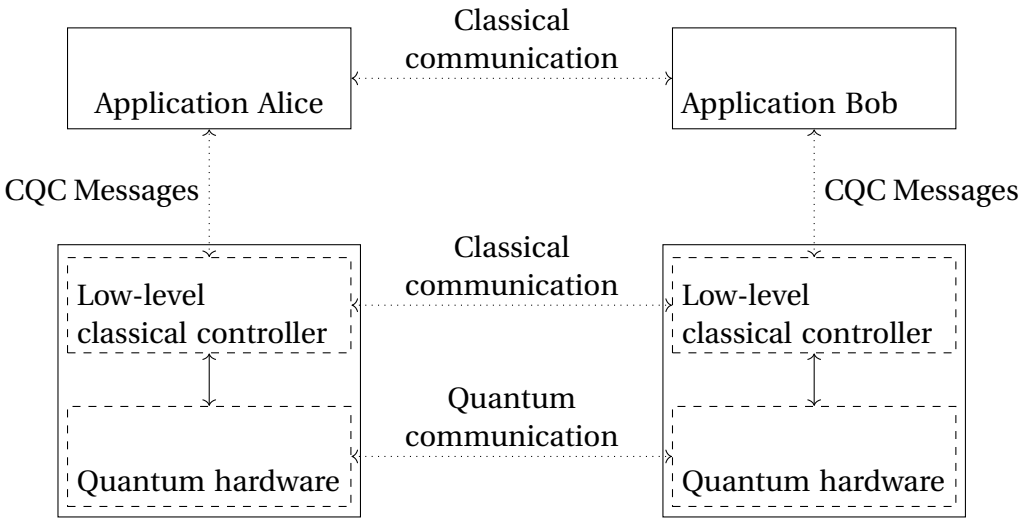

Figure 2.1: Overview of the architecture of a network with two nodes; Alice and Bob. Each solid box represents a process within in the network and can be substituted with another similar process without having to change the others. For example we can replace the quantum hardware with a simulation of the hardware or with a different kind of hardware. The low-level classical controller would also have to be adjusted for this hardware. Similarly we can write different applications that can all communicate with the backend. The low-level classical controller and the quantum hardware together form the backend of one node.
The dotted arrows indicate that the communication is over some kind of channel between the two processes. A solid arrow indicates direct communication.

2.1. CQC interface

CQC, the Classical Quantum Combiner, is an interface between the classical applications layers and the quantum backend. It receives classical low-level instructions from the higher level application and translates it to quantum instructions for its quantum hardware. So now if your backend has a low-level classical controller that can receive, parse and send CQC messages you can send it CQC messages to modify quantum bits, without knowing what the underlying hardware actually does. We can change the hardware with a completely different implementation without any change to the application. CQC is designed and developed by Axel Dahlberg and Stephanie Wehner at QuTech at Delft University of Technology [8].

2.1.1. CQC Message Format

The quantum processor can be physically at a different location than the classical computer instructing it. For this reason it should be possible to send the instructions over a classical connection to the server the backend is connected to. The packages to the CQC interface have the following format:

request type	rationale
TP_HELLO	Alive check, return backend information
TP_COMMAND	Execute one or more commands
TP_FACTORY	Execute one or more commands repeatedly
TP_GET_TIME	Get creation time of qubit

Table 2.1: Types of possible types from application to backend

- A type header.
- Optionally one or more command headers

The type header contains information on what type of message it is, the version number of the CQC format, a unique identifier for the application sending the message, and the length of the total message.
The unique identifier is for the backend to know which application the message was received from. One node in the network might have multiple applications running. This node is responsible for giving each application a unique identifier so the backend can distinguish messages from different applications from the same node.
Possible messages type are depicted in table 2.1.
From the length the backend can anticipate how many bytes it should still receive before executing the command.
If the first message is of type $TP\_COMMAND$ then one ore more messages are send with the information of the commands. This message consists of the following information:

- The command to perform (for example initializing a new qubit, what gate to do or a measurement)
- The qubit to perform the action on
- Whether to notify the sender when the action is done
- Whether the backend should be blocked from other messages until the action is done
- Whether there are any additional actions to perform after this action

In the last case we send another command of the same format.
Some commands require additional information. For example the backend needs to know what the qubit identifier of the second qubit for two-qubit gates is, the rotation angle of gates, or the ip and port information to which a qubit should be send to. This

command type	rationale
CMD_NEW	Request new qubit
CMD_MEASURE	Measure and destroy qubit
CMD_MEASURE_INPLACE	Measure qubit
CMD_RESET	Reset qubit to $\|0\rangle$
CMD_RELEASE	Mark qubit as free to use
CMD_SEND	Send qubit to other node
CMD_RECV	Receive qubit
CMD_EPR	Create EPR pair with other node
CMD_EPR_RECV	Receive half of EPR pair
CMD_I	Identity
CMD_X	Pauli X
CMD_Y	Pauli Y
CMD_Z	Pauli Z
CMD_H	Hadamard
CMD_K	K gate
CMD_T	T gate
CMD_ROT_X	Rotation over angle around X axis
CMD_ROT_Y	Rotation over angle around Y axis
CMD_ROT_Z	Rotation over angle around Z axis
CMD_CNOT	Controlled X gate
CMD_CPHASE	Controlled Z gate

Table 2.2: Possible CQC commands for the backend

information is send in a third message specific to the command.
Possible commands are depicted in table 2.2.

The rotation angles are depicted by one byte, and are therefore discrete with 256 possible angles. When the hardware finishes the command, it can send information back to the application in the same CQC format. Each message contains a response code, and optionally some content as well. Possible responses can be found in table 2.3.

The CQC format is useless if there is no application that can generate and send CQC messages and if there is no backend that can receive CQC messages and instruct some quantum processor.

command type	rationale
TP_EXPIRE	Qubit has expired
TP_DONE	Command successfully executed
TP_RECV	Qubit received. Returns qubit id
TP_EPR_OK	EPR pair created. Returns qubit id and entanglement information
TP_MEASOUT	Qubit measured. Returns outcome
TP_GET_TIME	Returns creation time of qubit
TP_INF_TIME	Returns timing information
TP_NEW_OK	Qubit created. Returns qubit id
ERR_GENERAL	General error happened
ERR_NOQUBIT	No more qubits available
ERR_UNSUPP	Command not supported
ERR_TIMEOUT	Timeout
ERR_INUSE	Qubit already in use
ERR_UNKNOWN	Qubit is unknown

Table 2.3: Possible CQC commands for the backend

2.2. Application frontend

In the frontend higher level platform independent applications, programs and protocols can be developed. These applications sends instructions to the backend in real time. The application needs to be able to receive and send these instructions to the backend for it to be used to develop software for a quantum internet. Higher level libraries can be written to help generating and parsing CQC messages. One such library we make use of in this thesis is the one written in python. Figure 2.2 shows what happens when the application sends a command to the backend.
Since the application is platform independent it should be possible to replace the backend with any other without loss of functionality.

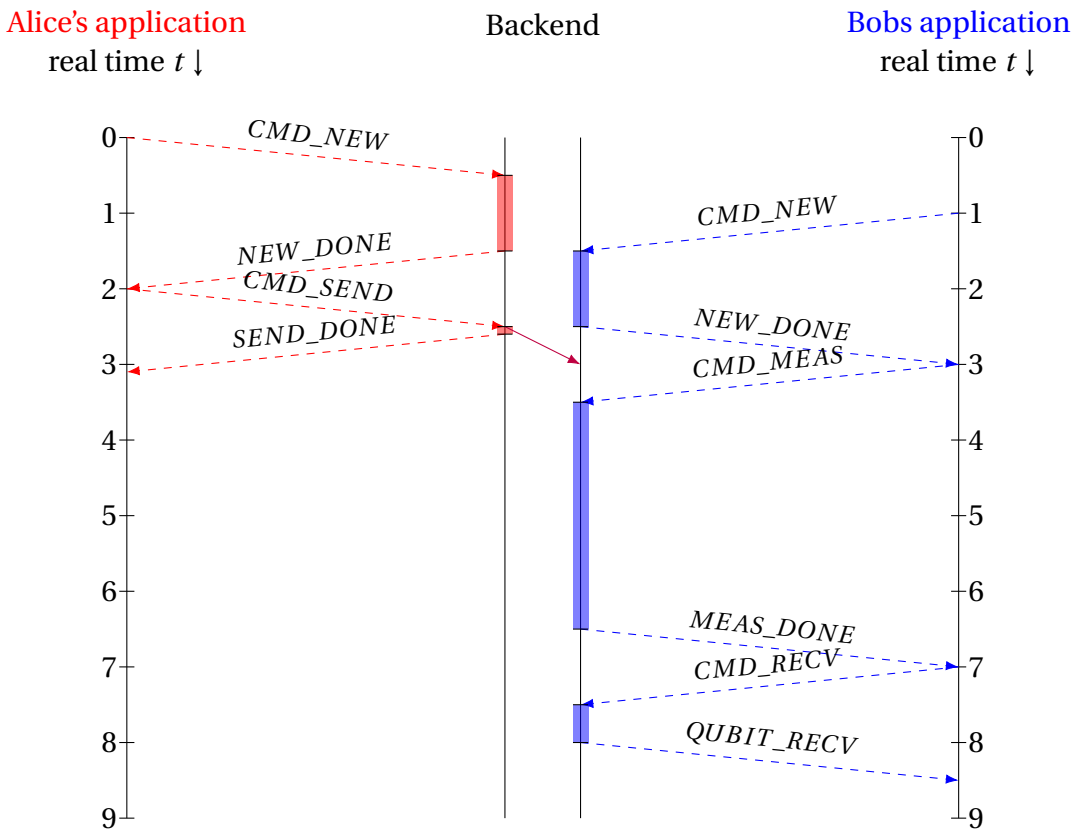

Figure 2.2: Timeline showing a simple application of an example program for Alice and Bob. Alice starts her program at $t = 0$ where she sends a CQC message to her backend to create a qubit. This command is received at $t = 0.5$. It takes $\Delta t = 1$ to create the new qubit, indicated by the red bar in the backend. Once the qubit is created its qubit id is send back to the application over a CQC message. Upon receiving this qubit id Alice sends it to Bob, again with a CQC message. The backend can already send back to the application that the send has been done before the qubit arrives at Bob. Meanwhile Bob started his program at $t = 1$. At $t = 3$ Bobs backend receives Alice's message. However Bob hasn't requested it yet. Only when a receive command comes in at $t = 7.5$ does Bobs backend receive the message and send a confirmation back to Bobs application.

2.3. SimulaQron

The backend of a node in a quantum network consists of two parts: a low-level classical controller that handles classical data and instructs the quantum hardware. This controller also can receive parse and send CQC messages. Based on the commands in these messages the hardware is instructed with certain commands such as creating new qubits, doing single or two-qubit gates on those qubits, sending them ore measuring them. The only quantum computers available at the moment do not have a lot of qubits available, and the ones it has are noisy and decohere fast. In order to test the CQC interface and the already written applications we simulate the qubits

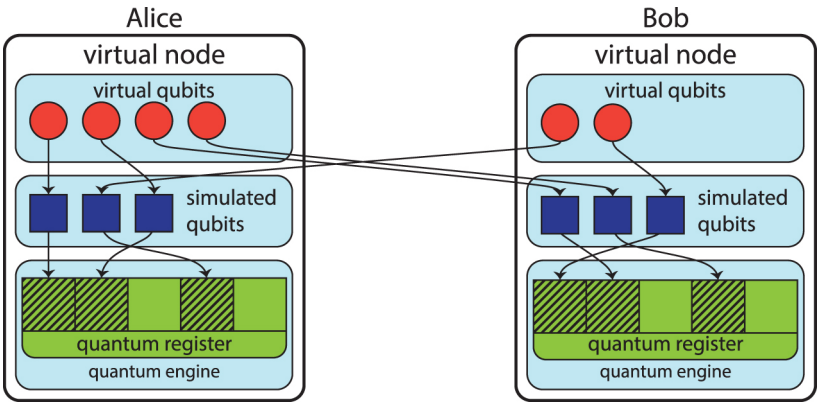

Figure 2.3: A visualization of the interplay between different internal components of SimulaQron. The $simulatedQubits$ (blue squares) are objects handled locally in a $virtualNode$. These $simulatedQubit$ s point to a part of the $quantumRegister$, which stores the quantum state simulated by the $virtualNode$. 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. Picture taken from [8].

.

instead on a classical processor.
For this the SimulaQron simulator is available [8]. SimulaQron simulates multiple local quantum processors and the classical and quantum communication channels between them. This makes it possible to simulate the transmission of qubits between different processors.
The quantum communication is done over already established classical network channels, which makes it possible to run SimulaQron on different remote classical computers.
We can simulate multiple quantum processors on one classical computer, have each classical computer simulate one processor or any combination between them. All simulated quantum processors are connected to each other over the classical network channel to transmit qubits and entanglement information.
A quantum simulator should be able to simulate entanglement between qubits. But entangled qubits can classically only be simulated within the same memory or memory register. This makes it impossible for entangled qubits to be simulated in two different classical processors, even though they are in different quantum processors. To solve this SimulaQron introduces the notion of 'simulated' and 'virtual' qubits. See figure 2.3.

Each node simulates a quantum register to function as a quantum memory with n qubits. Simulated qubits are simulated in these registers. They are stored as a matrix

at a certain index in the register. This allows easy manipulation of the qubit by gates or measurements. If qubits are measured or moved to other registers the size of the matrix in the register is shrunk to prevent it to grow arbitrarily when new qubits are created or added.
Each simulated qubit is associated with a virtual qubit. A virtual qubit 'belongs' to a certain node and is essentially a pointer to its simulated qubit. This means that a virtual qubit does not necessarily belong to the same as where it is simulated. A qubit can virtually belong to node A, but be simulated at node B. Only node A can perform operations on the qubit. If node A wants to apply an Hadamard gate to the qubit, the underlying structure sends an instruction to node B to perform the Hadamard gate on this qubit.
Node A can also send the qubit to another node C. From node A a message is send to node B to update the owner of the qubit, and to node C the information where the qubit resides.
The owners of each node do not know where its virtual qubits are simulated or whose qubits it simulates. This all happens in the background.

2.3.1. Register merging

When performing two qubit operations it is a bit more complicated. Such an operation, or more specifically, an entangling operation, can only be done if all qubits are simulated at the same location. So two qubits should not only belong to the same node, they also need to be simulated in the same register. This might require two registers to be merged before the operation can be done.
One way to prevent the need of register merges is to simulate all qubits on one single classical computer in the network. This classical computer had the computation load of the whole simulation, making it a single point of failure and a performance bottleneck. To prevent this we keep the simulation distributed by having remote registers, distributing the load over the network and allows for subsets of nodes not being influenced by other nodes.
As an example we look at a merge between a register at node A with a register at node B. Node B sends its register with simulated qubit to node A. And it tells the nodes that had their virtual qubits simulated at B that their qubits are now simulated at A.

2.4. Example

Let's look at a simple example to show how a program would look like. We have a network consisting of two nodes Alice and Bob. Alice creates a qubit and sends it Bob. Bob meanwhile also created a qubit, and measured it. Finally Bob receives Alice qubit.

Algorithm 2.1: Teleportation Alice

```
1 # Initialize the connection
2 with CQCConnection("Alice") as alice:
3   # create qubit
4   q = qubit(alice)
5   # send it to Bob
6   alice.sendQubit(q, "Bob")
```

Algorithm 2.2: Teleportation Bob

```
1  # Initialize the connection
2  with CQCConnection("Bob") as bob:
3    # create qubit
4    q = qubit(bob)
5    # measure and print it
6    outcome = q.measure()
7    print("Bob measured, outcome " + str(outcome))
8    # receive qubit from Alice
9    q2 = bob.recvQubit()
10   print("Bob received qubit from Alice")
```

2.4.1. The application layer

Algorithms 2.1 and 2.2 show the python code at the application level of Alice and Bob respectively. Figure 2.2 shows the process on a timeline.
On line 4 Alice creates a new qubit and sends it to Bob in line 6. Internally the library of Alice creates CQC message that are send to the classical controller of Alice's backend. In the case of the send command this are three headers; a type header, a command header and a communication header.
The type header tells the backend what to expect:
($VERSION\_NUMBER, TP\_COMMAND,$
$APP\_ID, CQC\_CMD\_HDR\_LENGTH + CQC\_COM\_HDR\_LENGTH$)
Each of these value is an integer. The first parameter indicates the version of CQC the application is running, the second is the integer that correspond to the *Command type*. The *APP_ID* is the application ID of Alice. Alice might be running multiple applications at once, the parameter indicates which application Alice sends the message from. And the last parameter is the total length of the messages still to come. We are sending a command header and a communication header, so the total amount of bytes the backend should still expect is the sum of the length of those two headers.

The command header tells the CQC interface what command should be done on which qubit. If for example the qubit identifier of the newly created qubit is 1 then the CQC message might look something like $(1, CMD\_SEND, True, False, False)$
The last three booleans indicate if we want to be notified when the command is done, if the other qubits of Alice should be inaccessible during the execution and if any additional actions are to be taken after this action. Let's say in our case we want to be notified when it is done.
And finally the third header is the communication header. The library has a method to look up the locations of the other nodes in a network, so it can lookup the IP address and port that Bob is listening to.
$(REMOTE\_APP\_ID, REMOTE\_IP, REMOTE\_PORT)$
Bob also might run multiple applications, so here the first parameter indicates for which application we want to send the qubit to.

Meanwhile Bobs application is waiting at line 9 to receive a qubit, after having created and measured a qubit.

2.4.2. CQC message handler

When an application sends a CQC message to the backend it is received by the CQC message handler in the low-level classical controller. This message handler reads the message and computes the instructions in the message on the simulation.
When it receives the CMD_NEW command it will create a new qubit object in the $|0\rangle$ state. The simulation stores the state of this new qubit in its register and the handler assigns a qubit identifier. It sends this qubit id back to the application so the application can refer to this qubit in the following send command.
A CQC message is send back to Alice's Application with the qubit id so she knows what id to use when doing future operations on this qubit. We send two headers back.
The first is as always a type header:
$(VERSION\_NUMBER, TP\_NEW\_OK, APP\_ID, length)$
The second header only contains the identifier of the qubit.
After Alice's backend sends the qubit to Bob it will send a notification back to Alice's application to indicate that the command has successfully finished, since Alice indicated that she wanted to be notified. This notification is again done with a CQC message. We only need to send one CQC message in this case with type TP_DONE.

Alice then sends an instruction to the backend to send the qubit to Bob. The simulation can do this in two different ways. It could either send the simulated qubit to Bob by sending the register it is simulated in or it can send the virtual qubit by sending information to Bob to where to find the simulated qubit in Alice's register. Sending the virtual qubit is less information, but might mean there is going to be more internal

traffic within the backend of the Alice's and Bobs simulator. When Bob wants to do additional commands on the qubit which is still simulated at Alice, then the backend of Bob has to send messages to Alice's backend to do those commands on Bobs qubit. What is done with the virtual qubit when Bobs backend receives it depends on whether Bob was already waiting to receive a qubit. If not then the virtual qubit is put in a queue. It will be removed from the queue when Bob does request to receive the qubit in a first in first out basis. If Bob was already waiting to receive it or there was a qubit in the queue when Bob called to receive a qubit, then the qubit information about where the qubit lives is send to the message handler. The handler gives the virtual qubit a qubit identifier and sends this with a CQC message to Bob in a similar way Alice had done.

3. NetSquid

What SimulaQron does not do is simulate noise and delays. Every operation is perfect and instantaneously.
To simulate these phenomena the NetSquid (Network Simulator for Quantum Information using Discrete events) simulator is developed at QuTech. NetSquids focus is to accurately simulate noises and timings in any quantum network.
The components of a quantum computer can be implemented and simulated in NetSquid. This way the performance of these implementations can be checked and bottlenecks can be found to determine were the biggest improvements in performance can be gained and to see if it is worth improving one component, as there might be greater grains to be found elsewhere.
NetSquid also simulates the timings of components. Operations on qubits and sending information through a fiber to other nodes take time, causing qubits to decohere. Decoherence is a large source of noise in quantum networks, so it is important to simulate this noise correctly.
One application for NetSquid is the simulation of quantum repeaters. We can simulate different repeater protocols and setups to see what configurations have the least overall decoherence times in quantum networks.

3.1. Discrete event simulator

As the name suggests NetSquid uses a discrete event simulator to simulate quantum networks. Under the hood a simulation engine keeps track of the simulation time. The engine can run the simulation up to a given time unit or for a certain amount of time. Entities within the simulation can schedule events on the timeline of the simulator. When the simulation engine runs, it sequentially runs from event to event. Discretely increasing the simulation time. See figure 3.1.

Each component in NetSquid is an entity. They can schedule events at the current time (`_schedule_now()`), at some relative time from now (`_schedule_after()`) or at some absolute time unit (`_schedule_at()`).
Entities can also listen to events. When listening to an event they subscribe to it with

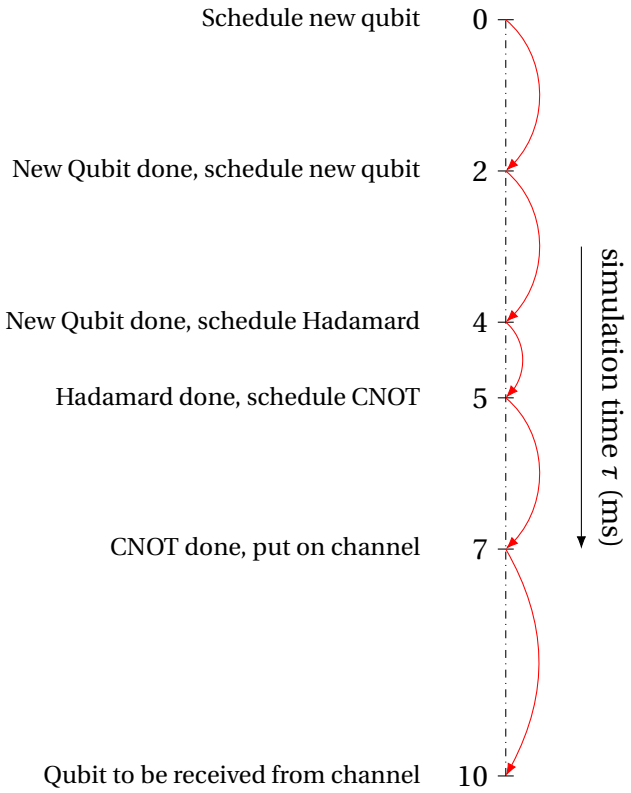

Figure 3.1: Example of a discrete-time simulation of creating a simple EPR pair between two nodes in a network. The timeline is not continuus indicated by the dashed line, it discretely increases the time. An arrow indicates the scheduling of a new event. For example at $\tau = 0$ we want to create a new qubit. Creating this qubit takes 2ms, so an event is created at $\tau = 2$. When the simulation jumps to this time the qubit is created and the next event is scheduled. The discrete-event simulation then jumps discretely from one event to the next.

an event handler; a callback function that fires when the event is triggered.
The event handler can in turn schedule and listen to other new events.

The discrete event simulator is oblivious to the world outside the simulation, it is independent on real time. The simulation steps from event to event as fast as the classical computer allows, increasing its simulated time discretely as it goes.

3.2. Components

NetSquid simulates the hardware of a quantum computer, so before running a simulation this hardware needs to be defined. This is done by defining components and their interaction with each other. The two main components are the channel and the quantum processor. Items are send over a channel by a sender to a receiver with some delay. Depending on the channel those items can be can be both classical or quantum bits.
Upon sending the items an event can be specified. When the items are ready to be retrieved the event will be called and event handlers that were subscribed to the event will be called.

Channels can be specified with delay, noise and loss models. The delay model describes the time delay of the channel as either some constant or a defined distribution. The noise model describes how much noise is applied to the information but on the channel. And the loss model describes how likely it is that the information is lost during transmission.

The quantum processor is where qubits are created, manipulated and measured. Depending on the processor these operations might behave very different. Simulating the quantum processor for NV-centers differs a lot from simulating the processor of ion-traps.
But on the surface they behave very similar as well. Each operation takes time and introduces some noise to the qubit. When building up the simulation the operation time and noise models all have to be defined. For example in figure 3.1 we set the operation time of initializing new qubits and doing a CNOT to two milliseconds and doing an Hadamard gate to one millisecond. Note that these times are not realist in a real implementation but are chosen arbitrary for the purpose of the example.
The processor of a node can be programmed by giving it a list of instructions. It will schedule the instructions sequentially (or in parallel if this is allowed by the type of processor) on the timeline. It can be given a callback to be called when the program finishes (or fails), this will prove useful in section 5.1.

3.3. Related work

Classical simulations that use discrete event simulation are avaible, as quantum networks are not the only networks where simulation is desired. For some classcial discrete even network simulators emulation is also possible. Emulating a network makes it possible to predict how it will behave in real applications, without having to implement them.
Since these emulators are classical they are often not focused on simulating noise, nor do they focus on internal computations that might take too long to compute. The main focus is on the correctness, loss and delays of sending data packets across the network. The physical systems that send packages to the emulator are assumed to run in real time [1, 2]. This makes having an internal clock within the emulator let alone have it synchronized with the real world less important [3, 6].
Emulating networks with discrete event simulators was introduced at the end of the twentieth century with the first *NETSIM* simulator [12]. Here the simulator introduces real time delay in order to slow the simulation down. The accuracy was improved using the second simulator *NS-2* by monitoring and correcting the virtual clock of the simulator to ensure the chronological order of events remained unchanged [20].
The biggest disadvantage of these emulators is simulation overload, where the simulator has to simulate too much that it starts running behind on real time. It is possible to use a large computing power to increase the execution speed [18]. This does only work up to a certain point and does not remove the initial issue.
In the case of NS-2 the simulation overload is because NS-2 is single threaded. A solution was brought in NS-3 where it is possible to use multi-core processors and distributed computing [22].
One platform, SliceTime, solves this by using time windows, or so called time slices [23, 24]. The simulation time is sliced up in different timewindows of given length. When the simulator hits the end of such a time slice it will halt the simulation and waits for the real time to synchronize. This makes it possible to have each component be simulated on a different machine, and the time drift or accuracy between the components is limited by the time of the slice.
We have a look at these implementations in section 4.3 to see which is suitable to implement in our quantum network emulator.

4. Design

We can now write high-level applications for a quantum network that creates CQC messages. We can test these programs on SimulaQron that can simulate a quantum network without noises and delays. And NetSquid can accurately simulate noises and delays quantum hardware components.
What we want now is to write high-level applications that can simulate accurate quantum network simulators including noise, decoherence and delays. We can do this by adding a CQC message handler in the backend to the NetSquid simulator. This handler receives CQC messages from the higher layers and translates them to programs that run on the simulated quantum processors in NetSquid. This chapter describes the design of the message handler.
The applications run in real time, where NetSquid keeps track of its own simulation time. If those times are independent on each other then messages received from the application might arrive later in simulation time than expected. We need to slow down the simulation time. This is quite challengiing, we describe why and how in the chapter.

4.1. Simulation setup

NetSquid runs its simulation all on a single machine in a single thread. This is in contrast to SimulaQron and real networks where the hardware of nodes can be physically separated, as shown in figure 2.1.
For NetSquid these nodes are all simulated within the same process. We still want each node to listen to its own higher level application. For this we add a low-level classical controller for each node but all within the same process, see figure 4.1.

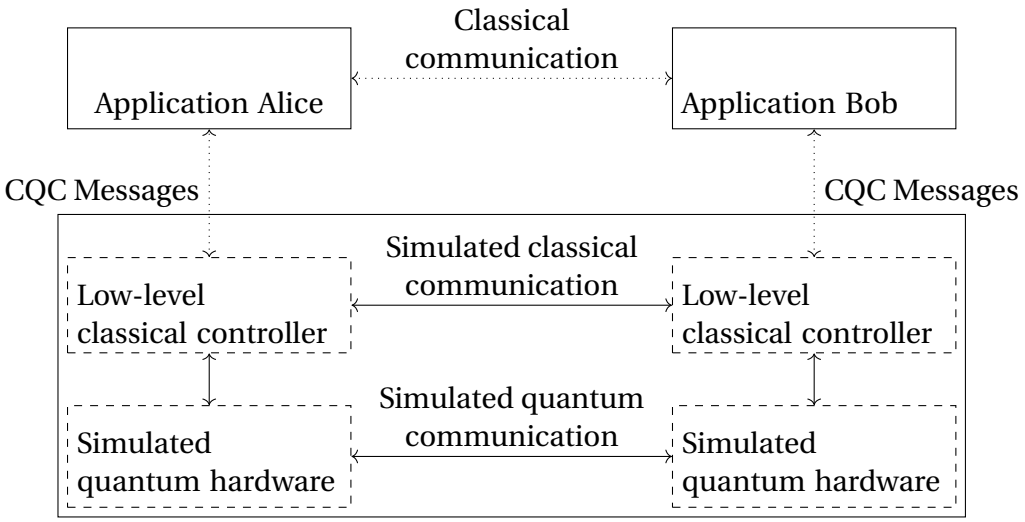

Figure 4.1: Overview of the architecture of a network with two nodes in the NetSquid simulator, similar to figure 2.1. The backends of every node run in the same process, where the classical controller of each node listen to their application level. Communication between the nodes is now done by simulated channels rather than real connections.

4.2. Quantum network emulation

We want the NetSquid simulation to behave as much like the real quantum hardware. This includes keeping track of some *simulation time* to simulate decoherence and delays. When running a simulation on NetSquid, the internal simulation time increases as fast as the machine the simulation is run on allows. This means the simulation can run much faster or much slower than it would have in a real scenario. We can simulate communication with a Mars base in seconds while it would take about half an hour in reality. Or simulation of a quantum state with 30 entangled qubits can take hours, where it would take a few microseconds when running on the real hardware. This is not a problem when running simulations on just NetSquid; we want the simulation to be done as fast as possible. It does become a problem when the simulation receives instructions from outside the simulation. The outside world of the simulation runs with a different internal clock than the simulated world. In our case the outside world is the real world running at one second per second, but we can also have another simulator communicating with the NetSquid simulator. A simulator that can receive inputs from outside the simulation is called an emulator [6, 12]. We need to turn NetSquid into an emulator.

For this we need some kind of time synchronization between the simulation and the real world. We can not run the simulation as fast as possible as the simulation allows. An instruction send to the simulation will be received too late in the simulation. For example Alice sends a command that advances the simulation time by one

hour, while Bob had a qubit stored in its memory. This qubit now has decohered by one hour when Bob next accesses the qubit, even though in real time Bob might only have had the qubit stored for a couple of milliseconds. To prevent this we can not let the simulation run faster than real time, we need to delay instructions in real time based on the operation time it would advance the simulation time. See figure 4.2.

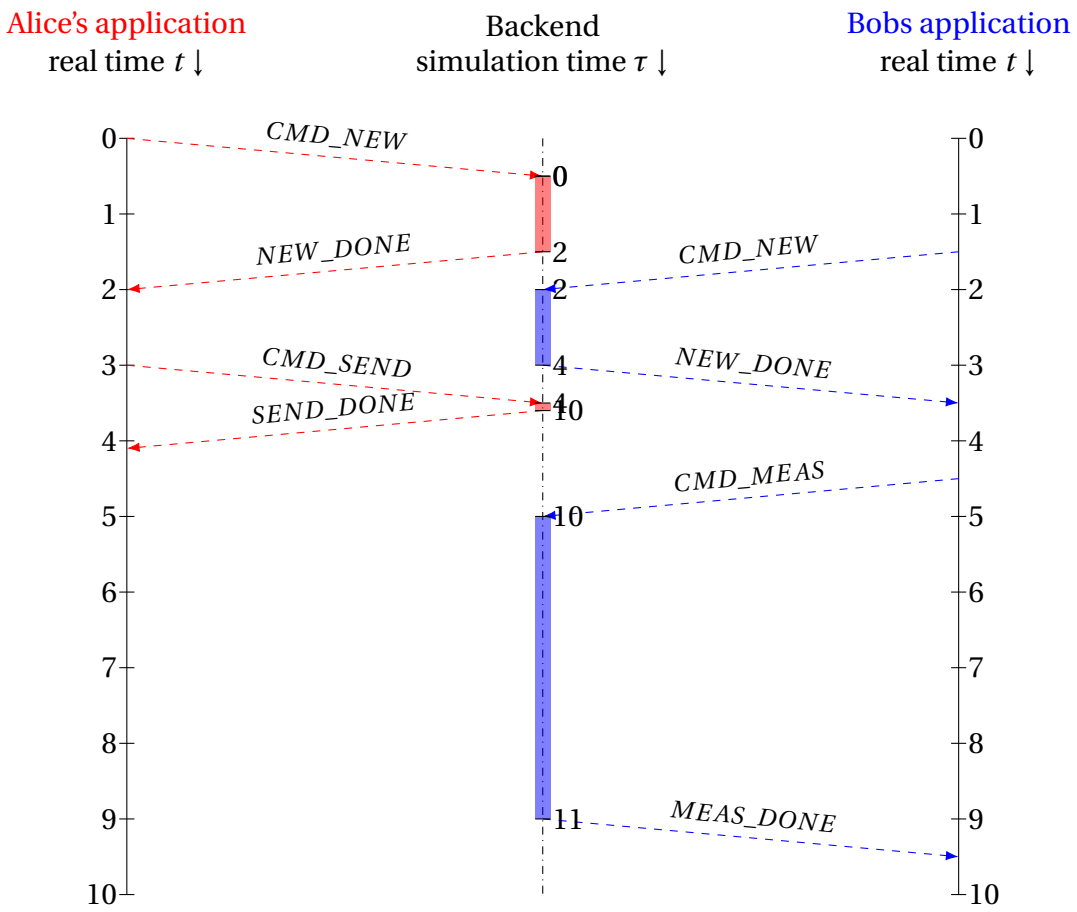

Figure 4.2: This timeline shows a **faulty** propagation of real and simulation time. Any command that comes in is directly simulated on the backend. CQC message from and Alice's application are indicated by red arrows on the left. Similarly Bobs CQC messages are indicated by blue arrows on the right. The red (Alice) and blue (Bob) bars indicate the backend is doing some indicate some computation. At $\tau = 4$ a command comes in to send a qubit to Bob, this command advances the simulation time by 6 time units. Now when Bob wants to measure its newly created qubit the qubit has already been alive for 6 time units. Much longer than the expected 1 time unit. Bobs qubit is much more decohered than expected. This is not desired, any application can forward the simulation time of the backend by any amount. This causes the simulations of all other applications to fast forward as well, so any messages that come in from those applications arrive much too late in the simulation time.

A desired timeline is shown in figure 4.3. However this scenario assumes that all instructions are computed instantaneously on the backend, this is not realistic.

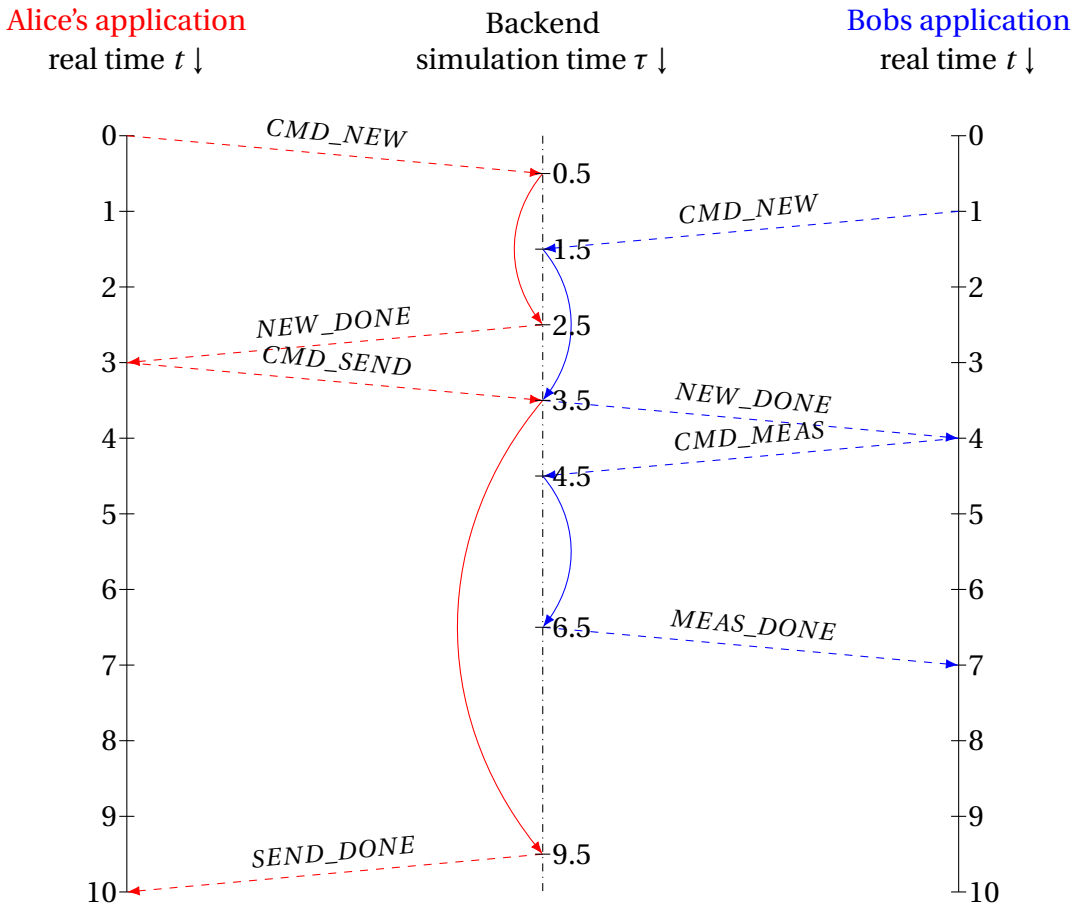

Figure 4.3: This imeline shows a **desired** order of events. Here the backend and application are 100% synchronized; when the discrete-time simulator increases its simulation time it always increases it to the current real time. At $t = 0.5$ a new command comes in to create a new qubit for Alice. Within the simulation it is defined that creating a new qubit would take $\Delta\tau = 2$, so the backend waits in real time for 2 time units until it creates the qubit at $t = 2.5$. The solid arrows indicate this delay in real time before the command is done. The time between Bobs new qubit and measuring it is now 1 as expected, no more decoherence is introduced. Note that receiving a message from the application also does some resynchronizing. This timeline does not consider the real time it takes to compute the instructions, so this timeline is not realistic to achieve.

What realistically happens is shown in figure 4.4 where we both slow down the simulation and instructions take some real computation time to compute.
Sometimes the simulation takes longer than expected to compute one instruction. This might happen for example when computing on a large quantum state, since

states grow exponentially in the amount of qubits, or when the simulation consist of many nodes that are all simulated on a single processor.
This can not be fixed. We can not speed up the simulation since it is bottlenecked by the computation. And we can not slow down real time.

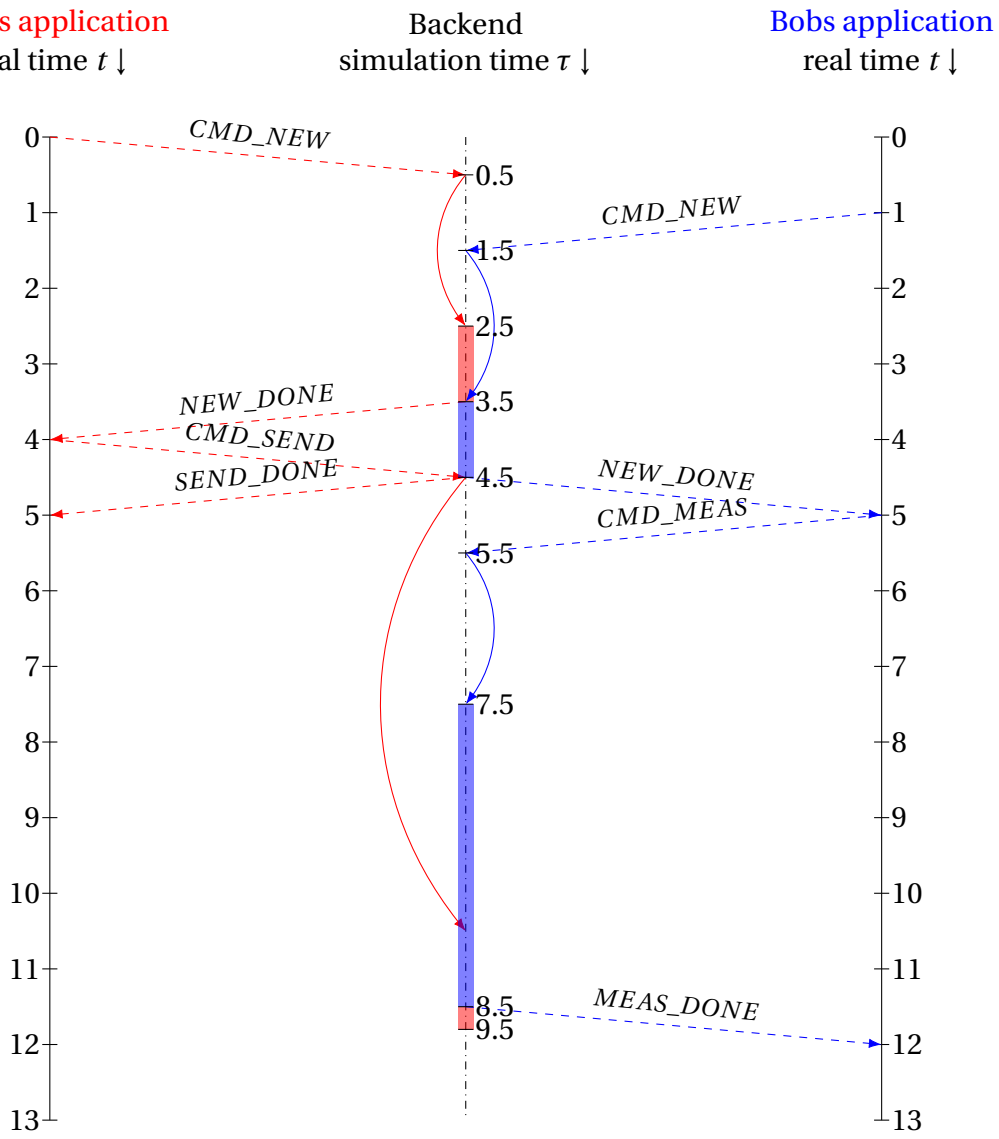

Figure 4.4: Timeline shows a **realistic** order of events where some operations take a real amount of time to compute on the backend. Here again the solid arrows indicate that the backend is waiting in real time to start the operation and the red and blue bars indicate the backend is doing some computation for Alice and Bob respectively. We now wait 6 time units in real time before doing the send. This gives the backend time to receive the measurement command from Bob and compute it. The measurement computation takes quite long. It is still being computed when Alice's qubit should be send. Instead it waits with this computation and only starts it after Bobs measurement has been finished.

4.3. Possible implementations

In chapter 3.3 some earlier works are discussed. In this chapter we go over some of those solutions and check their viability.

4.3.1. Time windows

In [23] a technique is implemented that slices up the simulation in discrete time windows. A synchronization component assigns the length to these time windows and acts as external time source. A scheduler keeps track of the current queue of events. If the time of execution of the next event is within the current time window it will be executed. Otherwise the synchronization component will cause the simulation to pause in real time until the start of the next window. If a new event is scheduled due to an instruction from outside the simulation, and this event resides in the current time window it will be executed immediately.
The length of a time window has to be carefully chosen. The optimal value depends on the protocols that are running, the simulated hardware, the machine the simulation runs on and the topology of your network.
If the length of the window is too **large** then events might happen too early, causing the simulation to run ahead of the real time. In fact if the length of the time window approaches to infinity, the emulation becomes a regular simulation where all computations are done as fast as possible.
If the length of a window is too **small** then there is a lot of synchronization needed, which in turn costs valuable computing time. Having such a small window also means that the computation time is often larger than the length of a window. This causes the next window to be delayed. The synchronizer will still try to start the window after at the correct time. This does mean the simulation will temporarily run faster than real time. Which, as explained earlier, risks processing incoming instructions too late.

4.3.2. Real time scheduler

With a real time scheduler [12, 20] the scheduler checks the simulation time of the next event to be executed, if this time is less than the current real time it will execute the event directly. This scheduler is part of the classical controller of the backend (see figure 2.1. Otherwise this event has to be done somewhere in the future, the scheduler will then in real time wait for the duration (in simulation time) of the event or if a new event is scheduled due to a new incoming message. A version of this was seen in figure 4.4.
If there are too many events or events are computationally heavy then the scheduler will fall behind in real time. Adding more computing power or distributing the

computation over multiple cores can lessen this problem, but does not solve the underlying issue.

4.3.3. Our solution

Using time windows is an infeasible solution. Picking the size of a time window proved to be a difficult task given the different use cases for our simulations. For networks with a large amount of nodes and high traffic between them it is desired to have a small time window so that in- and outgoing messages are scheduled accurately. But for networks that share large entangled states we want a larger time-window, since computations on these states take a long time, a large time window means the computation can sometimes start earlier, giving the window a bit more breathing time after the computation is finished.
With the time windows the synchronizer has to synchronize every time window. In the real time scheduler the scheduler synchronizes with every event.
We have chosen for an adjusted implementation of the real time scheduler. To reduce the amount of synchronizations we only synchronize when needed. Within the quantum simulator we have a lot of events that are irrelevant to the emulation, such as events that trigger within certain protocols that run in the background of the simulation. So instead we only synchronize on events that are fired when an instruction has finished.

An operation i has some operation time in the simulated world τ_i and some computation time t_i it takes to compute the operation on the machine that runs the simulation. τ_i is defined when setting up the simulation and is always constant under the same conditions within the simulation. The computation time t_i can vary due to small fluctuations within in the machine, or the machine might be busy computing something else.
With the implementation as in figure 4.4 every operation takes $t_i + \tau_i$ real time within the run of a simulation. One part waiting, one part computing. This can be improved by not waiting a full τ_i seconds after the previous operation has done, but rather we wait τ_i seconds after the previous operation has *started*.
It might be possible that at this time the previous operation is still being computed, so $t_{i-1} > \tau_i$. In that case we start our operation directly after when the previous has finished.
So when running a sequence of operations then the real time each operation contributes to the total computation time is $max(\tau_i, t_{i-1})$. And thus the total computation time is $\sum_i max(\tau_i, t_{i-1})$. Figure 4.5 shows this scenario.

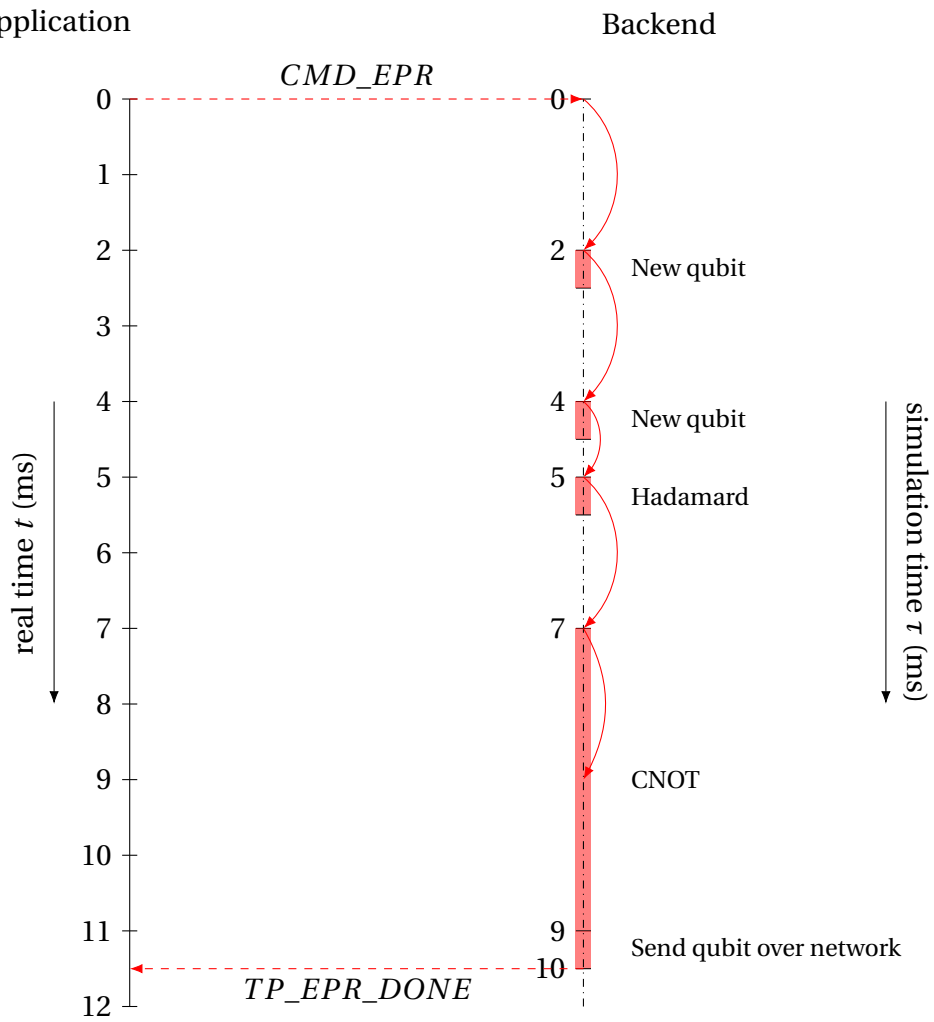

Figure 4.5: This timeline shows a slight optimization of event order. When creating an EPR pair the backend will schedule multiple events. Rather than scheduling the next event only after the simulation of the current event is finished, we can already schedule the next event from the start of the simulation of the current event. This means that the simulation is slightly faster overall in real time. The CNOT in this case takes longer to compute (4ms) then it would have on the real hardware (2ms). This might happen since combining two large quantum states is a quite resource heavy computation in a classical simulation. The backend still schedules the next operation after 2ms. However this event has to wait for the CNOT to be done computing, and is then done directly after the CNOT is finished.

4.3.4. Other optimizations and outlook

Other optimizations can be made in future works. The main one is that instead of first waiting for τ_i amount of time in real time and only then start the computation, we start the computation right away and then halt the backend for $min(\tau_i - t_i, 0)$ amount of time, as shows in figure 4.6. Unfortunately NetSquid was not well fitted yet for this implementation, but it is worth looking into this in future works.

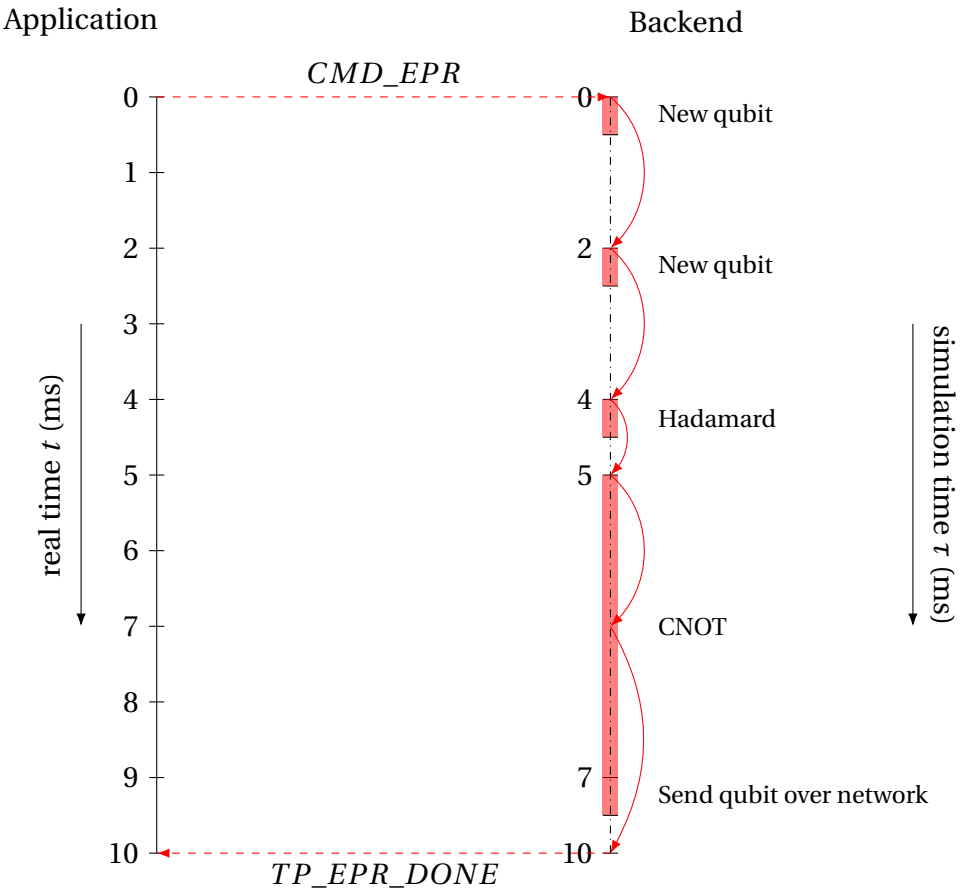

Figure 4.6: This timeline shows a possible optimization. Here when a command arrives the backend starts computing this instruction right away. Once the instruction is done in then waits for the remaining amount of time before starting the next instruction. This way the simulation does not run faster than real time, and we do not spend unnessary time waiting while we can already compute some instructions.

5. Implementation

5.1. Instructions

In order to instruct NetSquid with the commands send through the CQC interface a class is built to handle the messages.
The pseudocode in algorithm 5.1 shows what happens when this message handler receives a message.

Algorithm 5.1: Message Handler

```
pending_programs = {}
busy = False
quantum_program
def handle_cqc_message(header, message):
  quantum_program = new Program()
  add_commands(header, message)
  if header.should_notify:
    add_command(send_done)
  quantum_program.done_callback(execute_next_program)
  pending_programs.append(quantum_program)
  if not busy:
    execute_next_program()

def add_command(params*)
  quantum_program.add_instruction(params*,
    should_yield=False)
  quantum_program.add_instruction(command_done,
    should_yield=True)

def command_done():
  # Event that fires when an instruction has finished
  # Get duration time of the program upto the next yield
  instr_duration = quantum_processor.sequence_duration
  simulation_manager.new_event(instr_duration)
```

```
25
26    def execute_next_program():
27      if pending_programs is not empty:
28        busy = True
29        # Remove first program from program list
30        program = pending_programs.pop()
31        # Returns the duration time of the first instruction
32        # up to the first yield
33        instr_duration = quantum_processor.
34          execute_program(program)
35        simulation_manager.new_event(instr_duration)
36      else:
37        busy = False
```

On line 5 a new *Program* is created. This program is essentially a list with instructions for the underlying quantum processor. For each instruction a new event is created for discrete-even simulator. When this event is fired in simulation time depends on the instruction. For example doing an $XGATE$ takes some time to execute, so this needs to be scheduled within the simulation.
On line 6 the instructions in the CQC message are parsed and iteratively added to the list of instructions in the program by calling $add\_instruction()$.
Line 7 and 8 add two more instructions to send a CQC message back to the application to notify the command is done if required.
A quantum processor can only execute one program at the same time. It might be the case that the backend receives another set of instruction while another program is still being executed. Therefore this new program needs to wait before it can be executed. It is put in a queue of $pending\_programs$ in line 10. The program at the front of the queue should be executed either if the quantum processor is not busy (line 11 and 12) or when the previous program has finished. The latter is achieved in line 9 by adding a *callback* to the program that fires $execute\_next\_program$ when the program finishes.

When adding a command two instructions are scheduled. The first instruction is the physical instruction of the command (line 15-16). For example doing some quantum gate or measurement. The second instruction fires an event to indicate the previous instruction is done and the next simulation manager should schedule a time in *real time* to synchronize with this command (line 17-18). The $should\_yield$ parameter here tells the processor up to which instruction it should run until the simulation as to synchronize with the real world. Using that we can get the simulation time it would take to execute the next set of instructions up to the first yield. This time is used in the simulation manager (line 20-24, 33-35 and section 5.2).

Sometimes more instructions are scheduled. For example in the case of a measurement we schedule an additional instruction that would send the measurement result back to the application.

Instructions are removed from the instruction list after they are handled.

5.1.1. Dynamic program

Some commands cause the set of instructions to change based on what happens in the processor. For example when creating an EPR pair we need to know the memory position of the two created qubits before we can do any other operations on them. For this the set of instructions in a program dynamically changes while the program is already running. The psuedocode in algorithm 5.2 shows what happens when an *EPR command* is issued.

Algorithm 5.2: dynamic program

```
1  key = 0
2  def cmd_epr(header, command, extra):
3    key1 = key
4    key2 = key + 1
5    key = key + 2
6    add_command(INIT_QUBIT, output_key=key1)
7    add_command(INIT_QUBIT, output_key=key2)
8
9    def create_epr(event):
10     q_id1 = quantum_program.output[key1]
11     q_id2 = quantum_program.output[key2]
12     add_command(H, q_id1, instr_index=0)
13     add_command(CNOT, [q_id1, q_id2], instr_index=2)
14     send_CQC_back(CMD_EPR_OK, qubit_id = q_id1)
15     add_command(SEND, extra.receiver_details, instr_index=4)
16
17   event = EventHandler(create_epr)
18   add_command(FIRE_EVENT, event)
```

We first initialize two new qubits in line 6 and 7. We don't know yet what the memory positions of those two qubits are going to be. But we still need to know those memory positions in order to the additional operations on them. For this reason we tell the quantum program to store those memory positions in the output of the program by some key in a dictionary. When the memory position are available we could look in

this dictionary with the correct keys (line 10 and 11).
After the instructions to initialize two new qubits are added we add a command to call the $create\_epr$ function. This function will not run until the program has started and the two qubits are created. So when $create\_epr$ is called the program is already running, yet we still need to add additional instructions. Namely an *Hadamard* (line 12) a *CNOT* (line 13) and sending it to another node (line 15).
Normally when adding new instructions in a program they are just added sequentially. We now need to squeeze in new instructions at the start to prioritize them. For we this we give an additional parameter that tells the program at which index the instruction should be inserted. The *Hadamard* should be done right away so is inserted at the start of list of instructions ($instr\_index = 0$). After the Hadamard we do a *CNOT*, the Hadamard scheduled two instructions, one to do the command and one to notify that the command has finished, so the *CNOT* starts at index 2. And similarly the *SEND* command starts at index 4.
Once the *SEND* command has finished the program continues normally.

5.2. Simulation manager

Keeping the simulation time synchronous with the real time is a task of the simulation manager. The simulation manager starts and stops the simulation so that the time within the simulation does not increase faster than real time. The code in algorithm 5.3 shows a simplified version of the simulation manager.
When the backend receives one or multiple instructions from a higher layer it creates and schedules a *Quantum* program that can run on the quantum processor. This quantum program schedules one or more events. The *simulation* time τ_0 of the first fired event is known, this time is send to the simulation manager with a call of $SimulationManager.new\_event(\tau_0)$.
Upon receiving this time the simulation first synchronizes the simulation time. It keeps track of the real time $last\_sim$ the last simulation has finished, and advanced the simulation time by the difference between the current real time and $last\_sim$. In figure 4.4 the first message arrives at $\tau = 0.5$ in the backend. This is the time it took to the send the message from the application to the backend.
Once the backend has synchronized and the current simulation time is $\tau_{cur} = netsquid.simulation\_time()$ it will create a timer in real time for the duration of the event τ_0. When this timer is fired the simulation will advance the simulation time to $\tau_{cur} + \tau_0$.
While the timer is running it is possible that a new message comes in before the timer has finished. This happens in figure 4.4 at $\tau = 1.5$ when a message from Bobs comes in to create a new qubit, while Alice's timer is running. At this moment the simulation will synchronize again and starts the timer of this new command. This timer might be scheduled after the first event (first solid blue arrow from Bob in figure 4.4). Or it

might be scheduled earlier than the first event (second solid blue arrow from Bob in figure 4.4) in which case that message will be handled first.
When a timer finishes while the simulation is still computing something it will simply wait for that simulation to finish and schedule it right after. In Alice's second command in figure 4.4 the simulation is still doing a computation of Bob and is therefore locked, so when Alice's command comes in it starts after Bobs computation is done and the lock as been removed. Locking is done on line 24 of algorithm 5.3. Multiple threads are with timers might be running, the *lock* makes sure only one thread can enter its code at a time.
On line 7-9 we synchronize the simulation time with the real time. We compute the difference in real time since the last simulation has finished, and advance the simulation by this difference.

Algorithm 5.3: simulation manager

```
1  # Get the current real time
2  last_sim = get_current_time()
3  lock = threading.RLock()
4
5  def new_event(duration):
6    # synchronize simulation
7    sync_end_time = netsquid.simulation_time() +
8     get_current_time() - last_sim
9    run_simulation(end_time=sync_end_time)
10
11   end_time = netsquid.simulation_time() + duration
12
13   # Create timer in a new thread to simulate
14   # upto end_time after duration seconds
15   timer = threading.Timer(function=run_simulation,
16     parameters={end_time},
17     interval=duration)
18   timer.start()
19
20 def run_simulation(end_time):
21   # run the simulation up until end_time simulation time
22   # We need to lock the running of the simulation.
23   # Can't start multiple simulations
24   with _lock:
25     # It might be possible that another event already ran
26     # the simulation up to the current end_time
27     # So we don't need to run the simulation anymore
```

```
28          if end_time >= netsquid.simulation_time():
29              netsquid.run_sim(end_time=end_time)
30              # update the global last simulation time to
31              # the current real time.
32              last_sim = get_current_time()
```

6. Evaluation

The simulation manager is tested in different scenarios. We test for correctness and speed.

6.1. Correctness

For testing purposes we artificially increased the computation time by implementing a sleep method within the method. And the operation time for qubit operations has been set to high values to check if the delays are applied correctly.

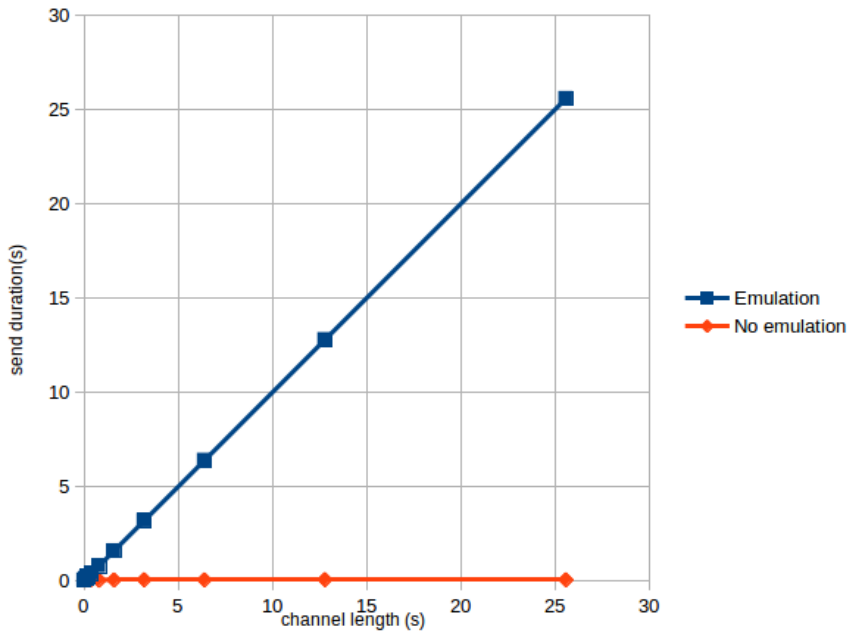

Figure 6.1: Real time that has advanced between sending and receiving a qubit as function of channel length between two nodes. With and without emulation.

We use the example as described in section 2.4 where we increase the length of the

channel between Alice and Bob. We run this example with and without synchronization. 6.1 shows the real time it takes for Bob to receive its qubit after Alice has send it. The figure shows that with emulation the real time that advanced is always slightly more than the time it takes for a qubit to be send over the channel between Alice and Bob. Without emulation the qubit are always send near instantly.

6.2. Speed

In order to test for speed in a realistic scenario we run an application on the SimulaQron backend and on the NetSquid backend. To compare the two simulations we now set the operation time of all operations within NetSquid to 0, this means the simulation time will not advance due to the operations. We create a GHZ state of n qubits [1]. When simulating the GHZ state on a classical computer the amount of information to store grows exponentially in n, and thus it is also expected that the time to create this state grows exponentially.
Within both simulators it is possible to store the state $|\psi\rangle$ in different ways. Either as a density matrix (DM) $|\psi\rangle\langle\psi|$, as regular KET vector $|\psi\rangle$ or as a stabilizer state. Not all states are stabilizer states, but stabilizer states can be simulated in efficiently polynomial time instead of in exponentially as $\mathcal{O}(2^n)$. All three formalisms are tested and timed for both simulators. The result of this is shown in figure 6.2.
As expected the density matrix formalism and KET formalism both scale exponentially when creating bigger GHZ states, where creating a stabilizer state is done linearly in both SimulaQron and NetSquid.
SimulaQron is slightly faster than NetSquid for each formalism. This might be because we now have a more roundabout way of doing the simulation with the emulation. The simulation constantly is started and stopped by the simulation manager, which adds unnecessary overhead.

Finally we also compare the speeds of NetSquid and SimulaQron by repeatedly creating and measuring qubits. This way we check which of the two simulators is better to use for non-computation heavy operations. We again set all operation times of each operation within the simulation to 0, so there is no delay added by the simulation manager. The results are shown in figure 6.3.
In this case SimulaQron is slightly faster. So for testing applications without noises or delays on channels SimulaQron is the better simulation still.

[1] A GHZ state is the state $\frac{1}{\sqrt{2}}(|0...0\rangle + |1...1\rangle)$

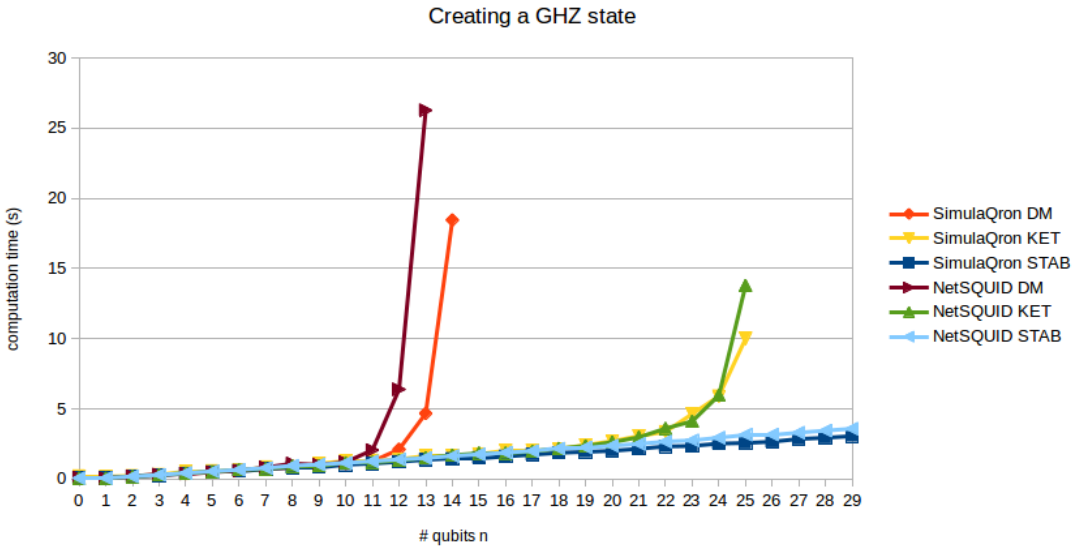

Figure 6.2: Time it takes to create a GHZ state of n qubits in different backends and formalizations. SimulaQron is faster for each qubit formalism. The three formalisms are density matrix formalism (DM), KET formalism and Stabilizer formalism.

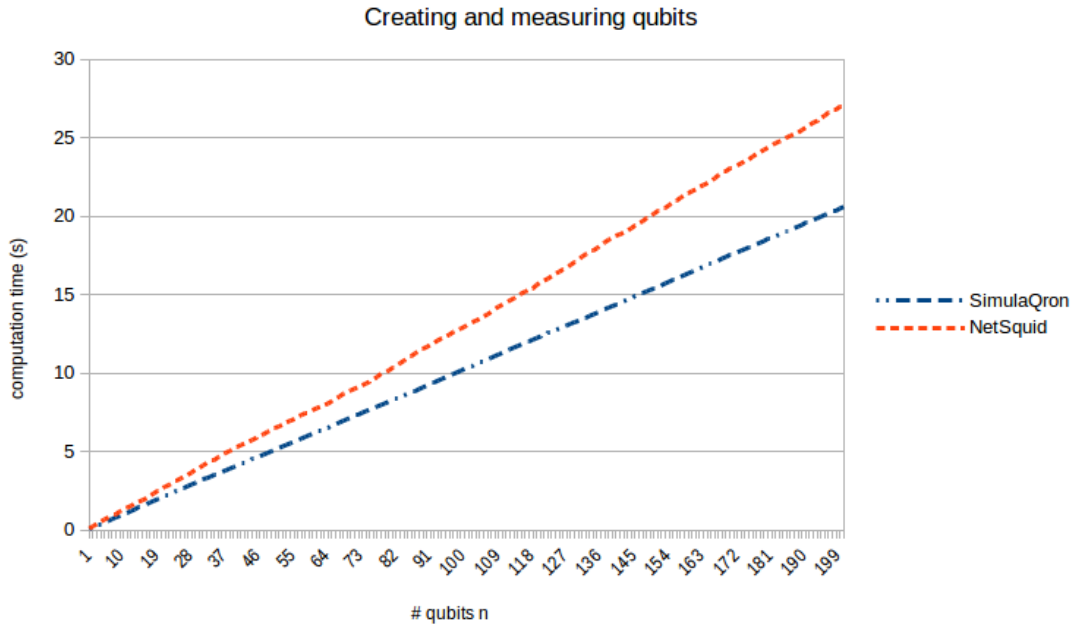

Figure 6.3: Time it takes to create and measure n qubits in SimulaQron and NetSquid.

7. Conclusion

We are in the process of building all layers of a fully realized quantum network stack. Each layer should be independent on the other layers. This makes it possible to already work on a layer when other layers are realized yet.
High-level applications are developed to be run on a quantum network. These applications talk to the quantum backend with so called CQC messages. Currently real scalable quantum computers are not available yet. As a substitute we simulate the quantum network classically instead. This way we can still test and develop the higher classical layers without the need of quantum hardware. One such simulator that can receive, parse and send CQC messages is SimulaQron.
If we want accurate simulations, then they need to behave as close as possible to the quantum hardware, so they should simulate noises, losses and delays as well. SimulaQron is not suited for this, but the NetSquid simulator is.
NetSquid did not have logic to handle CQC messages yet. This thesis has added this logic. For each node in the NetSquid simulation a message handler was added that handles the incoming and outgoing CQC messages from the higher level application for that node. It creates programs to be run on the simulated quantum processor. For this an interface is added between the classical and quantum layers that translates classical commands to quantum instructions. The application sends messages to the backend in real time, whereas the simulation in the backend runs in simulation time. The simulation and outside world run on different clocks, so these clocks have to be synchronized to prevent the simulation running faster than the wall time. This synchronization is done by a simulation manager that stops and runs the simulation when new instructions are scheduled.
We showed that the simulation is correctly slowed down when having large operation times within the simulation.
Running the emulation has similar computation times as SimulaQron, so there is no big drawback when running a program with the emulation or on SimulaQron if you do not care about simulated delays and operation times.
There are still some optimizations to be made. Such as moving the start of the computation of an operation to the moment this operation is scheduled, rather than to first wait for the duration of the operation in real time.

Bibliography

[1] J. Ahrenholz, T. Goff, and B. Adamson. Integration of the core and emane network emulators. In *2011 - MILCOM 2011 Military Communications Conference*, pages 1870–1875, Nov 2011. doi: 10.1109/MILCOM.2011.6127585.

[2] Jeff Ahrenholz, Claudiu Danilov, Thomas R. Henderson, and Jae Kim. Core: A real-time network emulator. pages 1–7, 11 2008. ISBN 978-1-4244-2676-8. doi: 10.1109/MILCOM.2008.4753614.

[3] M. Allman and S. Ostermann. One: The ohio network emulator. Technical Report 19972, Ohio University, 1997.

[4] C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing*, page 175, India, 1984.

[5] H.-J. Briegel, W. Dür, J. I. Cirac, and P. Zoller. Quantum repeaters: The role of imperfect local operations in quantum communication. *Phys. Rev. Lett.*, 81:5932–5935, Dec 1998. doi: 10.1103/PhysRevLett.81.5932. URL `https://link.aps.org/doi/10.1103/PhysRevLett.81.5932`.

[6] Mark Carson and Darrin Santay. Nist net: A linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, July 2003. ISSN 0146-4833. doi: 10.1145/956993.957007. URL `http://doi.acm.org/10.1145/956993.957007`.

[7] Andrew M. Childs. Secure assisted quantum computation. *Quantum Info. Comput.*, 5(6):456–466, September 2005. ISSN 1533-7146. URL `http://dl.acm.org/citation.cfm?id=2011670.2011674`.

[8] Axel Dahlberg and Stephanie Wehner. SimulaQron—a simulator for developing quantum internet software. *Quantum Science and Technology*, 4(1):015001, sep 2018. doi: 10.1088/2058-9565/aad56e. URL `https://doi.org/10.1088%2F2058-9565%2Faad56e`.

[9] Axel Dahlberg, Matthew Skrzypczyk, Tim Coopmans, Leon Wubben, Filip Rozpędek, Matteo Pompili, Arian Stolk, Przemysław Pawełczak, Robert Knegjens, Julio de Oliveira Filho, Ronald Hanson, and Stephanie Wehner. A link layer

protocol for quantum networks. *SIGCOMM Comput. Commun. Rev.*, 2019. doi: 10.1145/3341302.3342070.

[10] Ivan B. Damgård, Serge Fehr, Louis Salvail, and Christian Schaffner. Secure identification and qkd in the bounded-quantum-storage model. *Theoretical Computer Science*, 560(1):12–26, 12 2014. ISSN 0304-3975. doi: 10.1016/j.tcs.2014.09.014.

[11] Artur K. Ekert. Quantum cryptography based on bell's theorem. *Phys. Rev. Lett.*, 67:661–663, Aug 1991. doi: 10.1103/PhysRevLett.67.661. URL https://link.aps.org/doi/10.1103/PhysRevLett.67.661.

[12] K. Fall. Network emulation in the vint/ns simulator. In *Proceedings IEEE International Symposium on Computers and Communications (Cat. No.PR00250)*, pages 244–250, July 1999. doi: 10.1109/ISCC.1999.780820.

[13] Peter C. Humphreys, Norbert Kalb, Jaco P. J. Morits, Raymond N. Schouten, Raymond F. L. Vermeulen, Daniel J. Twitchen, Matthew Markham, and Ronald Hanson. Deterministic delivery of remote entanglement on a quantum network. *Nature*, 558(7709):268–273, 2018. ISSN 1476-4687. doi: 10.1038/s41586-018-0200-5. URL https://doi.org/10.1038/s41586-018-0200-5.

[14] I. V. Inlek, C. Crocker, M. Lichtman, K. Sosnova, and C. Monroe. Multispecies trapped-ion node for quantum networking. *Phys. Rev. Lett.*, 118:250502, Jun 2017. doi: 10.1103/PhysRevLett.118.250502. URL https://link.aps.org/doi/10.1103/PhysRevLett.118.250502.

[15] Richard Jozsa, Daniel S. Abrams, Jonathan P. Dowling, and Colin P. Williams. Quantum clock synchronization based on shared prior entanglement. *Phys. Rev. Lett.*, 85:2010–2013, Aug 2000. doi: 10.1103/PhysRevLett.85.2010. URL https://link.aps.org/doi/10.1103/PhysRevLett.85.2010.

[16] Jędrzej Kaniewski and Stephanie Wehner. Device-independent two-party cryptography secure against sequential attacks. *New Journal of Physics*, 18(5):055004, may 2016. doi: 10.1088/1367-2630/18/5/055004. URL https://doi.org/10.1088%2F1367-2630%2F18%2F5%2F055004.

[17] A Kellerer. Quantum telescopes. *Astronomy & Geophysics*, 55(3):3.28–3.32, 06 2014. ISSN 1366-8781. doi: 10.1093/astrogeo/atu126. URL https://doi.org/10.1093/astrogeo/atu126.

[18] Cameron Kiddle. *Scalable Network Emulation*. PhD thesis, Calgary, Alta., Canada, Canada, 2005. AAINR03870.

[19] Rob Knegjes and Julio Oliveira. Netsquid, 2018. URL https://netsquid.org/.

[20] D. Mahrenholz and S. Ivanov. Real-time network emulation with ns-2. In *Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 29–36, Oct 2004. doi: 10.1109/DS-RT.2004.34.

[21] A Pirker and W Dür. A quantum network stack and protocols for reliable entanglement-based networks. *New Journal of Physics*, 21(3):033003, mar 2019. doi: 10.1088/1367-2630/ab05f7. URL https://doi.org/10.1088%2F1367-2630%2Fab05f7.

[22] A. Sahu, A. Goulart, and K. Butler-Purry. Modeling ami network for real-time simulation in ns-3. In *2016 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–8, Oct 2016.

[23] Elias Weingaertner, Florian Schmidt, Hendrik Lehn, Tobias Heer, and Klaus Wehrle. Slicetime: A platform for scalable and accurate network emulation. 01 2011.

[24] Elias Weingärtner, Florian Schmidt, Tobias Heer, and Klaus Wehrle. Synchronized network emulation: Matching prototypes with complex simulations. *SIGMETRICS Perform. Eval. Rev.*, 36(2):58–63, August 2008. ISSN 0163-5999. doi: 10.1145/1453175.1453185. URL http://doi.acm.org/10.1145/1453175.1453185.