NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report Number: NU-CS-2020-15

December, 2020

A Simulator for Distributed Quantum Computing

Gaurav Chaudhary

Abstract

This project presents a simulator for distributed quantum computing by leveraging existing quantum hardware and simulators to model a system of interconnected quantum computers. Two well-known quantum computing simulators are used to achieve this: Qiskit and NetSquid. Qiskit by IBM is a quantum information processing simulator with approximate models for actual hardware by IBM and other companies. Qiskit supports two popular quantum computing technologies, superconducting transmon qubits and ion traps, which make it an ideal vehicle to model quantum processors. NetSquid is developed by QuTech at TUDelft. NetSquid is a network simulator with support for network nodes as noisy quantum processors. The work in this MS project leverages Qiskit to model quantum processors based on actual hardware and uses those as nodes in a network simulation in NetSquid. By doing this, this work aims to faithfully simulate distributed quantum computation.

Keywords

Quantum Computing, Distributed Quantum Computing, Quantum Networks, IBM Qiskit, NetSquid

This work was partially supported by NSF award CCF-1453853.

MS Project

A Simulator for Distributed Quantum Computing

By: Gaurav Chaudhary

Advisor: Professor Nikos Hardavellas

Committee: Professor Nikos Hardavellas (Chair), Professor Prem Kumar, Professor Randall Berry

Department of Computer Science, Department of Electrical and Computer Engineering, Northwestern University

Contents

1 Abstract

This project presents a simulator for distributed quantum computing by leveraging existing quantum hardware and simulators to model a system of interconnected quantum computers. Two well-known quantum computing simulators are used to achieve this: Qiskit and NetSquid. Qiskit by IBM is a quantum information processing simulator with approximate models for actual hardware by IBM and other companies. Qiskit supports two popular quantum computing technologies, superconducting transmon qubits and ion traps, which make it an ideal vehicle to model quantum processors. NetSquid is developed by QuTech at TUDelft. NetSquid is a network simulator with support for network nodes as noisy quantum processors. The work in this MS project leverages Qiskit to model quantum processors based on actual hardware and uses those as nodes in a network simulation in NetSquid. By doing this, this work aims to faithfully simulate distributed quantum computation.

2 Introduction

In the NISQ (Noisy Intermediate-Scale Quantum Computing) era, quantum computers are limited in the size of computations they can perform. It is challenging to increase the number of physical qubits on a single chip. One of the major contributors of this is the difficulty of maintaining a quantum state and preventing its interaction with the environment. To counter the problem of thermal coupling and interaction with the environment, quantum computers are cooled just above absolute zero using a dilution refrigerator. As a quantum computer scales up, the cost of logistics, like cooling, rises exponentially. The cost of maintaining an intermediate-scale quantum computer is enormous due to this. The largest quantum computer that exists at the writing of this report is around 72-qubits.

To get around the problem of scaling up a single quantum computer, researchers have proposed distributed quantum computers, which can leverage multiple intermediate-scale quantum computers connected in a network to create one big logical quantum computer. Distributed quantum computing is a broad area of research. There are architectural and system-level trade-offs which need to be studied extensively: hardware technologies, the distance between the nodes, entanglement generation and qubit transmission to name a few. Since distributed quantum computers are in a nascent stage, a high-fidelity simulator is needed to explore the design space.

Existing quantum simulators can simulate one aspect of the system well. For instance, Qiskit by IBM is good at simulating quantum information processors. On the other hand, NetSquid by QuTech is good at simulating quantum networks. This work aims to leverage the strengths of Qiskit and NetSquid to create a new framework which can simulate distributed quantum algorithms with high fidelity. The aim is to create a framework to help study system architecture and algorithms and to uncover implications of important design decisions.

3 Materials and Methods

This project leverages features from two well-known quantum computing simulators: Qiskit by IBM, and NetSquid by TUDelft. The requirement for a faithful distributed quantum computing simulator is that it should be capable of simulating all the aspects of the system with appropriate error models. Broadly, it should be able to simulate the following:

1. Quantum Processor or the nodes where the actual computation is performed
2. Quantum Information Channels or the physical link via which the quantum information flows
3. Quantum Memories or the physical devices used to store quantum information
4. The interaction of all these components in a system and the errors accumulated due to factors like latencies and stalling

The choice of two-component simulators: Qiskit and NetSquid is not a random one. The following subsections describe the relevant functionality of Qiskit and NetSquid in detail.

3.1 Qiskit

Qiskit can be used to automatically generate a basic noise model for an IBMQ hardware devices and use this model to perform simulations of Quantum Circuits to study the effects of errors which occur on real noisy devices.

These automatic models are only an approximation of the real errors that occur on actual devices, due to the fact that they must be built from a limited set of input parameters related to average error rates on gates.

The model of an actual machine is generated using the calibration information reported in the BackendProperties of a device which include:

1. T1-T2 constants for relaxation time
2. Readout error probabilities for each qubit
3. Gate errors for each basis gate on each qubit and allowed combinations for 2-qubit gates
4. Gate lengths of each basis gate on each qubit

The code to generate a basic device model of ibmq_16_melbourne (a 16-qubit quantum computer by IBM) is given in Figure 3-1.

The provider is the user account created on IBM Qiskit website. The get_backend call to ibmq_16_melbourne specifies the actual hardware device for which the simulation needs to be performed. The ibmq_16_melbourne can be replaced with any other backend supported by Qiskit. For instance, it can also be aqt_innusbruck which is a 6-qubit Trapped Ion quantum computer by AQT Technologies. This is another major advantage of Qiskit; it supports a wide variety of technologies to act as possible backends.

The from_backend call extracts out the noise model from a given backend, that is, the 4 points mentioned above. This is used as an argument into the execute function.

The coupling map describes the microarchitecture of the underlying hardware. It basically specifies which qubits are closer to each other in the hardware. Consequently, this provides a constraint list to perform two-qubit gates on the physical qubits.

```
from qiskit.visualization import plot_histogram
from qiskit.providers.aer.noise import NoiseModel

# Build noise model from backend properties
provider = IBMQ.load_account()
backend = provider.get_backend('ibmq_16_melbourne')
noise_model = NoiseModel.from_backend(backend)

# Get coupling map from backend
coupling_map = backend.configuration().coupling_map

# Get basis gates from noise model
basis_gates = noise_model.basis_gates

# Make a circuit
circ = QuantumCircuit(3, 3)
circ.h(0)
circ.cx(0, 1)
circ.cx(1, 2)
circ.measure([0, 1, 2], [0, 1, 2])

# Perform a noise simulation
result = execute(circ, Aer.get_backend('qasm_simulator'),
                 coupling_map=coupling_map,
                 basis_gates=basis_gates,
                 noise_model=noise_model).result()
counts = result.get_counts(0)
plot_histogram(counts)
```

Figure 3-1

The basis gate set in a noise model is a collection of the most basic gates which form the complete gate set for a quantum processor.

3.1.1 Basis Gate set for ibmq_16_melbourne

- Identity Gate - I gate
- U1 gate – Rotation along the Z axis

$$U1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$$

- U2 gate –

$$U2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{pmatrix}$$

- U3 gate –

$$U3(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\phi}\sin(\frac{\theta}{2}) & e^{i(\phi+\lambda)}\cos(\frac{\theta}{2}) \end{pmatrix}$$

- CX – Controlled X gate
- Measurement gate

All the gates in the system are modelled by some combination of these gates. The errors provided in the backend properties are given in terms of these basis gates.

Modelling of some common gates with basis gates in Qiskit.

Basis Gate	Common Gate
U2	H
U3	X
U1	Z
U3	Y
U1	S
CX	CX
U1(1)-->CX-->U1(1)	CY
U2(1)-->CX-->U2(1)	CZ
Measure	Measure

The following sub-section provides details on the error models provided by a given backend.

3.1.2 Error Models in Device Backend

3.1.2.1 T1-T2 constants

T1-T2 constants together form the thermal relaxation error for a given qubit. This is analogous to the decoherence model in quantum mechanics, where a quantum system couples with its environment and loses the coherence of its state. This process appears to be non-unitary if the system is studied in isolation but the overall system + environment comes out to be unitary.

T1 is known as the amplitude damping time and T2 is known as the dephasing time.

These can be extracted for a given backend by use of the following API

`device.thermal_relaxation_values` (properties)	Return T1, T2 and frequency values from a devices BackendProperties.

3.1.2.2 *Readout Errors*

Readout errors for a backend are provided as conditional probabilities. For an N-qubit readout error, the probabilities are given as:

```
probabilities[m] = [P(0|m), P(1|m), ..., P(2 ** N - 1|m)]
```

Where P(j|m) is the probability of recording a measurement outcome of m as the value j.

For one qubit, the representation is as follows:

```
probabilities[0] = [P("0"|"0"), P("1"|"0")]
probabilities[1] = [P("0"|"1"), P("1"|"1")]
```

Readout errors can be extracted from a given backend by using the following API

`device.readout_error_values` (properties)	Return readout error values from a devices BackendProperties.

Which returns the readout errors in the following format for each qubit.

{P(1|0), P(0|1)} probabilities for measuring an orthogonal state

3.1.2.3 *Gate Errors and Gate Lengths*

Gate errors in Qiskit refer to as the depolarization errors when a gate is applied to a particular qubit. Gate Length refers to the latency (ns) of the gate being applied on a particular qubit.

Gate Errors and Gate Lengths can be extracted from the device backend by the following API:

`device.gate_param_values` (properties)	Return parameter error values from a devices BackendProperties.

This call returns the gate errors and length values in the following format A list of tuples (name, qubits, time, error) for all physically possible combinations of qubits.

3.2 NetSquid

NetSquid is a quantum network simulation package for python. NetSquid incorporates all the major components to faithfully simulate a quantum network. It does so by integrating a quantum information processing simulator with an event-based classical network simulator. NetSquid is very well suited to simulate distributed quantum systems, given that an actual hardware processor is approximated as a quantum processor. NetSquid defines the abstraction of a quantum processor to be used as a node in the network. It also defines quantum/classical channels for quantum/classical information communication. It also enables the user to incorporate custom noise models to all the components. NetSquid has support for writing hardware-agnostic code by the quantum program abstraction. Consequently, whenever the underlying hardware technology is changed, no code rewrite is required. These components of NetSquid are described in detail in the following sub-sections.

3.2.1 Quantum Processor

A Quantum Processor extends the base class Quantum Memory and adds the functionality of quantum computation to the memory. Instructions on a Quantum Processor can be executed via a Quantum Program or directly by using the execute command. A Quantum Processor takes as input all the possible physical instructions that it can execute.

Here is an example declaration of a quantum processor:

```
qproc = QuantumProcessor(
    "TestQPD", num_positions=4,
    mem_noise_models=[DepolarNoiseModel(500)] * 4, phys_instructions=[
        PhysicalInstruction(INSTR_INIT, duration=2.5, parallel=True),
        PhysicalInstruction(INSTR_X, duration=1, topology=[1,2,3],
                            quantum_noise_model=T1T2NoiseModel(T1=0.5)),
        PhysicalInstruction(INSTR_CNOT, duration=2, topology=[(2, 1), (1, 3)]),
        PhysicalInstruction(INSTR_MEASURE, duration=3, parallel=True)])
```

Here, the name of the processor is “TestQPD” with 4 qubits in total. The processor has a DepolarNoise associated with all the qubits. Topology means the qubits that support the given instruction. The quantum_noise_model argument specifies the noise applied to that particular instruction. Duration specifies the time (ns) it takes to complete that instruction.

The example TestQPD processor supports the following instructions:

- INIT: Initializes the qubits to state |0>. All qubits support it (no topology input)
- X: BitFlip gate. Only qubits 1,3 and 3 support it with a relaxation error
- CNOT: Can only be performed between qubits (2,1) and (1,3)
- Measure: Applicable on all qubits

Now, instructions can be executed directly or via a quantum program:

- Directly executing instructions with an execute instruction call:

  execute_instruction(*self, instruction, qubit_mapping=None, str output_key='instr', bool physical=True, bool check_qubit_mapping=True, **parameters*)

  Example qmemory.execute_instruction(instr.INSTR_INIT,[0],physical=True)

- Executing via a quantum program:

```
class CustomQProgram(QuantumProgram):
    default_num_qubits = 2

    def program(self):
        q1, q2 = self.get_qubit_indices(2)
        self.apply(INSTR_INIT, q1)
        self.apply(INSTR_INIT, q2)
        yield self.run()
        self.apply(INSTR_X, q1)
        self.apply(INSTR_CNOT, [q1, q2])
        self.apply(INSTR_SIGNAL, physical=False)
        self.apply(INSTR_MEASURE, q1, output_key="m1")
        self.apply(INSTR_MEASURE, q2, output_key="m2")
        yield self.run(parallel=False)
```

  - Here, CustomQProgram extends the base class QuantumProgram
  - The program method needs to be overwritten with the user program
  - yield self.run() waits for the body of the program to finish (instructions can be executed in parallel or sequential order)

3.2.2 Quantum Channel

Netsquid.components.qchanel defines a quantum channel which is used to transmit quantum information. Now, gates can also be thought of as a quantum channel in time rather than space. This has support for both types of transmission. This API declares a one-way quantum channel capable of transmitting qubits with a specified delay, noise and loss.

As an example,

```
qchannel_c2a = QuantumChannel("qchannel_C2A", length=length / 2,
                              models={"quantum_loss_model": FibreLossModel(p_loss_init=0.98), "delay_model": FibreDel
```

Here, qchannel_c2a is a quantum channel from c to a with a standard FibreDelayModel and a specific FibreLossModel.

Similarly, NetSquid also has support for a classical channel capable of transmitting classical information with a given noise, loss and delay.

The above discussion brings us to the topic of a Quantum Network supported by NetSquid. This is where all the above components are integrated.

3.2.3 Quantum Network

Quantum Network is a component to help create and manage a network of nodes and connections.

As an example,

```
network = Network(name="test_network")
```

```
network.add_nodes(nodes=[Node("carl", qmemory=QuantumMemory("mem_carl")),
                         Node("evan", qmemory=QuantumProcessor("mem_evan"))])
```

Here, first a network is declared with the name “test_network”. Two nodes, carl and evan are added as nodes in the network. Carl has a QuantumMemory and evan has a QuantumProcessor included.

Connection of two nodes within a network can be done simply as follows:

```
network.add_connection("alice", "bob",
                       channel_to=QuantumChannel("quantum_channel"),
                       label="quantum")
```

Here, alice and bob are connected via a quantum channel and the connection is a part of the network.

Quantum memories, quantum processors and quantum/classical channels can have custom errors associated with them. The following subsection details the Error classes provided by NetSquid.

3.2.4 Error Models in NetSquid

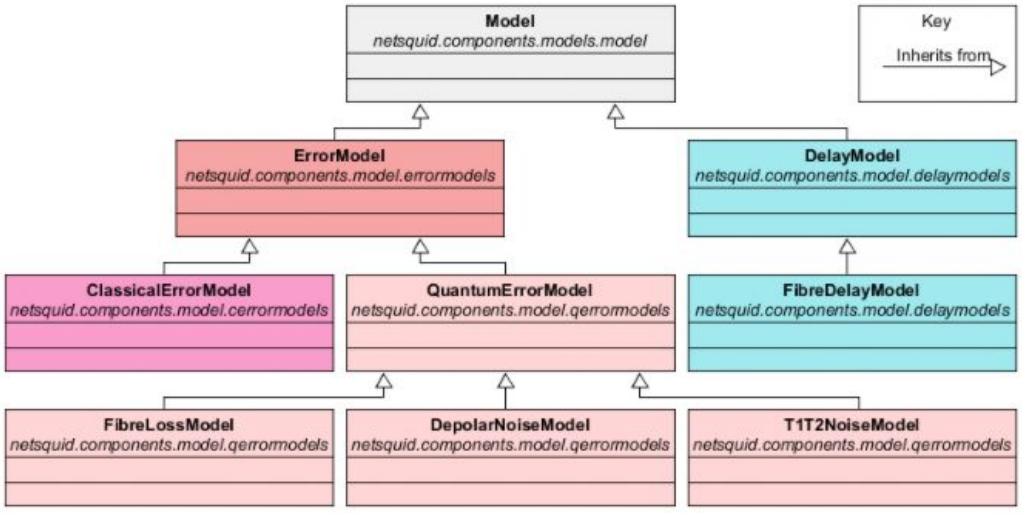


Figure 3-2

As shown in Figure 3-2, NetSquid supports Classical, Quantum as well as Delay models for Fibres.

The QuantumErrorModel class has three derived classes: FibreLossModel, DepolarNoiseModel and T1T2NoiseModel.

3.2.4.1 FibreLossModel

Describes a model for exponential photon loss on fibre optic channels.

class **netsquid.components.models.qerrormodels.FibreLossModel**(*p_loss_init=0.2, p_loss_length=0.25, rng=None*)

Where,

- p_loss_init: The initial probability of losing a photon when it enters the channel
- p_loss_length: The photon survival probability per unit length (dB/km)
- rng: Random number generator to use

This error model can be passed to a quantum channel to make the connection noisy and realistic.

3.2.4.2 *DepolarNoiseModel*

This model applies depolarization noise to the qubits.

```
class netsquid.components.models.qerrormodels.DepolarNoiseModel(depolar_rate,
time_independent=False, **kwargs)
```

Where,

- Depolar_rate: The probability that the qubit will depolarize with time.
- time_independent: If false, then this is the exponential depolarizing rate per unit time (Hz). If true, this is the probability.

3.2.4.3 *T1T2NoiseModel*

This is a commonly used phenomenological noise model based on T1 T2 times. This is more generally referred to as the decoherence model.

```
class netsquid.components.models.qerrormodels.T1T2NoiseModel(T1=0, T2=0, **kwargs)
```

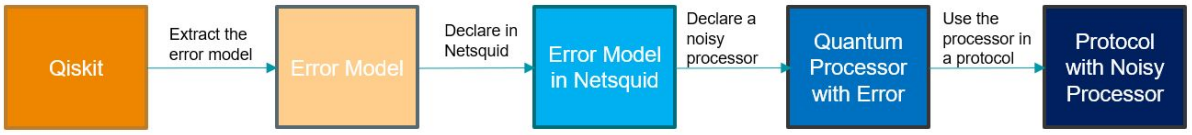
Where,

- T1: Time constant of exponential amplitude damping.
- T2: Time constant of exponential dephasing component.

3.3 Architecture

Now that the relevant features of Qiskit and NetSquid are covered, let's take a look at how these features can be used to perform a truly distributed quantum computing simulation.

The figure below depicts the steps we take to prepare NetSquid for distributed quantum computing simulations.



Step 1: Extract out the error model of a given backend in Qiskit.

Step 2: Recreate the Error model in NetSquid.

Step 3: Use the Error Model to declare noisy QuantumProcessors in NetSquid.

Step 4: Write the Node Protocol with Noisy Processors.

3.3.1 Step 1 – Extracting the Noise Model from Qiskit

As discussed in section 3.2, Qiskit provides a set of APIs to extract out a noise model from a given backend.

The following is the common code excerpt that applies to all the code snippets after this.

```
from qiskit.providers.aer.noise.device.models import basic_device_readout_errors
from qiskit.providers.aer.noise.device.models import basic_device_gate_errors
from qiskit.providers.aer.noise.device import gate_length_values
from qiskit.providers.aer.noise.device import readout_error_values
from qiskit.providers.aer.noise.device import thermal_relaxation_values
from qiskit.providers.aer.noise.device import gate_error_values
from qiskit.providers.aer.noise.device import gate_param_values
```

```
backend = provider.get_backend('ibmq_16_melbourne')
properties = backend.properties()
```

3.3.1.1 Readout Errors for ibmq_16_melbourne

Code to extract out the readout error values in Qiskit:

```
readout_errors = readout_error_values(properties)
for e1, e2 in readout_error_values(properties):
    print("Qubit: "+str(i)+" -->    P[1|0]: "+str(e1)+"  P[0|1]: "+str(e2))
```

Resulting error values of the ibmq_16_melbourne:

```
Qubit: 0 -->    P[1|0]: 0.0112  P[0|1]: 0.07120000000000004
Qubit: 1 -->    P[1|0]: 0.02080000000000004  P[0|1]: 0.067
Qubit: 2 -->    P[1|0]: 0.009199999999999986  P[0|1]: 0.0552
Qubit: 3 -->    P[1|0]: 0.1058  P[0|1]: 0.22
Qubit: 4 -->    P[1|0]: 0.0242  P[0|1]: 0.07720000000000005
Qubit: 5 -->    P[1|0]: 0.01319999999999999  P[0|1]: 0.083
Qubit: 6 -->    P[1|0]: 0.0106  P[0|1]: 0.04459999999999997
Qubit: 7 -->    P[1|0]: 0.06679999999999997  P[0|1]: 0.1296
Qubit: 8 -->    P[1|0]: 0.0376  P[0|1]: 0.0796
Qubit: 9 -->    P[1|0]: 0.04259999999999997  P[0|1]: 0.0984
Qubit: 10 -->    P[1|0]: 0.0146  P[0|1]: 0.09360000000000002
Qubit: 11 -->    P[1|0]: 0.1372  P[0|1]: 0.19420000000000004
Qubit: 12 -->    P[1|0]: 0.09199999999999997  P[0|1]: 0.1168
Qubit: 13 -->    P[1|0]: 0.0556  P[0|1]: 0.11819999999999997
Qubit: 14 -->    P[1|0]: 0.0364  P[0|1]: 0.06320000000000003
```

3.3.1.2 *Thermal Relaxation Values for ibmq_16_melbourne*

Code to extract out the Thermal Relaxation values from Qiskit:

```
for t1, t2, freq in thermal_relaxation_values(properties):
    print("Qubit: "+str(i)+" -->    T1: "+str(t1)+"  T2: "+str(t2))
```

Resulting error values for ibmq_16_melbourne:

```
Qubit: 0 -->    T1: 47066.97419877669  T2: 72813.08549522988
Qubit: 1 -->    T1: 52526.63323349799  T2: 61776.29720005739
Qubit: 2 -->    T1: 55578.654050151854  T2: 55862.436846446806
Qubit: 3 -->    T1: 79276.0188737155  T2: 16953.82899016347
Qubit: 4 -->    T1: 50998.48559584093  T2: 44778.50328360147
Qubit: 5 -->    T1: 18420.78190673815  T2: 33939.04535952175
Qubit: 6 -->    T1: 70854.2657475816  T2: 79983.17480736015
Qubit: 7 -->    T1: 39232.845641894986  T2: 14304.851612143295
Qubit: 8 -->    T1: 93839.02175863666  T2: 152115.68588624278
Qubit: 9 -->    T1: 46626.117888887966  T2: 56380.33782842999
Qubit: 10 -->    T1: 64069.65977930918  T2: 65144.28733236218
Qubit: 11 -->    T1: 56346.88810339914  T2: 79049.46331927988
Qubit: 12 -->    T1: 60098.57226242237  T2: 59798.80113649136
Qubit: 13 -->    T1: 32027.26475765039  T2: 31753.111374971857
Qubit: 14 -->    T1: 40069.55723129425  T2: 47811.23966861264
```

3.3.1.3 *Depolar gate errors and gate lengths*

Code to extract out the Gate Errors and Gate Lengths from Qiskit:

```
for name, qubits, time, error in gate_param_values(properties):
    print("Name: "+str(name)+"  Qubit: "+str(qubits)+"    Gate_Length(ns): "+str(time)+"  Gate_Depolar_Error: "+s
```

Resulting Error and Gate Lengths for ibmq_16_melbourne:

```
Name: u2  Qubit: [11]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0005173799851739677
Name: u3  Qubit: [11]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.0010344922882988827
Name: id  Qubit: [12]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0010377885660447055
Name: u1  Qubit: [12]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [12]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0010377885660447055
Name: u3  Qubit: [12]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.0020745001269817376
Name: id  Qubit: [13]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.001943855802352161
Name: u1  Qubit: [13]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [13]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.001943855802352161
Name: u3  Qubit: [13]    Gate_Length(ns): 284.44444444444446  Gate_Depolar_Error: 0.0038839330293238827
Name: id  Qubit: [14]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0006212687154268972
Name: u1  Qubit: [14]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [14]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0006212687154268972
Name: u3  Qubit: [14]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.0012421514560370106
Name: cx  Qubit: [0, 1]    Gate_Length(ns): 743.1111111111111  Gate_Depolar_Error: 0.017054713248095987
Name: cx  Qubit: [0, 14]    Gate_Length(ns): 970.6666666666666  Gate_Depolar_Error: 0.02542473202708037
Name: cx  Qubit: [1, 0]    Gate_Length(ns): 689.7777777777777  Gate_Depolar_Error: 0.017054713248095987
Name: cx  Qubit: [1, 2]    Gate_Length(ns): 355.55555555555554  Gate_Depolar_Error: 0.012528988755406784
Name: cx  Qubit: [1, 13]    Gate_Length(ns): 1137.7777777777778  Gate_Depolar_Error: 0.044176665543836896
Name: cx  Qubit: [2, 1]    Gate_Length(ns): 408.88888888888886  Gate_Depolar_Error: 0.012528988755406784
Name: cx  Qubit: [2, 3]    Gate_Length(ns): 988.4444444444443  Gate_Depolar_Error: 0.024714536632426898
Name: cx  Qubit: [2, 12]    Gate_Length(ns): 1696  Gate_Depolar_Error: 0.049049473998456966
Name: cx  Qubit: [3, 2]    Gate_Length(ns): 1041.7777777777778  Gate_Depolar_Error: 0.024714536632426898
Name: cx  Qubit: [3, 4]    Gate_Length(ns): 785.7777777777777  Gate_Depolar_Error: 0.01832611145947849
Name: cx  Qubit: [3, 11]    Gate_Length(ns): 807.1111111111111  Gate_Depolar_Error: 0.031113553874052235
Name: cx  Qubit: [4, 3]    Gate_Length(ns): 732.4444444444445  Gate_Depolar_Error: 0.01832611145947849
Name: cx  Qubit: [4, 5]    Gate_Length(ns): 789.3333333333333  Gate_Depolar_Error: 0.03004930207494594
Name: cx  Qubit: [4, 10]    Gate_Length(ns): 1169.7777777777776  Gate_Depolar_Error: 0.03556093589034437
Name: cx  Qubit: [5, 4]    Gate_Length(ns): 647.1111111111111  Gate_Depolar_Error: 0.03004930207494594
Name: cx  Qubit: [5, 6]    Gate_Length(ns): 1472  Gate_Depolar_Error: 0.04762032778709052
Name: cx  Qubit: [5, 9]    Gate_Length(ns): 860.4444444444443  Gate_Depolar_Error: 0.03666627390582228
Name: cx  Qubit: [6, 5]    Gate_Length(ns): 1614.2222222222222  Gate_Depolar_Error: 0.04762032778709052
Name: cx  Qubit: [6, 8]    Gate_Length(ns): 995.5555555555555  Gate_Depolar_Error: 0.022623481341894847
Name: cx  Qubit: [7, 8]    Gate_Length(ns): 775.1111111111111  Gate_Depolar_Error: 0.027860475752815778
Name: cx  Qubit: [8, 6]    Gate_Length(ns): 1137.7777777777778  Gate_Depolar_Error: 0.022623481341894847
Name: cx  Qubit: [8, 7]    Gate_Length(ns): 917.3333333333333  Gate_Depolar_Error: 0.027860475752815778
Name: cx  Qubit: [8, 9]    Gate_Length(ns): 1159.111111111111  Gate_Depolar_Error: 0.030946882576326384
Name: cx  Qubit: [9, 5]    Gate_Length(ns): 1002.6666666666666  Gate_Depolar_Error: 0.03666627390582228
Name: cx  Qubit: [9, 8]    Gate_Length(ns): 1016.8888888888888  Gate_Depolar_Error: 0.030946882576326384
Name: cx  Qubit: [9, 10]    Gate_Length(ns): 860.4444444444443  Gate_Depolar_Error: 0.030672986201688818
Name: cx  Qubit: [10, 4]    Gate_Length(ns): 1312  Gate_Depolar_Error: 0.03556093589034437
Name: cx  Qubit: [10, 9]    Gate_Length(ns): 1002.6666666666666  Gate_Depolar_Error: 0.030672986201688818
Name: cx  Qubit: [10, 11]    Gate_Length(ns): 945.7777777777777  Gate_Depolar_Error: 0.029146211166072244
Name: cx  Qubit: [11, 3]    Gate_Length(ns): 753.7777777777777  Gate_Depolar_Error: 0.031113553874052235
Name: cx  Qubit: [11, 10]    Gate_Length(ns): 1088  Gate_Depolar_Error: 0.029146211166072244
Name: cx  Qubit: [11, 12]    Gate_Length(ns): 597.3333333333333  Gate_Depolar_Error: 0.020257732923467886
Name: cx  Qubit: [12, 2]    Gate_Length(ns): 1642.6666666666665  Gate_Depolar_Error: 0.049049473998456966
Name: cx  Qubit: [12, 11]    Gate_Length(ns): 650.6666666666666  Gate_Depolar_Error: 0.020257732923467886
Name: cx  Qubit: [12, 13]    Gate_Length(ns): 753.7777777777777  Gate_Depolar_Error: 0.020048528380061548
Name: cx  Qubit: [13, 1]    Gate_Length(ns): 995.5555555555555  Gate_Depolar_Error: 0.044176665543836896
Name: cx  Qubit: [13, 12]    Gate_Length(ns): 611.5555555555555  Gate_Depolar_Error: 0.020048528380061548
Name: cx  Qubit: [13, 14]    Gate_Length(ns): 657.7777777777777  Gate_Depolar_Error: 0.02971525763666899
Name: cx  Qubit: [14, 0]    Gate_Length(ns): 917.3333333333333  Gate_Depolar_Error: 0.02542473202708037
Name: cx  Qubit: [14, 13]    Gate_Length(ns): 515.5555555555555  Gate_Depolar_Error: 0.02971525763666899
```

```
Name: id  Qubit: [0]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0006231992420577364
Name: u1  Qubit: [0]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [0]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0006231992420577364
Name: u3  Qubit: [0]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.001246010106820239
Name: id  Qubit: [1]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0006599902438002046
Name: u1  Qubit: [1]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [1]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0006599902438002046
Name: u3  Qubit: [1]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.0013195449004785864
Name: id  Qubit: [2]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0008881045631594462
Name: u1  Qubit: [2]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [2]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0008881045631594462
Name: u3  Qubit: [2]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.0017754203966038329
Name: id  Qubit: [3]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0005216837914342054
Name: u1  Qubit: [3]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [3]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0005216837914342054
Name: u3  Qubit: [3]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.0010430954288902505
Name: id  Qubit: [4]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0016942472506163867
Name: u1  Qubit: [4]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [4]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0016942472506163867
Name: u3  Qubit: [4]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.0033856240274865446
Name: id  Qubit: [5]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.002621838916525314
Name: u1  Qubit: [5]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [5]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.002621838916525314
Name: u3  Qubit: [5]    Gate_Length(ns): 284.44444444444446  Gate_Depolar_Error: 0.005236837937464695
Name: id  Qubit: [6]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.0010543979899390662
Name: u1  Qubit: [6]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [6]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.0010543979899390662
Name: u3  Qubit: [6]    Gate_Length(ns): 284.44444444444446  Gate_Depolar_Error: 0.002107684224757045
Name: id  Qubit: [7]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.0018152155267723913
Name: u1  Qubit: [7]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [7]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.0018152155267723913
Name: u3  Qubit: [7]    Gate_Length(ns): 284.44444444444446  Gate_Depolar_Error: 0.0036271360461362923
Name: id  Qubit: [8]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0005612989405485931
Name: u1  Qubit: [8]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [8]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0005612989405485931
Name: u3  Qubit: [8]    Gate_Length(ns): 106.66666666666666  Gate_Depolar_Error: 0.0011222828245964767
Name: id  Qubit: [9]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.0017273405189931858
Name: u1  Qubit: [9]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [9]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.0017273405189931858
Name: u3  Qubit: [9]    Gate_Length(ns): 284.44444444444446  Gate_Depolar_Error: 0.0034516973327177736
Name: id  Qubit: [10]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.00149219403111301
Name: u1  Qubit: [10]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
Name: u2  Qubit: [10]    Gate_Length(ns): 142.22222222222223  Gate_Depolar_Error: 0.00149219403111301
Name: u3  Qubit: [10]    Gate_Length(ns): 284.44444444444446  Gate_Depolar_Error: 0.0029821614191994916
Name: id  Qubit: [11]    Gate_Length(ns): 53.33333333333333  Gate_Depolar_Error: 0.0005173799851739677
Name: u1  Qubit: [11]    Gate_Length(ns): 0  Gate_Depolar_Error: 0
```

3.3.2 Step 2 – Recreating the Noise Model in NetSquid

This section describes the details of porting a given noise model to NetSquid. This contains details on how to port the readout errors, thermal relaxation errors, gate errors and gate lengths to NetSquid.

The following code is common to all the subsequent sections:

```
from netsquid.components.models.qerrormodels import DepolarNoiseModel, DephaseNoiseModel, T1T2NoiseModel, MeasureNoiseModel, FibreLossModel
from netsquid.components.models.delaymodels import FibreDelayModel, FixedDelayModel
from netsquid.util.datacollector import DataCollector
from netsquid.qubits.qformalism import QFormalism, set_qstate_formalism
import netsquid as ns
import pydynaa
import netsquid.qubits.ketstates as ks
import netsquid.qubits.qubitapi as qapi
import netsquid.components.instructions as instr
from qiskit import IBMQ, Aer
from qiskit.circuit.library import XGate
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise import QuantumError
from qiskit.providers.aer.noise.device.models import basic_device_readout_errors
from qiskit.providers.aer.noise.device.models import basic_device_gate_errors
from qiskit.providers.aer.noise.device import gate_length_values
from qiskit.providers.aer.noise.device import readout_error_values
from qiskit.providers.aer.noise.device import thermal_relaxation_values
from qiskit.providers.aer.noise.device import gate_error_values
from qiskit.providers.aer.noise.device import gate_param_values
```

3.3.2.1 Porting Readout Errors

Readout errors are measurement errors. Once a qubit is measured, the error is applied on the classical outcome with some probability.

Here is the code to port the readout errors from Qiskit to NetSquid:

```
for e1, e2 in readout_error_values(properties):
    #print("Qubit: "+str(i)+" -->    P[1|0]: "+str(e1)+"  P[0|1]: "+str(e2))
    measure_error = MeasureNoiseModel(e1,e2)
    physical_I.append(PhysicalInstruction(instr.INSTR_MEASURE, duration=1, parallel=False, topology=[i],
                                          q_noise_model=measure_error, apply_q_noise_after=False))
    i=i+1
```

- physical_I is just a python list to contain all the physical instructions that Melbourne supports.
- MeasureNoiseModel is a custom class created specifically for this port. Here are the details.

```
class MeasureNoiseModel(QuantumErrorModel):
    """Model for applying Measure noise to qubit(s).
    """
    def __init__(self, p10, p01, **kwargs):
        super().__init__(**kwargs)
        self.p01 = p01
        self.p10 = p10

    @property
    def p01(self):
        return self.properties['p01']

    @p01.setter
    def p01(self,p01):
        self.properties['p01'] = p01

    @property
    def p10(self):
        return self.properties['p10']

    @p10.setter
    def p10(self,p10):
        self.properties['p10'] = p10

    def error_operation(self, qubits, delta_time=0, **kwargs):
        """Error operation to apply to qubits.
        """
        pmax = max(self.p10,self.p01)
        probI = 1 - self.p01
        probX = self.p01
        probY = 0
        probZ = 0
        for qubit in qubits:
            ns.qubits.qubitapi.apply_pauli_noise(qubit, (probI, probX, probY, probZ))
        ns.qubits.qubitapi.combine_qubits(qubits)
```

The error operation defines how the measurement error in applied in NetSquid. Note that this is a pessimistic approximation of the error in Qiskit. The maximum of P[1|0] and P[0|1] is taken and the outcome is flipped with that probability. The for loop is present to take care of any qubits in shared or entangled states.

- Measure Noise is applied whenever the physical instruction INSTR_MEASURE is called on the qubit specified by the topology.
- Note that readout_error_values is a Qiskit construct and PhysicalInstruction is a NetSquid construct.

3.3.2.2 *Porting Thermal Relaxation Errors*

As discussed before, thermal relaxation errors define the decoherence model of quantum systems.

The code to port the thermal relaxation from Qiskit to NetSquid is given below:

```
i=0
relax_error = T1T2NoiseModel()
for t1, t2, freq in thermal_relaxation_values(properties):
    #print("Qubit: "+str(i)+" -->    T1: "+str(t1)+"  T2: "+str(t2))
    i=i+1
    relax_error += T1T2NoiseModel(T1=t1*1000, T2=t2*1000, qiskit_model=False)
```

- Thermal_relaxation_values is a Qiskit construct giving the T1 T2 values of Melbourne. The values are multiplied by 1000 because Qiskit provides the values in us whereas NetSquid expects the values in ns.
- relax_error is NetSquid T1T2NoiseModel, adding the relax errors for all the qubits concatenates the noise model and forms a single decoherence model for ibmq_16_melbourne.
- relax_error can be applied to quantum memory as follows:

```
processor = QuantumProcessor("quantum_processor", num_positions=15,
                             memory_noise_models=relax_error,
                             phys_instructions=physical_I)
```

Here, a 15-qubit quantum processor is declared in NetSquid with relax_error (decoherence model for Melbourne) as a memory_noise_model.

A limitation of NetSquid's T1T2NoiseModel is that it does not support a non-unitary model of decoherence (when T1<T2). To get around this problem, a custom decoherence model is written. The details are as follows:

```
if self.T2 > self.T1:
    t1 = self.T1
    t2 = self.T2
    time = t
    excited_state_population = 0
    if t1 == np.inf:
        rate1 = 0
        p_reset = 0
    else:
        rate1 = 1 / t1
        p_reset = 1 - np.exp(-time * rate1)
    # T2 dephasing rate
    if t2 == np.inf:
        rate2 = 0
        exp_t2 = 1
    else:
        rate2 = 1 / t2
        exp_t2 = np.exp(-time * rate2)
    # Qubit state equilibrium probabilities
    p0 = 1 - excited_state_population
    p1 = excited_state_population
    chan = Choi(
        np.array([[1 - p1 * p_reset, 0, 0, exp_t2],
                  [0, p1 * p_reset, 0, 0], [0, 0, p0 * p_reset, 0],
                  [exp_t2, 0, 0, 1 - p0 * p_reset]]))
    k = Kraus(chan)
    klist = k.data
    if k.is_cptp():
        self.apply_kraus_error(qubit, klist)
    return
```

Here, Choi and Kraus are Qiskit's constructs and apply_kraus_error is a custom Kraus model written for NetSquid.

```
def apply_kraus_error(self, qubit, operator):
    op = []
    i = 0
    for m in operator:
        s = "k" + str(i)
        op.append(ns.qubits.operators.Operator(s, m))
        i = i+1
    ns.qubits.qubitapi.multi_operate([qubit], op)
```

- The Kraus matrices given by k.data are converted into a list of operators.
- The multi_operate function does the following operation which is similar to the definition of Kraus' evolution model:
    - For a given list of operators O_i and a quantum state ρ, the evolution is defined as:

$$\varepsilon(\rho) = \sum_i w_i O_i \rho O_i^\dagger$$

3.3.2.3 *Porting Gate Errors and Gate Lengths to NetSquid*

Gate errors are the depolarization errors when a specific gate is applied to a specific qubit. Gate lengths represent the latency of the application of a gate on a specific qubit.

The code to port the gate errors and lengths from Qiskit to NetSquid is given below:

```
u2_list=[]
for name, qubits, time, error in gate_param_values(properties):
    print("Name: "+str(name)+"  Qubit: "+str(qubits)+"    Gate Length(ns): "+str(time)+"  Gate Depolar Error: "+str(error))
    Dep_Noise = DepolarNoiseModel(depolar_rate=error,time_independent=True)
    if name=="id":
        physical_I.append(PhysicalInstruction(instr.INSTR_I, duration=time, parallel=True, topology=qubits,
                          q_noise_model=Dep_Noise, apply_q_noise_after=False))
    elif name=="u1":
        physical_I.append(PhysicalInstruction(instr.INSTR_Z, duration=time, parallel=True, topology=qubits,
                          q_noise_model=Dep_Noise, apply_q_noise_after=False))
        physical_I.append(PhysicalInstruction(instr.INSTR_S, duration=time, parallel=True, topology=qubits,
                          q_noise_model=Dep_Noise, apply_q_noise_after=False))
    elif name=="u2":
        physical_I.append(PhysicalInstruction(instr.INSTR_H, duration=time, parallel=True, topology=qubits,
                          q_noise_model=Dep_Noise, apply_q_noise_after=False))
        u2_list.append([qubits,time,error])
    elif name=="u3":
        physical_I.append(PhysicalInstruction(instr.INSTR_X, duration=time, parallel=True, topology=qubits,
                          q_noise_model=Dep_Noise, apply_q_noise_after=False))
        physical_I.append(PhysicalInstruction(instr.INSTR_Y, duration=time, parallel=True, topology=qubits,
                          q_noise_model=Dep_Noise, apply_q_noise_after=False))
    elif name=="cx":
        q1,q2 = qubits
        physical_I.append(PhysicalInstruction(instr.INSTR_CNOT, duration=time, parallel=True, topology=[(q1,q2)],
                          q_noise_model=Dep_Noise, apply_q_noise_after=False))
        q_u2, time_u2, error_u2 = u2_list[q2]
        Dep_Noise_U2 = DepolarNoiseModel(depolar_rate=error_u2,time_independent=True)
        combined = Dep_Noise + Dep_Noise_U2*2
        physical_I.append(PhysicalInstruction(instr.INSTR_CZ, duration=time+2*time_u2, parallel=True, topology=[(q1,q2)],
                          q_noise_model=Dep_Noise, apply_q_noise_after=False))
```

The mapping of the basis gates of Qiskit to the gates used in NetSquid is given below:

Basis Gate in Qiskit	NetSquid Gate
U2	H
U3	X
U1	Z
U3	Y
U1	S
CX	CX
U1(1)-->CX-->U1(1)	CY
U2(1)-->CX-->U2(1)	CZ
Measure	Measure

After doing these steps, a complete list of noisy instructions (physical_I) and a complete decoherence model (relax_error) for the quantum processor is available. The next step is to declare a quantum processor with the given error model in NetSquid.

3.3.3 Step 3 – Declaring a Quantum Processor with the Noise Model

```
def create_processor(depolar_rate, dephase_rate):
    # We'll give both Alice and Bob the same kind of processor
    physical_I.append(PhysicalInstruction(instr.INSTR_INIT, duration=1, parallel=True))
    processor = QuantumProcessor("quantum_processor", num_positions=15,
                                 memory_noise_models=relax_error,
                                 phys_instructions=physical_I)
    return processor
```

The create_processor function creates a quantum processor with physical_I instructions and relax_error decoherence model. Note that relax_error and physical_I are globals in this piece of code.

3.3.4 Step 4 – Protocol with Noisy Processors

The following code shows a sample network using the noisy processors as nodes.

```
# Setup nodes Alice and Bob with quantum processor:
alice = Node("Alice", qmemory=create_processor(depolar_rate, dephase_rate))
bob = Node("Bob", qmemory=create_processor(depolar_rate, dephase_rate))
# Create a network
network = Network("Teleportation_network")
network.add_nodes([alice, bob])
```

Here, two nodes (Alice and Bob) are created using the create_processor function. A network called Teleportation network is declared and the quantum processors are added to the network.

For instance, the following code shows the protocol that Alice follows:

```
class BellMeasurementProgram(QuantumProgram):
    """Program to perform a Bell measurement on two qubits.

    Measurement results are stored in output keys "M1" and "M2"

    """
    default_num_qubits = 2

    def program(self):
        q1, q2 = self.get_qubit_indices(2)
        self.apply(instr.INSTR_CNOT, [q1, q2])
        self.apply(instr.INSTR_H, q1)
        self.apply(instr.INSTR_MEASURE, q1, output_key="M1")
        self.apply(instr.INSTR_MEASURE, q2, output_key="M2")
        yield self.run()
```

```
class BellMeasurementProtocol(NodeProtocol):
    """Protocol to perform a Bell measurement when qubits are available.

    """

    def run(self):
        qubit_initialised = False
        entanglement_ready = False
        qubit_init_program = InitStateProgram()
        measure_program = BellMeasurementProgram()
        self.node.qmemory.execute_program(qubit_init_program)
        while True:
            expr = yield (self.await_program(self.node.qmemory) |
                          self.await_port_input(self.node.ports["qin_charlie"]))
            if expr.first_term.value:
                qubit_initialised = True
            elif expr.second_term.value:
                entanglement_ready = True
            if qubit_initialised and entanglement_ready:
                # Once both qubits arrived, do BSM program and send to Bob
                yield self.node.qmemory.execute_program(measure_program)
                m1, = measure_program.output["M1"]
                m2, = measure_program.output["M2"]
                self.node.ports["cout_bob"].tx_output((m1, m2))
                self.send_signal(Signals.SUCCESS)
                qubit_initialised = False
                entanglement_ready = False
                self.node.qmemory.execute_program(qubit_init_program)
```

The protocol for the Node Alice can be defined as follows:

```
protocol_alice = BellMeasurementProtocol(alice)
```

This completes the port of the noise model from Qiskit to NetSquid. The following section describes our validation of the port and discusses important observations from the results.

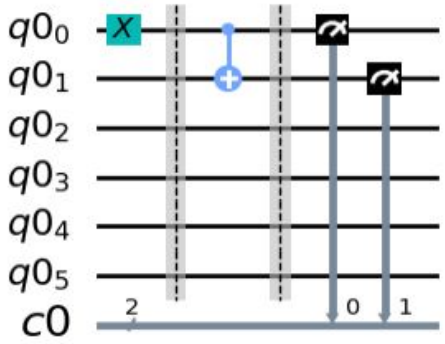
4 Port Validation and Discussion

This section details the experimental validation of porting the noise model from Qiskit to NetSquid. The validation of the port is performed by running three tests both on Qiskit and NetSquid and then comparing the results.

The three test cases used are CNOT, CNOT_Chain and Teleportation.

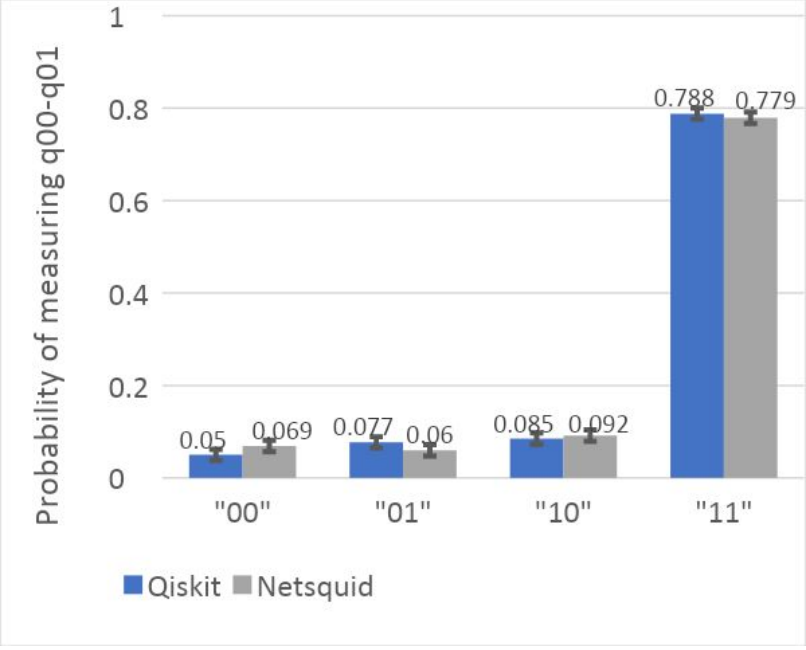
The following sub-sections list the details of each.

4.1 CNOT



The figure shows the schematic of the circuit. Qubit $q0_0$ is initialized in the |1> state and a CNOT is performed between $q0_0$ and $q0_1$. After that, the two qubits are measured, and the results of the measurements are recorded. This entire process is repeated 1000 times and the results are aggregated. A successful run results in measuring 11. Due to errors in the noisy quantum processor, other results will also be recorded with some probability.

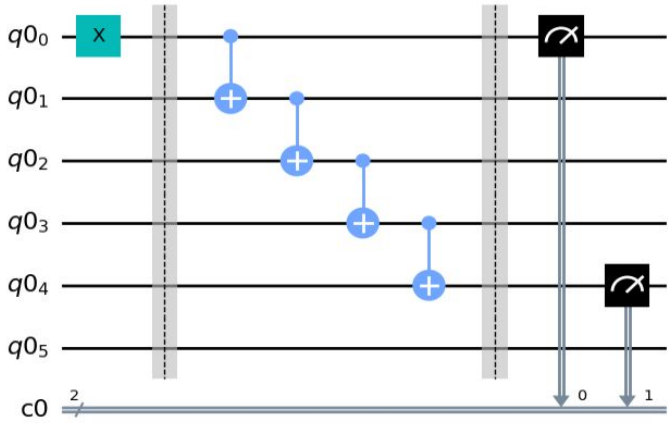
The following figure presents the average probability and 95% confidence interval of observing each of the possible measurement outcomes in Qiskit and our modified version of NetSquid when measuring qubits q00, q01:



As clear from the figure, the port successfully recreates the results on small circuits consisting of one or two qubits.

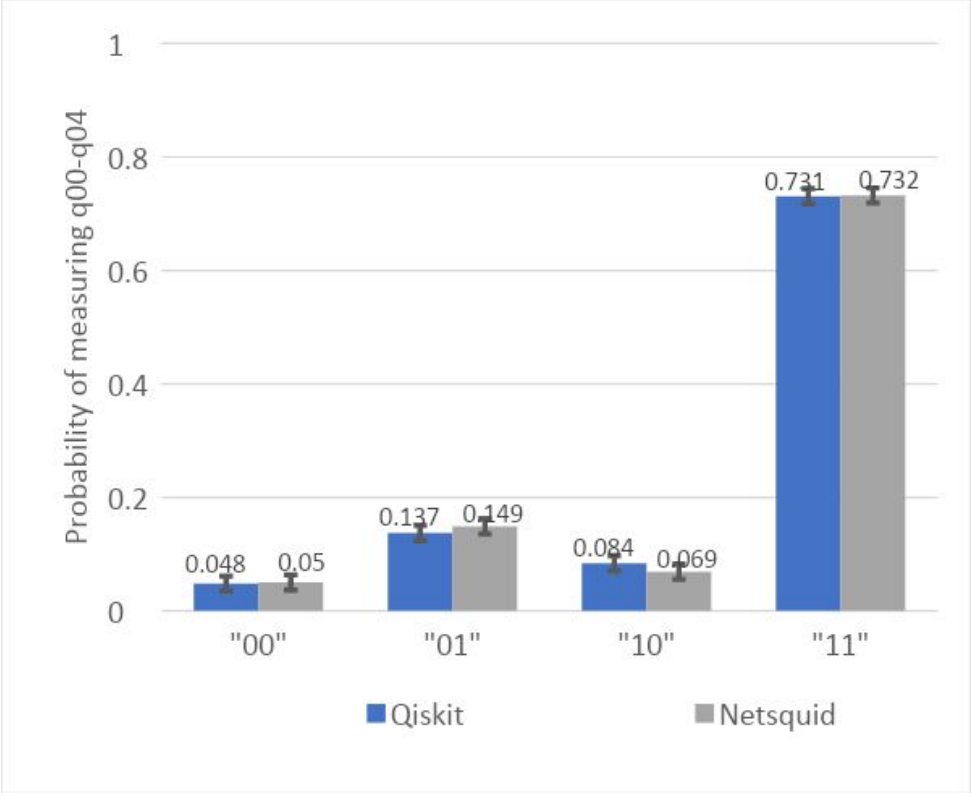
- Success probability with 95% Confidence Interval in Qiskit = 0.788 ± 0.0122
- Success probability with 95% Confidence Interval in NetSquid = 0.779 ± 0.0124

4.2 CNOT Chain



The figure shows the schematic of the CNOT chain circuit. Qubit $q0_0$ is initialized in the |1> state and a series of CNOT gates are performed as shown. In the end, qubits $q0_0$ and $q0_4$ are measured. This process is repeated 1000 times and the average rate of success ($q0_0$, $q0_4$ measurement outcome 11) is recorded for both Qiskit and NetSquid.

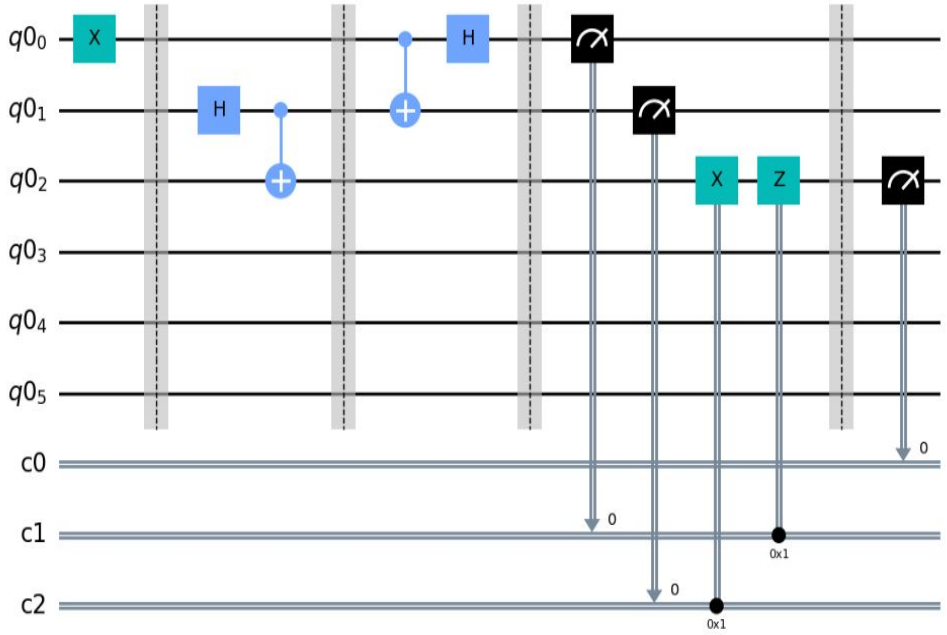
Here is the distribution of measurement for qubits $q0_0$, $q0_4$:



As shown in the chart, the success rate of Qiskit and NetSquid are very close (within 0.1% for the “11” case). This shows the port is successful for circuits with high latencies and a large number of gates spanning multiple qubits.

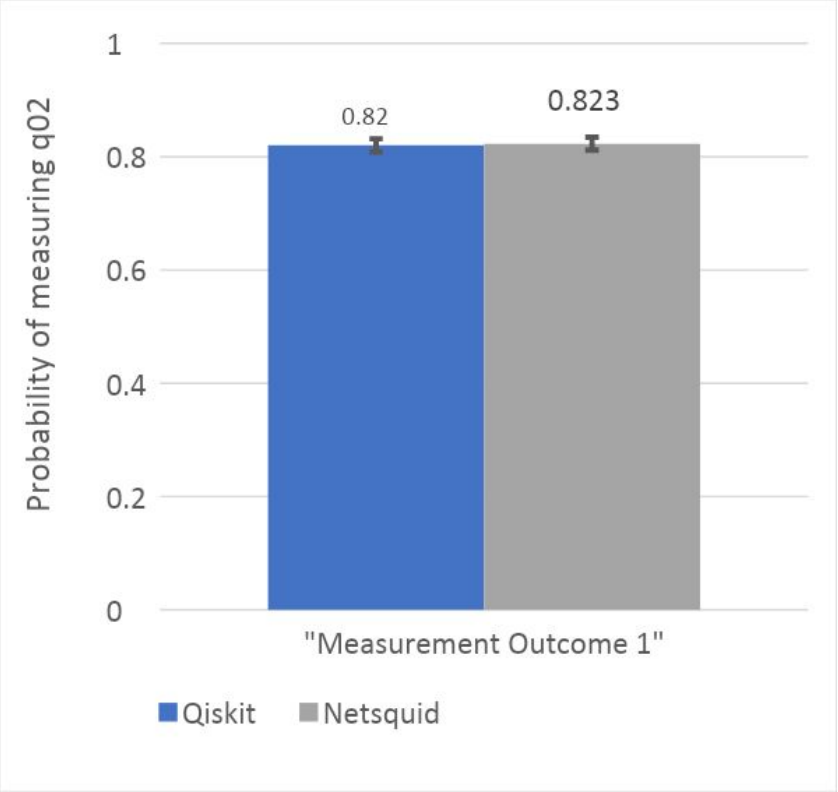
- Success probability with 95% Confidence Interval in Qiskit = 0.731 ± 0.013
- Success probability with 95% Confidence Interval in NetSquid = 0.732 ± 0.013

4.3 Teleportation



The circuit shown above is a teleportation circuit. Qubit $q0_0$ is prepared in the |1> state. Then, qubits $q0_1$ and $q0_2$ are entangled. Maximally entangling $q0_1$, $q0_2$ is not part of the teleportation protocol; we do it simply to place these qubits in a maximally entangled initial state before they participate in the teleportation of $q0_0$ to $q0_2$. Subsequently, teleportation is performed, where the state of qubit $q0_0$ is teleported to qubit $q0_2$. Lastly, the qubit $q0_2$ is measured. This process is repeated 1000 times and the rate of success ($q0_2$ measurement outcome 1) is recorded.

The following is the comparison of the experimental results in Qiskit and our modified version of NetSquid:



The chart above shows that the success rates are very close (within 0.3%). This proves that the port is successfully recreating the results even when maximally entangled states are involved in the computation.

The above three results prove that the port is successful and the quantum processors in NetSquid are behaving as if they were ibmq_16_mebourne.

- Success probability with 95% Confidence Interval in Qiskit = 0.82 ± 0.0115
- Success probability with 95% Confidence Interval in NetSquid = 0.823 ± 0.0114

4.4 Distributed CNOT simulation

To demonstrate the usefulness of the port, a simulation of a distributed CNOT gate is presented in this section.

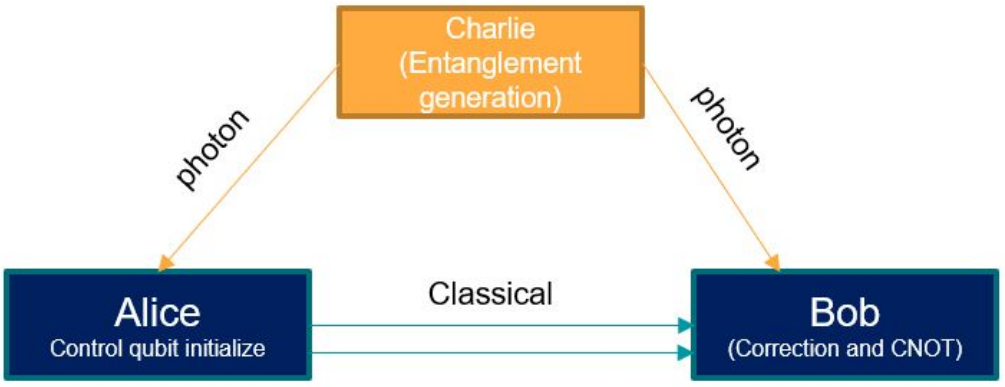


Figure 4-1

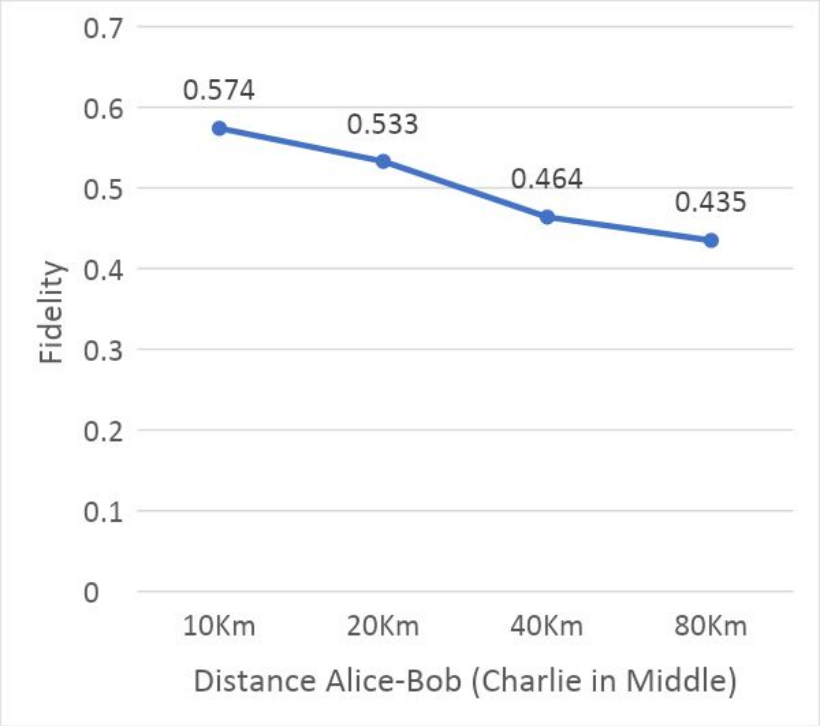
Figure 4-1 shows the network diagram of a distributed CNOT computation.

- Alice and Bob are noisy quantum processors based on the ibmq_16_melbourne noise model.
- Charlie is an entanglement generation source, generating an entangled pair of photons with an entanglement rate f (f = 4e4/node_distance) and fidelity 0.95, where node_distance is the distance in km between Alice and Bob. Charlie is assumed to be located exactly in the middle of the distance between Alice and Bob.
- The connections shown in orange are quantum connections and the connection shown in blue is a 2-bit classical connection.
- The error model of Quantum connections is described with a default FibreDelayModel() and a default FibreLossModel() available in NetSquid.
- The error on the classical connection is described by a default FibreDelayModel().
- Note that this circuit assumes perfect transduction, i.e., the photons sent by Carlie are transduced to a superconducting qubit perfectly.

The protocol of Distributed CNOT is as follows:

- Charlie generates an entangled pair every node_distance/4e4 seconds and sends one photon to Alice and one to Bob via the quantum channels shown in orange.
- Alice prepares a qubit in |1> state and performs a CNOT and Hadamard on the received qubit (part of teleportation).
- After those operations are finished, Alice measures the prepared qubit and the received qubit. Alice sends the 2-bit measurement results over to Bob via a classical connection.
- Bob waits for the photon from Charlie and the 2-bit classical data from Alice. Once bob has received both, Bob performs the correction protocol on the received qubit (CX and CZ based on classical data as per the teleportation protocol).
- Once bob has performed the correction, Bob performs a CNOT between the corrected qubit and another qubit (position 1) in the |0> state.
- Finally, the target qubit of the CNOT at Bob's side is measured and the results are recorded.
- Success is defined by measurement outcome 1 since the control should have been in |1> state and target in |0> state.
- The simulation is run for 1000 repetitions and the fidelity is plotted as a function of the distance between Alice and Bob, with Charlie exactly in the middle.

Here are the results of the simulation:



These numbers reflect the fidelity of a distributed CNOT in the case of a perfect Transduction. Taking transduction into account will render the distributed CNOT impractical in our setting (superconducting nodes connected by a photon-based quantum channel) due to the following reasons.

- The efficiency of transduction between superconducting transmons and photons is $0.88*10^{-5}$ [1].
- The entanglement generation and distribution rate to two nodes that are 10 Km apart is 4000 entangled photon pairs per second, which means that the entangling source will generate a successful entanglement once every (1/0.95) * 1/4000 = 263us (0.95 is the fidelity of entanglement generation).
- Now, since the transduction is performed on both Alice's and Bob's side, the probability of a successful transduction of an entangled pair is $(0.88*10^{-5})^2=0.77*10^{-10}$. Which means that a successful entanglement of superconducting qubits at Alice and Bob is generated every $263*10^{-6}/0.77*10^{-10} = 3.4*10^6$s or 39.53 days.
- On every iteration of the experiment, Charlie tries to establish and distribute entangled photons to Alice and Bob, who then transduce them to their own qubits. Meanwhile, at each iteration, Alice and Bob are executing their respective quantum circuits anew. After we wait for 39.5 days to get an entangled pair between Alice and Bob, we still must perform the rest of the CNOT circuit, given we already achieved transduction on both nodes. The fidelity of CNOT is still 0.57, which will translate in overall CNOT success rate of once every 39.53 / 0.57 = 69.35 days. The success rate is too low for this to be a practical scheme for most purposes.

5 Future Work

Future work related to this might entail modelling of Transduction as a gate in NetSquid and studying the effects of emerging transduction techniques in the context of distributed quantum computing. Our current model of transduction is done by simply manipulating the initial coupling probability of the quantum channel, resulting in most experiment trials losing the photon and hence the simulation cannot proceed. Modelling it as a gate will circumvent this problem and allow all runs to complete, and thus the simulator will produce results in a reasonable time frame.

Research on the architecture of distributed systems can be another area. A potential research question to address is whether a central entanglement service is better or a distributed entanglement service with entanglement swapping in the middle. Of course, these are big questions, and the answers depend on various system parameters and design choices.

Another avenue of research might be to automatically extract out noise models for Ion Traps or other technologies in Qiskit (or any other simulator), incorporate them in NetSquid, and compare the results with superconducting devices. This can be generalized, and any two device technologies can be compared to study the advantages and disadvantages over one another on specific algorithms.

6 References

[1] Mohammad Mirhosseini et al., Quantum transduction of optical photons from a superconducting qubit

[2] Stephanie Wehner et al., NetSquid, a discrete-event simulation platform for quantum networks

[3] Abraham Asfaw, Luciano Bello, Yael Ben-Haim, Sergey Bravyi, Nicholas Bronn, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Richard Chen, Albert Frisch, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, Francis Harkins, Takashi Imamichi, David McKay, Antonio Mezzacapo, Zlatko Minev, Ramis Movassagh, Giacomo Nannicni, Paul Nation, Anna Phan, Marco Pistoia, Arthur Rattew, Joachim Schaefer, Javad Shabani, John Smolin, Kristan Temme, Madeleine Tod, Stephen Wood, James Wootton. Qiskit

[4] Daniel Gottesman et al., Quantum teleportation is a universal computational primitive

[5] Tzvetan S. Metodi, Arvin I. Faruque, Frederic T. Chong, Quantum Computing for Computer Architects: Second Edition 2011, Morgan and Claypool Publishers

[6] Linran Fan et al, Superconducting cavity electro-optics: a platform for coherent photon conversion between superconducting and photonic circuits, Science Advances 17 Aug 2018