netQuil: A quantum playground for distributed quantum computing simulations

Matthew P. Radzihovsky, Zachary I. Espinosa, Yewon Gim
AT&T Foundry, Palo Alto, CA 94301, USA
(Dated: September 18, 2021)

NetQuil is an open-source Python framework for quantum networking simulations built on the quantum computing framework pyQuil, by Rigetti Computing. NetQuil is built for testing ideas in quantum network topology and distributed quantum protocol. It allows users to create multi-agent networks, connect parties through classical and quantum channels, and introduce noise. NetQuil also makes running multiple trials for non-deterministic experiments, reviewing traffic in real-time, and synchronizing agents based on local and master clocks simple and easy. We provide an overview of the state of distributed quantum protocol and a basic introduction to netQuil's framework. Finally, we present several demonstrations of canonical quantum information protocols built using netQuil's distributed quantum gates and pyQuil.

I. INTRODUCTION

Quantum information theory has garnered widespread attention since Shor's [1] factorization algorithm and Grover's [2] sublinear search algorithm emerged in the late 90s. A lack of reliable quantum hardware and increased access to classical computing systems has stimulated the emergence of quantum computing simulators (QCSs). QCSs have become an affordable and accessible testbed for quantum information theory research and include Qiskit n[3], Quil [4], Q# [5], QCL [6], Quipper [7], QX[8], and Strawberry Fields [9]. These simulators have created a public ecosystem for exploring noiseless, single-party quantum computations and are available to amateurs and experts alike.

Although QCSs enable simultaneous advancements in quantum algorithms and quantum hardware, modern quantum computing hardware remains marred by gate errors and decoherence [10], features which few QCSs account for. Additionally, state-of-the-art quantum computers are limited to computations involving tens of qubits and minimal gate operations [11]. Such limitations may be addressed by performing distributed quantum computations via a quantum network: a system of remote quantum computers that can distribute quantum and classical information between each other. Realizing a quantum network will require mitigating the impact of gate errors and decoherence, necessitating advancements in quantum error correction (QEC) [12] and indicating the need for a QCS ecosystem that moves beyond single party, noiseless quantum computations. Squanch [13] and Simulaqron [14] have emerged as two quantum networking and quantum internet simulators.

Here, we introduce netQuil, an open-source Python [15] framework for simulating quantum networks. Unlike existing distributed quantum computing (DQC) simulators, which build their computations from scratch, NetQuil is designed to be an extension of the popular quantum computing framework Quil [4], by Rigetti Computing. In addition to the quantum computing simulation capacity provided by Quil, NetQuil can be used to create multi-agent networks of quantum computers connected via classical and quantum channels. Agents can send quantum and classical information, introduce noise, simulate devices, run multiple trials for non-deterministic experiments, review traffic in real-time, and synchronize agents with master and local clocks. Each trial in a simulation is fully parallelized and generates a Quil program that can be run on a quantum virtual machine (qvm) or quantum processing unit (qpu).

This white-paper is organized as follows: section II provides a brief review of quantum information theory and quantum networking, followed by an overview of the state of distributed quantum protocol. Section III introduces netQuil and its relationship with Quil, implementation of quantum agents, quantum and classical channels, and devices and noise. This section also discusses two primitive distributed quantum protocol [16], their usage, and netQuil's implementation. Section IV offers a set of demonstrations of canonical quantum networking protocols using netQuil, such as teleportation, superdense-coding, and the middleman attack. Finally, section V describes netQuil's limitations and future work.

II. STATE OF DISTRIBUTED QUANTUM PROTOCOL

Distributed quantum computing (DQC) is a means of solving a problem cooperatively using quantum multiple devices. Each node on a quantum network is connected via a classical and quantum channel and manages its own classical for storing bits of information such as measurements. Nodes may not modify or interact with qubits that they do not physically possess without performing teleportation, using non-local operations, or physically receiving the qubits from a different agent.

Current experimental distributed quantum computing is quite primitive. Experimental realization requires advances in quantum processor, network, and transduction techniques. The state of each technology has been extensively reviewed and will not be explicitly discussed in this paper **(Remember for Yewon to come back and add citations)**. There have been demonstrations of simple protocols such as quantum teleportation proving the possibility of more advanced quantum computing

[17]. However, this next step of performing more advanced algorithms of larger entangled systems has yet to be demonstrated due to the limitations in current quantum technology. Furthermore, there are few platforms providing means to simulate distributed quantum computation. Our framework is loosely based on quantum network simulation platform SQUANCH [13], however netQuil focuses on distributing quantum computation among multiple quantum computing agents.

III. FRAMEWORK OVERVIEW

In this section we briefly outline the components of a quantum network in netQuil, and provide some basic formalism for describing distributed quantum protocols. Throughout this paper a working knowledge of linear algebra and quantum information is assumed.

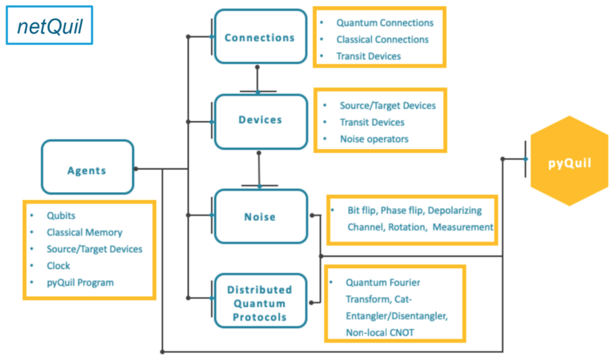

Figure 1. Layout of netQuil system

A. Quantum computation using pyquil

Pyquil is an open-source Python library for generating and executing quantum instruction language (Quil) programs on a quantum virtual machine (qvm) or on one of Rigetti's real quantum processing units (qpu). Quil programs are codified versions of quantum algorithms whose instructions correspond to specific quantum gates performed on a quantum system. Quil programs are line-based, assembly-like and provide conditional branching and classical feedback. Each trial run by a netQuil simulation operates on qubits managed by a single Quil program and returns the final Quil program to be run on a qvm or qpu. For this reason, agents do not pass qubits between each other, but rather, the index in the quantum system that the qubit corresponds to, along with the right to modify and operate using that qubit. By working within the bounds of pyQuil, netQuil both leverages its extensive quantum computing abilities and inherits its limitations, as discussed in section V.

B. Quantum agents

Multi-agent algorithms in quantum information theory are often described using the archetypal, fictional characters Alice and Bob. In netQuil, *Agent* is a base class, codified version of Alice or Bob, representing a single node in quantum network. Agents maintain a register for storing classical information, a local clock that increments based on traffic and device delay, and a set of qubits. Agents are connected via quantum and classical channels described by transit devices and maintain source and target devices that egress and ingress information travels through, respectively. Moreover, for parallelization, each agent runs on its own thread.

C. Quantum channels and devices

This subsection briefly explains quantum and classical channels and their properties and relationship to agents in a quantum network.

Conceptually, a channel is a communication line that allows the transmission of classical or quantum information between two or more agents. Channels can be modeled as a single device or a set of devices that quantum bits and classical bits must travel through. A device is a codified, single piece of hardware in the simulated quantum network. This would include devices such as a router, modem, repeater, and network switch in classical networking. In quantum networking, physical devices involved in quantum channels include fiber optics, transducers, free space, and quantum optical devices. Associated with each device in a quantum network is its physical effect on qubits, a consequence of undesirable environmental coupling (i.e., noise), and the time it takes qubits to pass through the device. In order to model this behavior, each device contains a noise model that is applied to a qubit in transit. Although there are a variety of errors that can occur during transmission, netQuil's noise models are limited to unitary operations and photon loss.

There are three types of devices in netQuil: source, transit, and target devices. Source and target devices are associated with an agent, while transit devices are associated with quantum channels. When a qubit leaves Alice and travels to Bob, the qubit originates from Alice's source devices, travels through the transit devices attached to their quantum channel and ends by arriving through Bob's target devices (Figure 2). The order in which a qubit passes through each device corresponds to the order in which those devices were added to either the agent or connection.

Most devices can be arbitrarily complex in their design and can depend on environmental factors such as temperature, humidity, or pressure. NetQuil has a host of built-in devices, but also allows users to build arbitrarily complex custom devices.

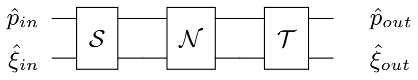

Figure 2. Circuit diagram representing a quantum state $\hat{p}_{in}$ traveling through source devices, $\mathcal{S}$, a quantum channel $\mathcal{N}$, and target devices $\mathcal{T}$ that each couple with the environment to produce a modified quantum state $\hat{p}_{out}$

D. Distributed Quantum Protocol

In this section we describe two primitive quantum protocol, the cat-entangler and cat-disentangler, for distributed quantum computing as introduced by Yimsiriwattana and Lomonaco [16, 18]. We then explain netQuil's implementation of the cat-entangler and cat-disentangler and how they can be cleverly combined to implement non-local CNOTS, non-local controlled gates, and teleportation. For a demonstration of teleportation using the cat-entangler and cat-disentanlger read Section IV D.

Eisert et al. [19] demonstrated that a universal set of gates for distributed quantum computing could be constructed through non-local CNOT gates and one-qubit gates. The cat-entangler and cat-disentangler are two primitive operations that were chosen as the basis for netQuils distributed quantum protocol for their ability to implement non-local operations such as the non-local CNOT.

1. *Cat-Entangler*

Using the cat-entangler, a single agent, Alice, in possession of an arbitrary control qubit, $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$, may distribute control among multiple agents, Bob and Charlie, given that all agents share a system of entangled qubits $|S\rangle = \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|111\rangle$ that can be placed in a 'cat-like' state, $\alpha|0000\rangle + \beta|1110\rangle$. The cat-entangler circuit is described in Figure 3. Non-local operations depend upon a means of distributing control among multiple qubits, making the cat-entangler a powerful primitive operator for distributed quantum computing.

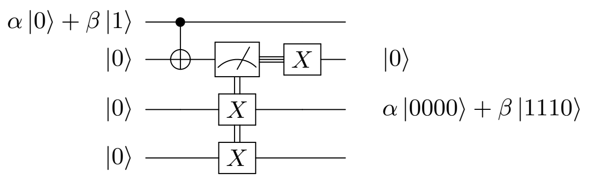
$\alpha|0\rangle + \beta|1\rangle$
$|0\rangle$
$|0\rangle$
$|0\rangle$
X
X
X
$|0\rangle$
$\alpha|0000\rangle + \beta|1110\rangle$

Figure 3. Cat-entangler circuit: The dark curved lines between wires two and four represent entangled qubits (i.e. $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$). In this case, wires one and two are owned by Alice, three by Bob, and four by Charlie. The double and triple lines represent a measurement result that is passed via a classical channel and used to control the **X** gates.

2. *Cat-Disentangler*

Once all agents have used the shared control bit to perform their local operation the cat-disentangler can be used to restore the system to its former state. Figure 4 outlines the cat-disentangler circuit.

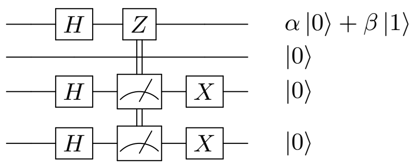

Figure 4. Cat-disentangler circuit: The **Z** gate on the first wire is controlled by the exclusive-or ($\oplus$) of the classical bits resulting from the measurements on qubits two and three. In netQuil, if **notify=True**, the cat-disentangler will send a classical bit to each participating agent (excluding the caller), notifying all parties that the entangler has finished. The caller is defined as the agent passed to the control.

IV. DEMONSTRATIONS

In this section we present a number of canonical quantum information protocol implemented using netQuil and pyQuil.

A. Quantum teleportation

Quantum teleportation is a protocol to deliver any arbitrary quantum state, $|\Psi\rangle$, between agents that share a maximally entangled state (a bell state pair) using a classical communication channel. Quantum teleportation is an extremely valuable and fundamental protocol for quantum networks, because it allows agents to transfer arbitrary quantum states to the sender.

Note that due to the requirement of classical communication, this protocol does not validate faster-than-light communication. Quantum teleportation transports a quantum state by sending two bits (one classical and one quantum), and thus is the inverse of superdense coding.

Quantum teleportation involves three agents, Alice, Bob, and Charlie. Charlie prepares a bell state pair and distributes the entangled qubits to Alice and Bob. Alice entangles her qubit $|\Psi\rangle$ with her bell state pair from Charlie, and then measures her qubits. Based on these measurements, Bob can recreate Alice's qubit $|\Psi\rangle$ by using X and Z gates. In this demonstration, we implement the canonical protocol using netQuil and pyQuil:

1. Charlie creates a bell state pair using a Hadamard (**H**) and controlled-not (**CNOT**) gate, $|AB\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. He then sends qubit A to Alice and qubit B to Bob.

2. Alice projects her arbitrary quantum state $|\Phi\rangle = \alpha|0\rangle + \beta|1\rangle$ onto qubit A using a controlled-not and a Hadamard gate. The 3-qubit quantum system, $|\Psi AB\rangle$, is in state: $\frac{1}{2}[|00\rangle(\alpha|0\rangle + \beta|1\rangle) + |01\rangle(\alpha|1\rangle + \beta|0\rangle) + |10\rangle(\alpha|0\rangle - \beta|1\rangle) + |11\rangle(\alpha|1\rangle - \beta|0\rangle)]$.

3. Alice measure $|\Psi\rangle$ and A and classically sends the results to Bob. As a result of the measurements, Bob's state collapses to one of the four bell states.

4. Bob recreates $|\Psi\rangle$ based on Alice's measurements, namely by applying a Pauli-X (**X**) gate if A is measured to be $|1\rangle$ or applying a Pauli-Z (**Z**) gate if Ψ is measured to be $|1\rangle$. Bob's qubit, B, is now in state $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$.

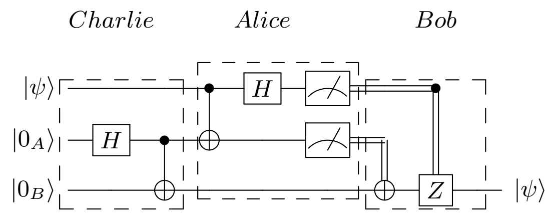

Figure 5. Teleportation: Alice wishes to send Bob her qubit state, $|\psi\rangle$, using classical information. She projects her state, $|\psi\rangle$, onto her bell state pair from Charlie and then collapses each qubit by measurement. She informs Bob of her measurements. Bob then applies a **X** gate, represented by a **CNOT** gate from the $|0_A\rangle$ measurement, and a **Z** from the $|\psi\rangle$ measurement. Thus, Bob can recreate Alice's state through classical information.

```
from netQuil import *
from pyquil import Program
from pyquil.api import QVMConnection
from pyquil.gates import *

class Charlie(Agent):
    '''Charlie distributes bell pair'''
    def run(self):
        # Create bell state pair
        p = self.program
        p += H(0)
        p += CNOT(0,1)

        self.qsend(alice.name, [0])
        self.qsend(bob.name, [1])

class Alice(Agent):
    '''Alice projects her state on pair'''
    def run(self):
        p = self.program

        # Define Alice's Qubits
        phi = self.qubits[0]
        qubits = self.qrecv(charlie.name)
        a = qubits[0]

        # Entangle Ancilla and Phi
        p += CNOT(phi, a)
        p += H(phi)

        # Measure Ancilla and Phi
        p += MEASURE(a, ro[0])
        p += MEASURE(phi, ro[1])

class Bob(Agent):
    '''Bob recreates Alice's state'''
    def run(self):
        p = self.program

        # Define Bob's qubits
        qubits = self.qrecv(charlie.name)
        b = qubits[0]

        # Prepare state
        p.if_then(ro[0], X(b))
        p.if_then(ro[1], Z(b))

p = Program()
p += H(2)

# Create classical memory
ro = p.declare('ro', 'BIT', 3)

# Create Alice, Bob, and Charlie.
alice = Alice(p, qubits=[2])
bob = Bob(p, name='bob')
charlie = Charlie(p, qubits=[0,1])

# Connect agents
QConnect(alice, bob, charlie)
CConnect(alice, bob)

# Run simulation
Simulation(alice, bob, charlie).run()
qvm = QVMConnection()
qvm.run(p)
```

B. Superdense coding

Superdense coding is a canonical protocol for delivering any two classical bits using a single quantum bit. The protocol allows two agents, Alice and Bob, to use a maximally entangled quantum system along with a single qubit to transmit two classical bits of information.

This protocol enables quantum computers to interact as a network in sharing classical information using qubits. Superdense coding transports two classical bits by sending a single qubit, and thus is the inverse of quantum teleportation. With stable qubit transit devices and a quantum memory, superdense coding in theory can double the information density of messages during the peak transmission and use down time to distribute the entangled pair.

Superdense coding involves three agents: Alice, Bob, and Charlie. Charlie prepares the bell state pair and distributes the entangled qubits to Alice and Bob. Alice operates on her bell state pair from Charlie based on the classical bits she wishes to send to Bob. Alice then sends her bell state pair to Bob. Finally, Bob switches back into a computational basis and measures each qubit to recreate Alice's classical bits.

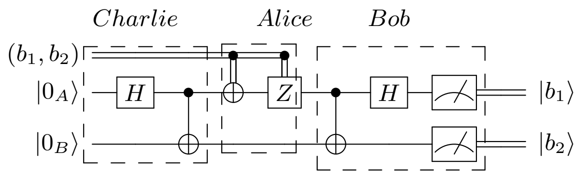

Figure 6. Superdense Coding: The b_1, b_2 bits are the classical bits that Alice is sending to Bob. Charlie creates and distributes the bell state pair to Alice and Bob. Alice projects her classical information onto her bell state pair qubit from Charlie and sends her bell state pair to Bob. Bob manipulates the qubits and measures the same classical information Alice originally wished to send to him.

1. Charlie creates a bell state pair using a Hadamard (**H**) and controlled-not (**CNOT**) gate, $|AB\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. He then sends qubit A to Alice and qubit B to Bob

2. Alice operates on her qubit based on the classical bits she wants to send to Bob. If her first classical bit is a 1, she operates on her qubit with an **X** gate. If her second classical bit is a 1, she operates on her qubit with a **Z** gate. Then, she sends her qubit to Bob.

   The 2-qubit quantum system, $|\Psi AB\rangle$, is one of the four bell states: $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle), \frac{1}{\sqrt{2}}(|10\rangle + |01\rangle), \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$.

3. Bob returns to the computational basis by applying a controlled-not and a Hadamard gate to the qubit from Charlie and Alice. Finally, Bob measures each qubit and now has both of Alice's classical bits.

```
from netQuil import *
from pyquil import Program
from pyquil.api import QVMConnection
from pyquil.gates import *

class Charlie(Agent):
    '''Charlie distributes bell pair'''
    def run(self):
        # Create Bell State Pair
        p = self.program
        p += H(0)
        p += CNOT(0,1)

        self.qsend(alice.name, [0])
        self.qsend(bob.name, [1])

class Alice(Agent):
    '''Alice sends superdense-encoded cbits'''
    def run(self):
        p = self.program
        qCharlie = self.qrecv(charlie.name)
        a = qCharlie[0]

        bit1 = self.cmem[0]
        bit2 = self.cmem[1]

        if bit2 == 1: p += X(a)
        if bit1 == 1: p += Z(a)
        self.qsend(bob.name, [a])

class Bob(Agent):
    '''
    Bob reconstructs Alice's classical bits
    '''
    def run(self):
        p = self.program

        # Get qubits from Alice and Charlie
        qAlice = self.qrecv(alice.name)
        qCharlie = self.qrecv(charlie.name)
        a = qAlice[0]
        c = qCharlie[0]

        p += CNOT(a,c)
        p += H(a)
        p += MEASURE(a, ro[0])
        p += MEASURE(c, ro[1])

p = Program()
p += H(2)

# Create classical memory
ro = p.declare('ro', 'BIT', 3)

# Create Alice, Bob, and Charlie.
alice = Alice(p, qubits=[2])
bob = Bob(p, name='bob')
charlie = Charlie(p, qubits=[0,1])

# Connect agents
QConnect(alice, bob, charlie)
CConnect(alice, bob)

# Run simulation
Simulation(alice, bob, charlie).run()
qvm = QVMConnection()
qvm.run(p)
```

C. Middle-man attack

Middle-Man Attack is a demonstration of quantum networks resistance against intruders. This demo extends superdense coding by allowing two agents, Alice and Bob, to send numerous classical bits using bell state pairs. However, in this protocol, there is an intruder agent, Eve, who attempts to intercept and measure the information. Despite successfully intercepting Alice and Bob's message, due to Eve not sharing a bell state pair with Alice, Eve only measures random noise. Moreover, Bob is able to detect if an intruder has intercepted the message.

The middle-man attack involves four agents, Alice, Bob, Charlie, and Eve. Following the superdense coding protocol, Charlie prepares the bell state pair and distributes the entangled qubits to Alice and Bob. Alice then operates on her bell state pair from Charlie based on the classical bits she wishes to send to Bob and sends her bell state pair to Bob. This process can be repeated for any number of classical bits that Alice wishes to send.

The attack occurs when Eve intercepts Alice's qubits on the way to Bob, measures the qubits, and re-transmits them to Bob. Due to Eve not having a bell state pair and her measurement collapsing the state of the qubit, Eve only intercepts random noise, while Bob can immediately detect the presence of an intruder.

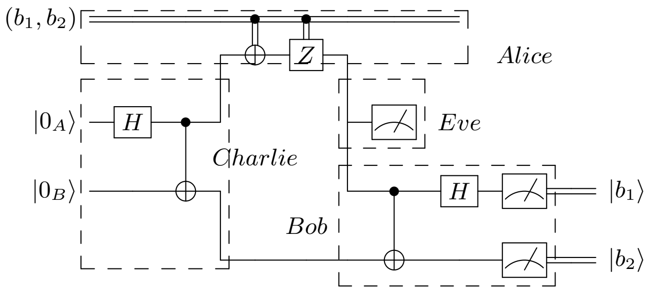

Figure 7. Middle-Man Attack Circuit: The b_1, b_2 bits are classical bits that Alice is attempting to send to Bob. Charlie creates and distributes the bell state pair to Alice and Bob. Eve intercepts and measures the qubit sent from Alice to Bob, which is only random noise as Eve does not have the bell state pair, and alerts Bob to an intruder.

1. Charlie creates a bell state pair using a Hadamard (**H**) and controlled-not (**CNOT**) gate, $|AB\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. He then sends qubit A to Alice and qubit B to Bob

2. Alice operates on her qubit based on the classical bits she wants to send to Bob. If her first classical bit is a 1, she operates on her qubit with a **X** gate. If her second classical bit is a 1, she operates on her qubit with a **Z** gate. Then, she sends her qubit (unknowingly) to the intruder, Eve.

3. Eve measures the qubit from Alice and re-transmits it to Bob.

4. Bob returns to the computational basis by applying a controlled-not and a Hadamard gate to the qubit from Charlie and from Eve (thinking it is from Alice). Finally, Bob measures each qubit and the results are equal to both of Alice's classical bits.

```
from netQuil import *
from pyquil import Program
from pyquil.api import QVMConnection
from pyquil.gates import *

class Charlie(Agent):
    '''Charlie distributes bell pair'''
    def run(self):
        # Create Bell State Pair
        p = self.program
        p += H(0)
        p += CNOT(0,1)

        self.qsend(alice.name, [0])
        self.qsend(bob.name, [1])

class Alice(Agent):
    '''Alice sends superdense-encoded cbits'''
    def run(self):
        p = self.program
        for i in range(0,len(self.cmem),2):
            bit1 = self.cmem[i]
            bit2 = self.cmem[i+1]
            qCharlie = self.qrecv(charlie.name)
            a = qCharlie[0]

            if bit2 == 1: p += X(a)
            if bit1 == 1: p += Z(a)

            self.qsend(eve.name, [a])

class Bob(Agent):
    '''
    Bob reconstructs Alice's classical bits
    '''
    def run(self):
        p = self.program
        for i in range(0,len(alice.cmem),2):
            qAlice = self.qrecv(eve.name)
            qCharlie = self.qrecv(charlie.name)
            a = qAlice[0]
            c = qCharlie[0]
            p += CNOT(a,c)
            p += H(a)
            p += MEASURE(a, ro[i])
            p += MEASURE(c, ro[i+1])
```

```
class Eve(Agent):
    '''
    Eve intercepts, measures, and sends to Bob
    '''
    def run(self):
        p = self.program
        for i in range(0,len(alice.cmem),2):
            qAlice = self.qrecv(alice.name)
            a = qAlice[0]
            p += MEASURE(a, ro[i+len(alice.cmem)])
            self.qsend(bob.name, [a])

import matplotlib.image as image

img = image.imread("./Images/Logo.jpeg")
img_bits = list(np.unpackbits(img))

program = Program()
ro = program.declare('ro', 'BIT', 2*len(img_bits))

qubitsUsed = list(range(len(img_bits)))
resultsEve = []
resultsBob = []

alice = Alice(program, cmem=img_bits)
bob = Bob(program)
charlie = Charlie(program, qubits=qubitsUsed)
eve = Eve(program)

QConnect(alice, bob, charlie, eve)

#define agents
alice = Alice(program, cmem=curImg_bits)
bob = Bob(program)
charlie = Charlie(program, qubits=qubitsUsed)
eve = Eve(program)

#connect agents
QConnect(alice, bob, charlie, eve)


#simulate agents
Simulation(alice,charlie,bob,eve).run()
qvm = QVMConnection()
results = qvm.run(program)

#window of current quantum bits
startWindow = 0
endWindow = 20

while end <= len(img_bits):
    curImg_bits = img_bits[start:end]
    qubitsUsed = list(range(len(curImg_bits)))

    program = Program()
    mem_len = 2*len(curImg_bits)
    ro = program.declare('ro', 'BIT', mem_len)
    #define agents
    alice = Alice(program, cmem=curImg_bits)
    bob = Bob(program)
    charlie = Charlie(program, qubits=qubitsUsed)
    eve = Eve(program)

    #connect agents
    QConnect(alice, bob, charlie, eve)

    #simulate agents
    Simulation(alice,charlie,bob,eve).run()
    qvm = QVMConnection()
    results = qvm.run(program)

    #record results
    resBob = results[0][0:len(curImg_bits)]
    resEve = results[0][len(curImg_bits):]
    resultsBob.extend(resBob)
    resultsEve.extend(resEve)

    #iterate
    start = end
    if end == len(img_bits):
        break
    elif len(img_bits) >= end+20:
        end += 20
    else:
        end = len(img_bits)
```

D. Distributed Protocol

In this demonstration, we introduce netQuil's distributed protocol library that implements a set of non-local operations. This library will introduce the primitive cat-entangler and cat-disentangler as introduced by Yimsiriwattana and Lomonaco [16], and their usage in non-local CNOTs, non-local controlled gates, and teleportation as described in section III D.

NetQuil's implementation of the cat-entangler requires that only one agent initiate and execute the circuit. NetQuil will transport the qubits and cbits between agents, update their clocks, and appropriately apply devices. If **notify=True**, the cat-entangler will send a classical bit to each participating agent (excluding the caller), notifying all parties that the entangler has finished. The caller is defined as the agent passed to the control. If **entangled=False**, the cat-entangler will entangle the target qubits and the measurement qubit before performing the circuit.

Once all agents have used the shared control bit to perform their local operation, the cat-disentangler can be used to restore the system to its former state.

It has been shwon that the controlled-NOT gate, Hadamard gate, and 45° phase gate together can be composed to create a universal quantum gate [19]. Therefore,

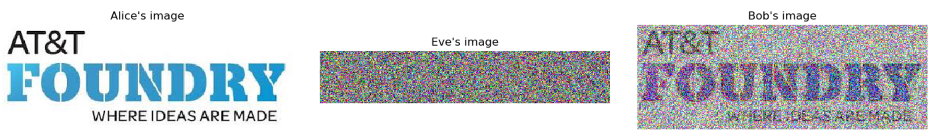

Figure 8. Middle Man Demo Image: Alice wants to send her image made up of classical bits to Bob using qubits. Eve intercepts the qubits from Alice to Bob, but only measures random noise. Eve's intrusion alerts Bob to her presence as half of his qubit measurements are corrupted.

in order to construct a universal set of operators for DQC, we must only construct a non-local CNOT gate, which can be done with the cat-entangler and cat-disentangler.

Below is an example of teleportation using the cat-entangler and cat-disentangler.

```
from netQuil import *
from pyquil import Program
from pyquil.api import WavefunctionSimulator,
QVMConnection
from pyquil.gates import *

class Alice(Agent):
    '''
    Alice uses cat-entangler and
    cat-disentangler to teleport psi to Bob
    '''
    def teleportation(self, psi, a, b):
        cat_entangler(
            control=(self, psi, a, ro),
            targets=[(bob, b)],
            entangled=False,
            notify=False
        )
        cat_disentangler(
            control=(bob, b, ro),
            targets=[(self, psi)],
        )

   def run(self):
        # Define Qubits
        a, psi = self.qubits
        b = bob.qubits[0]

        # Teleport
        self.teleportation(psi, a, b)

class Bob(Agent):
    '''
    Bob waits for teleportation to complete
    '''
    def run(self):
        # Receive Measurement from Cat-entangler
        self.crecv(alice.name)

p = Program()
# Prepare psi
p += H(2)
p += RZ(math.pi/2, 2)

# Create Classical Memory
ro = p.declare('ro', 'BIT', 3)

alice = Alice(p, qubits=[0,2], name='alice')
bob = Bob(p, qubits=[1], name='bob')

QConnect(alice, bob)
CConnect(alice, bob)

Simulation(alice, bob).run()
qvm = QVMConnection()
qvm.run(p)
```

E. Advanced usage

As described in section III C, netQuil's devices module has a number of built-in source, transit, and target devices. NetQuil's custom devices allow users to simulate arbitrarily complex devices. All devices must have an **apply** function that is responsible for the device's activity. The **run** function must return a dictionary that optionally contains the 'lost qubits' and 'delay'. The 'delay' represents the time it took qubits to travel through the device, while 'lost qubits' correspond to information completely lost due to attenuation (i.e., photon loss). In netQuil, If a qubit is lost the target will receive the negative index of the qubit lost. For example, if Alice sends qubit 3 and it is lost due to attenuation, Bob will receive -3, and neither Alice nor Bob will be able to operate on qubit 3. If the qubit lost is at index 0, then the value will be set to -inf. Remember that when we send qubits between agents we are sending the index of the qubit in the program and not the true qubit. If qubits are lost while passing through the device, return an entry in the dictionary **lost_qubits: [lost qubits]**.

In some situations, pyQuil programs generated between trials will be different depending on noise or the dynamic nature of your network. In order to for accommodate this, **Simulation().run()** will always return a list of programs (i.e. one program per trial) that can be run on a qvm or qpu. Pass the number of trials to be run into **Simulation().run(trials=5)**, as well as a list containing the class of each agent being run. Do NOT forget to pass **agent_classes** (i.e. **Simulation(Alice, bob).run(trials=5, agent_classes=[Alice, Bob]**), since this is required in order to reset the agents between trials.

It is also possible to see a list of transactions in the network, the time of each transaction, and information about network devices by setting **network_monitor=True** in **run**. In addition to individual agent clocks, a master clock is running throughout the network simulation and can be accessed through **agent.get_master_time()**. This may be useful for time-based encodings (e.g. time-bin encoding).

Here, we combine these advanced features into a simple example program:

```
class Simple_Fiber(Device):
    def __init__(self, len, fq, std):
        # fiber quality
        self.fq = fq
        self.length = len
        self.rot_std = std
        # speed of light in km/s
        self.signal_speed = 2.998 * 10 ** 5

    def apply(self, program, qubits):
        rot_std = self.rot_std
        for qubit in qubits:
          rot_angle = np.random.normal(0, rot_std)
          # Apply noise
          if np.random.rand() > self.fq:
              program += RX(rot_angle, qubit)

        delay = self.length/self.signal_speed

        return {
            'delay': delay,
        }

class Alice(Agent):
    def run(self):
        p = self.program
        for q in self.qubits:
            p += H(q)
            p += X(q)
            self.qsend('Bob', [q])

class Bob(Agent):
    def run(self):
        p = self.program
        for _ in range(3):
            q = self.qrecv(alice.name)[0]

            # Check if qubit is lost
            if q >= 0:
                p += MEASURE(q, ro[q])

p = Program()
ro = p.declare('ro', 'BIT', 3)

alice = Alice(p, qubits=[0,1,2])
bob = Bob(p)

# Define source device
laser = Laser(rotation_prob_variance=.9)
alice.add_source_devices([laser])

# Define transit devices and connection
length = 5
fiber_quality = .6
standard_dev_of_rotation = .1

custom_fiber = Simple_Fiber(
    len=length,
    fq=fiber_quality,
    std=stadard_dev_of_rotation
)
fiber = Fiber(
    length=5,
    attenuation_coefficient=-.20
)

QConnect(alice, bob,
  transit_devices=[fiber, custom_fiber]
)

# Run simulation
programs = Simulation(alice, bob).run(
    trials=5,
    agent_classes=[Alice, Bob]
)

# Run programs
qvm = QVMConnection()
for idx, program in enumerate(programs):
    results = qvm.run(program)
    print('Program {}: '.format(idx), results)
```

V. FUTURE WORKS

In this section we briefly discuss some of the current limitations of netQuil and propose solutions for significant improvement. NetQuil is an open-source project, and we encourage contributions at Github.

A. Quil Limitations

NetQuil is a framework for simulating quantum networks built on the open-source Python package, pyQuil. PyQyuil is a framework that allows users to create and execute quantum instruction language (Quil) programs.

Quil programs are assembly-like and intended to be run on a quantum virtual machine or quantum processing unit, representing a single, real or virtual quantum computer or one of its components. For this reason, Quil programs are not intended to be shared or modified between quantum computers and do not give developers access to individual qubits, but rather a reference to the qubits' indices. Each netQuil trial generates a single Quil program that is shared between users and modified during the simulation. As a result, rather than passing qubits (or their mathematical representation) between each other, agents must pass the index of the qubit in the Quil program, along with the right to modify and operate using that qubit. In this framework, qubits are never truly lost due to attenuation or decoherence. Instead all agents loss the right to modify the qubit or use it for operations. Client's are able to access lost qubits by examining the quantum system's state.

Finally, the execution of Quil programs slows down exponentially as the number of qubits in the system increases, because Quil maintains a representation of the entire quantum state. Generally, without allocating additional resources, QVMs typically cannot execute Quil programs beyond 30 qubits, and likewise, netQuil simulations are limited to 30 qubits.

B. Qubit representation - photons, etc..

In order to accurately simulate quantum computations, a mathematical formation of qubits must be chosen that is based upon a physical technology. The chosen representation of a qubit must emulate a two state quantum system that exhibits quantum mechanical properties, most notably superposition. Moreover, this representation must be easily manipulated and allow for modeling coherence and decoherence. In netQuil, devices are based off of photon qubits that use time-bin encoding [10]. NetQuil's noise and device module can be used to model any unitary quantum noise. Time-bin encoding is a promising quantum networking qubit candidate as it is resistant to decoherence, however, time-binned qubits are often difficult to manipulate and entangle in large systems.

C. Distributed netQuil Simulations

NetQuil currently runs quantum networking simulations on a single classical computer. As a result, when a classical channel is established between agents, netQuil simply simulates the classical network by passing bits between agents within the program. However, for quantum networks that depend on classical channels for communication, as in the case of networks that use quantum teleportation for quantum state transfer, netQuil could be improved by allowing for simulations that run across a cluster of computers or agents that run on individual servers.

VI. CONCLUSION

NetQuil offers users of all backgrounds a framework for developing, simulating, and studying quantum information theory. NetQuil is particularly useful for algorithms that involve many-agent computations, distributed quantum gates, entangled qubit systems, and realistic noise and device models. We hope netQuil allows users to explore the possibilities of distributed quantum computing.

VII. ACKNOWLEDGEMENTS

We wish to thank the entire team at AT&T Foundry for providing support and helpful feedback throughout development. All quantum circuits depicted in this paper were drawn using the *Q-circuit* LaTeXpackage. Questions can be directed to mattradz@stanford.edu and zespinos@stanford.edu.

[1] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[2] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.

[3] IBM-Research. Ibm q experience, 2016.

[4] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[5] Microsoft. Quantum development kit, 2016.

[6] Bernhard Ömer. A procedural formalism for quantum computing. 1998.

[7] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Notices*, volume 48, pages 333–342. ACM, 2013.

[8] Nader Khammassi, Imran Ashraf, Xiang Fu, Carmen G Almudever, and Koen Bertels. Qx: A high-performance quantum computer simulation platform. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 464–469. IEEE, 2017.

[9] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry fields: A software platform for photonic quantum computing. *Quantum*, 3:129, 2019.

[10] Thaddeus D Ladd, Fedor Jelezko, Raymond Laflamme, Yasunobu Nakamura, Christopher Monroe, and Jeremy Lloyd O'Brien. Quantum computers. *nature*, 464(7285):45, 2010.

[11] Ni-Ni Huang, Wei-Hao Huang, and Che-Ming Li. Identification of networking quantum teleportation on 14-qubit ibm universal quantum computer. *Scientific reports*, 10(1):1–12, 2020.

[12] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. Quantum computation by adiabatic evolution. *arXiv preprint quant-ph/0001106*, 2000.

[13] Ben Bartlett. A distributed simulation framework for quantum networks and channels. *arXiv preprint arXiv:1808.07047*, 2018.

[14] Axel Dahlberg and Stephanie Wehner. Simulaqron—a simulator for developing quantum internet software. *Quantum Science and Technology*, 4(1):015001, 2018.

[15] Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, page 36, 2007.

[16] Anocha Yimsiriwattana. Generalized ghz states and distributed quantum computing anocha yimsiriwattana and samuel j. lomonaco, jr. In *Coding Theory and Quantum Computing: An International Conference on Coding Theory and Quantum Computing, May 20-24, 2003, University of Virginia*, volume 381, page 131. American Mathematical Soc., 2005.

[17] Raju Valivarthi, Samantha I Davis, Cristián Peña, Si Xie, Nikolai Lauk, Lautaro Narváez, Jason P Allmaras, Andrew D Beyer, Yewon Gim, Meraj Hussein, et al. Teleportation systems toward a quantum internet. *PRX Quantum*, 1(2):020317, 2020.

[18] Anocha Yimsiriwattana and Samuel J Lomonaco Jr. Distributed quantum computing: A distributed shor algorithm. In *Quantum Information and Computation II*, volume 5436, pages 360–372. International Society for Optics and Photonics, 2004.

[19] Jens Eisert, Kurt Jacobs, Polykarpos Papadopoulos, and Martin B Plenio. Optimal local implementation of non-local quantum gates. *Physical Review A*, 62(5):052317, 2000.

Appendix A: Appendix A: Full source code

The full source code for netQuil is available on GitHub at github.com/att-innovate/netQuil or from The Python Package Index. Documentation is at att-innovate.github.io/netQuil. The source code for netQuil can be found in the /demos directory of the GitHub repository.