IEEE Transactions on Quantum Engineering

Received March 25, 2021; revised June 7, 2021; accepted June 22, 2021; date of publication June 25, 2021; date of current version July 20, 2021.

Digital Object Identifier 10.1109/TQE.2021.3092395

QuNetSim: A Software Framework for Quantum Networks

STEPHEN DIADAMO, JANIS NÖTZEL, BENJAMIN ZANGER, AND MEHMET MERT BEŞE

Technische Universität München, 80333 Munich, Germany

Corresponding author: Stephen Diadamo (stephen.diadamo@gmail.com).

This work was funded by the Deutsche Forschungssgemeinschaft (DFG) under Grant 1129/2-1 and the Unitary Fund.

ABSTRACT As quantum network technologies develop, the need for teaching and engineering tools such as simulators and emulators rises. QuNetSim addresses this need. QuNetSim is a Python software framework that delivers an easy-to-use interface for simulating quantum networks at the network layer, which can be extended at little effort of the user to implement the corresponding link layer protocols. The goal of QuNetSim is to make it easier to investigate and test quantum networking protocols over various quantum network configurations and parameters. The framework incorporates many known quantum network protocols so that users can quickly build simulations using a quantum-networking toolbox in a few lines of code and so that beginners can easily learn to implement their own quantum networking protocols. Unlike most current tools, QuNetSim simulates with real time and is, therefore, well suited to control laboratory hardware. Here, we present a software design overview of QuNetSim and demonstrate examples of protocols implemented with it. We describe ongoing work, which uses QuNetSim as a library, and describe possible future directions for the development of QuNetSim.

INDEX TERMS Quantum Internet, quantum networking, quantum simulation, quantum software.

I. INTRODUCTION

A quantum network is a network of physical devices that are able to transmit quantum information and distribute quantum entanglement amongst themselves. As developments are made toward realizing a standardized quantum Internet [1], [2], there is a stronger need to efficiently develop and test quantum networking protocols and applications. Recently, there has been much effort into developing quantum simulation software for quantum computing [3], whereas simulation software for quantum networks has received considerably less attention. An aspect that is much more critical to networking than to computing is asynchrony between nodes in the network, and an aspect that is important to the engineer is to deploy code that is written once in a simulation directly to a device. The initial release of QuNetSim addresses this need by providing a lightweight, easy-to-use, open-source quantum network simulation framework, which runs in real time and is, therefore, in principle also able to interact with laboratory equipment.

As it has been done for the classical Internet with, for example, the NS-3 [4] and Mininet [5] platforms, work toward a similar open-source simulation platform with many contributors should be developed for quantum networking. Although there have been developments in quantum network simulators, as we discuss in Section III, presently, we think there is a gap between network simulators that work on a low level and network simulators that are easy-to-use and can be used in a testing phase of protocol development for quantum networks.

The goal for QuNetSim is to provide a high-level framework that allows users to quickly develop quantum networking protocols without having to invest time on purely software-related tasks, such as managing threading and synchronization, writing thread-safe logic, or repeatedly implementing basic protocols that can be used as building blocks for new protocols. QuNetSim further aims to provide an open-source simulation platform offering a high degree of freedom to attract contributors, enabling even more simulation possibilities. As a consequence of meeting these goals, the learning curve needed to begin developing protocols for quantum networks is flattened, since QuNetSim makes it easier to write them. QuNetSim allows users to create examples of quantum networking protocols that are along the lines of how protocols are developed as a first stage for research papers, which helps us to develop protocols as well as educate students about quantum networks. In the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

examples, we provide, we see how there is an almost one-to-one correspondence between how protocols are written in quantum network protocol research papers as to how simulations are developed in QuNetSim. In future releases, we aim to subsequently professionalize the process of emulating parts of quantum networks interfacing with QuNetSim, or to use QuNetSim for the purpose of creating emulations. Although the currently limited availability of off-the-shelf hardware components for quantum networks leaves the network engineer with many unfulfilled needs, QuNetSim offers, at least in principle, already today the possibility for using hardware in the loop.

In this article, we will give an overview of the design and implementation of QuNetSim, detailing its layered structure and how it works in principle. We then review the other quantum network simulation platforms that are available. Lastly, we demonstrate with some examples quantum network protocols implemented in QuNetSim.

II. OVERVIEW OF QUNETSIM

The current main purpose of QuNetSim, as the name suggests, is to simulate quantum networks. To this end, we aim to allow for the writing and testing of robust protocols for multihop quantum transmission with various network parameters and configurations. QuNetSim allows users to create network configurations of nodes connected via classical or quantum links and then program the behavior of each node in the network as they choose to. To simulate the asynchrony of networks, QuNetSim runs in a multithreaded environment using first-in-first-out queues for packets. The generally challenging programming problem of managing multiple threads is simplified as QuNetSim provides the methods to synchronize the nodes in the network even when they are all performing their actions independently and asynchronously. Furthermore, QuNetSim comes with many built-in protocols such as teleportation, EPR generation, GHZ state distribution and more, over arbitrary network topologies that make it easier to develop more complex protocols, using the basic ones as a toolbox. It also provides an easy way of constructing a complex network topology such that one can design and test routing algorithms for quantum networks.

QuNetSim uses a network layering model inspired by the OSI model [6]. It naturally incorporates control information together with any payload type, but is open to modifications where control information is explicitly transmitted separate from payload. In future quantum network implementations, it is likely that exact OSI model layering model will not be used and it could be that new layers are introduced. For example, an already proposed network layering includes a "connectivity layer" between the link and physical layer [7]. We anticipate that the basic concept of layering of classical communication networks will be carried over to quantum networks, so that included will be layers such as application, transport, and network. Quantum information carriers will be encoded into some form of packets, which will then be routed through the network to the desired destination. In its initial

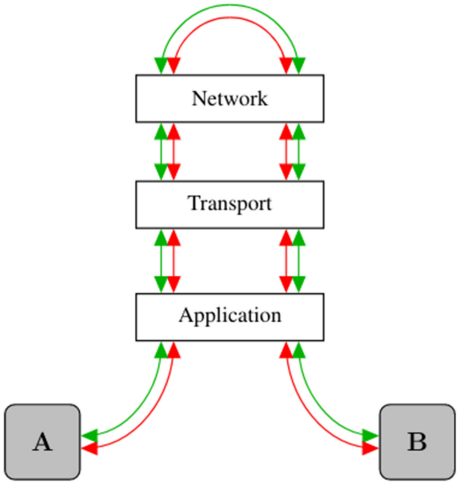

FIG. 1. Design depiction of QuNetSim. Here, the green line represents a classical channel and the red a quantum channel. QuNetSim attempts to simulate the process of moving both classical and quantum packets through a set of network layers as does the classical Internet. Here, host A has a virtual connection to host B, and so all of their communication is processed one layer at a time. QuNetSim does not explicitly incorporate features of the higher layers such as the link-layer or the physical layer.

form, QuNetSim implements the network layer and assumes link- and physical-layer deliver error-free bit and qubit transmission capabilities. While it is relatively straightforward to model link layer (in the sense of a logical but potentially not error-free qubit channel between two nodes) behavior in QuNetSim, the modeling of a physical layer (in the sense of, e.g., a continuous-variable quantum system) is currently not considered as within the scope of QuNetSim. By design, the accurate modeling of lower layers is left to simulators, which are better suited for that task.

In Fig. 1, we depict the high-level design structure of QuNetSim. At this depth, it resembles a virtual connection between two nodes in a classical network, where "virtual connection" means that node **A** has the perspective that it is directly connected to node **B** even though the information sent from **A** is routed through the network with potentially many relaying hops. In the figure, the two nodes are connected (virtually) by a classical channel, represented by the green lines, and a quantum channel, represented by the red lines. Both modes of communication are processed through the same layering mechanism as the network is able to route both kinds of information but makes decisions based on the payload of the packets. This allows users to use the same programming logic for sending classical messages as for sending quantum, leaving it to the lower layers to work out the differences.

III. COMPARISON TO OTHER QUANTUM NETWORK SIMULATORS

A detailed list of quantum software libraries hosted at [3] contains approximately 100 different flavors of quantum simulation software. Most of these are directed at simulating

quantum computation and circuitry on various hardware configurations with various levels of realism. With regards to quantum networking, as far as we know at this time, there are five publicly available quantum network simulators: SimulaQron [8], NetSquid [9], simulator for quantum networks and channels (SQUANCH) [10], quantum Internet simulation package (QuISP) [11], and SeQUeNCe [12]. Of these, all are libraries are freely available. SQUANCH, QuISP, SeQUeNCe, and SimulaQron are open-source under the MIT license or 3-Clause BSD License. NetSquid uses a license restricting its use for educational and noncommercial research and development purposes. Indeed, there are simulators that simulate quantum key distribution (QKD) [13], [14], but this are single use case simulators and do not allow for arbitrary application layer protocols and we do not compare them to QuNetSim for that reason.

SimulaQron [8] is a simulator that can be used for developing quantum Internet software. It simulates several quantum processors that are located at the end nodes of a quantum network and are connected by simulated quantum links. The main purpose of SimulaQron is to simulate the application layer of a network; tasks such as routing are left to the user to implement using their own approach if needed. SimulaQron further offers the ability to run simulations across a distributed system, that is, simulations can be set up to run on multiple computers. What we found slightly lacking in SimulaQron is a way to easily synchronize the parties regarding qubit arrival. A key difference in QuNetSim is that it adds a layer of synchronization. Built into QuNetSim is the approach of acknowledging when information arrives at the receiver. One can more naturally write protocols in a standard way, where one handles the information arriving or not before proceeding. SimulaQron also has hosts, which have features such as sending qubits, establishing Einstein-Podolsky-Rosen (EPR) pairs, and sending classical information. To simplify the task of developing protocols on top of existing protocols, we try to include more built-in tasks such as sending teleportation qubits, establishing a GHZ state, and establishing a secret key using QKD.

NetSquid [9] is a powerful event-driven quantum network simulator. It can simulate the physical properties of quantum devices such as quantum gate and memory, noise and loss of a quantum channel, and time-dependent quantum state decoherence. NetSquid can be used as a benchmarking tool to test robustness of quantum network protocols against the physical and link layer effects of the network. It uses a modular approach for network configuration, allowing users to customize their simulations in many ways. A key strength of NetSquid is its ability to incorporate the time-dependent effects of quantum systems. With these features comes a layer of added complexity that falls to the user. To use NetSquid to its full extent, the user should have a fairly good understanding of the hardware structure of a quantum network. In contrast, QuNetSim is designed to more easily develop quantum protocols and test them for correctness over networks, and not for benchmarking against physical properties of the network. This greatly reduces the need to understand quantum networks at a deep level, but does remove the ability to benchmark quantum networking protocols against any hardware specifications. One could use both NetSquid and QuNetSim together to develop their ideas as a first prototyping step with QuNetSim and then benchmark the protocols in NetSquid to see how it performs with the added physical models.

SQUANCH [10] achieves similar functionality as SimulaQron but allows for customizable physical layer properties and error models. It allows for creating simulations of distributed quantum information processing that can be parallelized for more efficient simulation. It is designed specifically for simulating quantum networks to test ideas in quantum transmission and networking protocols. SQUANCH can be used to simulate many qubits and can allow a user to add their own error models, which we think allows for a more realistic quantum network simulator. SQUANCH also allows one to separate the quantum and classical networks of a complete network and as well as adding length dependent noise to the channel. A key difference between SQUANCH and QuNetSim is that in SQUANCH, as far as we know, a node can run one set of instructions at a time and not more in parallel. This may not be so restrictive, but in multiparty protocols, it may become challenging to develop all of the behavior in one set of instructions. QuNetSim allows one to develop multiparty protocols one at a time and run them in parallel. Furthermore, synchronization between parties is again potentially an issue with SQUANCH. QuNetSim gives each host an addressable quantum memory such that given an ID, they can fetch a qubit and can manipulate it as desired. In SQUANCH, one should initialize their qubits before the start of the simulation whereas with QuNetSim qubits are initialized at run time. We think this adds more flexibility when writing protocols and allows for more natural logic in the code.

QuISP [11] is also an event-driven simulation of quantum repeater networks. The goal of QuISP is to simulate a full quantum Internet consisting of up to 100 networks of up to 100 nodes each and 100 qubits at each node. Its focus is on protocol design and studying emergent behaviors of complex, heterogeneous networks at large scale, while keeping the physical layer as realistic as possible. In comparison to QuNetSim, QuISP uses a different approach in regards to how qubits are simulated. The quantum state vector of the qubits in the system are not represented. To simulate a large scale network as QuISP does it is not possible to store the state information of many mutually entangled qubits, as the size of their state vector grows exponentially fast with the number of qubits. Instead, QuISP stores the errors applied on the qubits, which simplifies the qubit data structure. In QuNetSim, in order to get a better sense of the network's effects, we do provide the qubit state vector, albeit with the tradeoff that large scale simulations that are possible with QuISP are not with QuNetSim. QuISP also follows the "RuleSet" paradigm for programming the logic of the

network [15]. QuNetSim uses standard Python programming to define the actions in the network, in-line with the goal of simplicity.

SeQUeNCe [12] is a discrete-event quantum network simulator as are QuISP and NetSquid. It is customizable, so users can define the network topology, the hardware parameters of the nodes, and the actions the nodes take regarding storing qubits. SeQUeNCe has physical models built in and follows a modularized design with cross-module communication to allow for flexibility of the simulation. In this regard SeQUeNCe and NetSquid are the most similar from the collection. In comparison to QuNetSim, again the main difference is QuNetSim is not a discrete-event simulator and focuses on simplifying protocol development rather than benchmarking.

A real-time simulator such as QuNetSim allows the user to easily connect to physical hardware devices, allowing for a "hardware-in-the-loop" type of simulation. Indeed we have demonstrated splitting the quantum simulation backend aspect from the application development part of QuNetSim, controlling the quantum simulation through microcontroller instructions. With this, we can replace the quantum simulation devices with physical hardware for "lab-in-the-loop" behavior. QuNetSim can be integrated as an external library for simulating distributed quantum systems. Indeed an initial implementation of a distributed quantum computing emulation has been demonstrated [16]. The Interlin-q project simulates the execution of distributed quantum algorithms using QuNetSim to perform entanglement distribution and classical communication, the necessary ingredients for distributed quantum computing.

IV. ASSUMPTIONS MADE ABOUT FUTURE QUANTUM NETWORKS IN QUNETSIM

Although much research is directed at building a quantum Internet, or more generally and abstractly a network of nodes with the ability to create, distribute, and store quantum states, currently such a network does not exist. In order to build a simulation software framework as general as possible while attempting to keep it simple, we, therefore, make only few assumptions that we think will most likely be met by future quantum networks.

QuNetSim assumes that both classical and quantum information transmitted via future quantum networks will use signal processing that is modeled based on quantum mechanics. However, following the principles of network layering, not all information that is available at the lower layers can be accessed by the upper layers. Therefore, the class `quantum_connection` in QuNetSim is assumed to model an error-free logical qubit channel between network hosts where the physical means of transmitting a qubit necessarily involve also sending—potentially classical—control information such as desired destination node, type of error correction applied, and synchronization information all of which might even rely on an exchange of information via a feedback loop from receiver to sender. In future networks, not every connection between network nodes will necessarily be able to transmit quantum information. Therefore, QuNetSim offers the additional class `classical_connection`, which allows the transmission of classical data alone.

Another assumption we make is that quantum nodes will be able to detect that a qubit has arrived in their qubit storage. The practical methods (e.g., heralding) used to achieving this functionality are left open. QuNetSim aims to make it easy to synchronize between hosts, and therefore, we allow acknowledgment of the arrival of qubits. Since QuNetSim simulates up to the network layer, we assume that the lower layers of the quantum network will be able to provide protocols to accomplish this. This is different than current QKD networks that simply measure the qubits as they arrive without storage. In such an implementation, the measurement output acts as a herald but inevitably destroys quantum information and is, therefore, not suitable for a quantum network.

QuNetSim is a tool for protocol development in quantum communication networks modeled based on a structure similar to current classical communication networks. The state of the quantum Internet is still very primitive and although we attempted to design QuNetSim in a way that safeguards against future physical implementations, it is still difficult to clearly foresee how a quantum Internet will be designed and implemented and which features it will and will not have. We, therefore, make no claim that a simulation implemented in QuNetSim is guaranteed to work in a future quantum Internet, but we hope that by using it one can more easily envision the potential structures of and experiment with possible quantum networks.

V. LIMITATIONS OF QUNETSIM

QuNetSim provides a high-level framework for developing quantum protocols. There are, however, some limitations in the current implementation. QuNetSim relies on existing qubit simulators (see Section VII-A), which in some cases perform well and in some cases do not. This causes QuNetSim to periodically run more slowly than desired and, because we are using full qubit simulators, where the alternative is error tracking only as it is in QuISP [11], running large scale simulations can consume much of the user's computer's resources. We have found that QuNetSim works well for smaller scale simulations using ten to fifteen hosts that are separated by a small number of network hops. QuNetSim tends to reach its limits when many entangled qubits are being generated across the network with many parallel operations. As this is an ongoing project, we will investigate performance improvements as a priority during future iterations. Moreover, QuNetSim does not aim at perfect physical realism, and so the physical properties of quantum networks are mostly neglected. One can although in principle recover physical realism by integrating QuNetSim with physical devices as described earlier.

VI. DESIGN OVERVIEW OF QUNETSIM

The aim of QuNetSim is to allow for the development of simulations that contain enough realism that applications of

IEEE Transactions on Quantum Engineering

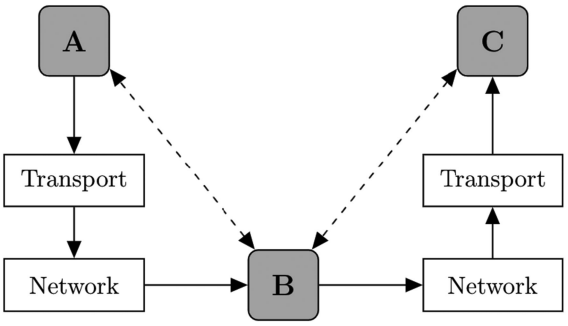

FIG. 2. Example of a communication process in QuNetSim with three hosts. In this example, there are three hosts, A, B, and C. hosts A and B are connected via a two-way channel (represented by the dashed line), as are hosts B and C. When host A executes an application that transmits information to host C, since there is no direct connection, the information must first be routed though host B. QuNetSim uses a layered approach such as the Internet. First, application data are filtered through a transport preparation layer so that the information packet is prepared for the network. From there, the transport layer packet is put into the network. The network also encodes the packet with its own header information and begins to route the packet through the network. The network packet is moved first to host B, and host B relays the data to host C to complete the transmission.

quantum networks can be developed, tested, and debugged for a proof of principle step. With this in mind, we have designed the software such that we remove the need for the large overhead of setting up new simulations and added built-in features that are potentially repeated across many simulations. Another design aspect we aim for is that a prior deep level of understanding of quantum networks and software development should not be required to use QuNetSim. To allow for as many users as possible to develop their applications, we keep the functionality at a high level such that protocols written with QuNetSim are as easy or easier to understand as the protocols written in scientific papers. QuNetSim allows users to merge various protocols, which can be easily modified and simulated or chained together to run in parallel or sequential configurations.

The general architecture layout of QuNetSim is depicted in Fig. 2. Here, there are three network nodes, which are hosts in QuNetSim. Hosts **A** and **B** are connected via a communication link as are hosts **B** and **C**. As in a classical network, hosts run such that they sit idle awaiting any incoming packets and then act when packets arrive, or optionally they can be programmed to act during idle periods. Hosts in QuNetSim run applications asynchronously and transfer quantum and classical messages to other hosts in the network. In the figure, host **A** runs an application that sends a packet to host **C**. Host **A** has no direct connection to host **C**, and therefore, its information must be routed through **B** in order to arrive at **C**. In a layered network architecture, since host **A** is running on the application layer, it should not be concerned with how the information arrives at **C**, and it is the duty of transport layer to prepare metadata of the application's action and the network layer to route the information to **C** if a route exists.

The transport layer prepares the information sent from **A** for the network by encoding necessary information in a packet header such as the sender and receiver IDs, the protocol, and the packet sequence number. As the layering of quantum networks is not specified yet, the transport layer of QuNetSim has only few responsibilities. For example, it can be used to ensure the availability of pre-established entanglement between network hosts prior to executing protocols such as dense coding or teleportation. The transport layer processes incoming and outgoing packets as a layer between the network and the hosts. Defined in the transport layer is a set of protocols so that packets are encoded and decoded properly. Once information is encoded into a packet, the transport layer moves the packet to the network. In the current state of QuNetSim, the transport layer is not configurable, other than the ability to disable the check for EPR pairs for the built-in teleportation and superdense coding features. In the next major release of QuNetSim, we aim to allow users to define their own packet structures and have a more configurable transport layer behavior.

Once the packet is added to the network the network layer routes it to the destination receiver. The path from **A** to **C** is through **B** and so a transport layer packet is encoded in a network packet and then moved through **B** to **C**. When host **B** receives a packet from **A**, since it is not the intended receiver, it relays the network packet onward. Finally, when the network packet arrives at **C** the packet is unpacked in the network layer and again in the transport layer. The payload can then be processed according to the decoded information in the header. This separation of responsibility per layer is a fundamental element of the QuNetSim design. The network of QuNetSim behaves much like the network in the Internet with some key differences. In QuNetSim, the network is composed of two separate but parallel networks, one for quantum information and one for classical information. When quantum information is sent from a host, the network routes it through the quantum links in the network as it does for classical information. Another responsibility of the network layer that differs from the classical setting is that the network layer is responsible for establishing an EPR pair between nodes that do not share a direct connection via an entanglement swapping routine. It can trigger a chain of hosts to perform an entanglement swapping so that in the end, the sender and the intended receiver will share an EPR pair.

QuNetSim does not currently go above the network layer in terms of simulation of quantum networks. As more features are developed, the code structure allows us to replace pieces that we currently omit, such as link layer functionality. We do, however, allow the user to integrate their qubit channel models, and in subsequent versions will incorporate this into the design such that these things will be easy to change.

We now discuss the software implementation of QuNetSim. The QuNetSim framework is developed in the Python programming language [17] as a software library. It uses Python's multithreading library to simulate the asynchronous networks behavior. Each host has a processing queue and

runs in a thread so that when a host performs an action the actions run in a first-in-first-out ordering. These actions are then processed in the transport layer where the header information contains sender and receiver information, along with information on how to process the payload. The processed packet is put into the network for routing. The network acts like a host where it has a packet queue, which is being monitored for changes. Once a packet is put into the queue, it is analyzed and processed.

In the network-packet can be a signal to generate entanglement between hosts prior to executing a given task. Since the swap procedure—as mentioned above—consumes a large amount of communication resources, and since this reflects on the run-time of QuNetSim, the default behavior of QuNetSim is to first check if it is possible to form a SWAP chain from sender to receiver. If it is possible to form an entanglement swap chain, then all corresponding Bell measurements and the transmission of classical messages associated with executing the entanglement swap protocol will not be performed explicitly. Instead, only the end result of the protocol (deletion of the consumed EPR pairs and establishment of an EPR pair between sender and receiver) is directly provided to the backend. This way, the execution time of the already well-known procedure is not contradicting the design goal of an efficient protocol testing. If no EPR pair is needed the payload is checked for the type of data it contains. A network in QuNetSim contains two directed graphs to represent disjoint networks for the respective data types. If the packet payload is classical data, it is routed over the classical network and if it is quantum data, it is routed via the quantum network. By default, the routing algorithm is shortest path, but it can be changed via parameter settings. We see an example of how this is done in Section VIII. Currently in QuNetSim there are simple error models available that can be turned ON as an optional feature. The error models are configured as network property and include packet loss and Pauli errors applied to Qubits en route. A more complex treatment of the link-layer will be considered in future iterations of QuNetSim.

Finally, the packet arrives at the receiver host after being routed in the network. The payload is then processed at the receiver side according to the header information. Once processed, the classical or quantum data is stored in either the classical memory of the host or in one of the two quantum memories. There are two quantum memories, as this allows users to more easily distinguish between qubits that arrive as qubits encoded by the sender or qubits specifically generated using one of the built-in entanglement generation host methods.

In summary, QuNetSim implements a layered model of component objects much like the OSI model [6]. The host and network components are implemented using threading and observing queues. The queues are monitored constantly and queue changes trigger an event. Extensive use of threading allows each task to wait without blocking the main program thread, which simulates the behavior of sending information and waiting for an acknowledgment, or expecting information to arrive for some period of time from another host.

VII. USING QUNETSIM

In this section, we introduce the key features of QuNetSim for implementing protocols. A full set of documentation is also available [18].

A foundational data structure used in QuNetSim is the Qubit. When a Qubit is created, it is created with a specified host and gets assigned a unique ID. A qubit is generated using `Qubit(host)`. Once a qubit is created by a host, logic gates can be applied to it, it can be stored, or it can be transmitted to another host. To send a qubit to another host, one can send it directly or by using teleportation. The two Qubit methods that accomplish this are as follows:

1) `send_qubit`;
2) `send_teleport`.

Hosts can easily establish entangled qubits with other hosts in QuNetsim with two host methods. Built in, hosts can generate EPR pairs with another host or a GHZ or W-state with many hosts. To do so, the following host methods are in place:

1) `send_epr`;
2) `send_ghz`;
3) `send_w`.

Hosts can send classical messages in three ways; they can send an arbitrary string over a classical connection, a binary message via superdense coding, or classically broadcast messages through the network. These are accomplished with the following host methods:

1) `send_classical`;
2) `send_superdense`;
3) `send_broadcast`.

Sending a message using superdense coding [19] requires that a quantum channel is also available between hosts, as it requires a pre-established EPR pair.

For synchronization between communicating hosts, it is sometimes beneficial to wait for acknowledgment from the receiving host. "Waiting" in QuNetSim implies blocking a line of code. Waiting for an acknowledgment is possible for all sending method, which is done by setting the flag in the methods called `await_ack`. By setting the flag to false, the host does not wait before executing the actions that follow and there is no thread blocking done. Moreover, it can be specified how long a host should wait for acknowledgments by setting the `max_ack_wait` host property. QuNetSim uses real-time for this, so this parameter is the number of seconds to wait. When `await_ack` is false, an acknowledgment is still sent, but there is no waiting. One can further specify that no acknowledgment should be sent by using the `no_ack` flag in sending methods.

Hosts can be programmed to retrieve an incoming classical or quantum message and also wait for it to arrive. When a

classical or quantum message arrives, it is stored at the host in its respective memory structure—there is a distinct memory for classical and quantum information. Hosts have the option to fetch the data from their memories so that actions can be performed on it. These methods are as follows:

1) `get_classical`;
2) `get_data_qubit`;
3) `get_epr`;
4) `get_ghz`;
5) `get_w`.

Much like awaiting acknowledgments, hosts can wait until a message or qubit arrives for a fixed amount of time before proceeding. For each "get" method, there is a parameter `wait=n` where `n` is a floating point number of seconds to wait. For example, `get_epr("Alice," wait=5)` will wait for five (real) seconds for an EPR to arrive from Alice.

In near-future quantum hardware, it is expected that quantum memories will be limited in their ability to store large numbers of qubits. QuNetSim supports limiting the number of qubits stored at a host. The number of entangled qubits and data qubits can be limited separately or a limit for the combined number of qubits can be set. The host methods for setting the limits are as follows:

1) `set_epr_memory_limit`;
2) `set_data_qubit_memory_limit`.

These parameters will enforce that no more than the set number of qubits can be stored. When the limit is reached, the qubits are lost when they arrive.

To construct a network of hosts, hosts methods are provided with for adding and removing connections. Connections in QuNetSim are unidirectional and can be either purely classical, purely quantum, or both classical and quantum. For example, if two hosts are connected by only a classical connection, then qubits cannot be transmitted between the two hosts. These host methods are as follows:

1) `add_c_connection`;
2) `add_q_connection`;
3) `add_connection`.

Connections can also be removed at run time with

1) `remove_connection`;
2) `remove_c_connection`;
3) `remove_q_connection`.

Hosts have the ability to eavesdrop on communications. This is accomplished by first setting the following host properties to `true`:

1) `c_relay_sniffing`;
2) `q_relay_sniffing`.

From there, a custom Python function can be ran to manipulate the payload of the relaying packets. To do this, functions are set to the host properties:

1) `c_relay_sniffing_fn`;
2) `q_relay_sniffing_fn`.

These features allow hosts to intercept the qubits that pass through them en route and manipulate them by measuring them or performing a unitary operation on them before relaying it onward. With classical messages, the hosts can manipulate the classical payloads, changing the messages for example.

Host are initialized in an "OFF" state, and so to start hosts one uses the `start` host method. Once started, hosts can run custom protocols using the `run_protocol` method taking any Python written function as a parameter. We will see examples of this in the following section. A running protocol can be blocking or nonblocking. The Python thread object containing the running protocols are provided and can be handled by the user as desired.

Building the network topology is also a critical part of every simulation. QuNetSim uses a `Network` singleton object to abstract the classical and quantum networks. Once the network topology is established between the hosts, hosts are added to the network using the network method `add_host`. The network builds a graph structure using the connections of the host to be used for routing. As we will see in the following section, this involves adding hosts to the network and then calling the Network method `start`.

Networks can be programmed to run custom routing functions for the classical and quantum messages separately. By default, the network uses shortest-path routing for packets in the network, but by setting the following network properties:

1) `quantum_routing_algo`;
2) `classical_routing_algo`;

the network will use the routing algorithm specified by the user. The custom routing algorithm's output need just be an ordered list, which is the path to the receiver. The network can reroute at each hop as well by setting the `use_hop_by_hop` flag accordingly.

Overall, with these features one can already develop a variety of protocols with varying network topologies. Furthermore, by programming eavesdropping hosts, one can easily test protocols against malicious third parties, programming various attacks, as is the norm in the development of QKD protocols.

A. QUANTUM BACKENDS

QuNetSim relies on open-source qubit simulators that we use to simulate the physical qubits in the network. At the current stage, we are using three qubit simulators that each have their own benefits: CQC/SimulaQron [20], ProjectQ [21], EQSN [22], and recently QuTiP [23]. EQSN is the default backend and it is created by the TQSD group. Users are free to change the backend of QuNetSim to use different qubit simulators, and we also explain how new backends can be easily added in our full documentation [18].

TABLE I Benchmark of the Various Backends With Different Networking Tasks

Backend	Teleportation	Superdense	GHZ State
ProjectQ	102 ± 30	82 ± 15	N/A
EQSN	283 ± 33	296 ± 72	2765 ± 245
CQC	301 ± 75	533 ± 70*	215 ± 17*
QuTiP	111 ± 28	92 ± 2	351 ± 40

Various protocols are ran over networks with ten hops between the end nodes. Listed is the average duration in milliseconds to run the specified protocol 10 times repeating the procedure 30 times over to collect the samples. The GHZ state is distributed to nine other hosts from the first host in the chain. The benchmarking is done using an average laptop with a 2.7 GHz Dual-Core Intel Core i5 processor with Python version 3.6.4. *represents tested with just two nodes to work with the benchmarking tool.

Other than SimulaQron/CQC, using these quantum simulators independent of QuNetSim removes the ability to easily mimic transporting of quantum information in a network setting. For example, one can indeed implement a simulation of quantum teleportation in ProjectQ. With only a quantum circuit alone, which ProjectQ uses for simulation input, one may not immediately know that the circuit represents a multiparty network protocol. In QuNetSim, one programs multiple entities independently, clarifying the network aspects. With the circuit model, there is also no aspect of waiting and eavesdropping, both of which are available in QuNetSim. SimulaQron most closely resembles QuNetSim, but QuNetSim adds a lair of synchronization, such as waiting for arrivals and acknowledgment to the hosts, on top of SimulaQron and adds addition network features as mentioned in the earlier section.

Each backend has advantages and disadvantages and we give a general benchmarking overview in Table I. The advantage of using ProjectQ is that it is the fastest in terms of run-time of the three implemented backends. We have found, however that it tends to slow when there is a high volume of qubit entanglement and less robust when protocols run for longer durations. ProjectQ is not designed to run in a multithreaded environment and so we observed run-time errors when many measurements are made on qubits in multiple threads. We predict that there could be some additional thread synchronization done here to prevent these errors. Generally, these are not major issues and can be avoided by slowing the simulation slightly using the delaying mechanisms built-in to QuNetSim. EQSN and CQC both work well in terms of threading and processing speed, but are indeed slower in speed than ProjectQ. They are generally more reliable under many qubit operations and tend not to be affected as quickly when many entangled states are being generated. QuTiP currently shows the best performance. It is optimized in terms of representing a density matrix with as little data as possible as well as not using a single matrix to represent the entire quantum system. This allows for generating many qubits, even with bipartite entanglement, efficiently. Overall, the user can use the same code for their network simulation and easily change the underlying quantum simulation backend to find the one that best suits their needs. In future iterations, we aim to integrate more qubit backends to allow for an even more diverse set of simulation possibilities.

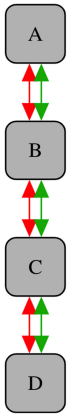

FIG. 3. Network depiction for example A.

VIII. EXAMPLE SIMULATIONS

In this section, we cover three examples using QuNetSim. The first example is an explanatory example that demonstrates establishing a network and sending qubits between hosts in the network. The following example is of establishing quantum entanglement between two parties where the knowledge of which hosts share the entangled pair is anonymous to all but the two hosts. The example is based on the protocol in [24]. In the final example, we establish a network and define a custom routing mechanism. In this example, hosts are constantly generating entanglement when possible, and super-dense encoded messages are then transmitted between parties using the route with the most established entanglement. This routing concept is further established in [25].

A. SENDING DATA QUBITS

We demonstrate here the simple task of sending qubits that have been encoded with information, or as they are called in QuNetSim, "data" qubits. We send the qubits from host **A** to host **D** over the network in Fig. 3.

```
# Network is a singleton
network = Network.get_instance()
# Start the network with the nodes defined above
network.start()

# Define the Hosts
host_A = Host('A')
# Define the Host's connections
host_A.add_connection('B')
# Start the Host
host_A.start()

host_B = Host('B')
host_B.add_connections(['A', 'E'])
host_B.start()

host_E = Host('E')
host_E.add_connections(['B', 'D'])
host_E.start()

host_D = Host('D')
host_D.add_connection('E')
host_D.start()

# Add the Hosts to the network to build the network
# graph
network.add_hosts([host_A, host_B, host_E, host_D])
```

In the following, we want to generate the protocols for **A** and **D** to run. Protocols are the functionality of a host. Protocols are very flexible with the only exception that the

protocol function must take the host as the first parameter. Following is the sample protocol and code to launch the protocol for a host. In this example, host **A** is sending five data qubits to **D**.

```
def sender(host, receiver):
 """
 Sends 5 qubits to Host *receiver*.
 Args:
   host (Host): The Host object running the protocol
   receiver (str): The name of the receiver
 """
 for i in range(5):
   # The Host creates a qubit
   qubit = Qubit(host)
   # Perform a Hadamard operation on the qubit
   qubit.H()
   # The Host sends the qubit to the receiver
   # and awaits an ACK from the receiver that
   # the qubit arrived for some fixed amount of time.
   ack_arrived = host.send_qubit(receiver, qubit,
                                 await_ack=True)
   if ack_arrived:
       print('Qubit sent successfully.')
   else:
       print('Qubit did not transmit.')
```

A protocol for receiving qubits must also be written.

```
def receiver(host, sender):
 """
 Sends 5 qubits to Host *receiver*.
 Args:
   host (Host): The Host object running the protocol
   receiver (str): The name of the sender
 """
 for i in range(5):
   # The Host awaits a data qubit for 10 seconds maximum
   qubit = host.get_data_qubit(sender, wait=10)
   # If the qubit arrived, measure it
   if qubit is not None:
       m = qubit.measure()
       print("%s received qubit in state %d"
                     % (host.host_id, m))
   else:
       print("Qubit did not arrive.")
```

To run the protocols, we have the following lines of code.

```
# A runs the sender protocol with D
host_A.run_protocol(sender, ('D',))
# D runs the receiver protocol A
host_D.run_protocol(receiver, ('A',))
```

In summary, with these code snippets, we can simulate the transmission of five qubits from **A** to **D** over the network in Fig. 3. Of course, this simple example is intended to give an gentle introduction into how QuNetSim works. In the following examples, we develop more complex protocol simulations to see further the simplicity of using QuNetSim to write quantum network simulations.

B. GHZ-BASED QUANTUM ANONYMOUS DISTRIBUTION

In this example, we will demonstrate how to simulate an instance of GHZ-based quantum anonymous transmission [24]. The goal of the protocol is to hide the creation of an entangled pair between two parties. This protocol implementation demonstrates the simplicity of translating a protocol from a high-level mathematical syntax into simulation using QuNetSim. The protocol involves establishing GHZ states amongst n parties as well as broadcasting measurement outcomes. These types of tasks would involve a relatively high level of software synchronization in order to program a simulation from scratch. Here, we demonstrate that in QuNetSim, synchronization logic is programmatically kept at a high level.

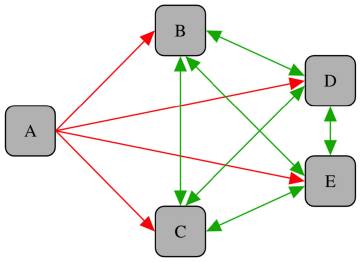

FIG. 4. Network depiction for example B.

As a first step, as always, we generate a network. Following is the code to generate such a network, which is depicted in Fig. 4.

```
network = Network.get_instance()
network.start()

host_A = Host('A')
host_A.add_connections(['B', 'C', 'D', 'E'])
host_A.start()

host_B = Host('B')
host_B.add_c_connections(['C', 'D', 'E'])
host_B.start()

host_C = Host('C')
host_C.add_c_connections(['B', 'D', 'E'])
host_C.start()

host_D = Host('D')
host_D.add_c_connections(['B', 'C', 'E'])
host_D.start()

host_E = Host('E')
host_E.add_c_connections(['B', 'C', 'D'])
host_E.start()

network.add_hosts([host_A, host_B, host_C,
                   host_D, host_E])
```

The following step is to write the behavior of the GHZ distributor, which in this example is host **A**. QuNetSim provides a function for distributing GHZ states so the function `distribute` simply takes the distributing host as the first parameter and the list of receiving nodes as the second. One notices that the flag `distribute` has been set to true in the `send_ghz` method call. This tells the sending host that it should not keep part of the GHZ state, rather, it should generate a GHZ state amongst the list given and send it to the parties in the node list, keeping no part of the state for itself.

```
def distribute(host, nodes):
   """
   Args:
       host (Host): The Host running the protocol
       nodes (list): The list of nodes to distribute the
                     GHZ to
   """
   # distribute=True => don't keep part of the GHZ
   host.send_ghz(nodes, distribute=True)
```

The next type of behavior we would like to simulate is that of a node in the group that is not attempting to establish an EPR pair. In the anonymous entanglement protocol, the behavior of such a node is to simply receive a piece of a GHZ state, perform a Hadamard operation on the received qubit, measure it, and broadcast to the remaining participating parties the outcome of a measurement in the computational basis. In the following, we see how to accomplish this.

In the `node` function, the first parameter is, as always, the host that is performing the protocol. The second is the ID of the node distribution the GHZ state, which in this example, is host **A**. The host fetches its GHZ state where, if it is not available at the time, they will wait ten seconds for it, accomplished by setting the `wait` parameter. If they have not received part of the GHZ state, then the protocol has failed, otherwise they simply perform the Hadamard operation on the received qubit, measures it, and broadcasts the message to the network. QuNetSim includes the task of broadcasting as a built-in function, and therefore, the task of sending classical messages to the whole network is simplified to one line of code.

```
def node(host, distributor):
    """
    Args:
        host (Host): The Host running the protocol
        distributor (str): The ID of the distributor
                           for GHZ states
    """
    q = host.get_ghz(distributor, wait=10)
    if q is None:
        print('failed')
        return
    q.H()
    m = q.measure()
    host.send_broadcast(str(m))
```

We implement next the behavior of the party in the protocol acting as one end of the EPR link that we label the *sender*. Here, we take three parameters (other than the host running the protocol), the ID of the distributor, the receiver that is the holder of the other half of the EPR pair, and an agreed upon ID for the EPR pair that will be generated. In QuNetSim, qubits have IDs for easier synchronization between parties. For EPR pairs and GHZ states, qubits share and ID, that is, the collection of qubits would all have the same ID. This is done so that when parties share many EPR pairs, they can easily synchronize their joint operations. The `sender` protocol is the following: first they receive part of a GHZ state, they select a random bit and then broadcast the message so that they appear as just any other node. They then manipulate their part of their part of the GHZ state according to what the random bit was. If the bit was 1, then a Z gate is applied. The sending party can then add the qubit as an EPR pair shared with the receiver. This EPR pair can then be used as if the sender and receiver established an EPR directly.

```
def sender(host, distributor, receiver, epr_id):
    """
    Args:
        host (Host): The Host running the protocol
        distributor (str): The ID of the distributor
                           for GHZ states
        receiver (str): ID to teleport the qubit to after
                        EPR is established
        epr_id (str): ID for the EPR pair established
                      ahead of time
    """
    q = host.get_ghz(distributor, wait=10)
    b = random.choice(['0', '1'])
    host.send_broadcast(b)
    if b == '1':
        q.Z()

    host.add_epr(receiver, q, q_id=epr_id)
    qubit_to_send = Qubit(host)
    host.send_teleport(r, qubit_to_send)
    host.empty_classical()
```

Finally, we establish the behavior of the *receiver*. The receiver here behaves as follows: First, in order to mask their behavior they randomly choose a bit and broadcast it to the network. Once complete, they await the remainder of the broadcast messages. In QuNetSim, classical messages are stored as a list in the `classical` field. Since there are three other parties, other than the receiver themself, they await the other three messages. Once they arrive, the receiver computes a global parity operation by taking the XOR of all received bits along with their own random choice. With this, the receiver can apply a controlled Z gate, which establishes the EPR pair with the correct sender. They simply add the EPR pair and complete the protocol.

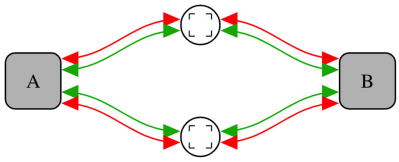

FIG. 5. **Network depiction for example C.**

```
def receiver(host, distributor, sender, epr_id):
    q = host.get_ghz(distributor, wait=10)
    b = random.choice(['0', '1'])
    host.send_broadcast(b)

    messages = []
    # Await broadcast messages from all parties
    while len(messages) < 3:
        messages = host.classical

    parity = int(b)
    for m in messages:
        if m.sender != s:
            parity = parity ^ int(m.content)
    if parity == 1:
        q.Z()

    # Established secret EPR, add it
    host.add_epr(sender, q, q_id=epr_id)

    # Await a teleportation from the anonymous sender
    q = host.get_data_qubit(s, wait=10)
```

The last step of writing a QuNetSim simulation is to run the protocols for each desired host. In the following, we let host **A** act as the GHZ state distributor, **B** and **C** are neutral parties running the `node` behavior, **D** acts as the sender and **E** acts as the receiver. The following code initiates the simulation.

```
epr_id = '12345'
host_A.run_protocol(distribute, (['B', 'C', 'D', 'E'],))
host_B.run_protocol(node, ('A',))
host_C.run_protocol(node, ('A',))
host_D.run_protocol(sender, ('A', 'E', epr_id))
host_E.run_protocol(receiver, ('A', 'D', epr_id))
```

C. ROUTING WITH ENTANGLEMENT

In this example, we demonstrate how one can use QuNetSim to test a custom routing algorithm. We consider the network shown in Fig. 5. The example here uses the approach of [26] for choosing the route and using dense coding for classical message transmission. For this example, we assume the network is using entanglement resources to transfer classical information via superdense coding from host **A** to host **B**. The sending and receiving parties must first establish an EPR pair to send messages via superdense coding. The sender performs a specific set of operations on its half of the EPR pair

and then transmits it through the network. When the receiver received the qubit, it performs a specific set of operations such that it recovers two bits of classical information. What is important here is that **A** and **B** are separated by one hop. In order to share an EPR pair, an entanglement swap routing has to be made.

The strategy for this routing algorithm is to first build a graph of the entanglement shared amongst the hosts in the network. The strategy, since superdense coding consumes entanglement pairs, will then be to route information through the path that contains the most entanglement. In this example, we show how this can be accomplished. As always, we first generate the network topology.

```
nodes = ['A', 'node_1', 'node_2', 'B']
network.use_hop_by_hop = False
network.use_ent_swap = True
network.set_delay = 0.1
network.start(nodes)

A = Host('A')
A.add_connections(['node_1', 'node_2'])
A.start()

node_1 = Host('node_1')
node_1.add_connections(['A', 'B'])
node_1.start()

node_2 = Host('node_2')
node_2.add_connections(['A', 'B'])
node_2.start()

B = Host('B')
B.add_connections(['node_1', 'node_2'])
B.start()

network.add_hosts([A, node_1, node_2, B])
```

```
def generate_entanglement(host):
 """
 Generate entanglement if the Host is idle.
 """
 while True:
   # Check if the Host is not processing
   if host.is_idle():
     for connection in host.quantum_connections:
        host.send_epr(connection)
```

```
def routing_algorithm(network_graph, source, target):
 """
 Entanglement based routing function.

 Args:
    network_graph (networkx.DiGraph): The directed graph
                                      representation of
                                      the network.
    source (str): The sender ID
    target (str): The receiver ID
 Returns:
    (list): The route ordered by the steps in the route.
 """

 # Generate entanglement network
 entanglement_network = nx.DiGraph()
 nodes = network_graph.nodes()
 # A relatively large number
 inf = 1000000
 for node in nodes:
   host = network.get_host(node)
   for connection in host.quantum_connections:
     num_epr_pairs = len(host.get_epr_pairs(connection))
     if num_epr_pairs == 0:
         entanglement_network
           .add_edge(host.host_id,
                     connection,
                     weight=inf)
     else:
         entanglement_network
             .add_edge(host.host_id,
                       connection,
                       weight=1. / num_epr_pairs)

 try:
   return nx.shortest_path(entanglement_network,
                           source,
                           target,
                           weight='weight')
 except Exception as e:
   print('Error getting route.')
```

We can now begin to simulate the network using this configuration. In this simulation, the nodes in the middle are sending entanglement to the other nodes as often as they can, establishing the most EPR pairs while they are free to do so. We start them on this process as so:

```
node_1.run_protocol(generate_entanglement)
node_2.run_protocol(generate_entanglement)
```

Now we tell the network to use a different routing algorithm for the quantum information in the network.

```
network.quantum_routing_algo = routing_algorithm
```

Finally, we trigger host **A** to begin transmitting 100 messages via superdense coding to host **B**.

```
choices = ['00', '11', '10', '01']
for _ in range(100):
   m = random.choice(choices)
   A.send_superdense('B', m, await_ack=True)
```

IX. CONCLUSION AND OUTLOOK

Overall, we have introduced the open-source, Python-based, real-time quantum network simulation framework QuNetSim. We reviewed the design choices and architecture of QuNetSim and the built-in features that allow for complex quantum networking protocols to be developed in a straightforward way. We have shown that the high-level functionality of QuNetSim allows beginners to easily learn about quantum networks as well as quantum network protocol developers to quickly develop and test their protocols, making QuNetSim a good choice for teaching and research.

The current state of the quantum networking field is at the near beginning in regards to which applications will be implemented in reality, the network layer structure, the standardization of protocols, and more. QuNetSim uses a minimal set of assumptions as to allow for a maximization of variation and configuration. QuNetSim's main focus is quantum network application development, but that is not the end of its capabilities. QuNetSim being a Python library allows for other distributed quantum network applications to be built on top of it, as we have already seen with the Interlin-q project [16]. Moreover, as a real-time simulator, it allows for future emulation features and for integration of "lab-in-a-loop" behavior.

The future of QuNetSim is to maintain its ease-of-use design while integrating the features needed to be a first-choice tool for those in the quantum network field. QuNetSim's target user base and use cases will continue to be students for learning, lectures as a teaching instruments, quantum network engineers as a protocol development platform, and quantum hardware developers for test-bed development. As an open-source tool, building a strong collaborative developer network is also an important goal. Quantum networks are an inevitable technology and with them comes many open questions. QuNetSim aims to be a tool to help with answering these questions.

ACKNOWLEDGMENT

The authors would like to thank all external contributors to QuNetSim, especially A. Muzzi for implementing

W-state distribution and Anirban Majumder for implementing noise models in channels. The work of Stephen Diadamo was supported by the Unitary Fund under the QuNetSim project.

REFERENCES

[1] H. J. Kimble, "The quantum Internet," *Nature*, vol. 453, no. 7198, pp. 1023–1030, 2008, doi: 10.1038/nature07127.
[2] S. Wehner, D. Elkouss, and R. Hanson, "Quantum Internet: A vision for the road ahead," *Science*, vol. 362, no. 6412, 2018, Art. no. eaam9288, doi: 10.1126/science.aam9288.
[3] Q. O. S. Foundation, "List of open quantum projects," 2020. [Online]. Available: https://qosf.org/project_list/
[4] G. F. Riley and T. R. Henderson, "The NS-3 network simulator," in *Modeling Tools Network Simulation*. Berlin, Germany: Springer, 2010, pp. 15–34, doi: 10.1007/978-3-642-12331-3.
[5] R. R. Fontes, S. Afzal, S. H. Brito, M. A. Santos, and C. E. Rothenberg, "Mininet-WiFi: Emulating software-defined wireless networks," in *Proc. 11th Int. Conf. Netw. Service Manage.*, 2015, pp. 384–389, doi: 10.1109/CNSM.2015.7367387.
[6] H. Zimmermann, "OSI reference model—The ISO model of architecture for open systems interconnection," *IEEE Trans. Commun.*, vol. 28, no. 4, pp. 425–432, Aug. 1980, doi: 10.1109/TCOM.1980.1094702.
[7] A. Pirker and W. Dür, "A quantum network stack and protocols for reliable entanglement-based networks," *New J. Phys.*, vol. 21, no. 3, 2019, Art. no. 033003, doi: 10.1088/1367-2630/ab05f7.
[8] A. Dahlberg and S. Wehner, "SimulaQron—A simulator for developing quantum Internet software," *Quantum Sci. Technol.*, vol. 4, no. 1, 2018, Art. no. 015001, doi: 10.1088/2058-9565/aad56e.
[9] T. Coopmans *et al.*, "Netsquid, a discrete-event simulation platform for quantum networks," 2020, *arXiv:2010.12535*.
[10] B. Bartlett, "A distributed simulation framework for quantum networks and channels," 2018, *arXiv:1808.07047*.
[11] T. Matsuo *et al.*, "QuISP—Quantum Internet simulation package," 2020. [Online]. Available: https://aqua.sfc.wide.ad.jp/quisp_website/
[12] X. Wu *et al.*, "Sequence: A customizable discrete-event simulator of quantum networks," 2020, *arXiv:2009.12000*.
[13] M. Mehic, O. Maurhart, S. Rass, and M. Voznak, "Implementation of quantum key distribution network simulation module in the network simulator NS-3," *Quantum Inf. Process.*, vol. 16, no. 10, pp. 1–23, 2017, doi: 10.1007/s11128-017-1702-z.
[14] R. Chatterjee, K. Joarder, S. Chatterjee, B. C. Sanders, and U. Sinha, "QKDSIM, a simulation toolkit for quantum key distribution including imperfections: Performance analysis and demonstration of the b92 protocol using heralded photons," *Phys. Rev. Appl.*, vol. 14, Aug. 2020, Art. no. 024036, doi: 10.1103/PhysRevApplied.14.024036
[15] T. Matsuo, "Simulation of a dynamic, ruleset-based quantum network," 2019, *arXiv:1908.10758*.
[16] R. Parekh and S. DiAdamo, "Interlin-Q: A quantum interconnect simulator for distributed quantum algorithms," 2021. [Online]. Available: https://github.com/Interlin-q/Interlin-q
[17] G. Van Rossum, and F. L. Drake, "Python 3 Reference Manual," Scotts Valley, CA, US, CreateSpace, 2009, isbn: 1441412697.
[18] S. DiAdamo, J. Nötzel, B. Zanger, and M. Mert Beşe, "QuNetSim: A software framework for quantum networks," 2020. [Online]. Available: https://tqsd.github.io/QuNetSim
[19] C. H. Bennett and S. J. Wiesner, "Communication via one- and two-particle operators on einstein-podolsky-rosen states," *Phys. Rev. Lett.*, vol. 69, pp. 2881–2884, Nov. 1992, doi: 10.1103/PhysRevLett.69.2881
[20] A. Dahlberg *et al.*, "A link layer protocol for quantum networks," in *Proc. ACM Special Int. Group Data Commun.*, 2019, pp. 159–173, doi: 10.1145/3341302.3342070.
[21] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, 2018, doi: 10.22331/q-2018-01-31-49.
[22] B. Zanger and S. DiAdamo, "EQSN: Effective Quantum Simulator for Networks," 2020. [Online]. Available: https://github.com/tqsd/EQSN_python
[23] J. R. Johansson, P. D. Nation, and F. Nori, "QuTiP: An open-source python framework for the dynamics of open quantum systems," *Comput. Phys. Commun.*, vol. 183, no. 8, pp. 1760–1772, 2012, doi: 10.1016/j.cpc.2012.02.021.
[24] M. Christandl and S. Wehner, "Quantum anonymous transmissions," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2005, pp. 217–235, doi: 10.1007/11593447_12.
[25] J. Nötzel and S. DiAdamo, "Entanglement-enhanced communication networks," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2020, pp. 242–248, doi: 10.1109/QCE49297.2020.00038.
[26] J. Nötzel and S. DiAdamo, "Entanglement-enhanced communication networks," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2020, pp. 242–248, doi: 10.1109/QCE49297.2020.00038.

Stephen Diadamo received the B.Sc. (Hons.) degree in computer science and software engineering from the University of Toronto, Toronto, ON, Canada, in 2014 and the M.Sc. degree in mathematics from the Technical University of Munich, Munich, Germany, in 2018. He is currently working toward the Ph.D. degree in electrical engineering with the Technical University of Munich, Munich, Germany.

His research interests include quantum networks, quantum information theory, and distributed quantum computing.

Mr. Diadamo has been awarded a microgrant from the Unitary Fund program in order to assist in the development of this work.

Janis Nötzel received the Dipl. Phys. degree in physics from the Technische Universität Berlin, Berlin, Germany, in 2007 and the Ph.D. degree (Dr. rer. nat) in pure Mathematics from Technische Universität München, München, Germany, in 2012.

From 2008 to 2010, he was a Research Assistant with the Technische Universität Berlin, and from 2011 to 2015, with Technische Universität München. In 2015 and 2016, he was a DFG Research Fellow with the Universitat Autònoma de Barcelona, Bellaterra, Spain. From September 2016 to November 2018, he led a research transfer with the 5G Lab, Technische Universität Dresden, Dresden, Germany, resulting in the spin-off ZentiConnect. In December 2018, he became an Emmy-Noether Research Group Leader with the Technische Universität München. His research interests include quantum information processing, classical information theory, and signal processing algorithms.

Benjamin Zanger received the B.Sc. degree in electrical engineering and the B.Sc. degree in computer science, with a focus on scientific computing and quantum computing, in 2018 and 2020, respectively, from the Technical University of Munich, Munich, Germany, where he is currently working toward the M.Sc. degree in electrical engineering.

Before joining the Chair of Theoretical Information Technology, he was a working student in embedded software development for over two years.

Mehmet Mert Beşe received the B.Sc. degree in electrical engineering from Middle East Technical University, Ankara, Turkey, in 2018, with a focus on mobile communications, and the M.Sc. degree in communications engineering from the Technical University of Munich, Munich, Germany, in 2020.

Before joining the Chair of Theoretical Information Technology, he was a working student for Airbus.