NetQASM—a low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet

Dahlberg, E.A.; van der Vecht, B.; Delle Donne, C.; Skrzypczyk, M.D.; te Raa, I.; Kozlowski, W.; Wehner, S.D.C.

DOI
10.1088/2058-9565/ac753f

Publication date
2022

Document Version
Final published version

Published in
Quantum Science and Technology

Quantum Science and Technology

PAPER • OPEN ACCESS

NetQASM—a low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet

To cite this article: Axel Dahlberg *et al* 2022 *Quantum Sci. Technol.* **7** 035023

View the article online for updates and enhancements.

Quantum Science and Technology

OPEN ACCESS

RECEIVED
17 December 2021

REVISED
5 May 2022

ACCEPTED FOR PUBLICATION
1 June 2022

PUBLISHED
20 June 2022

PAPER

NetQASM—a low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet

Axel Dahlberg[1,2,3], Bart van der Vecht[1,2,3,*], Carlo Delle Donne[1,2], Matthew Skrzypczyk[1,2], Ingmar te Raa[1,2], Wojciech Kozlowski[1,2] and Stephanie Wehner[1,2,*]

[1] QuTech, Lorentzweg 1, 2628 CJ Delft, The Netherlands
[2] Kavli Institute of Nanoscience, Delft, The Netherlands
[*] Authors to whom any correspondence should be addressed.
[3] These authors contributed equally.

E-mail: b.vandervecht@tudelft.nl and s.d.c.wehner@tudelft.nl

Keywords: quantum network, quantum internet, instruction set architecture, software stack, remote entanglement, hybrid program

Abstract

We introduce NetQASM, a low-level instruction set architecture for quantum internet applications. NetQASM is a universal, platform-independent and extendable instruction set with support for local quantum gates, powerful classical logic and quantum networking operations for remote entanglement generation. Furthermore, NetQASM allows for close integration of classical logic and communication at the application layer with quantum operations at the physical layer. This enables quantum network applications to be programmed in high-level platform-independent software, which is not possible using any other QASM variants. We implement NetQASM in a series of tools to write, parse, encode and run NetQASM code, which are available online. Our tools include a higher-level software development kit (SDK) in Python, which allows an easy way of programming applications for a quantum internet. Our SDK can be used at home by making use of our existing quantum simulators, NetSquid and SimulaQron, and will also provide a public interface to hardware released on a future iteration of Quantum Network Explorer.

1. Introduction

Quantum mechanics shows that if one is able to communicate quantum information between nodes in a network, one is able to achieve certain tasks which are impossible using only classical communication. There are many applications [1] where a *quantum network* has advantage over a *classical* (*non-quantum*) *network*, either by (1) enabling something that is theoretically impossible in a classical network, such as the establishment of an unconditionally secure key [2] and secure blind quantum computing [3] or (2) allowing something to be done faster or more efficiently such as exponential savings in communication [4] and extending the baseline of telescopes [5]. In recent years, many experiments have been conducted to show that a quantum network is not only a theoretical concept, and indeed advancements have been made to implement such a quantum network on various hardware platforms [6–12]. However, these experiments alone do not yet make a quantum network *programmable*, since the program logic was hard-coded into the experimental hardware ahead of time[4].

Before considering how to program quantum network applications, let us first briefly sketch the system our applications are run on. Abstractly, quantum networks consist of *nodes* that are connected by *channels* (figure 3). Classical channels enable classical communication between nodes, while quantum channels are used for *entanglement* generation between nodes. So-called *end-nodes* may contain *quantum processors* that can run arbitrary (quantum) programs. They have access to a quantum memory consisting of qubits, on which they can perform operations, including quantum computations. Some of these qubits may be used

[4] There have been examples of experiments with some simple logic but only with a very limited number of pre-loaded decision-branches.

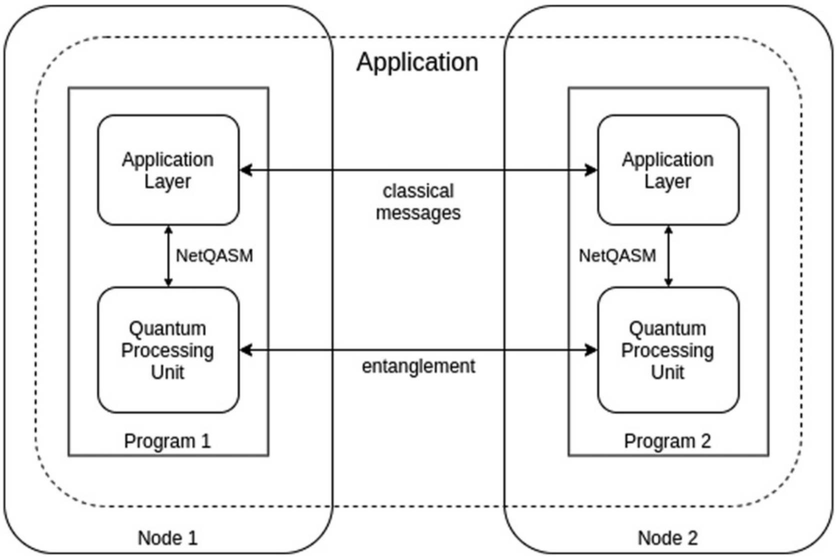


Figure 1. A quantum network application consists of a program for each of the nodes involved in the application. Each program is locally executed by the node. Program execution on each node is split into execution in an application layer, which can send and receive classical messages, and a quantum processor, which can create entanglement with another node. The communication between nodes can hence be both classical and quantum. Communication instructions need to be matched by corresponding instructions in the other program. There is no global actor overseeing execution of each of the programs, and the nodes may be physically far apart.

for establishing an entangled quantum state with a remote node. An end-node also possesses a classical processor and a classical memory. Furthermore, an end-node can send and receive classical messages to and from other end-nodes in the network. A network of quantum networks may be a called a *quantum internet*.

Quantum (network) processors differ from classical processors in a number of ways. Firstly, quantum memory has limited lifetime, meaning that its quality degrades over time. For example, quantum memories based on nitrogen-vacancy (NV) centers in diamond have impressively been optimized to achieve lifetimes in the order of seconds [13]; however, this is still very short compared to classical memories, which generally do not have a limited lifetime at all. Therefore, the quality of program execution is time-sensitive. Secondly, physical devices are prone to inaccuracies which lead to decreased quality of (quantum) computation. For example, applying an operation (like a gate) on a qubit affects that qubit's quality. We note that the two challenges mentioned so far are also inherent to non-network quantum processors. Quantum *network* processors have additional challenges: (1) the processor may have to act as a local computation unit and a network interface at the same time; for example, in NV centers, an electron spin qubit is used for generating entanglement with a remote node but is also needed to do local two-qubit gates, (2) remote-entanglement operations may not have a fixed time in which they complete, which makes scheduling and optimization more difficult.

Quantum network *applications*, also called *protocols*, are multi-partite programs that involve entanglement generation and classical communication between different end-nodes, as well as local computation. Examples include quantum key distribution (QKD) [2, 14], leader election protocols [15, 16], and blind quantum computation (BQC) [1]. Such applications are split into distinct *programs* each of which runs on a separate end-node. The programs consist of both local operations (classical and quantum) and network operations (classical and quantum), see figure 1. That is, the programs communicate either by passing classical messages, or by establishing quantum entanglement. For example, BQC involves a *client* node and a *server* node, both of which run their own program. Their joint execution looks roughly as follows: (1) the client and server engage in remote entanglement generation such that the server's quantum memory ends up being in a certain state, (2) the client sends instructions to the server in the form of a classical message, (3) the server performs a measurement-based computation on its own quantum memory based on the client's instructions, (4) the server sends measurement results back to the client, (5) the client sends new instructions based on the measurement results, (6) repeat steps (3) to (5) until the client obtains its desired result.

The example above illustrates that quantum network programs consist of different types of operations. Indeed, program code consists of *classical code*, containing local classical operations and classical communication with other nodes, and *quantum code*, which are operations on quantum memory (such as *gates*) and remote entanglement generation. Blocks of these types of code may depend on each other in multiple ways, as depicted in figure 2. Programs with mixed classical and quantum operations have also

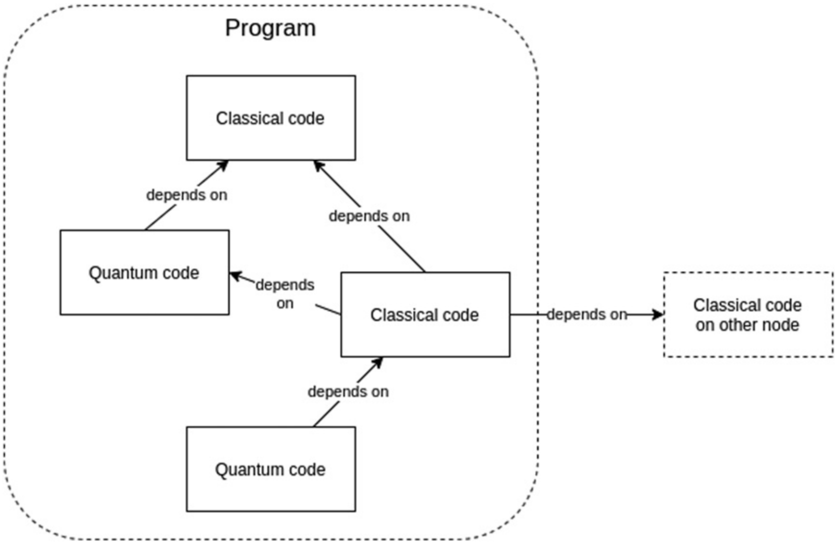


Figure 2. A program on a single node consists of different blocks of code, which can be quantum (pure quantum instructions with classical control in between), or classical (no quantum operations at all). These blocks may depend on each other in various ways. For example, the outcome of a measurement happening in one of the quantum blocks may be used in a calculation performed in one of the classical blocks. Blocks may also depend on other nodes. For instance, the value of a message coming from another node can influence the branch taken in one of the classical blocks.

been called *dynamic quantum circuits* [17, 18], but these do not cover the networking dimension found in programs we consider here, such as the dependency on remote information and entanglement generation operations.

Due to the nature of quantum network programs, execution may have to *wait* for some time. For example, the program needs to wait until another node sends a classical message, or until remote entanglement has been established. Therefore, it makes sense to run multiple (independent) quantum network programs on a node at the same time (interleaved), so that processor idle times can be filled by execution of other programs. This is something that typically does not happen on local quantum computers, and therefore introduces new challenges.

Quantum network applications may be programmed by a single actor. For example, a developer may program a QKD application in the form of a two programs, and distribute these two programs to two end-nodes in the network. Alternatively, a single-node quantum network program may be developed separately from other programs, possibly not knowing how these other programs are implemented. For example, a BQC service provider could have already implemented the server-side program of a specific BQC protocol. A client may then write the client-side of this protocol, without having control over the server-side implementation.

The aim of this work is to propose a way to program quantum network programs and execute them on the end-nodes of a quantum network.

1.1. Contribution

In this work we introduce an abstract model—including a quantum network processing unit (QNPU)—for end-nodes in a quantum network, which we define in section 2. We then propose NetQASM, an instruction set architecture that can be used to run arbitrary programs (of the form described in figure 2) on end-nodes, as long as the end-nodes realize the model including the QNPU.

NetQASM consists of a specification of a low-level assembly-like language to express the quantum parts of quantum network program code. It also specifies how the application layer should interact with the QNPU and how the assembly language can be used to execute (network) quantum code. This is not possible using other QASM languages.

The NetQASM language is extendible using the concept of *flavors*. The core language definition consists of a common set of instructions that are shared by all flavors. This common set contains classical instructions for control-flow and classical memory operations. This allows the realization of low-level control logic close to the quantum hardware; for example, to perform branching based on a measurement outcome. Quantum-specific instructions are bundled in flavors. We introduce a *vanilla* flavor containing universal platform-independent quantum gates. Using this flavor of the NetQASM language enables the platform-independent description of quantum network programs. Platform-*specific* flavors may be created to have quantum operations that are native and optimized for a specific hardware platform. As an example, we show a flavor tailored to the NV hardware, a promising platform for quantum network end-nodes [19, 20].

In our model, application-specific classical communication only happens at the application layer (figure 1). In particular, this means that NetQASM contains no provision for classical communication with the remote node. We remark that of course, classical control communication may be used by the QNPU to realize the services of the quantum network stack accessed through NetQASM.

We note that NetQASM is used for representing and running code that runs on a single node in a quantum network. Synchronization between the (NetQASM) programs of multiple nodes is the responsibility of the programmer. For example, in a client-server application, if the client code contains a ‘receive classical message’ operation, it is the responsibility of the server node that its program code contains a ‘send classical message’ operation at the right moment. The same holds for instructions for creating remote entanglement. In terms of precise timing, which is needed for entanglement generation, it is the QNPU that is responsible to communicate and synchronize with the QNPU of the other node to make sure entanglement attempts are synchronized.

With NetQASM we solve various problems that are unique to quantum internet programming: (1) for remote entanglement generation, we introduce new instruction types for making use of an underlying quantum network stack [21, 22], (2) for the close interaction between classical and quantum operations, we use a shared-memory model for sharing classical data between the application layer and the QNPU, (3) in order to run multiple applications on the same quantum node—which may be beneficial for overall resource usage (see section 4)—we make use of virtualized quantum memory, similar to virtual memory in classical computing [23], (4) since on some platforms, not all qubits may be used to generate remote entanglement, we introduce the concept of unit-modules describing qubit topologies with additional information per (virtual) qubit about which operations are possible.

Since NetQASM is meant to be low-level, similar in nature to classical assembly languages, we have also developed a higher-level software development kit (SDK), in Python, to make it easier to write applications. This SDK and related tools are open-source and freely available at [24], as part of our Quantum Network Explorer [25]. Through the SDK we have also enabled the quantum network simulators NetSquid [26] and SimulaQron [27] to run any application programmed in NetQASM.

We have evaluated NetQASM by simulating the execution of a teleportation application and a BQC using NetQASM. Hereby we have shown that interesting quantum internet applications can indeed be programmed using NetQASM. Furthermore, the evaluations argue certain design choices of NetQASM, namely the use of so-called unit-modules, as well as platform-specific *flavors*.

We remark that NetQASM has already been used on a real hardware setup in the lab, in a highly simplified test case that only produces entanglement [28].

1.2. Related work

In the field of quantum computing, a substantial amount of progress has been made related to developing architectures (e.g. [29–36]), instruction sets (e.g. [37–45]) and compilers [46–59]. One example is QASM, an instruction set framework, borrowing ideas from classical assembly languages, which has gained a lot of popularity over the years and has been successfully integrated in software stacks for quantum computers. There are in fact many variants of QASM such as OpenQASM [37], cQASM [38], eQASM [39], f-QASM [40]. Some of these variants are at a level closer to the physical implementation, such as eQASM, allowing for specifying low-level timing of quantum operations, while others, such as f-QASM, are at a higher level. Together with the definition of these QASM-variants, progress has also been made in compilation of applications programmed in QASM to hardware implementations. More abstract languages and programming frameworks for quantum programs include Quil [41], Qiskit [42], Cirq [43], Q# [44], QuEST [45].

None of these instruction sets or languages contain elements for remote entanglement generation (i.e. between different nodes), which NetQASM does provide. A NetQASM program that uses the vanilla flavor and only contains local operations would look similar to an OpenQASM program. However, the instruction set is not quite the same, since NetQASM uses a different memory model than OpenQASM. This is due to the hybrid nature of quantum network programs, which has more interaction between classical data and quantum data than non-networking programs (for which OpenQASM might be used). So, NetQASM is not just a superset of the OpenQASM instruction set (in the sense of adding entanglement instructions).

In [27], we introduced the CQC interface, which was a first step towards a universal instruction set. However, CQC had a number of drawbacks, in particular: (1) CQC does not have a notion of virtualized memory (see section 4), which meant that applications needed to use qubit IDs that were explicitly provided by the underlying hardware. This introduced more communication overhead and fewer optimization opportunities for the compiler. (2) CQC does not provide as much information about hardware details. Therefore, platform-specific compilation and optimization is not possible. (3) Furthermore, CQC does not match entirely with the later definition of our quantum network stack [21, 22]. For example, it was not clearly defined how CQC relates to the definition of a network layer.

Many of the ideas from e.g. QASM for how to handle and compile local gates can be reused also for quantum network applications. For example, version 3 of OpenQASM [17] which is under development, proposes close integration between *local* classical logic and quantum operations, which is something we also propose in this work. However, there are two key differences that we need to address:

(a) Instructions for generating entanglement between remote nodes in the network need to be handled and integrated with the rest of the application, see section 2.2 below.

(b) The local operations performed by a node might depend on information communicated by another node and only known at runtime. Note that this is different from the conditionals on *local* classical information, proposed in for example OpenQASM version 3, which does not require communication between remote nodes in a network. This brings new constraints in how to handle memory allocation, scheduling and addressing. We discuss this point in further detail in the coming sections.

NetQASM solves the above two points and improves upon CQC.

1.3. Outline

In section 2 we define relevant concepts and introduce the model of end-nodes that we use, including the QNPU. In section 3 we discuss use-cases of a quantum network which NetQASM should support. In section 4 we consider requirements and considerations any instruction set architecture for quantum networks should fulfill which then lay the basis for the decisions that went into developing NetQASM, see section 5. In sections 6 and 7 we describe details about the NetQASM language and associated SDK. In section 8 we quantitatively evaluate some of the design decision of NetQASM by benchmarking quality of execution using the quantum network simulator NetSquid [26, 60]. We conclude in section 9.

2. Preliminaries and definitions

2.1. Quantum networks

A schematic overview of quantum networks is given in figure 3. A quantum network consists of *end-nodes* (hereafter also: *nodes*), which contain quantum network processors as well as classical processors. Nodes are connected by *quantum channels* or *links* that can be used to generate *entangled* quantum states across nodes. End-nodes possess quantum memory in the form of qubits, which can be manipulated by performing *operations* such as initialization, readout, and single- or multi-qubit *gates*. Each quantum memory has a certain *topology* that describes which operations can be applied on which (pair of) qubits. Some of the qubits in a quantum memory may be used to generate an entangled state with another node. These qubits are called *communication qubits* [21], in contrast to *storage qubits* which can only directly interact with other qubits part of the same local node[5].

Some platforms only have a single communication qubit and multiple storage qubits [61], whereas others can have multiple communication qubits [11]. Qubits are sensitive to *decoherence* and have limited lifetimes. Therefore, the timing and duration of operations (such as local gates or entanglement generation with another node) have an impact on the quality of quantum memory. Classical processors control the quantum hardware, and also perform classical computation. Finally, classical links exist between nodes for sending classical messages.

Since end-nodes can control their memory and entanglement generation, they can run arbitrary *user programs*. End-nodes can both communicate classically and generate entanglement between each other, either directly or through repeaters and routers, (figure 3). Nodes in the network other than end-nodes, such as repeaters and routers, do not execute user programs; rather these run protocols that are part of some level in the network stack [21, 22]. These internal nodes in the network perform elementary link generation and entanglement swapping in order to generate long-distance remote entanglement between end-nodes [21].

[5] A storage qubit may however hold a state that is entangled with a qubit in another node: after remote entanglement generation using a communication qubit, the state in that local qubit could be transferred to one of the storage qubits, preserving the remote entanglement.

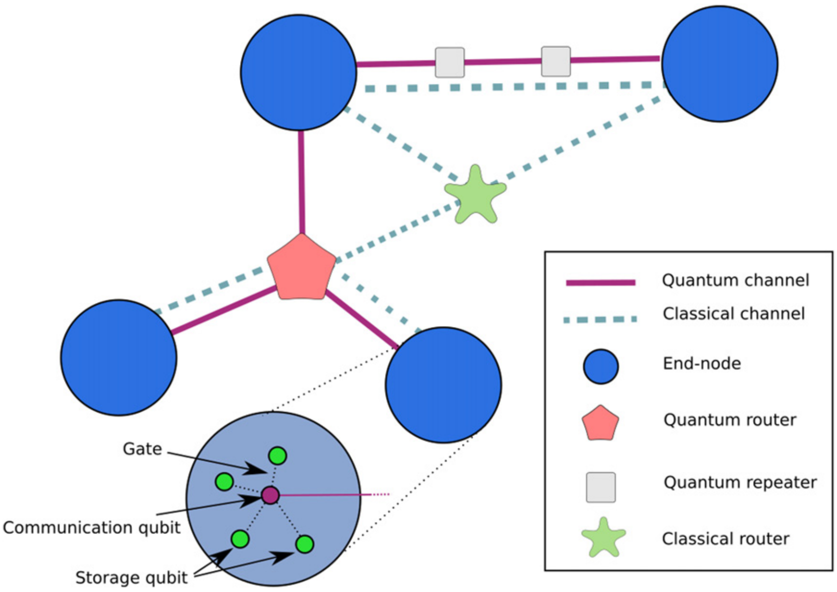


Figure 3. Abstract model of a quantum network and its components. Quantum network applications run on the *end-nodes* (blue). Their communication via classical message passing and quantum entanglement (figure 1) is abstracted away by a network stack. That is, it is not visible at the application layer how entanglement generation or classical message passing is realized. This may be via direct physical connections, or intermediary repeaters and/or routers. End-nodes hold two types of qubits: (1) *communication qubits* which can be used to generate entanglement with remote nodes and (2) *storage qubits* which can be used to store quantum states and apply operations. A communication qubit may also be used as a storage qubit. The qubits within an end-node can interact through quantum gates and their state can be measured.

There are various quantum hardware implementations for quantum network processors, such as NV centers in diamond [61], ion traps [8], and neutral atoms [9, 62], which all have different capabilities and gates that can be performed.

In contrast to classical networks, we consider the end-nodes in a quantum network to not have a network interface component that is separated from the main processing unit. Having local and networking operations combined in a single interface reflects the physical constraint on current and near-term hardware. Current state-of-the-art hardware for quantum networking devices can make use of up to the order of 10 qubits [63]. Furthermore, certain hardware implementations, such as NV centers in diamond [61], only have a single communication qubit, which also acts as a mediator for any local gate on the storage qubits. This prevents dedicating some qubits for purely local operations and some for purely networking operations. Rather, to make maximal use of near-term quantum hardware, a multi-purpose approach needs to be supported.

2.2. Application layer and QNPU

In this work we will assume an abstract model of the hardware and software architecture of end-nodes in a quantum network. Specifically, we assume each end-node to consist of an application layer and a QNPU. The application layer can be also be seen as a the user space of a classical computer, and the QNPU as a coprocessor.

This model takes into account both physical- and application-level constraints found in quantum network programming. The QNPU can be accessed by the application layer, at the same node, to execute quantum and classical instructions. We define the capabilities of the QNPU, and roughly their internal components, but do not assume how exactly this is implemented. In the rest of this work, we simply use the QNPU as a black box.

The QNPU can do both classical and quantum operations, including (1) local operations such as classical arithmetic and quantum gates and (2) networking operations, i.e. remote entanglement generation. The application layer cannot do any quantum operations. It can only do local computation and classical communication with other nodes. In terms of classical processing power, the difference between the application layer and the QNPU is that the application layer can do heavy and elaborate computation, while we assume the QNPU to be limited in processing power.

The application layer can interact with the QNPU by for example sending instructions to do certain operations. The application layer and the QNPU are logical components and may or may not be the same

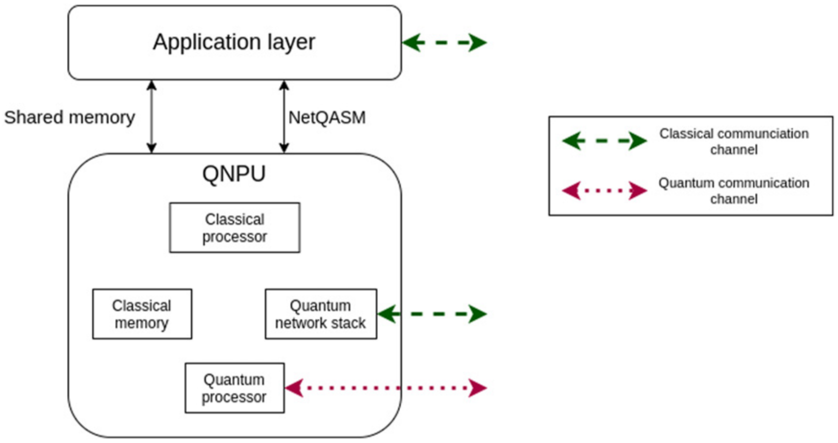


Figure 4. Overview of QNPU components and interfaces. The application layer talks to the QNPU using NetQASM. The processor inside the QNPU can interact with all other components. Channels are connecting components with corresponding components in adjacent nodes in the network.

physical device. It is assumed that there is low latency in the communication between these components, and in particular that they are physically part of the same node in the network.

One crucial difference between the application layer and the QNPU is that the application layer can do application-level classical communication with other end-nodes, while the QNPU cannot. The QNPU can communicate classically to synchronize remote entanglement generation, but it does not allow arbitrary user-code classical communication. We use this restriction in order for the QNPU to have relatively few resource requirements.

The QNPU consists of the following components, see figure 4:

- **Processor.** The processor controls the other components of the QNPU and understands how to execute the operations specified by the application layer. It can read and write data to the classical memory and use this data to make decisions on what operations to do next. It can apply quantum gates to the qubits in the quantum memory and measure them as well. Measurement outcomes can be stored in the classical memory.
- **Classical memory.** Random-access memory storing data produced during the execution of operations, such as counters, qubit measurement outcomes, information about generated entangled pairs, etc.
- **Quantum memory.** Consists of communication and storage qubits, see section 2.1, on which quantum gates can be applied. The qubits can be measured and the resulting outcome stored in the classical memory by the processor. The communication qubits are connected through a quantum channel to adjacent nodes in the quantum network, through which they can be entangled. This quantum channel may also include classical communication needed for synchronization, phase stabilization or other mechanisms needed in the specific realization.
- **Quantum network stack.** Communicates classically with other nodes and quantum repeaters in the network to synchronize the generation of remote entanglement, and issues low-level instructions to execute the entanglement generation procedures, see [21, 22].

We stress that the internals of the QNPU are not relevant to the design of NetQASM. We do assume that the QNPU only has limited classical processing power, and can therefore be implemented on for example a simple hardware board.

2.3. Applications and programs

As mentioned in section 1, quantum network *applications* (or protocols) are multi-partite and distributed over multiple end-nodes. The unit of code that is executed on each of the end-nodes that are part of the application, is called a *program*. We will use this terminology throughout the rest of the paper.

As mentioned in the previous section, the end-nodes are modeled such that there is an application layer and a QNPU. We assume that execution of quantum network programs is handled by the application layer. How exactly the program is executed, and how the QNPU is involved herein, is part of the NetQASM proposal.

3. Use-cases

In the next section we will discuss the design considerations taken when developing NetQASM. These design considerations are based on a set of use-cases listed in this section which we intend for NetQASM to support. Applications intended to run on a quantum network will often depend on a combination of these use-cases.

- **Local quantum operations.** Applications running on a network node need to perform quantum operations on local qubits, including initialization, measurement, and single- or multi-qubit gates. Such local qubit manipulation is well known in the field of quantum computing. For example, OpenQASM [37] describes quantum operations. Quantum *network* applications should be able to do these local operations as well.
- **Local quantum operations depending on local events or data.** The next use-case stems from applications consisting of programs in which limited classical computation or decision making is needed in-between performing quantum operations. Here we consider only dependencies in a program between quantum operations and information that is produced locally, that is, on the node that this program is being executed. For instance, a program might only apply a quantum gate on a qubit depending on the measurement outcome of another qubit, or choose between execution branches based on evaluation of a classical function of earlier measurement outcomes. An example is for the server-side of BQC, which performs a form of measurement-based quantum computation (MBQC). In each step of the MBQC, the server performs certain gates on a qubit, depending on results of measuring previous qubits [64]. These applications need classical operations to not take too much time, so that qubit states stay coherent during these operations. This implies that switching between classical and quantum operations should have little overhead.
- **Entanglement generation.** Crucial to quantum networks is the ability to generate remote *entanglement*. Applications should be able to specify requests for entanglement generation between remote nodes. In some cases, a measure-directly [21] type generation is required, where entangled state is measured directly, without storing in memory, to obtain correlated classical bits, such as in QKD. However, in many cases a create-keep [21] type is needed, where the entanglement needs to be stored in memory and further operations applied involving other qubits. We want applications to be able to *initiate* or *receive* (await) entanglement of both forms with nodes in the network.
- **Local quantum operations depending on remote events or data.** We already mentioned the use-case of having conditionals based on *local* information. We also envision applications that need to store qubits and subsequently perform local quantum operations on them and other local qubits, based on classical information coming from *another node*. An example is *teleportation* in which the receiver—after successful entanglement generation—needs to apply local quantum corrections based on the measurement outcomes of the sender. Another application is BQC, where the server waits for classical instructions from the client about which quantum operations to perform. Hence, there need to be integration of classical communication (sending the measurement results or further instructions) and the local quantum operations. Furthermore, since classical communication has a non-zero latency (and is in general even non-deterministic), it should be possible to suspend doing quantum operations while waiting for communication or performing classical processing, while quantum states stay coherent.
- **Waiting time.** We consider the scenario where an application requires two nodes to communicate with each other, and where communication takes a long time, for example since they are physically far apart. It should be possible for a program to suspend doing quantum operations while waiting for communication or performing classical processing, while quantum states stay coherent. Furthermore, in order to maximize the usage of the QNPU we want to have a way to fill this waiting time in a useful way.

4. Design considerations

In this section we review the most important design considerations and requirements that were applied when developing NetQASM. Our proposed solutions to these design considerations are presented in the next section, with more details about NetQASM as a language in the subsequent sections.

- **Remote entanglement generation.** One of the main differences compared to the design considerations of a quantum computing architecture is that of remote entanglement generation (see the use-case in section 3). Nodes need to be able to generate entanglement with a remote node, which requires the collaboration and synchronization of both nodes, and possibly intermediate nodes, which is handled by the network stack (section 2).

Further requirements arise in platforms with a limited number of communication qubits. The extreme case is NV centers in diamond which have a single communication qubit that additionally is required for performing local operations. For this reason it is not possible to decouple local gates on qubits from entanglement generation. We note the contrast with classical processors, where networking operations are typically intrinsically separate kinds of operations. For example, operations such as sending a message may simply involve moving data to a certain memory (e.g. that of a physically separate network interface), which is often abstracted as a system call.

A quantum network stack has already been proposed in [21, 22], and we expect the QNPU of the end-node to implement such a stack, including a *network layer* that exposes an interface for establishing entanglement with remote nodes. The way in which a program creates remote entanglement should therefore be compatible with this network layer.

- **Conditionals.** In section 3 we mentioned the need to do local quantum operations conditioned on classical data that may be generated locally or by remote nodes. Such classical data include for example measurement results or information communicated to or from other nodes in the network. We distinguish between real-time and near-time conditionals [17]. Real-time conditionals are time-sensitive, such as applying a certain quantum operation on a qubit depending on a measurement outcome. For such conditionals, we would like to have fast feedback, in order for quantum memory not to wait too long (which would decrease their quality). Near-time conditionals are not as sensitive to timing. For example, a program may have to wait for a classical message of a remote node, while no quantum memory is currently being used. Although it is preferably minimized, the actual waiting time does not affect the overall execution quality.
- **Shared memory.** As described in section 2, we expect end-nodes to consist of an application layer and a QNPU. These two components have different capabilities. For example, only the application layer has the ability to do arbitrary classical communication with other nodes. Only the QNPU can do quantum operations. These restrictions lead the design in a certain way. The two components hence need to work together somehow. There needs to be model for interaction between the two, and also for shared memory.

Executing programs on an end-node is shared by the application layer and the QNPU (see section 2.2). Indeed, only the QNPU can do quantum-related operations, whereas the application layer needs to do classical communication. In order to make these work together, the two components have to share data somehow. This includes the application layer requesting operations on the QNPU, and sending the following from the QNPU to the application layer: (1) measurement outcomes of qubits, (2) information about entanglement generation, in particular a way to identify entangled pairs. This communication between application layer and QNPU needs to be done during runtime of the program. This is in contrast to local quantum computation, where one might wait until execution on the QNPU is finished before returning all data. The challenge for quantum network programs is to have a way to return data while quantum memory stays in memory.

- **Processing delay.** Since we assume that the application layer and the QNPU have to share execution of a single program, the interaction between the two layers should be efficient. Unnecessary delays lead to reduced quality (see section 1). The challenge is therefore to come up with an architecture for the interaction between the application layer and the QNPU, as well as a way to let QNPU execution not take too long.
- **Platform-independence.** As explained in section 1, hardware can have many different capabilities and gates that can be performed. However, application programmers should not need to know the details of the underlying hardware. For this reason, there needs to be a framework through which a programmer can develop an application in a platform-independent way which compiles to operations the QNPU can execute.
- **Potential for optimization.** Since near-term quantum hardware has a limited number of qubits and qubits have a relatively short lifetime, the hardware should be utilized in an effective way. There is therefore a need to optimize the quantum gates to be applied to the qubits. This includes for example choosing how to decompose a generic gate into native gates, rearranging the order of gates and measurements and choosing what gates to run in parallel. Since different platforms have vastly different topologies and gates that they can perform, this optimization needs to take the underlying platform into account. The challenge is to have a uniform way to express both platform-independent and platform-specific instructions.

- **Multitasking.** The 'waiting time' use-case in section 3 describes that a node's QNPU may have to wait a long time. We consider the solution that the QNPU may do multitasking, that is, run multiple (unrelated) programs at the same time. Then, when one program is waiting, another program can execute (partly) and fill the gap. To make our design compatible with such multitasking, we need to provide a way such that programs can run at the same time as other programs, but without having to know about them.
- **Ease of programming.** Even though NetQASM provides an abstraction over the interaction with the QNPU, it is still low-level and hence not intended to be used directly by application developers. Furthermore, applications also contain classical code that is not intended to run on the QNPU. Therefore it should be possible to write programs consisting of both classical and quantum (network) operations in a high-level language like Python, and compile them to a hybrid quantum–classical program that uses NetQASM.

5. Design decisions

Based on the use-cases, design considerations and requirements, we have designed the low-level language NetQASM as an API to the QNPU. In this section we present concepts and design decisions we have taken. Details on the mode of execution and the NetQASM-language are presented in section 6.

5.1. Interface between application layer and QNPU

5.1.1. Execution model

As described in section 2, and also in section 6 program execution is split across the application layer and the QNPU. Since the QNPU is assumed to have limited processing power (section 2), our design lets the application layer do most of the classical processing. The program blocks (figure 2) are hence spread over two separate systems: blocks of purely classical code are executed by the application layer, and blocks of quantum code (containing both quantum operations and limited classical control) are executed by the QNPU.

The quantum code (including limited classical control) is expressed using the NetQASM language. The classical code is handled completely by the application layer, and we do not impose a restriction to its format. In our implementation (section 7), we use Python. This classical code on the application layer also handles all application-level classical communication between nodes, since it cannot be done on the QNPU.

We let the application layer initiate a program. Whenever quantum code needs to be executed, the application layer delegates this to the QNPU. Since processing delay should be minimized (section 4), the communication between application layer and QNPU should be minimized. Therefore, NetQASM bundles the quantum operations together into blocks of instructions, called *subroutines*, to be executed on the QNPU. A program, then, consists of both classical code and quantum code, and the quantum code is represented as one or more subroutines. These subroutines can be seen as the quantum code blocks of figure 2.

For most programs, we consider subroutines to be sent consecutively in time. However, if the QNPU supports it, NetQASM also allows to send multiple subroutines to be executed on the QNPU at the same time, although this requires some extra care when dealing with shared memory. From the perspective of the QNPU, a program consists of a series of subroutines sent from the application layer. Before receiving subroutines, the application layer first *registers* a program at the QNPU. The QNPU then sets up the classical and quantum memories (see below) for this program. Then, the application layer may send subroutines to the QNPU for execution.

5.1.2. Shared classical memory

Since classical and quantum blocks in the code (as per figure 2) can depend on each other, the application layer and the QNPU need to have a way to communicate information to each other. For example, a subroutine may include a measurement instruction; the outcome of this measurement may be used by the application layer upon completion of the subroutine. Therefore, NetQASM uses a shared memory model such that conceptually both layers can access and manipulate the same data. This solves the need to return data, and to do conditionals (section 4).

Each program has a classical memory space consisting of *registers* and *arrays*. Registers are the default way of storing classical values, like a measurement outcome. In the example of the application layer needing a measurement outcome, there would be an instruction in the subroutine saying that a measurement outcome needs to be placed in a certain register. The application layer can then access this same register (since they share the memory space) and use it in further processing. The number of registers is small, and constant for each program. Arrays are collections of memory slots (typically the slots are contiguous),

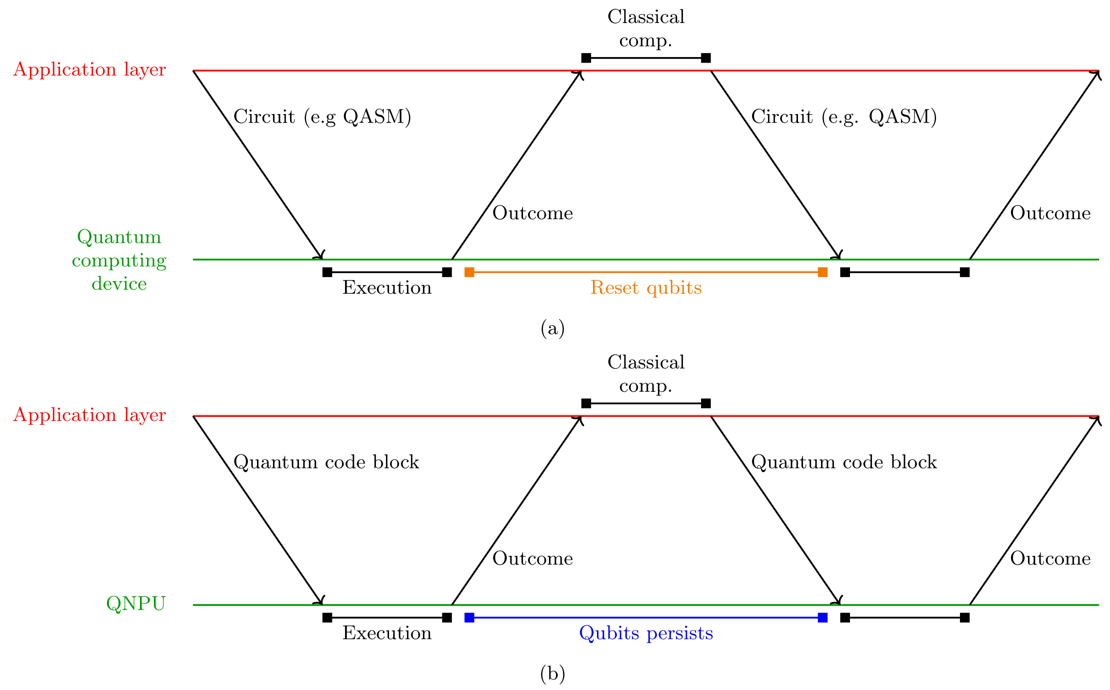


Figure 5. Program interaction between the application layer and a quantum device in both the case of *hybrid-quantum computing* (figure 5(a)) and quantum networks (figure 5(b)). In the case of hybrid-quantum computing, qubits are reset in between circuits (in e.g. QASM). For quantum internet programs the qubits should on the other hand be kept in memory, since they might be entangled with another node and intended to be used further.

which can be allocated by the program at runtime. Arrays are used to store larger chunks of data, such as parameters for entanglement requests, entanglement generation results, or multiple measurement outcomes when doing multiple entangle-and-measure operations. The application layer may only read from the shared memory; writing to it can only be done by issuing NetQASM instructions such as `set` (for registers) and `store` (for arrays). The QNPU may directly write to the shared memory, for example when entanglement finished and it writes the results to the array specified by the program.

5.1.3. Unit modules

In order to support systems with multitasking (section 4), NetQASM provides a virtualized model of the quantum memory to the program. This allows the QNPU to do mapping between the virtualized memory and the physical memory and perform scheduling between programs.

The quantum memory for a program is represented by a unit-module (figure 6). A unit-module defines the topology of the available qubits (which qubits are connected, i.e. on which qubit pairs a two-qubit gate can be executed), plus additional information on each qubit. This additional information consists of which gates are possible on which qubit or qubit pair. It also specifies if a qubit can be used for remote entanglement generation or not. The extra information is needed since on some platforms, not all qubits can be used for entanglement generation and different qubits may support different local gates. For example, in a single NV-centre, there is only one communication qubit and any additional qubits are storage qubits. Also, the communication qubit can do different local gates than the storage qubits (figure 7).

A single program has a single quantum memory space, which is *not* reset at the end of a subroutine, which is in contrast with quantum computing. This allows the application layer to do processing while qubits are in memory. The following sequence of operations provides an example. (1) The application layer first sends a subroutine containing instructions for entanglement generation with a remote node R. (2) The QNPU has finished executing the subroutine, and informs the application layer about it. There is now a qubit in the program’s memory that is entangled with some qubit in R. (3) The application layer does some classical processing and waits for a classical message from (the application layer of) R. (4) Based on the contents of the message, the application layer sends a new subroutine to the QNPU containing instructions to do certain operations on the entangled qubit. The subroutine can indeed access this qubit by using the same identifier as the first subroutine, since the quantum memory is still the same. We note the contrast with (non-network) quantum computing, where quantum memory is reset at the end of each block of instructions (figure 5).

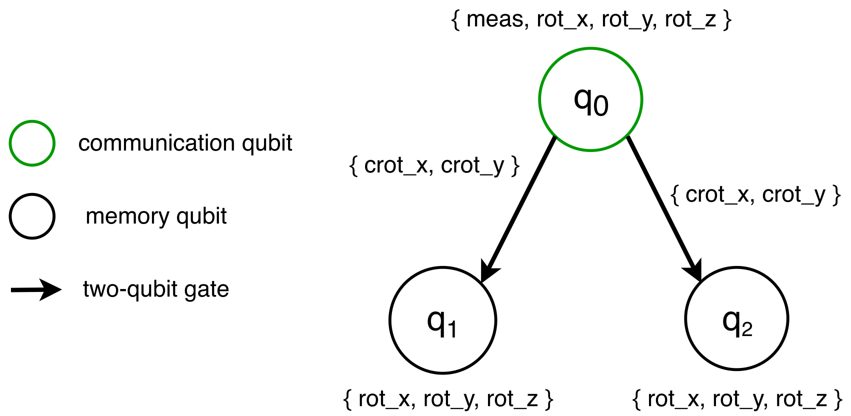


Figure 6. Example of a unit-module topology on a platform using NV centers in diamond. A unit-module is a hypergraph [65], with associated information on both nodes and edges. Each node represents a virtual qubit, containing information about (1) its qubit type (communication or storage), (2) physical properties of the qubit, such as decoherence times and (3) which single-qubit gates are supported on the qubit, together with their duration and noise. Each edge represents the possibility of performing joint operations on those qubits, such as two-qubit gates, and also containing information about gate durations and noise.

```
array 10 @0                // array for writing EPR results to
array 1 @1                 // array with virtual IDs for enangled qubits to be generated
store 0 @1[0]              // set virtual ID of the only generated qubit to 0
array 20 @2                // array for holding EPR request parameters
store 0 @2[0]              // set request type to 0 (Create and Keep)
store 1 @2[1]              // set number of requested EPR pairs to 1
create_epr(1,0) 1 2 0      // send command to create EPR pair
wait_all @0[0:10]          // wait until results for first pair (10 elements) are available
set Q0 0
meas Q0 M0                 // measure the entangled qubit
qfree Q0
ret_reg M0                 // return measurement outcome
```

Figure 7. Example of NetQASM code for generating a single entangled pair with another node followed by a measurement. See the appendix for more details of the instructions.

unit-modules contain *virtual qubit IDs*. This is because of the requirement that it should be possible to run multiple programs at the same time on a single QNPU (multitasking consideration in section 4). We use an approach that is similar to virtual memory in classical systems [23]. Each application has control over a set of physical qubits, but the application does not (need to) know which physical qubits these are exactly. The unit-module provides a virtualized view of this available memory. This view contains virtual IDs each representing a single qubit, called a virtual qubit. The QNPU maintains a mapping of virtual IDs (per application) to physical qubits. The QNPU may change this mapping over time, without the applications knowing. We stress that our virtualization hence only involves a mapping from IDs to physical qubits. There is no copying of quantum states involved.

We note that this design decision meets our multitasking consideration (section 4). By using virtualized unit-modules, the QNPU is free to map qubit IDs of the application to physical qubits as it sees fit. For example, consider a node with a physical memory consisting of one communication qubit, and multiple memory qubits. Application A creates entanglement with a remote node such that its half of the pair is in the communication qubit. Then, application A needs to wait for a long time before further processing the quantum state in this qubit, for example since it needs to wait for a classical message from a remote node. Meanwhile, application B is waiting to be executed on the QNPU, and it also requires the communication qubit for entanglement generation. The QNPU can now move the state from the communication qubit to one of the memory qubits, and update the mapping of application A's ID to this physical memory qubit. Then, the QNPU can run application B while A is waiting for the classical message. When B has finished, the QNPU can move A's state back to the communication qubit. Since application A uses the unit-module and does not know about the physical memory, it (1) does not care that its state was temporarily moved to a different physical qubit, and (2) can remain oblivious about any other application being run (like B) while it is waiting.

5.2. NetQASM language

5.2.1. Instructions

As explained in section 5.1, the application layer delegates quantum code (including limited classical control) of the program to the QNPU by creating blocks of instructions and sending these to the QNPU for execution. These blocks are called subroutines and contain NetQASM *instructions*. Since the QNPU is meant to be limited in processing power, the instruction set that it interprets should also be simple and low-level. The NetQASM instruction set contains instructions for simple arithmetic, classical data manipulation, and simple control flow in the form of (un)conditional branch instructions. Although conditional control-flow can be done at the application layer as well, NetQASM branching instructions allow for much faster feedback since they are executed by the QNPU, and hence cover the design consideration of real-time conditionals (section 4). We note the obvious performance gain by being able to do control logic without having to go back to the application layer. There are no higher-level concepts such as functions or for-loops, which would require more complicated and resource-demanding parsing for the QNPU, such as constructing an abstract syntax tree.

A single instruction specifies an operation, possibly acting on classical or quantum data. For example, a single-qubit rotation gate is represented as an instruction containing the type of gate, the classical register containing the rotation angle, and the classical register containing the virtual ID of the qubit (as specified in the unit-module) to act on. NetQASM specifies a set of *core* instructions that are expected to be implemented by any QNPU. These include classical instructions like storing and loading classical data, branching, and simple arithmetic. Different hardware platforms support different quantum operations. NetQASM should also support platform-specific optimization (section 4). Therefore, NetQASM uses *flavors* of quantum instructions (section 5.2.3). The *vanilla* flavor consists universal of a set of platform-independent quantum gates. Particular hardware platforms, such as the NV-centre, may use a special NV flavor, containing NV-specific instructions. A QNPU implementation may use a custom mapping from vanilla instructions to platform-specific ones. The instructions in a flavor are also called a software-visible gate set [31]. See appendix F for more details on NetQASM instructions.

5.2.2. Remote entanglement generation

Generating entanglement with a remote node is also specified by instructions. These are however somewhat special compared to other instructions. First, entanglement generation has a non-deterministic duration. Therefore, when an entanglement instruction is executed, the request is forwarded to the part of the system responsible for creating entanglement, but the instruction itself immediately returns. A separate *wait* instruction can be used to block on entanglement generation to actually be completed. Second, entanglement generation requests should be compatible with the network stack proposed in [21], including the network layer from [22]. These requests need to be accompanied by information such as the number of EPR pairs to generate or the minimum required fidelity. Third, this information should be able to depend on runtime information. For example, the required fidelity may depend on an earlier measurement outcome. Therefore, entanglement generation parameters cannot be static data, and must be stored in arrays. Furthermore, the result of entanglement generation with the remote node consists of a lot of information, such as which Bell state was produced, the time it took, and the measurement results in case of measuring directly. This information is written by the QNPU to an array which is specified by the entanglement instruction. Finally, since writing the information to the array indicates that entanglement generation succeeded, the wait instruction can be used to wait until a certain array is filled in, such as the one provided by the entanglement instruction. Since the entanglement instruction is non-blocking, it is possible to continue doing local operations while waiting for entanglement generation to complete.

We assume that the QNPU implements a network stack where connections need to be set-up between remote nodes before entanglement generation can happen [21, 22]. NetQASM provides a way for programs to open such connections in the form of *EPR sockets*. The application layer can ask the QNPU to open an EPR socket with a particular remote node. The QNPU is expected to set up the required connections in the network stack, and associates this program socket with the connection. When the program issues an instruction for generating entanglement, it refers to the EPR socket it wants to use. Based on this, the QNPU can use the corresponding connection in the network.

5.2.3. Flavors

We want to keep NetQASM platform-independent. However, we also want the potential for platform-specific optimization (section 4). Therefore we introduce the concept of *flavors*. Flavors only affect the quantum instruction set of the language, and not the memory model or the interaction with the QNPU. We use the *vanilla* or generic flavor for a general, universal gate set. Subroutines may be written or generated in this vanilla flavor. Platform-independent optimization may be done on this level. A QNPU may

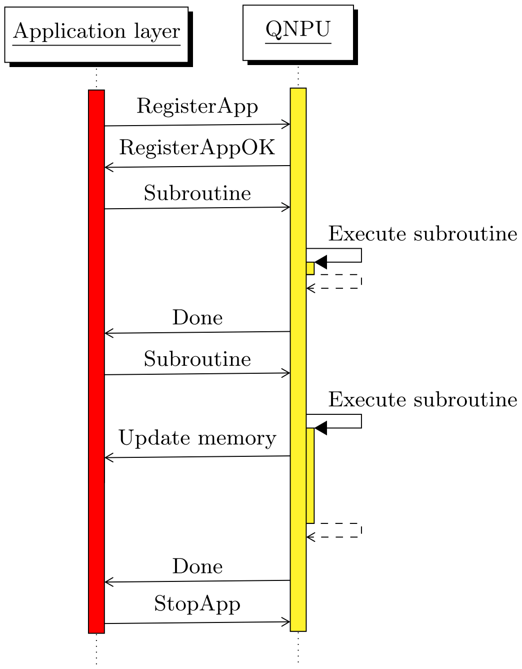


Figure 8. Flow of messages between the application layer and the QNPU.

directly support executing vanilla-flavored NetQASM. Platform-specific translations may then be done by the QNPU itself. It can also be that a QNPU only supports a specific flavor of NetQASM. A reason for this could be that the QNPU does not want to spend time translating of the instructions at runtime. In this case, the application layer should perform a translation step from the vanilla flavor to the platform-specific flavor. In such a case, the vanilla flavor can be seen as an *intermediate representation*, and the translation to a specific flavor as a back-end compilation step.

5.2.4. Programmability

Since the NetQASM instructions are relatively low-level, we like to have a higher-level programming language for writing programs, that is automatically compiled to NetQASM. We introduce a higher-level SDK in section 7. However, we do not see this as part of the NetQASM specification itself. This decoupling allows the development of SDKs to be independent such that these can be provided in various languages and frameworks.

We still want NetQASM instructions to be suitable for manual writing and inspection. Therefore, instructions (and subroutines) have two formats: a binary one that is used when sending to the QNPU, and a text format that is human-readable. The text format resembles assembly languages including OpenQASM. Example are given in section 7.1 and the appendix.

6. Implementation

6.1. Interface between application layer and QNPU

Here we explain the flow of messages between the application layer and the QNPU. The application layer starts by declaring the registration of an application, including resource-requirements for the application. After this, the application layer sends some number of subroutines for the QNPU to execute before declaring the application is finished. See figure 8 for a sequence diagram and below for a definition of the messages. In section 6.2 we will describe in more details the content of the subroutines and the format of instructions. The QNPU returns to the application layer an assigned application ID for the registered application and returns data based on the subroutines executed.

The application layer and the QNPU are assumed to run independently and in parallel. For example, while a subroutine is being executed by the QNPU, the application layer could in principle do other operations, such as heavy processing or communication with another node.

Figure 8 shows an example of a message exchange between the application layer and the QNPU. The content of these messages is further detailed in appendix A.

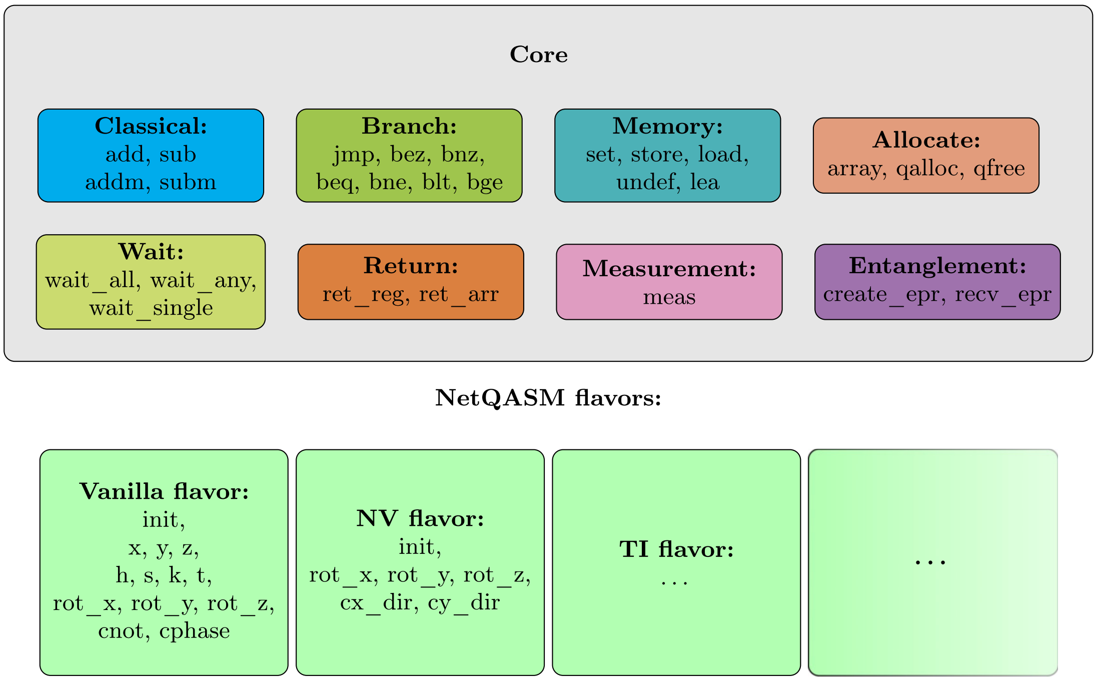

Core
Classical: add, sub addm, subm
Branch: jmp, bez, bnz, beq, bne, blt, bge
Memory: set, store, load, undef, lea
Allocate: array, qalloc, qfree
Wait: wait_all, wait_any, wait_single
Return: ret_reg, ret_arr
Measurement: meas
Entanglement: create_epr, recv_epr
NetQASM flavors:
Vanilla flavor: init, x, y, z, h, s, k, t, rot_x, rot_y, rot_z, cnot, cphase
NV flavor: init, rot_x, rot_y, rot_z, cx_dir, cy_dir
TI flavor: ...
...

Figure 9. The **core** of NetQASM consists of eight groups of instructions. The quantum gates are defined as a set of software-visible gates part of a NetQASM **flavor**. The **vanilla flavor** is the unique platform-independent NetQASM **flavor** of NetQASM, which can be used by a compiler.

6.2. The language

The syntax and structure of NetQASM resemble that of classical assembly languages, which in turn inspired the various QASM-variants for quantum computing [37–40].

A NetQASM instruction is formed by an instruction name followed by some number of operands:

```
instr operands
```

where instr specifies the instruction, for example add to add numbers or h to perform a Hadamard. The operands part consists of zero or more values that specify additional information about the instruction, such as which qubit to act on in the case of a gate instruction. Instructions and operands are further specified in appendix B.

6.3. Instructions

There are eight groups of instructions in the **core** of NetQASM. Also summarized in figure 9, these are:

- **Classical.** Classical arithmetic on integers.
- **Branch.** Branching operations for performing conditional logic.
- **Memory.** Read and write operations to classical memory (register and arrays).
- **Allocate.** Allocation of qubits and arrays.
- **Wait.** Waiting for certain events. This can for example be the event that entanglement has been generated by the network stack.
- **Return.** Returning classical values from the QNPU to the application layer. In our implementation we implement this by having the QNPU write to the shared memory so that the application layer can access it.
- **Measurement.** Measuring a qubit.
- **Entanglement.** Creating entanglement with a remote node using the quantum network stack.

Quantum gates are specific to a NetQASM **flavor** and given as a set of software-visible gates of a given platform, see section 4. There is a single platform-independent NetQASM **flavor** which we call the **vanilla flavor**, see figure 9. The **vanilla flavor** can be used as an intermediate representation for a compiler.

6.4. Compilation

Although application programmers could write NetQASM subroutines manually, and let their (classical) application code send these subroutines to the QNPU, it is useful and more user-friendly to be able to write quantum internet applications in a higher level language, and have the quantum parts compiled to NetQASM subroutines automatically. For this, we use the compilation steps depicted in figure 10. The format and compilation of the higher-level programming language is not part of the NetQASM specification. However, we do provide an implementation in the form of an SDK, see section 7.

7. Python SDK

We implemented NetQASM by developing a SDK in Python. This SDK allows a programmer to write quantum network programs as Python code, including the quantum parts. These parts are automatically translated to NetQASM subroutines. The SDK contains a simulator that simulates a quantum network containing end-nodes, each with a QNPU. The SDK can execute programs by executing their classical parts directly and executing the quantum parts as NetQASM subroutines on the simulated QNPU. By executing multiple programs at the same time, on the same simulated network, a whole multi-partite application can be simulated. In section 8 we use this SDK to evaluate some of the design decisions of NetQASM.

We refer to the docs at [24] for the latest version of the SDK. Below, we give an example of an application written in the SDK to give an idea of how development in the SDK looks like. In appendix H.2 we provide a few more examples of applications in the SDK and their corresponding NetQASM subroutines.

All code can be found at [24, 66], including: (1) tools for serializing (de-serializing) to (from) both human-readable text form and binary encoding, (2) the NetQASM SDK, together with compilers (no optimization yet), (3) support for running applications written in the SDK on the simulators NetSquid [26, 60] and SimulaQron [27], and (4) implemented applications in NetQASM, including: anonymous transmission [67], BB84 [2], blind quantum computing [68, 69], CHSH game [70], performing a distributed CNOT [71], magic square game [72], teleportation [73].

7.1. SDK

The SDK of NetQASM uses a similar framework to the SDK used by the predecessor CQC [74]. Any program on a node starts by setting up a NetQASMConnection to the QNPU-implementation in the *backend*. The NetQASMConnection encapsulates all communication that the application layer does with the QNPU. More information about supported backends can be found below in section 7.1.1. Using the NetQASMConnection one can for example construct a Qubit object. The qubit object has methods for performing quantum gates and measurements. When these methods are called, corresponding NetQASM instructions are included in the current subroutine being constructed. One marks the end of a subroutine, and the start of another, either by explicitly calling flush on the NetQASMConnection or by ending the scope of the `with NetQASMConnection ...` context.

The following Python code shows a basic application written in the NetQASM SDK. The application will be compiled into a single subroutine executed on the QNPU, which creates a qubit, performs a Hadamard operation, measures the qubit and returns the result to the application layer.

```
# Setup connection to backend
# as the node Alice
with NetQASMConnection("Alice") as alice:
  # Create a qubit
  q = Qubit(alice)
  # Perform a Hadamard on the qubit
  q.H()
  # Measure the qubit
  m = q.measure()
  # The end of the context also marks
  # the end of the subroutine
  # automatically but can also be done
  # explicitly using ‘alice.flush()‘
```

The following NetQASM subroutine is the result of translating the above Python code to NetQASM of the vanilla (platform-independent) flavor.

```
# NETQASM 1.0
# APPID 0
// Set the virtual qubit ID to use
set Q0 0

// Allocate and initialize a qubit
qalloc Q0
init Q0

// Perform a Hadamard gate
h Q0

// Measure the qubit
meas Q0 M0

// Return the outcome
ret_reg M0
```

7.1.1. Backends

As mentioned above, the NetQASMConnection in the SDK is responsible for communicating with the implemented QNPU in the *backend*. The *backend* can either be a simulator or an actual QNPU using real quantum hardware. Currently supported backends are the simulators SquidASM [66] (using NetSquid [26, 60]) and SimulaQron [27]. A physical implementation of QNPU running on quantum hardware is being worked on at the time of writing. Using the SDK provided at [24], one can for example simulate a set of program files for the nodes of a quantum network on NetSquid using a density matrix formalism with the command:

```
netqasm simulate --simulator=netsquid --formalism=dm
```

For more details see the docs at [24].

8. Evaluation

We evaluate two of the design choices that we made for NetQASM: (1) exposing unit-modules to the application layer and (2) adding the possibility to use platform-specific flavors of instructions. For both elements we study the difference in including them in NetQASM versus not including them. We do this by simulating a teleportation application and a BQC application. These examples also showcase the ability of NetQASM to express general quantum internet applications.

We have implemented a simulator, called SquidASM [66], that simulates a network in which end-nodes have the internal architecture as described in section 2, that is, with an application layer and a QNPU. The simulator internally uses NetSquid [60], which was made specifically for the simulation of quantum networks. SquidASM executes programs written using the SDK (section 7), including sending NetQASM subroutines to the (simulated) QNPU. The code and data that were used to produce the results in this section can be found at [75].

We evaluate the performance of NetQASM by looking at the runtime quality of two applications, both consisting of two programs (one per node). The first is a teleportation of a single qubit from a sender node to a receiver node. We define the quality as the fidelity between the original qubit state at the sender and the final qubit state at the receiver. The second application is a blind computation protocol which involves a client and a server. The server effectively performs, blindly, a single-qubit computation on behalf of the client. The protocol is a so-called *verifiable* BQC [69]. This means that some of the rounds of the protocols are *trap rounds*. We define the quality that we evaluate as the error rate of these trap rounds, since this indicates the blindness of the server.

We run these applications on SquidASM, where we simulate realistic quantum hardware. Specifically, we simulate nodes based on NVs in diamond, that can do heralded entanglement generation between each other. The simulated hardware uses noise models that are also used in [26]. For more details, see appendix I.

A note on how we chose what to evaluate and what not. We listed several design considerations in section 4. We addressed these in our design decisions (section 5). For some of these, it is straightforward to see how they address a certain consideration, such as conditionals allowing for fast runtime feedback, and

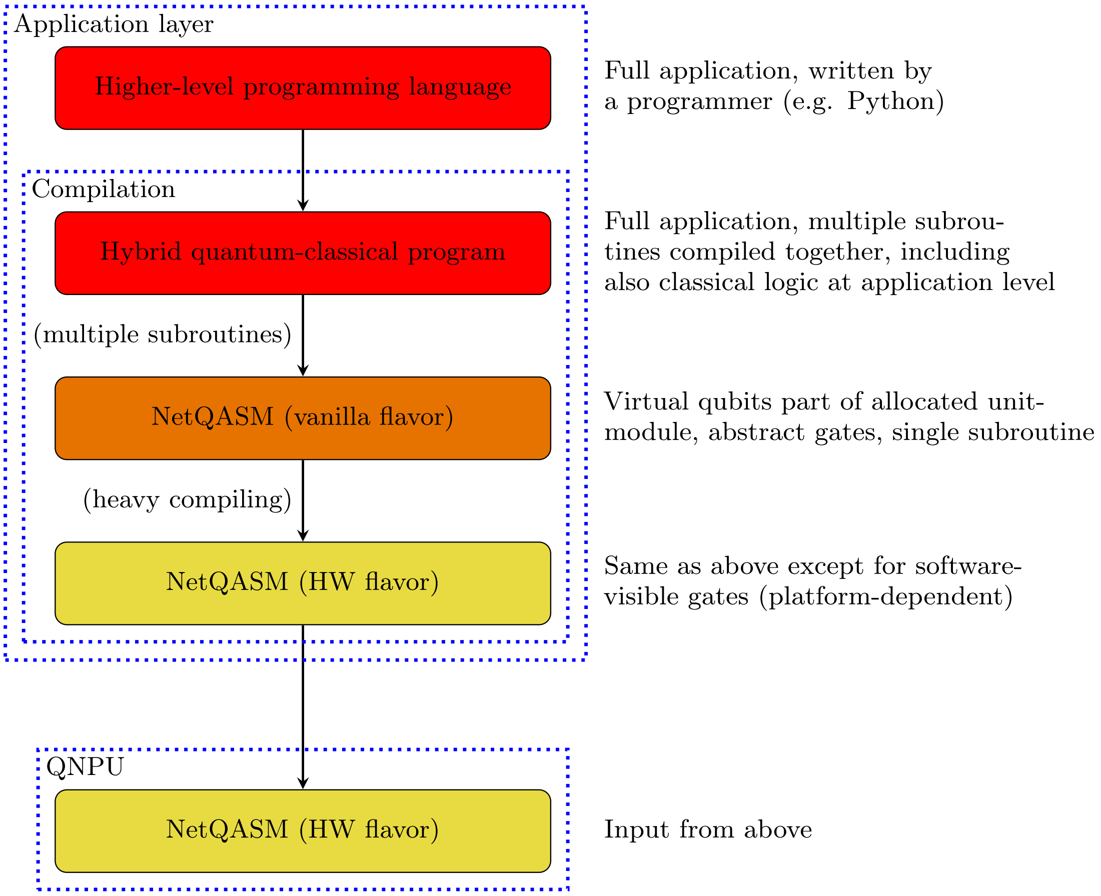

Application layer
Higher-level programming language
Full application, written by a programmer (e.g. Python)
Compilation
Hybrid quantum-classical program
Full application, multiple subroutines compiled together, including also classical logic at application level
(multiple subroutines)
NetQASM (vanilla flavor)
Virtual qubits part of allocated unit-module, abstract gates, single subroutine
(heavy compiling)
NetQASM (HW flavor)
Same as above except for software-visible gates (platform-dependent)
QNPU
NetQASM (HW flavor)
Input from above

Figure 10. Compilation steps from higher-level programming language, to the `NetQASM` **flavor** exposed by the specific platform. What is contained at each level is further specified to the right of the diagram.

unit-modules for allowing multitasking, as explained in section 5. Also, fundamental requirements like remote-entanglement generation and shared memory have been addressed. The remaining considerations, and our solutions, namely platform independence and memory virtualization using unit-modules, are less trivial to evaluate just by looking at the design. Therefore, we focus on the evaluation of these two design decisions.

In our evaluation, we focus specifically on the NV hardware for our nodes. This has two reasons. First, it is a promising hardware platform for quantum network nodes [19] which we know quite well since it is available in the lab, and we have even used `NetQASM` in a simple test case running on nodes based on NV [28]. Second, the NV hardware is interesting since it has a restricted gate set and qubit topology, which is explained in more detail below. Therefore, we expect that the use of unit-modules and an NV-specific flavor makes a difference in terms of runtime quality.

8.1. Unit modules

We ask ourselves the question whether it pays off to expose unit-modules, that is, a qubit topology with gate- and entanglement information. Specifically, we want to know if there are situations where knowing the unit-module gives the application layer an opportunity to optimize the application in a way that is not possible when not knowing the unit-module. If so, we are interested in how much advantage this gives (in terms of the runtime quality defined above).

In the next section we show that there are indeed situations where knowledge of the unit-module is advantageous. It can be that the order in which `NetQASM` instructions are issued in a subroutine is sub-optimal, since virtual qubit IDs may be mapped in such a way that the `QNPU` has to move virtual qubits to different physical qubits in order to execute the instructions. If the application layer does not know this mapping, it cannot know that the instructions are ordered sub-optimally. With knowledge of the unit-module, on the other hand, the application layer can optimize the order and the overall application performance is improved.

We consider a teleportation application where a *sender* program teleports a single qubit to another *receiver* program. It is assumed that the underlying platform is based on NV centers in diamond (NV) and use well-established models for both the noise and operations supported on such platforms, see appendix I. The sender program uses two qubits: one to create entanglement with the receiver (qubit E), and one to send (teleport) to the receiver (qubit T). At some point, the sender measures both qubits, after which it sends the outcomes to the receiver so that it can do the relevant corrections on its received qubit. We assume that the sender program is written in a higher-level language like, like in our SDK (section 7.1), and in such a way that it first issues a measurement operation on qubit T, and then on E. However, due to the differences in characteristics of the physical qubits, as will be explained below, it is more efficient to first do the measurement on E, and then on T. Now we consider two scenarios, namely

- **Unit-modules (UM).** We assume that the sender program is written and executed on a software stack implementating `NetQASM`, which means that the application's view of its quantum working memory is in the form of a unit-module. This unit-module contains information about the above-mentioned hardware restrictions, and therefore a compiler can take advantage of it by re-ordering the measurement operations while generating the `NetQASM` subroutines to be sent to the `QNPU`.
- **No unit-modules (NUM).** In this case the software stack also implements `NetQASM`, but without unit-modules. Specifically, the application sees its quantum memory as just a number of uniform qubits. Therefore, a compiler for this application does not know about the hardware restrictions, and will construct `NetQASM`-subroutines sent to the `QNPU` without doing any optimization and leaves the order of the operations to be performed as they are specified in the high-level SDK.

Let us first go through the steps of the teleportation application:

Sender:

1. Initialize qubit q_t to be teleported in a Pauli state.
2. Create entanglement with *receiver* using qubit q_s.
3. Perform CNOT gate with q_t as control and q_s as target.
4. Perform Hadamard gate on q_t.
5. Measure qubit q_t and store outcome as m_1.
6. Measure qubit q_s and store outcome as m_2.
7. Send m_1 and m_2 to *receiver.*

Receiver:

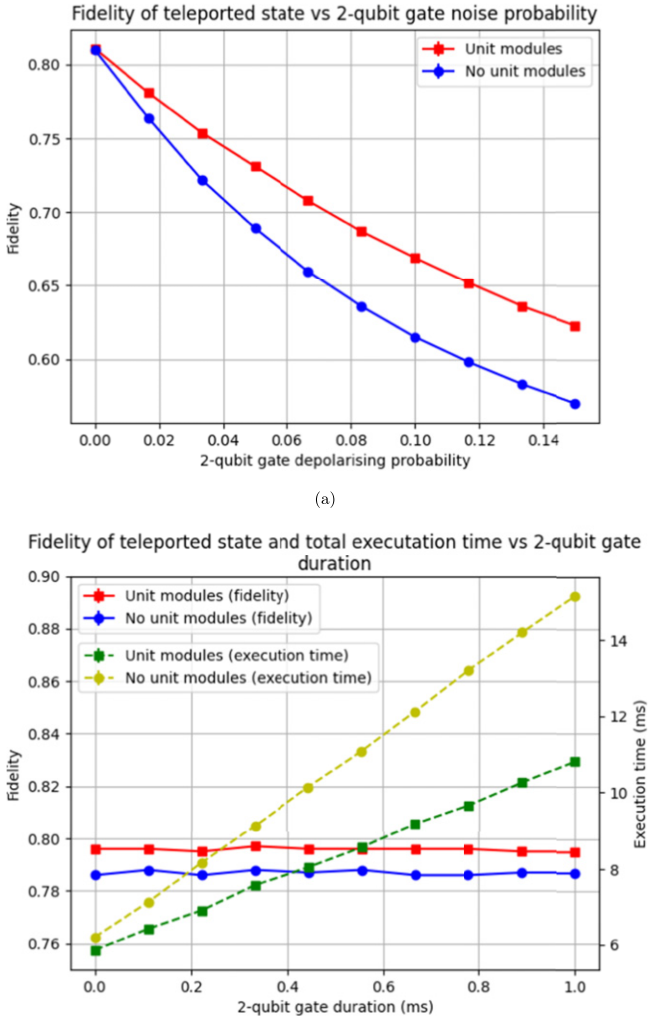
1. Receive entanglement with *sender* using qubit q_r.
2. Receive measurement outcomes from *sender.*
3. Apply correction operations on q_r based on measurement outcomes.

We will now consider the order of the steps of the *sender*. Firstly, we assume that the qubit to be teleported, q_t, is always created before the entanglement. We motivate this assumption below. For this reason, steps 1–3 and 7 are fixed and cannot change. However, we are free to do step 6 before steps 4 and 5, since these single-qubit operations and measurements commute, as long as we are consistent with the outcomes m_1 and m_2. Let us now consider what impact this decision of measuring q_s before q_t or not has on the quality of execution for a NV-platform.

One of the biggest restrictions on a NV-platform is the topology of the qubits. In particular, the NV-platform has a single communication-qubit (electron) surrounded by some number of storage qubits (carbon spins), see for example figure 6. The single communication qubit is not only responsible for any remote entanglement generation but also for any two-qubit gate and is the only qubit that can be directly measured. These restrictions require qubit states to be moved back and forth between the communication qubit and the storage qubits in order to free up the communication qubit, to create new entanglement or to measure another qubit. Since the operation of moving a qubit state is relatively slow on this platform (up to a millisecond [7]) and adds noise to the qubits, it is important to try to minimize the number of moves needed. For more details on the NV-platform, see for example [61] or [21].

In the steps of the *sender* above, the communication qubit is first initialized to a Pauli state. This state is then moved to a storage qubit to free up the communication qubit in order to create entanglement with the *receiver*. Then in step 5, q_t should be measured, which is currently in the storage qubit. This requires the qubit state to first be moved to the communication qubit. However, at this point the communication qubit is occupied by the entangled pair and therefore first needs to be moved to a second storage qubit. Qubit q_t can then be moved to the communication qubit to be measured and then the same is done for q_s, requiring in total four move operations and three physical qubits.

We can now see that performing step 6 before 4 and 5 has the advantage that this qubit is already in the communication qubit and can be measured directly without moving it first. Afterwards, q_t can be moved to



(a)

(b)

Figure 11. (a) Average fidelity between the original state at the sender and the final state at the server, as a function of the depolarizing noise of the native two-qubit gate of the NV-platform, both for the case of performing step 6 after (**NUM**) and before (**UM**) steps 4 and 5. Execution time of the native two-qubit gate is set to 0.5 ms. The rest of the parameters used are listed in appendix I. Each point is the average over each of the six Pauli states as initial state, repeated 100 times. (b) Average fidelity of the teleported state (left y-axis, solid lines) and total execution time of the teleportation application (right y-axis, dashed lines) as a function of the execution time of the native two-qubit gate of the NV-platform, both for the case of performing step 6 after (**NUM**) and before (**UM**) steps 4 and 5. Dephasing parameter of the native two-qubit gate is set to 0.02. The rest of the parameters used are listed in appendix I. Each point is the average over each of the six Pauli state as initial state, repeated 100 times. In both figures, error bars are smaller than the drawn dots.

the communication qubit, which is cleared after the measurement, requiring in total only 2 move operations and only two physical qubits. The decision of performing step 6 before 4 and 5 is highly dependent on the NV-platform and can only be made by a compiler that is aware about these restrictions. The inclusion of unit-modules and qubit types in the `NetQASM`-framework, which are exposed to the compiler at the application layer, allows for these optimization decision and can therefore improve the quality of execution.

For the two scenarios we consider, i.e. performing step 6 before 4 and 5 (**UM**) or not (**NUM**), we check the average fidelity of the teleported state as a function of the gate noise (figure 11(a)), as well as the average fidelity and execution time as a function of gate duration (figure 11(a)), of the native two-qubit gate of the NV-platform. We see that performing step 6 before 4 and 5 improves both total execution time and average fidelity. This can be explained by the fact that using unit-modules allowed a compiler to produce `NetQASM` code containing fewer two-qubit gates. Therefore, an increase in two-qubit gate noice leads to a lower fidelity. Also, an increase in two-qubit gate duration leads to higher execution time difference between the two scenarios. Finally, figure 11(a) shows that the two-qubit gate duration does not affect the final fidelity in this situation, but the difference between using unit-modules versus not using them remains.

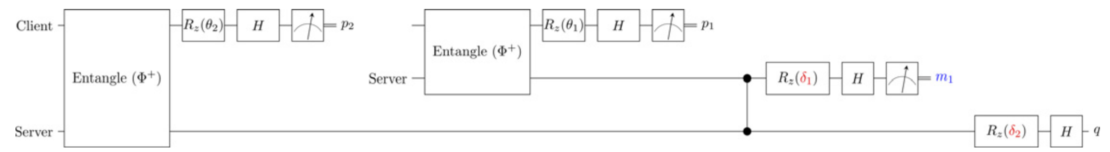


Figure 12. Circuit representation of the simulated BQC application. The client remotely prepares two qubits on the server, by twice creating an entangled pair with the server followed by a local measurement. The server locally entangles its two qubits (cphase gate). Then, the client and server use classical communication to further guide the server's quantum operations. The client computes $\delta_1 = \alpha - \theta_1 + p_1 \cdot \pi$ and sends this to the server. The server uses the received value to do a local rotation and later sends measurement outcome m_1 back to the client. The client then sends $\delta_2 = (-1)^{m_1} \cdot (\beta - \theta_2 + p_2 \cdot \pi)$ to the server. The qubit state q is the result of this application.

8.2. Flavors

While aiming to let NetQASM be mostly platform-independent, we did also choose to allow platform-specific instructions, bundled in flavors. The idea is that this allows for platform-specific optimization leading to better application performance. Here we evaluate if flavors really impact potential performance, and if so how much.

We show that platform-specific optimization can indeed improve application performance, and that there are such optimizations that are not possible without flavors. We see that it has impact mostly on the execution time, but not necessarily on outcome quality.

We consider the blind computation application depicted in figure 12, where both the client and server node implement the NV hardware. Again we compare two scenarios, in this case:

- **Vanilla.** We compile both the client's and server's application code to NetQASM subroutines with the vanilla flavor. The QNPU, controlling NV hardware which does not implement all vanilla gates natively, needs to translate the vanilla instructions on the go. We assume this translation is ad-hoc and does not do any optimizations like removing redundant gates.
- **NV.** The code is compiled to NetQASM subroutines containing instructions in the NV flavor, and redundant gates are optimized away. The QNPU can directly execute the instructions on the hardware.

We implemented this by writing two separate programs in the SDK, one for the client and one for the server. The SDK automatically compiles the relevant parts of these programs into NetQASM subroutines. Classical communication (values δ_1, m_1 and δ_2) is done purely between the two simulated application layers, so these operations are not compiled to NetQASM subroutines. More details about the simulation can be found in appendix I.

The protocol is a verifiable blind quantum protocol [69], which means that the circuit in figure 12 is run multiple times, namely once per round. Some of these rounds are *trap rounds* in which the client chooses a special set of input values. Such a trap round can either succeed or fail, depending on the values returned by the server. The fraction of trap rounds that fail is called the error rate. The error rate should stay low in order for the computation to be blind.

We simulate the BQC application by running the client's and server's programs in SquidASM. We look at the error rate of the trap round as a function of the two-qubit gate noise. The result can be seen in figure 13. It can be seen that using the NV flavor provides a better (lower) error rate than using the vanilla flavor. This can be explained by noting that NetQASM instructions in the vanilla flavor are mapped ad-hoc to native NV gates by the QNPU at runtime, which leads to more two-qubit gates in total.

To gain some more insight into why using the NV flavor provides a lower error rate we also look at the fidelity of the two quantum states on the server before any local gates are applied. As can be seen in figure 12, the client remotely prepares two states on the server by twice creating entanglement and measuring its own half of the EPR pair. In figure 14 we see that already during this remote state preparation phase the NV flavor outperforms the vanilla flavor in terms of the fidelity of the prepared states.

8.3. Relation to other results

We note that a similar question of how many physical details to expose from lower-level layers (in our case the QNPU) to higher-level layers (in our case the application layer) has also been evaluated in [31]. Their conclusion is that exposing and leveraging some of these details can indeed improve certain program success metrics. That result agrees with that of ours, which shows that program execution quality can improve by exposing and leveraging unit-modules and platform-specific NetQASM flavors.

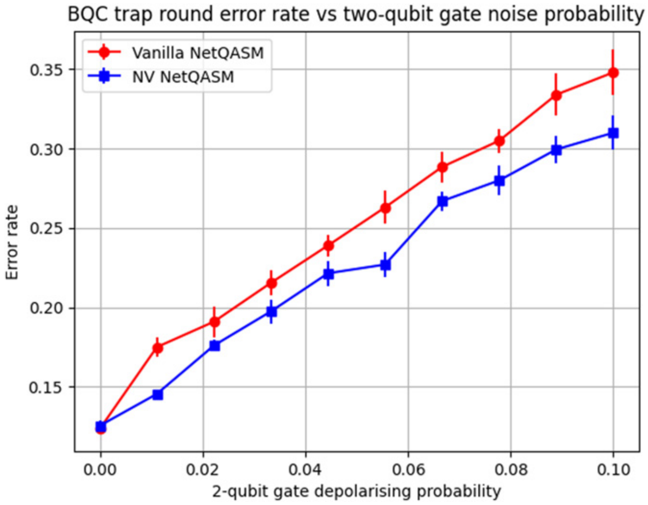


Figure 13. Average error rate of trap rounds for the circuit of figure 12. Each point is the average over four combinations of θ_1 and θ_2, each used in 500 trap rounds. It can be seen that using the vanilla (platform-independent) `NetQASM` flavor results in a worse (higher) error rate on average.

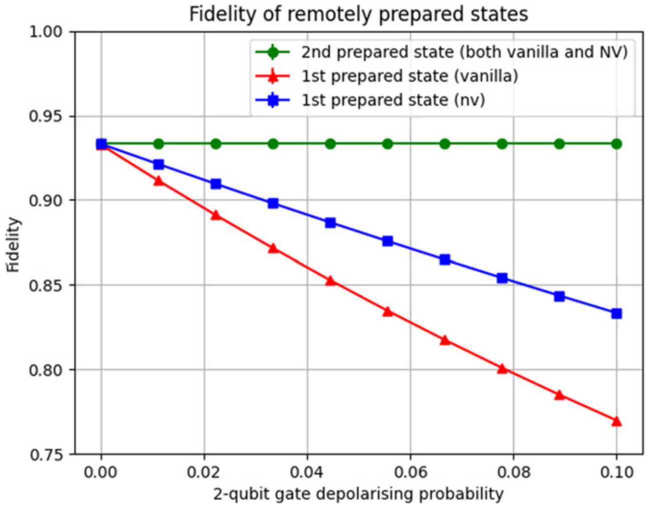


Figure 14. Fidelity of the two remotely prepared states on the server in the BQC application. To remotely prepare a state, the client and server first create an EPR pair, and the client then measures its half in a specific basis while the server keeps its half stored in the communication qubit. This first prepared state is then moved to a memory qubit to free up the communication qubit for preparing the second state. This move operation has a negative effect on the fidelity of the first prepared state. Since the fidelity of the second prepared state only depends on the link entanglement generation, there is no difference between using vanilla or NV instructions. The values are from the same simulation experiment as for figure 13. Error bars are negligible.

9. Conclusion

`NetQASM` enables the development of quantum internet applications in a platform-independent manner. It solves the question of dealing with the complexity of having both classical and quantum operations in a single program, while at the same time providing a relatively simple format for `QNPU`-like layers to handle. Multiple applications, such as remote teleportation and BQC, have already been implemented. A simple compiler has been implemented that can translate code written in the higher-level SDK into `NetQASM`.

Additionally to the work in this paper, we are also developing a physical implementation of the `QNPU`. One key component in this implementation is the *quantum node operating system* (`QNodeOS`), which acts as the bridge between the applications and the physical layer. `QNodeOS` will be presented in a dedicated paper including results of a first integration test between `NetQASM`, `QNodeOS` and underlying physical quantum hardware. This will mark the first time a quantum network node has been programmed using platform-independent code.

Acknowledgments

We thank Arjen Rouvoet and Önder Karpat for valuable discussions. This work was supported by ERC Starting Grant, EU Flagship on Quantum Technologies, Quantum Internet alliance, NWO VIDI.

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://github.com/QuTech-Delft/netqasm-paper-data.

Appendix A. Flow of messages

Here we define the content of each of the messages being sent between the application layer and the QNPU. Each message has an ID chosen by the application layer which is used to associate replies from the QNPU to the application layer.

- RegisterApp. Sent once from the application layer to the QNPU whenever a new application starts. Contains information on what resources are required by the application, in particular:
    - ∗ unit_module_spec. Specification of unit-module needed, e.g. number of qubits.
    - ∗ epr_socket_spec. Specification of EPR sockets needed, see [22], containing (1) EPR socket ID, (2) remote node ID, (3) remote EPR socket ID and (4) minimum required fidelity.
- RegisterAppOK. Returned from the QNPU when application is registered, containing an application ID to be used for future messages.
    - ∗ app_id. Application ID.
- RegisterAppErr. Returned from the QNPU when registration of application failed. For example if required resources could not be met.
    - ∗ error_code. Error code specifying what went wrong.
- Subroutine. Message from the application layer to the QNPU, containing a subroutine to be executed. Details on the content are presented in later sections.
    - ∗ app_id. Application ID.
    - ∗ subroutine. The subroutine to be executed.
- Done. Message from the QNPU to the application layer, indicating that a subroutine has finished. Which subroutine is indicated by the message ID.
    - ∗ message_id. Message ID used for the subroutine-message.
- **Update memory**. The application layer will have access to a copy of the memory allocated by the QNPU for certain registers and arrays, see section 6.2. This memory is read-only by the application layer. Updates to the copy of the memory are performed by the end of a subroutine or if the subroutine is waiting. Furthermore, updates need to be explicitly specified in the subroutine by using one of the return-commands. How the actual update is implemented depends on the platform and can either be done by message-passing or with an actual shared memory. However, the subroutine is independent from this implementation. The application layer will be notified by an explicit message whenever the memory is updated.
- StopApp. Sent from the application layer to the QNPU indicating that an application is finished.

Appendix B. Operands

In this section we give the exact definition of the types of operands used in the NetQASM language. Each instruction of NetQASM takes one or more operands. There are five types of operands, which are listed and described below. Each instruction has a fixed types of operands at each position. The exact operands for each instruction is listed in appendix F. We note also that in the human-readable text-form of NetQASM, there are also *branch variables*. However, these are always replaced by **IMMEDIATE**s (constants), corresponding to the instruction number of the subroutine, before serializing, see appendix C.

The operand types of NetQASM are:

- **IMMEDIATE** (constant). An integer seen as it is value. The following instruction, `beq` *branch-if-equal*, branches to instruction index `12` since the number `0` equals the number `0`.

```
beq 0 0 12
```

In the binary encoding used at [24], **IMMEDIATE**s are `int32`.

- **REGISTER.** A register specifying a register name and a index. The following instruction sets index `0` of the register name `R` to be `0`.

```
set R0 0
```

In the current version of `NetQASM` there are four register names and the indices are relative to the names. They are all functionally the same but are meant to be used for different purposes and increase readability:

∗ `C`. Constants, meant to only be `set` once throughout a subroutine.
∗ `R`. Normal register, used for looping etc.
∗ `Q`. Stores virtual qubit IDs.
∗ `M`. Stores measurement outcomes.

In the binary encoding used at [24], **REGISTER**s are specified by one byte and hold one `int32`.

- **ADDRESS.** Specifies an address to an array. Starts with `@`. The following instruction declares an array of length `10` at address `0`.

```
array 10 @0
```

For more information about arrays, see below. The address here is just an identifier of the array and does not refer to a actual memory address. For this reason `@1` above does not mean the second entry of the declared array but simply a different array. Addresses are relative to the application ID and are valid across subroutines.

- **ARRAY_ENTRY.** Specifies an entry in an array. Takes the form `@a[i]`, where `a` specifies the address and `i` the index. The following instruction stores the value of `R0` to the second entry of the array with address `0`.

```
store R0 @0[1]
```

In the text-form `i` can either be an **IMMEDIATE** or a **REGISTER**, however in the binary encoding used at [24], `i` is always a **REGISTER**. This is handled by the compiler by using a `set`-command before.

- **ARRAY_SLICE.** Specifies a slice of an array. Takes the form `@a[s:e]`, where `a` specifies the address, `s` the start-index (inclusive) and `e` the end-index (exclusive). The following instruction waits for the second to the fourth entry of array with address `0` to become not `null`, see appendix F.7.

```
wait_all @0[1:4]
```

In the text-form s and e can either be an **IMMEDIATE**s or a **REGISTER**s, however in the binary encoding defined used at [24], s and e are always a **REGISTER**s. This is handled by the compiler by using a set-commands before.

Appendix C. Branch variables

The human-readable text-form of NetQASM supports the use of *branch variables*. Branch labels are declared as VAR: before the instruction to branch to. Before serializing a NetQASM-subroutine, all branch variables are replaced with **IMMEDIATE**s corresponding to the correct *instruction index*. Delaying this replacement to the end is useful if the compiler wants to move around instructions. For example if a subroutine is as follows:

```
# NETQASM 1.0
# APPID 0
set R0 0

// Loop entry
LOOP:
beq R0 10 LOOP_EXIT

// Loop body
// If statement
bge R0 5 ELSE
// true block
add R0 R0 1
jmp IF_EXIT
// false block
ELSE:
add R0 R0 2
IF_EXIT:

// Loop exit
jmp LOOP
LOOP_EXIT:
```

Which effectively does the same as the following program written in Python (where the variable i corresponds to the register R0 above).

```
i = 0
while i != 10:
  if i < 5:
    i += 1
  else:
    i += 2
```

After replacing the branch labels the body of the subroutine will instead look:

```
store R0 0
beq R0 10 7
bge R0 5 5
add R0 R0 1
jmp 6
add R0 R0 2
jmp 1
```

Appendix D. Arrays

Classical data produced during the execution of a subroutine are stored in either fixed registers or allocated arrays. Arrays in NetQASM have fixed-length, which is specified when declared using the array-instruction. Each entry of an array is an *optional* **IMMEDIATE**, meaning that the entry is an integer (e.g. int32) or not defined (null). The arrays can be used to collect measurement outcomes to be returned to the application layer but also other data such as information about the generated remote entanglement [21, 22]. All wait-instructions of NetQASM wait for one or more entries in an array to become defined (i.e. not null). The main use-case is for the execution of the subroutine to wait until the quantum network stack of the QNPU has finished generated an entangled pair with a remote node. The subroutine will be waiting for information about the entangled pair to be stored in a given array. Once this is done, the execution can proceed.

The following subroutine for example creates and array with three elements, stores the values `1` and `2` to the array and reads them and adds them up, storing the value in the third entry.

```
// Create two constant registers
set C1 1
set C2 2
// Make an array of three entries
array 3 @0
// Load the constants to the array
store C1 @0[0]
store C2 @0[1]
// Load the array entries to two other registers
load R0 @0[0]
load R1 @0[1]
// Add the registers and store the result in the first
add R0 R1 R0
// Store the sum in the third entry of the array
store R0 @0[2]
```

Appendix E. Qubit address operands

Commands that perform actions on qubits have **REGISTER**-operands which specify the virtual address of the qubit to act on. It is good practice to use register name `Q` for these registers. The following subroutine performs a Hadamard gates on qubits with virtual addresses `0`, `1` and `2`.

```
set Q0 0
set Q1 1
h Q0
h Q1
set Q0 2
h Q0
```

Note that `Q0` is used twice but the value of the register is different.

Appendix F. Instructions

Here we list the current instructions part of the **vanilla flavor** of `NetQASM`. For the most up to date version of the language, refer to [24]. Commands are specified as follows:

- `name`. Description of instruction.
  - ∗ **IMMEDIATE.** Description of op1
  - ∗ **REGISTER.** Description of op2

where `name` is the name of the instruction, followed by the list of operands, specified by their type and description. We note that in the human-readable text-form of `NetQASM`, it is allowed to provide an **IMMEDIATE** for operands that are specified as **REGISTER**. The compiler will then replace these, using the `set`-command.

F.1. Allocation

- `qalloc`. Start using a qubit in the unit-module.
  - ∗ **REGISTER.** The virtual address of the qubit.
- `array`. Creates an array of a certain length (width is fixed)
  - ∗ **IMMEDIATE.** Number of entries in the array.
  - ∗ **ADDRESS.** Address of array

F.2. Initialization

- `init`. Initializes a qubit to $|0\rangle$
  - ∗ **REGISTER.** The virtual address of the qubit.
- `set`. Set a register to a certain value.

- **REGISTER.** The register to assign a value to.
- **IMMEDIATE.** The value to assign.

F.3. Memory operations

- `store`. Stores the value in a register to an index of an array.
  - **REGISTER.** The register holding the value to store.
  - **ARRAY_ENTRY.** The array-entry to store the value to.
- `load`. Loads the value from an index of an array to a register.
  - **REGISTER.** The register to store the value to.
  - **ARRAY_ENTRY.** The array-entry holding the value.
- `undef`. Sets an entry of an array to `null`, see appendix F.7.
  - **ARRAY_ENTRY.** Array-entry to make `null`.
- `lea`. Loads a given address of an array to a register.
  - **REGISTER.** The register to store the address to.
  - **ADDRESS.** The address to the array.

F.4. Classical logic

There are three groups of branch instructions: nullary, unary and binary.

Nullary branching

- `jmp`. Jump to a given line (unconditionally).

IMMEDIATE. Line to branch to.

Unary branching. There are two unary branching instructions: `beq` and `bnz`, which both have the following structure:

- `b{ez,nz}`. Branch to a given line if condition fulfilled, see below.
  - **REGISTER.** Value v in condition expression.
  - **IMMEDIATE.** Line to branch to.

Branching occurs if:

- `bez`: $v = 0$ (branch-if-zero)
- `bnz`: $v \neq 0$ (branch-if-not-zero)

Binary branching. There are four binary branch instructions: `beq`, `bne`, `blt` and `bge`, which all have the following structure:

- `b{eq,ne,lt,ge}`. Branch if condition fulfilled, see below.
  - **REGISTER.** Value 1 v_1 in conditional expression.
  - **REGISTER.** Value 1 v_1 in conditional expression.
  - **IMMEDIATE.** Line to branch to.

Branching occurs if:

- `beq`: $v_0 = v_1$ (branch-if-equal)
- `bne`: $v_0 \neq v_1$ (branch-if-not-equal)
- `blt`: $v_0 < v_1$ (branch-if-less-than)
- `bge`: $v_0 \geqslant v_1$ (branch-if-greater-or-equal)

F.5. Classical operations

There are currently four binary classical operations: addition (`add`), subtraction (`sub`) and addition (`addm`), subtraction (`subm`) modulo a number. The first two have the following structure:

- `{add,sub}`. Perform a binary operation and store the result.
  - **REGISTER.** Register to write result (r) to.
  - **REGISTER.** First operand in binary operation (v_0).
  - **REGISTER.** Second operand in binary operation (v_1).

The second two have an additional operand to specify what module should be taken for the result:

- `{add,sub}m`. Perform a binary operation modulo `mod` and store the result.
  - **REGISTER.** Register to write result (r) to.
  - **REGISTER.** First operand in binary operation (v_0).

  - ∗ **REGISTER.** Second operand in binary operation (v_1).
  - ∗ **REGISTER.** Modulo in binary operation (m).

Binary operations are the following:

- `add`, $r = (v_0 + v_1)$
- `sub`, $r = (v_0 - v_1)$
- `addm`, $r = (v_0 + v_1) \pmod m$
- `subm`, $r = (v_0 - v_1) \pmod m$

F.6. Quantum gates

Single-qubit gates. There is a number of single-qubit gates which all have the following structure

- `instr`. Perform a single-qubit gate.
  - ∗ **REGISTER.** The virtual address of the qubit.

Single-qubit gates without additional arguments are the following.

- `x`. X-gate.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{F1}$$

- `y`. Y-gate.

$$Y = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix} \tag{F2}$$

- `z`. Z-gate.

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{F3}$$

- `h`. Hadamard gate.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{F4}$$

- `s`. S-gate (phase)

$$S = \begin{pmatrix} 1 & 0 \\ 0 & \mathrm{i} \end{pmatrix} \tag{F5}$$

- `k`. K-gate.

$$K = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -\mathrm{i} \\ \mathrm{i} & -1 \end{pmatrix} \tag{F6}$$

- `t`. T-gate.

$$T = \begin{pmatrix} 1 & 0 \\ 0 & \mathrm{e}^{\mathrm{i}\pi/4} \end{pmatrix} \tag{F7}$$

Single-qubit rotations. Additionally one can perform single-qubit rotations with a given angle. The angles `a` are specified by two integers `n` and `d` as:

$$a = \frac{n\pi}{2^d} \tag{F8}$$

These instructions have the following structure

- `rot_{x,y,z}`. Perform a single-qubit rotation.
  - ∗ **REGISTER.** The virtual address of the qubit.
  - ∗ **IMMEDIATE.** `n`, for angle, see above.
  - ∗ **IMMEDIATE.** `d`, for angle, see above.

Single-qubit rotations are the following.

- `rot_x`. Rotation around X-axis.
- `rot_y`. Rotation around Y-axis.
- `rot_z`. Rotation around Z-axis.

Two-qubit gates. There are two two-qubit gates which have the following structure

- `{cnot,cphase}`. Perform a two-qubit operation.
  - ∗ **REGISTER.** The virtual address of the control qubit.
  - ∗ **REGISTER.** The virtual address of the target qubit.

Two-qubit gates are the following.

- `cnot`. Controlled X gate.

- cphase. Controlled Z gate.

Measurement

- meas. Measure a qubit in the standard basis.
  - ∗ **REGISTER.** The virtual address of the qubit.
  - ∗ **REGISTER.** The register to write outcome address to.

Pre-measurement rotations. To measure in other bases one can perform gates/rotations before the measurement. If the same measurement basis is used a lot, one can also make use of pre-measurement rotations which can reduce the amount of communication needed internally in the QNPU. A pre-measurement rotations is specified by either the pmr_xyx, pmr_zxz or pmr_yzy which have the following structure. With any two of the bases X, Y and Z, one can do any rotation.

- pmr_{xyx, zxz, yzy}. Specify a pre-measurement rotation.
  - ∗ **IMMEDIATE.** n0, for angle of first rotation, see below.
  - ∗ **IMMEDIATE.** d0, for angle of first rotation, see below.
  - ∗ **IMMEDIATE.** n1, for angle of second rotation, see below.
  - ∗ **IMMEDIATE.** d1, for angle of second rotation, see below.
  - ∗ **IMMEDIATE.** n2, for angle of third rotation, see below.
  - ∗ **IMMEDIATE.** d2, for angle of third rotation, see below.

If a pre-measurement rotation is specified, then three rotations are performed before measuring using a meas_rot-command, see below. The axes of these rotations as given in the instruction name.

The angles of the rotations are specified by the integers n{0, 1, 2} and d{0, 1, 2} in the same way as for single-qubit rotations. That is, rotation i is done by angle $\frac{\pi n_i}{2^{d_i}}$.

Entanglement generation

There are two commands related to entanglement generation. A node can initiate entanglement generation with another node by using the create_epr-command. This command is *not* blocking until entanglement has been generated but a wait-instruction (see below) can be used to block until certain a certain array has been written to, indicating that entanglement has been generated. The remote node should also provide a recv_epr-command. This command does not initiate the entanglement generation but is used to provide the virtual qubit IDs that should be used for the entangled qubits.

- create_epr. Create an EPR pair with a remote node.
  - ∗ **REGISTER.** Remote node ID.
  - ∗ **REGISTER.** EPR socket ID.
  - ∗ **REGISTER.** Provides the address to the array containing the virtual qubit IDs for the entangled pairs in this request. The value of the register should contain the address to an array with as many virtual qubit IDs stored as pair requested.
  - ∗ **REGISTER.** Provides the address to the array which holds the rest of the arguments of the entanglement generation to the network stack [21, 22]. The value of the register should contain the address to an array with as entries as arguments in the entanglement generation request to the network stack [21, 22] (except remote node ID and EPR socket ID).
  - ∗ **REGISTER.** Provides the address to the array to which information about the entanglement should be written. The value of the register should contain the address to an array with as many entries as $n_{\text{pairs}} \times n_{\text{args}}$, where num_{args} is the number of arguments in the entanglement information provided by the network stack [21, 22].
- recv_epr. Receive an EPR pair from a remote node.
  - ∗ **REGISTER.** Remote node ID.
  - ∗ **REGISTER.** EPR socket ID.
  - ∗ **REGISTER.** Provides the address to the array containing the virtual qubit IDs for the entangled pairs in this request. The value of the register should contain the address to an array with as many virtual qubit IDs stored as pair requested.
  - ∗ **REGISTER.** Provides the address to the array to which information about the entanglement should be written. The value of the register should contain the address to an array with as many entries as $n_{\text{pairs}} \times n_{\text{args}}$, where n_{args} is the number of arguments in the entanglement information provided by the network stack [21, 22].

F.7. Waiting

There are three wait-commands that can wait for entries in arrays to become *defined*, i.e. not `null`. Entries in a new array is by default `null` (*undefined*).

- `wait_all`. Wait for all entries in a given array slice to become not `null`.
  - **ARRAY_SLICE.** Array slice to wait for.
- `wait_any`. Wait for any entry in a given array slice to become not `null`.
  - **ARRAY_SLICE.** Array slice to wait for.
- `wait_single`. Wait for a single entry in an array to become not `null`.
  - **ARRAY_ENTRY.** Array entry to wait for.

F.8. Deallocation

- `qfree`. Stop using a qubit in the unit-module.
  - **REGISTER.** The virtual address of the qubit.

F.9. Return

There are two commands for returning data to the application layer. These commands indicate that the copy of the memory on the application layer side should be updated, see above.

- `ret_reg`. Return a register.
  - **REGISTER.** The register to return.
- `ret_arr`. Return an array,
  - **ADDRESS.** The address of the array to return.

Appendix G. Preprocessing

A subroutine written in text form will first be preprocessed, which does the following:

- Parses preprocessing commands and handles these. Any preprocessing command starts with # and should be before any command in the body of the subroutine. Allowed preprocessing commands are:

∗ a `NetQASM` (required). Sets the `NetQASM` version in the metadata.

```
# NETQASM 1.0
```

∗ `APPID` (required). Sets the application ID in the metadata.

```
# APPID 0
```

∗ `DEFINE` (optional). Defines a macro with a key and a value. Any occurrence of the key prepended by $ will be replaced with the value in the subroutine. Values containing spaces should be enclosed with { }.

```
# DEFINE q 0
# DEFINE add {add @0 @0 @1}
```

First command replaces any occurrence of `$q` with `0` and second `$add` with `add @0 @0 @1`.

Appendix H. Examples

Here we list some examples of programs written in `NetQASM`. In appendix H.1, we show some examples written directly in the `NetQASM`-language. In appendix H.2, we show the corresponding examples, instead written in the Python SDK.

H.1. NetQASM

H.1.1. Classical logic (if-statement)

A subroutine which creates a qubit, puts in the $|+\rangle$ state, measures it and depending on the outcome performs an X-gates such that by the end of the subroutine the qubit is always in the state $|0\rangle$.

```
# NETQASM 1.0
# APPID 0
// Set the virtual qubit ID to use
set Q0 0

// Allocate and initialize a qubit
qalloc Q0
init Q0

// Perform a Hadamard gate
h Q0

// Measure the qubit
meas Q0 M0

// Branch to end if m = 0
bez M0 EXIT

// Perform X gate
x Q0

EXIT:
```

H.1.2. Classical logic (for-loop)

A subroutine which performs a for-loop which body creates a qubit, puts in the $|+\rangle$ state and measures it. The outcomes are stored in an array. In a higher-level language (using python syntax) the below subroutine might be written as follows:

```
ms = [None] * 10

for i in range(10):
  q = Qubit()
  q.H()
  m = q.measure()
  ms[i] = m
```

The equivalent `NetQASM` subroutine is:

```
# NETQASM 1.0
# APPID 0
# DEFINE ms @0
# DEFINE i R0
# DEFINE q Q0
# DEFINE m M0
// Create an array with 10 entries (all null)
array 10 $ms

// Initialize loop counter
store $i 0

// Set the virtual qubit ID to use
set $q 0

// Loop entry
LOOP:
beq $i 10 EXIT

// Loop body
qalloc $q
init $q
h $q
meas $q $m
store $m $ms[$i]
qfree $q
add $i $i 1

// Loop exit
jmp LOOP
EXIT:
```

In the above subroutine DEFINE statements have been used to clarify what registers/arrays correspond to the variables in the higher-level language example above.

H.1.3. Create and recv EPR

This code is for the side initializing the entanglement request.

```
# NETQASM 1.0
# APPID 0
# DEFINE qubits @0
# DEFINE args $1
# DEFINE entinfo @2
// Initilizer side

// Setup array with virtual qubit IDs to be used
// for the EPR pairs
array 1 $qubits
store 0 $qubits[0]

// Setup array to store other arguments to entanglement
// generation request
array 20 $args

// Setup array to store entanglement information
array 10 $entinfo

// Create entanglement
// Remote node ID 0 and EPR socket ID 0
// NOTE that these IMMEDIATEs will be replaced by
// REGISTERs when pre-processing.
create_epr 1 0 $qubits $args $entinfo

// Wait for the entanglement to succeed
// i.e. that all entries in the entinfo array becomes
// valid.
wait_all $entinfo[0:10]

// Measure the entanglement qubit
load Q0 $qubits[0]
meas Q0 M0

// Return the outcome
ret_req M0
```

This code is for the receiving side.

```
# NETQASM 1.0
# APPID 0
# DEFINE qubits @0
# DEFINE entinfo @1
// Receiver side (very similar to the initializer side)

// Setup array with virtual qubit IDs to be used
// for the EPR pairs
array 1 $qubits
store 0 $qubits[0]

# Setup array to store entanglement information
array 10 $entinfo

// Receive entanglement
// Remote node ID 1 and EPR socket ID 0
// NOTE that these IMMEDIATEs will be replaced by
// REGISTERs when pre-processing.
recv_epr 1 0 $qubits $entinfo

// Wait for the entanglement to succeed
wait_all $entinfo[0:10]

// Measure the entanglement qubit
load Q0 $qubits[0]
meas Q0 M0

// Return the outcome
ret_req M0
```

H.2. SDK

Each of the examples in this section are functionally the same as the examples in appendix H.1. A compiler will produce a similar subroutine as the examples in the previous section but might vary depending on the exact implementation of the compiler.

H.2.1. Classical logic (if-statement)

Functionally the same as the `NetQASM`-subroutine (appendix H.1.1).

```python
# Setup connection to backend
# as the node Alice
with NetQASMConnection("Alice") as alice:
  # Create a qubit
  q = Qubit(alice)
  # Perform a Hadamard on the qubit
  q.H()
  # Measure the qubit
  m = q.measure()
  # Conditionally apply a X-gate
  with m.if_eq(1):
    q.X()
```

H.2.2. Classical logic (for-loop)

Functionally the same as the `NetQASM`-subroutine (appendix H.1.2).

```python
# Setup connection to backend
# as the node Alice
with NetQASMConnection("Alice") as alice:
  # Create an array for the outcomes
  outcomes = alice.new_array(10)
  # For-loop
  with alice.loop(10) as i:
    # Create a qubit
    q = Qubit(alice)
    # Perform a Hadamard on the qubit
    q.H()
    # Measure the qubit
    m = q.measure()
    # Add the outcome to the array
    outcomes[i] = m
```

H.2.3. Create and recv EPR

Functionally the same as the `NetQASM`-subroutine (appendix H.1.3).

This code is for the side initializing the entanglement request.

```python
# Setup an EPR socket with the node Bob
epr_socket = EPRSocket("Bob")
# Setup connection to backend
# as the node Alice
with NetSquidConnection(
"Alice",
epr_sockets=[epr_socket],
):
  # Create entanglement
  epr = epr_socket.create()[0]
  # Measure the entangled qubit
  m = epr.measure()
```

This code is for the receiving side.

```python
# Setup an EPR socket with the node Alice
epr_socket = EPRSocket("Bob")
# Setup connection to backend
# as the node Bob
with NetSquidConnection(
"Alice",
epr_sockets=[epr_socket]
):
  # Create entanglement
  epr = epr_socket.recv()[0]
  # Measure the entangled qubit
  m = epr.measure()
```

Table 1. Gate durations for scenario **B** of section 8. t is the value being swept in figure 11(b). All values are from [21].

Gate	Durations (ns)	Explanation
`electron_init`	2×10^3	Initialize a communication qubit (electron) to $\lvert 0\rangle$
`electron_rot`	5	Single-qubit rotation on communication qubit (electron)
`measure`	3.7×10^3	Measure communication qubit (electron)
`carbon_init`	3.1×10^5	Initialize a storage qubit (carbon) to $\lvert 0\rangle$
`carbon_xy_rot`	t	X/Y-rotation on storage qubit (carbon)
`carbon_z_rot`	5	Z-rotation on storage qubit (carbon)
`ec_controlled_dir_xy`	t	Native two-qubit gates, see equations (I1) and (I2)

Appendix I. Simulation details

In this section we detail how simulations in section 8 were performed and what models and parameters were used. All simulations used the NetQASM SDK [24], using NetSquid [26, 60] as the underlying simulator. All code used in these simulations can also be found at [66].

I.1. Noise model

In both the teleportation and the blind quantum computing scenario we used the same model for NV centres in diamonds as was used in [21, 26]. All gates specified by the application in the SDK were translated to NV-specific gates, see table 1, using a simple compiler without any optimization. The parameters used in the model from [21] are listed in tables 1 and 2, together with an explanation and a reference. `ec_controlled_dir_xy` are the native two-qubit gates of the NV-platform, ideally performing one of the unitary operations

$$U_{\text{ecx}}(\alpha) = \begin{pmatrix} R_x(\alpha) & 0 \\ 0 & R_x(-\alpha) \end{pmatrix} \tag{I1}$$

$$U_{\text{ecy}}(\alpha) = \begin{pmatrix} R_y(\alpha) & 0 \\ 0 & R_y(-\alpha) \end{pmatrix} \tag{I2}$$

where $R_x(\alpha)$ and $R_y(\alpha)$ are the rotation matrices around X and Y, respectively. When sweeping the duration and noise of this two-qubit gate the same value is also used for the `carbon_xy_rot` (X- and Y-rotations on the carbon) on the storage qubits, since these are also effectively done with a similar operation also involving the communication qubit (electron). All noise indicated by a fidelity in table 2 are applied as depolarising noise by applying the perfect operation, producing the state ρ_{ideal}, and mapping this to

$$\rho_{\text{noisy}} = (1-p)\rho_{\text{ideal}} + \frac{p}{3}X\rho_{\text{ideal}}X + \frac{p}{3}Y\rho_{\text{ideal}}Y + \frac{p}{3}Z\rho_{\text{ideal}}Z \tag{I3}$$

where X, Y and Z are the Pauli operators in equations (F1) to (F3), $p = \frac{4}{3}(1-F)$, with F being the value specific in table 2. Decoherence noise is specific as T_1 (energy/thermal relaxation time) and T_2 (dephasing time) [76].

I.2. BQC application and flavors

In section 8.2 we simulated the BQC application from figure 12. The code for this is available at [66].

In the scenario when the application code was compiled to subroutines with the vanilla lavour, the QNPU had to map the vanilla instructions to NV-native operations on the fly. We used the gate mappings listed below. For convenience we use PI and PI_OVER_2 for π and $\frac{\pi}{2}$ respectively.

A h (Hadamard) vanilla instruction was mapped to the following NV instruction sequence:

```
rot_y PI_OVER_2
rot_x PI
```

Table 2. Noise parameters for used in the simulations of section 8. f is the value being swept in figures 11(a) and 13. All fidelities are realized by a applying depolarising noise as in equation (I3). All values are from [26], except `link_fidelity` which is set to relatively high value to avoid this being the major noise-contribution and preventing any conclusions to be made.

Parameter	Value	Explanation
`electron_T1`	1 h	T_1 of communication qubit (electron)
`electron_T2`	1.46 s	T_2 of communication qubit (electron)
`electron_init`	0.99	Fidelity to initialize communication qubit (electron)
`electron_rot`	1.0	Fidelity for Z-rotation on communication qubit (electron)
`carbon_T1`	10 h	T_1 of storage qubit (carbon)
`carbon_T2`	1 s	T_2 of storage qubit (carbon)
`carbon_init`	0.997	Fidelity to initialize storage qubit (carbon)
`carbon_z_rot`	0.999	Fidelity for Z-rotation on storage qubit (carbon)
`carbon_xy_rot`	f	Fidelity for X/Y-rotation on storage qubit (carbon)
`ec_controlled_dir_xy`	f	Fidelity for native two-qubit gate
`prob_error_meas_0`	0.05	Probability of flipped measurement outcome for $\|0\rangle$
`prob_error_meas_1`	0.005	Probability of flipped measurement outcome for $\|1\rangle$
`link_fidelity`	0.9	Fidelity of generated entangled pair

A `cnot C S` vanilla instruction between a communication qubit (C) and a storage qubit (S) (as specified in the unit-module) was mapped to the following NV instruction sequence:

```
cx_dir C S PI_OVER_2
rot_z C -PI_OVER_2
rot_x S -PI_OVER_2
```

A `cnot S C` vanilla instruction between a store qubit (S) and a communication qubit (C) (as specified in the unit-module) was mapped to the following NV instruction sequence:

```
rot_y C PI_OVER_2
rot_x C PI
rot_y S PI_OVER_2
cx_dir C S PI_OVER_2
rot_z C -PI_OVER_2
rot_x S -PI_OVER_2
rot_y S PI_OVER_2
rot_y C PI_OVER_2
rot_x C PI
```

A `cphase C S` vanilla instruction between a communication qubit (C) and a storage qubit (S) (as specified in the unit-module) was mapped to the following NV instruction sequence:

```
rot_y S PI_OVER_2
cx_dir C S PI_OVER_2
rot_z C -PI_OVER_2
rot_x S -PI_OVER_2
rot_y S -PI_OVER_2
```

ORCID iDs

Axel Dahlberg https://orcid.org/0000-0003-2479-7424
Bart van der Vecht https://orcid.org/0000-0002-1048-5588
Carlo Delle Donne https://orcid.org/0000-0003-2834-4334
Ingmar te Raa https://orcid.org/0000-0002-9504-9370

References

[1] Wehner S, Elkouss D and Hanson R 2018 Quantum internet: a vision for the road ahead *Science* **362** eaam9288
[2] Bennett C H and Brassard G 1984 Quantum cryptography: public key distribution, and coin-tossing *Proc. 1984 IEEE Int. Conf. Computers, Systems, and Signal Processing* pp 175–9
[3] Childs A M 2005 Secure assisted quantum computation *Quantum Inf. Comput.* **5** 456–66

[4] Buhrman H, Cleve R, Massar S and de Wolf R 2010 Nonlocality and communication complexity *Rev. Mod. Phys.* **82** 665–98

[5] Gottesman D, Jennewein T and Croke S 2012 Longer-baseline telescopes using quantum repeaters *Phys. Rev. Lett.* **109** 070503

[6] Hensen B *et al* 2015 Loophole-free Bell inequality violation using electron spins separated by 1.3 kilometres *Nature* **526** 682–6

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

[8] Moehring D L, Maunz P, Olmschenk S, Younge K C, Matsukevich D N, Duan L-M and Monroe C 2007 Entanglement of single-atom quantum bits at a distance *Nature* **449** 68–71

[9] Hofmann J, Krug M, Ortegel N, Gérard L, Weber M, Rosenfeld W and Weinfurter H 2012 Heralded entanglement between widely separated atoms *Science* **337** 72–5

[10] Kalb N *et al* 2017 Entanglement distillation between solid-state quantum network nodes *Science* **356** 928–32

[11] Inlek I V, Crocker C, Lichtman M, Sosnova K and Monroe C 2017 Multispecies trapped-ion node for quantum networking *Phys. Rev. Lett.* **118** 250502

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

[13] Abobeih M H, Cramer J, Bakker M A, Kalb N, Markham M, Twitchen D J and Taminiau T H 2018 One-second coherence for a single electron spin coupled to a multi-qubit nuclear-spin environment *Nat. Commun.* **9** 2552

[14] Ekert A K 1991 Quantum cryptography based on Bell's theorem *Phys. Rev. Lett.* **67** 661

[15] Kobayashi H, Matsumoto K and Tani S 2014 Simpler exact leader election via quantum reduction *Chicago J. Theor. Comput. Sci.* **2014** 10

[16] Ganz M 2009 Quantum leader election (arXiv:0910.4952)

[17] Cross A W *et al* 2021 OpenQASM 3: a broader and deeper quantum assembly language (arXiv:2104.14722)

[18] Burgholzer L and Wille R 2021 Towards verification of dynamic quantum circuits (arXiv:2106.01099)

[19] Taminiau T H, Cramer J, van der Sar T, Dobrovitski V V and Hanson R 2014 Universal control and error correction in multi-qubit spin registers in diamond *Nat. Nanotechnol.* **9** 171–6

[20] Pompili M *et al* 2021 Realization of a multi-node quantum network of remote solid-state qubits *Science* **372** 259–64

[21] Dahlberg A *et al* 2019 A link layer protocol for quantum networks *ACM SIGCOMM 2019 Conf. (SIGCOMM '19)* (New York: ACM) p 15

[22] Kozlowski W, Dahlberg A and Wehner S 2020 Designing a quantum network protocol (arXiv:2010.02575)

[23] Arpaci-Dusseau R H and Arpaci-Dusseau A C 2018 *Operating Systems: Three Easy Pieces* (Madison, WI: Arpaci-Dusseau Books LLC)

[24] van der Vecht B and Dahlberg A 2021 Git repository with code for NetQASM https://github.com/QuTech-Delft/netqasm

[25] Quantum Network Explorer 2022 https://quantum-network.comhttps://quantum-network.com

[26] Coopmans T *et al* 2021 NetSquid, a network simulator for quantum information using discrete events *Commun. Phys.* **4** 164

[27] Dahlberg A and Wehner S 2018 SimulaQron—a simulator for developing quantum internet software *Quantum Sci. Technol.* **4** 015001

[28] Pompili M *et al* 2021 Experimental demonstration of entanglement delivery using a quantum network stack (arXiv:2111.11332)

[29] Fu X *et al* 2017 An experimental microarchitecture for a superconducting quantum processor *Proc. 50th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO-50 '17)* (New York: Association for Computing Machinery) pp 813–25

[30] Eli Bourassa J *et al* 2020 Blueprint for a scalable photonic fault-tolerant quantum computer (arXiv:2010.02905)

[31] Murali P, Linke N M, Martonosi M, Abhari A J, Nguyen N H and Alderete C H 2019 Full-stack, real-system quantum computer studies: architectural comparisons and design insights *Proc. 46th Int. Symp. Computer Architecture (ISCA '19)* (New York: Association for Computing Machinery) pp 527–40

[32] Wecker D and Svore K M 2014 LIQUi$|\rangle$: a software design architecture and domain-specific language for quantum computing (arXiv:1402.4467)

[33] Khammassi N, Ashraf I, Someren J v, Nane R, Krol A M, Rol M A, Lao L, Bertels K and Almudever C G 2020 OpenQL: a portable quantum programming framework for quantum accelerators (arXiv:2005.13283)

[34] Amy M and Gheorghiu V 2020 Staq—a full-stack quantum processing toolkit *Quantum Sci. Technol.* **5** 034016

[35] Green A S, Lumsdaine P L, Ross N J, Selinger P and Valiron B 2013 Quipper: a scalable quantum programming language *Proc. 34th ACM SIGPLAN Conf. Programming Language Design and Implementation* vol 48 pp 333–42

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

[37] Cross A W, Bishop L S, Smolin J A and Gambetta J M 2017 Open quantum assembly language (arXiv:1707.03429)

[38] Khammassi N, Guerreschi G G, Ashraf I, Hogaboam J W, Almudever C G and Bertels K 2018 cQASM v1.0: towards a common quantum assembly language (arXiv:1805.09607)

[39] Fu X *et al* 2019 An executable quantum instruction set architecture *2019 IEEE Int. Symp. High Performance Computer Architecture (HPCA)* pp 224–37

[40] Liu S, Wang X, Zhou L, Guan J, Li Y, He Y, Duan R and Ying M 2017 $Q|SI\rangle$: a quantum programming environment (arXiv:1710.09500)

[41] Smith R S, Curtis M J and Zeng W J 2016 A practical quantum instruction set architecture (arXiv:1608.03355)

[42] IBM (2020) Qiskit https://qiskit.org/

[43] Google (2020) Cirq https://cirq.readthedocs.io/en/stable/

[44] Microsoft (2020) Q# https://docs.microsoft.com/en-us/quantum/

[45] Jones T, Brown A, Bush I and Benjamin S C 2019 Quest and high performance simulation of quantum computers *Sci. Rep.* **9** 10736

[46] Zulehner A and Wille R 2019 Compiling SU (4) quantum circuits to IBM QX architectures *Proc. 24th Asia and South Pacific Design Automation Conf.* pp 185–90

[47] Häner T, Steiger D S, Svore K and Troyer M 2018 A software methodology for compiling quantum programs *Quantum Sci. Technol.* **3** 020501

[48] Gokhale P, Koretsky S, Huang S, Majumder S, Drucker A, Brown K R and Chong F T 2020 Quantum fan-out: circuit optimizations and technology modeling (arXiv:2007.04246)

[49] Liu L and Dou X 2020 A new qubits mapping mechanism for multi-programming quantum computing *ACM International Conference on Parallel Architectures and Compilation Techniques* (September 2020) pp 349–50

[50] Gokhale P, Javadi-Abhari A, Earnest N, Shi Y and Chong F T 2020 Optimized quantum compilation for near-term algorithms with openpulse arXiv 2004.11205

[51] Ding Y, Wu X-C, Holmes A, Wiseth A, Franklin D, Martonosi M and Chong F T 2020 SQUARE: Strategic quantum ancilla reuse for modular quantum programs via cost-effective uncomputation *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* pp 570–83

[52] Smith R S, Peterson E C, Skilbeck M G and Davis E J 2020 An open-source, industrial-strength optimizing compiler for quantum programs *Quantum Sci. Technol.* **5** 044001

[53] Sivarajah S, Dilkes S, Alexander C, Simmons W, Edgington A and Duncan R 2020 t|ket⟩: a retargetable compiler for NISQ devices *Quantum Sci. Technol.* **6** 014003

[54] Hietala K, Rand R, Hung S-H, Wu X and Hicks M 2019 A verified optimizer for quantum circuits (arXiv:1912.02250)

[55] Zhang Y, Deng H and Li Q 2020 Context-sensitive and duration-aware qubit mapping for various NISQ devices (arXiv:2001.06887)

[56] Niu S, Suau A, Staffelbach G and Todri-Sanial A 2020 A hardware-aware heuristic for the qubit mapping problem in the NISQ era (arXiv:2010.03397)

[57] Bryan D and Di Matteo O 2020 A QUBO formulation for qubit allocation (arXiv:2009.00140)

[58] Pozzi M G, Herbert S J, Sengupta A and Mullins R D 2020 Using reinforcement learning to perform qubit routing in quantum compilers (arXiv:2007.15957)

[59] Nishio S, Pan Y, Satoh T, Amano H and Meter R V 2020 Extracting success from IBM's 20-qubit machines using error-aware compilation *ACM J. Emerg. Technol. Comput. Syst.* **16** 1–25

[60] QuTech 2020 *NetSQUID* https://netsquid.org/

[61] Bernien H 2014 Control, measurement and entanglement of remote quantum spin registers in diamond *PhD Thesis* TU Delft

[62] Ritter S *et al* 2012 An elementary quantum network of single atoms in optical cavities *Nature* **484** 195

[63] Bradley C E *et al* 2019 A 10-qubit solid-state spin register with quantum memory up to one minute *Phys. Rev.* X **9** 031045

[64] Fitzsimons J F 2017 Private quantum computation: an introduction to blind quantum computing and related protocols *npj Quantum Inf.* **3** 23

[65] Berge C 1984 *Hypergraphs: Combinatorics of Finite Sets* (Amsterdam: North-Holland)

[66] van der Vecht B and Dahlberg A 2021 Git repository with code for SquidASM https://github.com/QuTech-Delft/squidasm

[67] Christandl M and Wehner S 2005 Quantum anonymous transmissions *Advances in Cryptology—ASIACRYPT 2005* ed B Roy (Berlin: Springer) pp 217–35

[68] Broadbent A, Fitzsimons J and Kashefi E 2009 Universal blind quantum computation *Proc. 2009 50th Annual IEEE Symp. Foundations of Computer Science (FOCS '09)* (IEEE) pp 517–26

[69] Fitzsimons J F and Kashefi E 2017 Unconditionally verifiable blind quantum computation *Phys. Rev.* A **96** 012303

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

[71] Denchev V S and Pandurangan G 2008 Distributed quantum computing: a new frontier in distributed systems or science fiction? *ACM SIGACT News* **39** 77–95

[72] Brassard G, Cleve R and Tapp A 1999 Cost of exactly simulating quantum entanglement with classical communication *Phys. Rev. Lett.* **83** 1874–7

[73] Bennett C H, Brassard G, Crépeau C, Jozsa R, Peres A and Wootters W K 1993 Teleporting an unknown quantum state via dual classical and Einstein–Podolsky–Rosen channels *Phys. Rev. Lett.* **70** 1895

[74] Dahlberg A 2021 Git repository with code for CQC https://github.com/SoftwareQuTech/CQC-Python

[75] van der Vecht B 2022 Git repository with simulation code and data used for the evaluation in this paper https://github.com/QuTech-Delft/netqasm-paper-data

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