TU Delft

Delft University of Technology

Document Version
Final published version

Licence
Dutch Copyright Act (Article 25fa)

Citation (APA)
Oslovich, S., Van Der Vecht, B., & Wehner, S. (2025). Compilation Strategies for Quantum Network Programs Using Qoala. In C. Culhane, G. Byrd, H. Muller, A. Delgado, & S. Eidenbenz (Eds.), *Technical Papers Program* (pp. 731-741). (Proceedings - IEEE Quantum Week 2025, QCE 2025; Vol. 1). IEEE. https://doi.org/10.1109/QCE65121.2025.00084

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

Compilation strategies for quantum network programs using Qoala

Samuel Oslovich
QuTech, Delft University of Technology and Kavli Institute of Nanoscience
Delft, The Netherlands
s.o.oslovich@tudelft.nl

Bart van der Vecht
QuTech, Delft University of Technology and Kavli Institute of Nanoscience
Delft, The Netherlands
b.vandervecht@tudelft.nl

Stephanie Wehner
QuTech, Delft University of Technology and Kavli Institute of Nanoscience
Delft, The Netherlands
s.d.c.wehner@tudelft.nl

2025 IEEE International Conference on Quantum Computing and Engineering (QCE) | 979-8-3315-5736-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/QCE65121.2025.00084

***Abstract*—Execution of quantum network applications requires a software stack for nodes. Recently, the first designs and demonstrations have been proposed for such software stacks, including QNodeOS and its extension, Qoala. The latter enables compilation strategies previously not possible in QNodeOS. Here, we show how the extensions provided by Qoala can be used by a compiler to improve the performance of quantum network applications. We define new compilation strategies that allow the compiler to influence the scheduling and execution of quantum programs on a quantum network node. Through simulation, we demonstrate that our compilation strategies can reduce the execution time by up to 29.53% and increase the success probability by up to 25.12%. Our work highlights the potential of compiler optimizations for quantum network programs.**

***Index Terms*—quantum network, application, compiler, multi-tasking, scheduling**

I. Introduction

Quantum networks consist of multiple connected quantum processors that use entanglement as a means to transfer quantum information and to realize correlations that are not possible classically. Such networks enable applications [1] not possible on classical networks, including Quantum Key Distribution (QKD) [2], Blind Quantum Computation (BQC) [3], [4], data consistency in the cloud [5], privacy-enhancing proofs of deletion [6], and exponential savings in communication [7].

We envision a future *quantum internet* globally connecting quantum processors, in which independent nodes run their own programs that communicate with other nodes, realizing multi-program applications like those mentioned above. We consider applications all the way up to the most advanced stage [1], including *interactive hybrid quantum-classical* applications [8]. This means that programs consist of a hybrid of classical and quantum code segments, and that these segments may run concurrently, while influencing each other. For example, a quantum code segment may 'pause' halfway, keeping quantum states in memory, while waiting for a classical code segment to receive and process a message from another node. This message could then determine further execution of the quantum code.

To realize quantum network applications in a scalable way, a set of software tools and abstractions is needed. First, arbitrary programs need to be written in a high-level (quantum-) hardware-agnostic programming language, allowing developers to focus on behavior rather than physical details. Second, a compiler must transform program code into efficient executable code. Finally, a runtime environment on a node must execute the compiled code. These constructs allow the development and execution of arbitrary quantum network applications, including applications that are still to be invented.

Recently, advancements have been made regarding architectures that can execute arbitrary quantum network applications on quantum processors. QNodeOS was the first such full-stack system architecture, and it was demonstrated on real quantum hardware that it could execute applications programmed in a high level [9]. Recently, QNodeOS was extended to improve compilation and scheduling, in an extension called Qoala [8].

A. Contributions

We validate that hybrid quantum-classical compilation for network programs can provide performance improvements compared to QNodeOS style program execution (without hybrid compilation). Hybrid compilation refers to compilation applied on the joint quantum-classical program code, rather than on the quantum code and classical code separately. For our example program, hybrid compilation can increase the success probability by up to 25.12%. When considering BQC applications executed on near-term hardware, we identify limiting factors that reduce the improvements of hybrid compilation. We identify new compilation strategies—*block compilation*, *deadline compilation*, and *critical section compilation*—and show that they allow the compiler to influence how programs are executed, leading to improved program performance. We discover scenarios in which block compilation can decrease execution time by up to 29.53%. We evaluate deadline compilation and identify its shortcomings. We identify conditions where critical section compilation can result in a decrease in execution time of up to 6.12%.

The remainder of this paper is structured as follows. Section II provides context and revisits the Qoala architecture, highlighting the most relevant extensions to QNodeOS. In section III we briefly describe the Qoala simulator and relevant quantum hardware parameters. Section IV demonstrates performance improvements made possible through hybrid compilation. In sections V, VI, and VII we evaluate our new compilation strategies.

979-8-3315-5736-2/25/$31.00 ©2025 IEEE
DOI 10.1109/QCE65121.2025.00084

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

II. Background

A. Qoala recap

We briefly summarize the Qoala architecture put forward in [8]. This architecture can execute any arbitrary applications, such as QKD, BQC, or other (new) applications, as long as the application adheres to the Qoala format. *Applications* are realized as a collection (typically two or more) of single-node *programs* that interact with each other. A program runs on a single quantum network node, and its code is organized into *blocks*. Each block is of one of four types:

- *CL (classical local)*: classical computation. A block can contain arbitrary classical code; it is typically executed very fast. Using the classical processor from the QNodeOS experiment, it would take roughly 15ns per classical instruction [9], [10]. See Appendix A for more details.
- *CC (classical communication)*: classical messaging with other programs (on other nodes). E.g. sending a quantum measurement outcome (classical bit). The time to receive a classical message depends on the distance and number of hops between nodes. In experiments in a lab setting (0 hops, directly connected via ethernet), classical communication times have been measured at roughly 155μs [9], [11].
- *QL (quantum local)*: local quantum computation (such as gates and measurement). A QL block is expressed as a NetQASM subroutine [12] containing local quantum instructions. Executing quantum instructions takes much longer than classical instructions (roughly 50μs per quantum instruction in the QNodeOS experiment [9]).
- *QC (quantum communication)*: entanglement generation with other programs (on other nodes). A QC block involves asking the network (see below) for the creation of one or more entangled pairs. Entanglement generation is a slow and non-deterministic process, with the expected time to generate an entangled pair varying based on the distance between nodes and the hardware used. Generating entanglement on current NV-center hardware takes approximately 1s on average (lab setting) [9]. Using anticipated values for trapped ion hardware, the expected entanglement generation time is as low as 15ms on average (lab setting) [13]. For more details see Appendix A.

The programs constituting an application can be developed separately, e.g. a client programming a client program and a server programming a server program. Each node has its own Qoala runtime, which executes Qoala programs. Qoala requires nodes to have a Classical Processing Unit (CPS) (executing CL and CC blocks) and a Quantum Processing Unit (QPS) (executing QL and QC blocks).

A node can also execute multiple programs concurrently. This enables better usage of quantum resources, by e.g. executing (part of) one program while another program is waiting for some event [9].

Qoala assumes the existence of a network control plane (either a central controller or a distributed protocol) [14], which takes an application's demand for entanglement and distributes a network schedule [15] to the nodes. This schedule consists of sequential time slots specifying when nodes are allowed to use the network to generate entanglement. Execution of QC blocks (which request entangled pairs) consists of triggering corresponding generation procedures in the allowed time slots. We refer to these slots as *time bins*, and the duration of each slot as the *bin length*.

Qoala extends the QNodeOS architecture [9], which is also able to (concurrently) execute arbitrary quantum network applications. QNodeOS treats the classical and quantum parts of the program separately, limiting the implementation of hybrid quantum-classical compilation strategies. Furthermore, in the current implementation the CPS and QPS schedulers operate completely independently, preventing advanced scheduling algorithms that oversee execution of the whole program.

Qoala addresses these issues by presenting two new elements. First, Qoala defines a unified, hybrid quantum-classical executable format for quantum network programs. The executable format covers both classical and quantum code (in the form of CL, CC, QL, and QC blocks), enabling full-program code analysis and optimization by a compiler.

Second, Qoala specifies how programs are executed. A single program is *instantiated* one or more times by the user. For each such program instance, program blocks are converted into *tasks* for the instance. For simplicity, in this work we only consider *blocks*. We assume the scheduler directly executes blocks and do not deal with the underlying runtime mechanism of tasks. Blocks are treated as non-preemptable atomic blocks of execution. The scheduler stores these blocks in a graph, which induces precedence and deadline (see below) constraints. When the precedence constraints for a block are satisfied it is considered to be available for execution.

At any point in time, there may be multiple program instances running concurrently in the Qoala runtime of a node. These instances may be of the same program, different programs, or both. A *node scheduler* (hereafter just 'scheduler') decides at runtime which of the available blocks to execute when, based on its scheduling policy. When executing multiple instances concurrently, the scheduler may interleave execution of blocks of different instances.

The scheduler oversees execution of both CPS and QPS blocks. Blocks may specify *deadlines*, either relative to other blocks or as absolute deadlines. These deadlines are transferred to their respective blocks at runtime and may be used by the scheduler. Blocks may also be grouped into *critical sections*, which forces the scheduler to treat such groups as single units of execution.

B. Compilation opportunities

The features mentioned above (hybrid quantum-classical format with blocks, and a node-wide scheduler) enable particular compilation strategies not possible without the Qoala extension of QNodeOS. Indeed, a compiler for Qoala programs may apply optimization techniques across classical and quantum blocks. Moreover, a compiler can guide the scheduler's behavior at runtime: the compiler may insert deadlines, use critical

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

sections, or use a particular division of code into blocks, in order to increase runtime performance. These scheduler-guiding compilation strategies are most impactful in scenarios where multiple programs are running concurrently on the node.

While there has not yet been developed a full compilation stack that converts a high-level program (written in e.g. Python) into the Qoala program format, we already investigate here which different compilation outputs we want to see.

We focus on (1) hybrid quantum-classical compilation, (2) different strategies of dividing program code into blocks, (3) different ways of assigning deadlines to blocks, (4) the impact of (not) using critical sections.

We note that we do this investigation in the context of a quantum internet consisting of autonomous nodes. Therefore, we only consider compilation of single programs (which communicate—using classical communication and entanglement generation—with other programs on other nodes realizing an application), rather than joint compilation of multiple distributed quantum programs. Furthermore, just like in classical multiprocessing systems, we assume the compiler is not aware of any other programs running on the same node. It is up to the runtime environment and its scheduler to orchestrate the concurrent execution of programs. A program compiler can only guide the scheduler as mentioned above.

C. Compilation Strategy Definitions

- **Hybrid Compilation** refers to the new compilation possibilities made available with the unified Qoala program format. For example, with hybrid compilation it is possible to reorder classical and quantum blocks (as long as data dependencies remain satisfied).
- **Block Compilation** determines the number of blocks in a compiled Qoala program. Specific compilation options can be selected to allow for more (or fewer) blocks in a Qoala program.
  - **Selfish Block Compilation** of a Qoala program minimizes the number of quantum local blocks by placing quantum gates in as few blocks as possible. We call this selfish since a program with fewer blocks allows for less interleaving with other programs.
  - **Cooperative Block Compilation** of a Qoala program increases the total number of quantum local blocks by distributing quantum gates over multiple blocks. We call this cooperative since increasing the number of blocks in a program allows for more opportunities for interleaving. N-cooperative block compilation enforces that no more than N quantum gates are in a single block. A block is still allowed to have as many instructions as necessary for loading the quantum memory register and measurement(s).
- **Deadline Compilation** After a Qoala program is split into blocks, the compiler can add deadlines. We consider deadline compilation for an Earliest Deadline First (EDF) scheduler. These (soft) deadlines specify the maximum amount of time that should occur between the completion of a block and the next block.
  - **Deadline Free Compilation**: No deadlines are specified. This results in the first available block being executed by the processing units.
  - **Selfish Deadline Compilation** of a Qoala program assigns the block deadlines such that it is not possible for another block to be inserted between sequential blocks (if multiple are available). This is achieved by setting the deadline equal to the processing time of a single instruction. For quantum blocks the deadline is set to be the *quantum instruction execution time* (see A) and for classical blocks the deadline is set to be the *classical instruction execution time*. We call this selfish since it prevents blocks from other programs from being interleaved.
  - **Cooperative Deadline Compilation** of a Qoala program increases the deadlines to allow for the interleaving of blocks. We call this cooperative since it allows for more interleaving of blocks between program instances. An M-cooperative deadline compilation specifies the deadline of a block to be $M \cdot t_i$, where t_i is the time to complete an instruction of that block type. For quantum blocks we have deadlines set to $M\cdot$(*quantum instruction execution time* + max *gate duration*), and for classical blocks we have deadlines set to $M\cdot$ *classical instruction execution time*.
- **Critical Section Compilation** uses critical sections to enforce a group of blocks to be executed as a single atomic unit. In this compilation method, a critical section begins at the final entanglement generation, and ends at the last quantum measurement. All of the blocks (quantum and classical) in between are part of the critical section. This prevents any other program from executing blocks until the final measurement has been made.

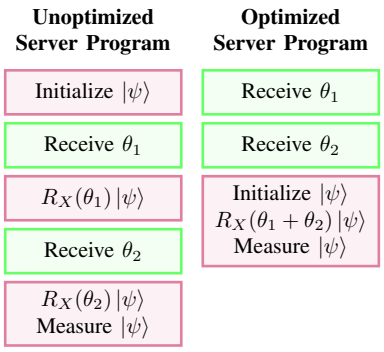
D. Evaluation Metrics

When evaluating application performance we use two different metrics: *execution time* and *success probability*. Execution time is a classical metric that measures how long it takes an application to run from start to finish. A shorter execution time is better. Success probability is a quantum metric that measures how likely an application is to succeed. An application execution is considered a success when it returns an expected outcome, e.g. a Z basis measurement outcome of 0 when our qubit is expected to be in the $|0\rangle$ state. We run many application executions and take the average in order to determine the overall success probability of the application. A larger success probability is better since it gives us greater confidence in our results.

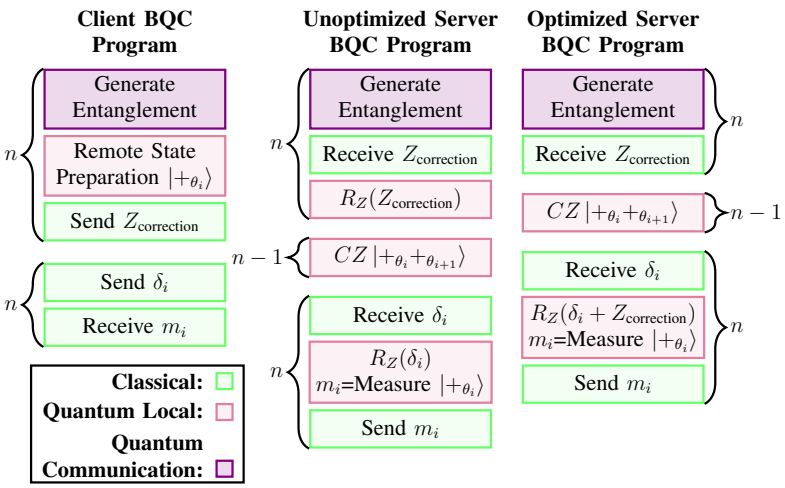
III. Simulation Configuration

For simulating the execution of applications on quantum processing nodes in a quantum network we make use of the simulator used in the Qoala paper [8], [16]. This simulator allows for low level details when simulating the timings between different components in the architecture, such as the classical/quantum instruction execution time, shared memory

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.



(a) An illustration of the optimizations performed on the rotation application for $n = 2$. In the optimized program the classical and quantum blocks have been reordered so that the quantum state $|\psi\rangle$ is not initialized until all of the rotation angles have been received. This minimizes the time the state spends deteriorating in memory. Additionally, the quantum blocks have been combined into a single block to reduce classical processing overhead. The X rotation gates have been combined into a single X rotation, reducing the effects of gate fidelity on the qubit.



(b) An illustration of the client and server programs for an n qubit BQC application. Each bracketed group of blocks is repeated the labeled number of times. The client program: (1) generates entanglement with the server, (2) remotely prepares a state on the server (by performing a local measurement on the entangled qubit), (3) sends the correction for the prepared state, (4) sends the desired measurement angle, δ_i, and (5) receives the measurement outcome from the server. The unoptimized server program: (1) generates entanglement with the client, (2) receives the correction, (3) corrects the prepared state, (4) applies CZ gates to the prepared states (if $n > 1$), (5) receives the measurement angle, δ_i, (6) measures the prepared state, and (7) sends the measurement outcome to the client. In the optimized server program the classical and quantum local operations are reordered, which allows the R_Z gates to be combined.

Fig. 1

processing time, classical communication latency, quantum gate duration, and more. For a detailed description see A.

Our evaluation parameters are based on real and anticipated values for trapped-ion hardware [13]. The default parameters for our trapped-ion hardware configuration are shown in Table I.

We make the following simplifying assumptions: (1) gates operating on the same number of qubits have identical gate duration and fidelity, (2) a gate set of $\{X, Z, H, R_X(\theta), R_Z(\theta), R_Y(\theta), CZ\}$ is supported, (3) gates can be applied between any pair of qubits, and (4) all qubits can be used for entanglement generation.

We model entanglement generation as a stochastic process where each attempt takes a fixed amount of time, and succeeds with some probability. We average our simulation results over 10 different initial random seeds generated using a publicly available quantum random number generator [17].

TABLE I: Default trapped-ion hardware values.

Parameter	Value
Single qubit gate duration	26.6μs [18]
Two qubit gate duration	107μs [19]
Single qubit gate fidelity	0.99 [18]
Two qubit gate fidelity	0.95 [19]
Dephasing (T_2) time	10s [13]
Quantum instruction execution time	50μs [9]
Expected entanglement generation time (lab setting)	15.26ms [13]
Entanglement generation probability (lab setting)	0.013 [13]
Duration of entanglement generation attempt (lab setting)	200 μs [13]
Entangled pair fidelity	0.95 [13]
Classical communication latency (lab setting)	0.155ms [11]

IV. Hybrid Quantum-Classical Compilation

As mentioned in II-A, the Qoala program format enables hybrid quantum-classical compilation. This allows for the reordering and combining of blocks.

A. Rotation Application

Can hybrid compilation improve program performance compared to QNodeOS without hybrid compilation? To explore this, we first propose a simple client-server application where hybrid compilation provides a clear advantage. We call this example the *rotation application*. Our client program sends n angles over a classical communication channel to the server. These angles are chosen at random with the constraint that the angles sum to 2π. Our server program performs R_X gates on a single qubit based on the measurement angles received.

The unoptimized server program, referred to as S_{unopt}, will:

1. Initialize the qubit to one of the six $\pm\{X, Y, Z\}$ basis states (chosen uniformly at random).
2. Receive an angle θ_i from the client and will perform an X rotation by θ_i, repeating this processing n times
3. Measure the qubit in the same basis it was initialized.

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

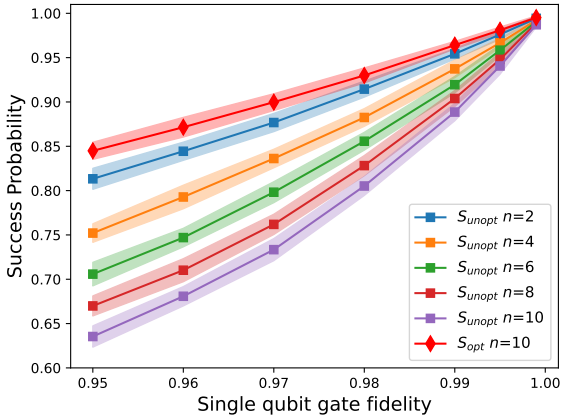


Fig. 2: Success probability of the rotation application for varied single qubit gate fidelities. S_{opt}'s success probability is independent of n, the number of angles sent, hence a single line is shown.

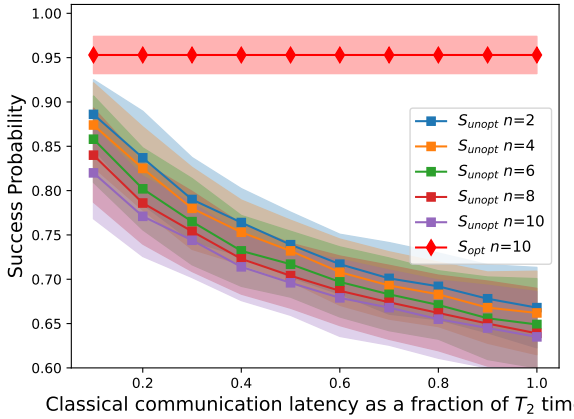


Fig. 3: Success probability of the rotation for varied classical communication latencies. S_{opt}'s success probability is independent of n, the number of angles sent, hence a single line is shown.

Using hybrid compilation, the classical and quantum blocks in the program can be reorganized. This allows for the quantum blocks to be combined.

The resulting optimized server program, called S_{opt}, will:

1. Receive an angle θ_i from the client. Repeat this n times.
2. Sum the received angles $\theta = \sum_i \theta_i$.
3. Initialize the qubit to one of the six $\pm\{X, Y, Z\}$ basis states (chosen uniformly at random).
4. Perform an X rotation by θ.
5. Measure the qubit in the same basis it was initialized.

An illustration of the difference between S_{unopt} and S_{opt} can be seen in Fig. 1a.

Notably, in S_{opt}, there is a single rotation gate, whereas in S_{unopt} the number of rotation gates scales with n. When considering imperfect quantum gates (fidelity < 1), we expect S_{opt} to have a higher success probability than S_{unopt} due to executing fewer (noisy) gates.

1) Simulation Configuration: We separately vary the single qubit gate fidelity and the classical communication latency, comparing how they effect the unoptimized and optimized rotation application variants. For single qubit gate fidelity we sweep over values of $\{0.95, 0.96, 0.97, 0.98, 0.99, 0.995, 0.999\}$. In order to isolate the effect of gate fidelity on success probability, we set all other parameters to the ideal scenario (infinite T_2 time). When varying the classical communication latency we choose values that are a fraction of the T_2 time $\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ in order to see the difference in memory decoherence between S_{unopt} and S_{opt}. We run each program 1000 times and plot the average over our 10 initial seeds.

2) Results: In Fig. 2 we see that S_{opt} outperforms S_{unopt} when gate fidelities are imperfect. The success of S_{opt} is independent of n, the number of angles sent. This independence is due to S_{opt} only performing a single R_X gate, regardless of n. As n increases, so does the improvement in the success probability of S_{opt}, with increases of 1.86%, 5.70%, 9.35%, 12.85%, and 16.11% for $n = 2, 4, 6, 8, 10$, respectively.

TABLE II: Distances and hop counts between nodes in the SURF network [20], [21].

Server	Client	Distance (km)	Hops
Delft 1	Delft 1	0	0
Delft 1	Delft 2	2.2	0
Delft 1	Rotterdam 1	16.8	1
Delft 1	Den Haag 2	19.8	0
Delft 1	Den Haag 1	26.3	1
Delft 1	Leiden 1	30.6	0
Delft 1	Rotterdam 2	33.1	0
Delft 1	Rotterdam 3	40.2	1
Delft 1	Leiden 2	47.9	2
Delft 1	Leiden 3	55.2	3

In Fig. 3 we see that S_{opt}'s success probability is unaffected by the classical communication latency. This is due to S_{opt} receiving all the measurement angles before initializing the qubit. In S_{unopt}, the qubit is stored in memory while waiting for classical communications to be received. As the classical communication latency approaches the T_2 time we see the success probability decrease. Varying the classical communication latency has an equal effect on the execution time of S_{opt} and S_{unopt} since they receive an equal number of classical communications.

B. BQC Application

Can hybrid compilation provide an advantage for more realistic applications? We chose to evaluate BQC, since it is commonly seen as a future application for quantum networks and has been demonstrated in experimental settings [22]–[24], including from QNodeOS [9].

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

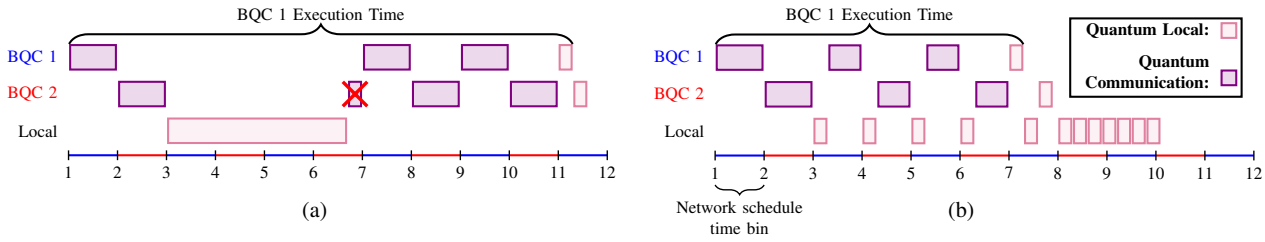


Fig. 4: Shown here is a simplified execution of quantum blocks on the server QPS for two 3 qubit BQC applications and a local program. In (a) the local program is compiled block selfishly, and in (b) the local program is compiled block cooperatively. The network schedule is depicted on the x axis, and the length of each time bin is the expected time to generate a single entangled pair. The color corresponds to which BQC application is allowed to generate entanglement during that bin. In (a) the local program prevents both BQC applications from generating entanglement and increases their execution times. In the time bin from 6-7 the second BQC application fails to generate entanglement due to not enough time remaining in the bin. In (b), block cooperative compilation of the local program allows it to be interleaved with the entanglement generation of the BQC applications. The execution times for BQC are reduced at the cost of increasing the execution time of the local program.

See Fig. 1b for an overview of the BQC protocol and program structure. In our unoptimized BQC server program, two Z rotations are executed on each qubit. One to correct the qubit after remote state preparation, and one before measuring the qubit. In our optimized version of the server program, these two Z rotations are combined into a single rotation by adding the rotation angles, similar to the optimization for S_{opt}.

1) Simulation Configuration: We evaluate BQC applications with a varying number of qubits, $n = \{3, 5\}$. We examine the effects of varying the single qubit gate fidelity, and the distance between the server and client. When varying the single qubit gate fidelity we sweep over values of $\{0.95, 0.96, 0.97, 0.98, 0.99, 0.995, 0.999\}$, and set all other parameters to be perfect (entanglement fidelity of 1.0, two qubit gate fidelity of 1.0, infinite T_2 time). Varying the distance between the client and the server modifies two parameters: the classical communication latency, and the time to generate entanglement. We use a real network topology, SURFnet [20], [21], a research and education network in The Netherlands, and consider placing our server and client at different locations in the network as shown in Table II. We calculate the entanglement generation time as a function of the distance, and the classical communication time as a function of the distance and hop count. For more details, see appendix A.

2) Results: For BQC there is very little increase in success probability for the optimized program when varying the distance (0.42% $n = 3$, 0.86% $n = 5$ on average). The dominating sources of noise are the fidelities of two qubit gates (0.95), and entangled pairs (0.95). When isolating single qubit gate fidelity, and considering all other noise sources to be ideal, there is a larger improvement for the optimized program (0.63% $n = 3$, 1.43% $n = 5$ on average). Similarly, time to generate entanglement is the dominating factor for execution time. The reduction in number of single qubit gates (76.6μs per gate) in the optimized program results in a negligible decrease in execution time when varying the distance (0.52% $n = 3$, 0.63% $n = 5$ on average).

V. Block Compilation

As defined in II-C, block compilation allows programs to influence the scheduler by increasing or reducing the number of blocks their code is split between. We explore three different scenarios, identifying benefits and trade-offs of this compilation method.

A. Scenario 1

We consider a single server S, with c clients $C_1, \ldots C_c$, where each (S, C_i) pair runs its own BQC application. Each client runs a single BQC client program, and the server runs c BQC server programs. Additionally, the server runs a single local program which does the following:

1. Initialize a qubit to $|0\rangle$.
2. Execute 8 single qubit gates on the qubit.
3. Measure the qubit.
4. Repeat steps 1-3 200 times.

The execution time of this local program is approximately 3.5 times the expected duration of generating entanglement (for anticipated trapped ion hardware parameters [13]). In the selfishly block compiled version of the local program, all four steps are contained in a single quantum block, preventing the BQC applications from interleaving (see Fig. 4a). We expect the execution of the selfishly block compiled local program to increase the execution time of the BQC applications by delaying their entanglement generation attempts.

Using 8-cooperative block compilation on the local program, each iteration of steps 1-3 will be contained in a separate block. This enables the BQC applications to interleave entanglement generation while the local program is being executed (see Fig. 4b). We expect this to increase the execution time of the local program, and decrease the execution time of the BQC applications. We do not expect the success probability of the local program to be affected since the qubit is measured and reset in each block.

1) Simulation Configuration: We fix our parameters (see Table I), and vary the length of time bins in the network schedule. Bin length is varied as a multiple of

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

TABLE III: Scenario 1 execution time impacts of compiling the local program 8-block cooperatively vs. block selfishly.

Number of clients (c)	BQC Program Size $n = 3$		BQC Program Size $n = 5$	
	BQC Execution Time Decrease	Local Execution Time Increase	BQC Execution Time Decrease	Local Execution Time Increase
2	29.53% ± 5.95%	191.30% ± 30.21%	23.58% ± 2.06%	313.24% ± 24.55%
3	20.59% ± 6.00%	230.20% ± 59.90%	17.21% ± 2.03%	397.04% ± 65.47%
4	15.93% ± 5.73%	256.84% ± 86.98%	13.60% ± 1.98%	465.26% ± 108.20%
5	13.08% ± 5.26%	280.38% ± 110.27%	11.51% ± 1.89%	518.41% ± 153.44%

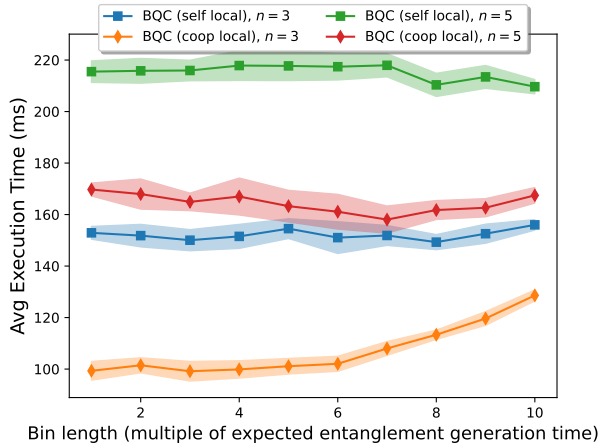


Fig. 5: Block compilation Scenario 1 execution time results for two clients each executing a BQC application with a single server, for varied network schedule bin lengths. The server executes a single instance of local program with varying compilation strategies (selfish block, vs. 8-cooperative block). The BQC applications are compiled block selfishly.

the expected entanglement generation time, with lengths of $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ (times the expected generation time). Time bins follow a fixed repeating pattern that is randomly permuted for each experiment execution. The number of clients, c, is also varied from $\{2, 3, 4, 5\}$. The server node has sufficient quantum memories to execute all of its programs concurrently. Each simulation run executes the experiment 100 times. We plot the average over 10 simulation runs.

2) Results: In Fig. 5, we show the execution time of the BQC application for Scenario 1 with 2 clients. Compiling the local program cooperatively results in a reduced execution time for the BQC applications, 29.53% for $n = 3$, and 23.58% for $n = 5$. This is due to the local program no longer preventing the BQC applications from generating entanglement (as we hypothesized in Fig. 4). This improvement for the BQC applications comes at the cost of an increased execution time for the local program. The interleaving of entanglement generation with the local program increases the execution time of the local program by 191.30% for $n = 3$, and 313.24% for $n = 5$. As we increase the number of clients, the reduction in execution time for the BQC applications decreases (see Table III). Conversely, as the number of clients increases, the increase in execution time of the local program increases.

Our T_2 times are significantly larger than the application execution times ($T_2 = 10$s, execution time <1s), so success probability (for BQC or the local program) is unaffected when varying the compilation strategy for the local program.

For $n = 3$ when running BQC with a cooperatively block compiled local program we see the execution time increase once the bin length exceeds 6. For a bin of length 6, it is highly likely (≈93%) that each BQC application will complete within a single time bin. The other BQC application must wait until its time bin before attempting entanglement generation. As bin length increases past 6, the BQC applications waiting time increases, resulting in longer execution times for BQC.

TABLE IV: Scenario 2 success probability impacts of compiling the local program 8-block cooperatively vs. block selfishly.

Number of clients (c)	Local Success Probability Decrease	
	BQC Program Size ($n = 3$)	BQC Program Size ($n = 5$)
2	4.31% ± 1.28%	7.27% ± 0.50%
3	4.93% ± 1.77%	8.58% ± 1.44%
4	5.26% ± 2.16%	10.33% ± 2.36%
5	5.83% ± 2.56%	10.93% ± 2.69%

B. Scenario 2

We now make a slight modification to the local program seen in Scenario 1 (see V-A). Steps 1 and 2 are repeated 200 times, but only a single measurement is performed at the end of the program. We expect the impact on execution times to be identical to Scenario 1 (see V-A2). However, we now expect the success probability of the local program to decrease when compiled 8-block cooperatively due to the qubit staying live in memory for the duration of the BQC applications.

1) Simulation Configuration: Single qubit gate fidelity is set to 1 in order to isolate the effects of decoherence on our local program. The rest is identical to Scenario 1 (see V-A1).

2) Results: Trends for the execution time are identical to Scenario 1. This is expected since the only difference is where measurements are placed in the local program. The local program experiences a decrease in success probability when compiled block cooperatively (see Table IV), due to the qubit remaining in noisy quantum memory for a longer duration.

C. Scenario 3

We now eliminate the local program, and consider a single server, with c clients, each running a single BQC application. We compare compiling all of the server BQC programs using

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

selfish block compilation vs. 1-cooperative block compilation. The individual blocks in the BQC programs are significantly smaller than the expected time for entanglement generation (roughly 1% or less), so we do not expect to see a decrease in execution time when compiling cooperatively. In fact, we expect a small increase in execution time due to overhead of adding additional blocks. This overhead is due to instructions for loading quantum memory (50μs per instruction) in each additional quantum block.

1) Simulation Configuration: Same as Scenario 1 (V-A1).

2) Results: As expected, compiling the BQC applications on the server block cooperatively vs. block selfishly has a negligible effect on the execution time and success probability. There is a very slight increase in execution time (0.20% $n = 3$, 0.77% $n = 5$, 5 clients), however, this is within the error bars. We observe similar results for 2, 3, and 4 clients.

VI. Deadline Compilation

We consider a single server with c clients, $C_1, \ldots C_c$, where each (S, C_i) pair runs a single BQC application. We envision this scenario like an HPC (high performance computing) environment, where the server is the HPC. We compare the performance of user C_1's BQC application with the average performance of users C_2, ..., C_c. We fix the server compilation strategy of users C_2, ..., C_c to be 100-cooperative deadline compilation. If user C_1 has the default HPC priority (e.g. student access) their program will be compiled on the server with 100-cooperative deadline compilation. However, if user C_1 is a high priority user (e.g. research lab) their program is compiled on the server using selfish deadline compilation. We expect when all users compile with 100-cooperative deadline compilation, there will be no difference in performance between C_1 and the other users. When C_1 uses selfish deadline compilation, we expect their BQC application to have a shorter execution time than the other users since the execution of their program blocks will be prioritized.

1) Simulation Configuration: Our configuration is identical to that of Block Compilation Scenario 1 (see V-A1).

2) Results: Fig. 6 shows that varying user C_1's BQC application compilation strategy on the server does not result in a reduction in execution time. Deadlines have little impact for classical blocks since their execution time is already quite short ($<1\mu$s). For the quantum blocks, these deadlines are almost never used. In BQC before local quantum blocks can be executed, entanglement must first be established. Since entanglement generation can only occur during specific slots in the network schedule, the availability of quantum blocks to be executed is staggered, which causes their deadlines to remain unused. The other user's (C_2, ..., C_5) BQC applications execution time and success probability remain constant regardless of user C_1's compilation strategy on the server. And the success probability for user C_1 is unaffected. Similar results are observed for 2, 3, and 4 total clients. Ultimately, deadlines are irrelevant in this situation, as entanglement generation and the network schedule strongly influence node scheduling.

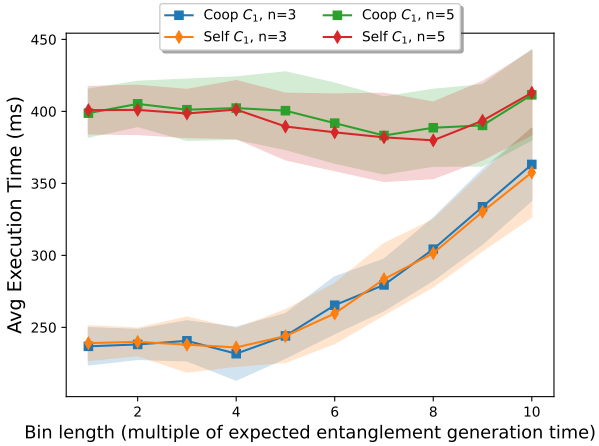


Fig. 6: Deadline compilation execution time results for user C_1's BQC application with a total of 5 clients. User C_1's application is compiled on the server with 100-cooperative deadline compilation vs. with selfish deadline compilation.

The large error bars can be attributed to the network schedule. Consider user C_1's application for $n = 3$ and a network schedule with bin lengths of 3. Once the simulation begins, all applications begin executing immediately. If user C_1's application is first in the network schedule, all entangled pairs will be generated and the application finishes shortly after. However, if user C_1's application is last in the network schedule, it waits for $c - 1$ time bins before generating entanglement and completing execution. Since entanglement generation is the dominating factor in execution time (roughly 88% of execution time for $n = 3$), permutations of the network schedule cause significant variation of execution time.

VII. Critical Section Compilation

We consider an identical setup to deadline compilation (see VI), except now the default compilation strategy for low priority users is to not use critical sections (see II-C). The compilation strategy for high priority users is to compile with critical sections. Using critical section compilation for user C_1's server program enforces that after entanglement generation is completed, the remaining blocks (the last four blocks for the optimized server program in Fig. 1b) will be executed without interleaving. When all of the server programs are compiled without critical sections we expect no difference in execution time or success probability between C_1 and the other users. However, when C_1 is a high priority user compiling with critical sections, we expect C_1's application will experience a decrease in execution time.

1) Simulation Configuration: Our configuration is identical to that of Block Compilation Scenario 1 (see V-A1).

2) Results: Fig. 7 shows no advantage for user C_1 when using critical section compilation. Once entanglement has been generated, the blocks until the final measurement are short (0.33-0.5% of expected entanglement generation time) and

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

TABLE V: Critical section execution time impacts of compiling user C_1's BQC application with vs. without critical sections.

Number of clients (c)	BQC Program Size $n = 3$		BQC Program Size $n = 5$	
	C_1 Execution Time Decrease	Avg C_i Execution Time Increase	C_1 Execution Time Decrease	Avg C_i Execution Time Increase
2	5.05% ± 3.3%	18.90% ± 11.83%	5.85% ± 1.40%	20.58% ± 3.10%
3	5.57% ± 2.81%	16.84% ± 11.65%	6.12% ± 1.66%	18.11% ± 2.79%
4	5.17% ± 3.31%	16.92% ± 11.29%	4.77% ± 1.88%	17.23% ± 2.23%
5	4.38% ± 2.87%	15.44% ± 11.10%	4.99% ± 1.59%	16.51% ± 3.68%

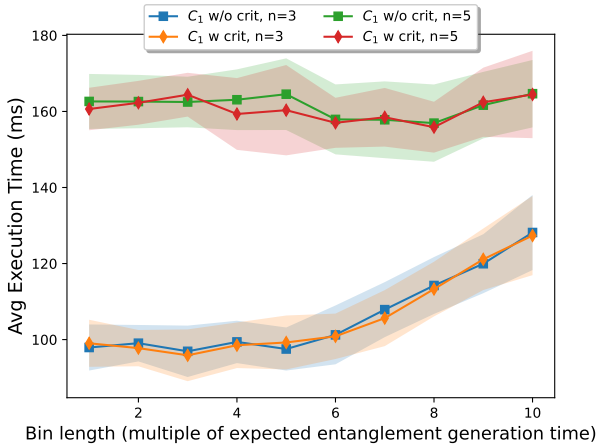


Fig. 7: Critical section compilation execution time results for user C_1's BQC application with a total of 2 clients. User C_1's application is compiled on the server with vs. without critical section compilation.

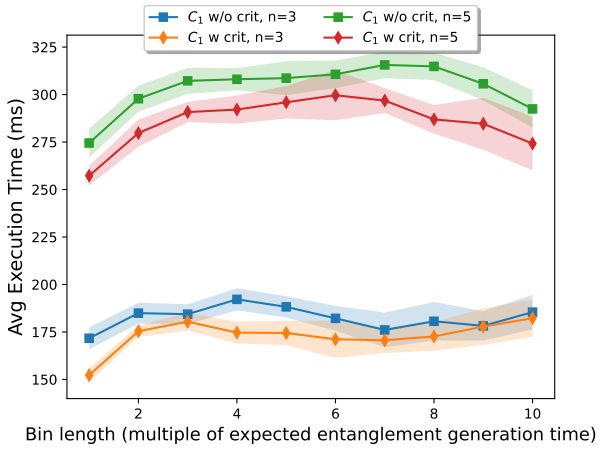


Fig. 8: Critical section compilation execution time results for user C_1's BQC application with a total of 2 clients. Local quantum blocks are modified to have a duration of 12-18% of expected entanglement generation time (0.33-0.5% previously). User C_1's application is compiled on the server with vs. without critical section compilation.

easily interleaved with the blocks of other programs. However, critical sections cause the other server program to have an increased execution time (9.93% $n = 3$, 9.57% $n = 5$, 2 clients). By preventing interleaving, all other BQC programs on the server are forced to halt their execution until C_1's last measurement. We observe similar results for 3, 4, and 5 clients.

We hypothesize that quantum programs with larger quantum blocks (i.e. more quantum gates) would see a benefit from critical sections due to reduced interleaving. We increase size of quantum blocks to vary between 12-18% of expected entanglement generation time (previously 0.33-0.5%) and re-run the simulations, showing the results in Fig. 8. When compiled with critical sections, user C_1's BQC application experiences a decrease in execution time of up to 6.12% (see Table V). Once again, the other BQC applications on the server experience an increase in execution time. The cause of the large errors bars is the same as VI-2.

VIII. Conclusion

Our work opens the door to compiler optimizations for quantum network programs. We demonstrate that our compilation techniques can reduce execution time by up to 29.53%, and increase success probability by up to 25.12% when considering realistic hardware parameters. Further work could include expanding on the present compilation ideas, or developing entirely new ones. It would also be interesting to understand how such techniques perform on other existing applications. Additional hardware constraints, such as limited quantum memories and restrictions on which qubits can be used to generate entanglement also provide avenues for further research.

IX. Acknowledgments

This work was funded by the European Union's Horizon Europe research and innovation programme under grant agreement No. 101102140 – QIA Phase 1. We also acknowledge funding from the NWO VICI grant. We thank Sacha Bernheim, Luca Marchese, and Diego Rivera for valuable feedback on an earlier version of this manuscript.

X. Data Availability

The code and data used to obtain the results of this paper can be found at [25].

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

References

[1] S. Wehner, D. Elkouss, and R. Hanson, "Quantum Internet: A Vision for the Road Ahead," *Science*, vol. 362, no. 6412, pp. 1–9, 2018.

[2] C. H. Bennett and G. Brassard, "Quantum cryptography: Public key distribution and coin tossing," in *Proceedings of the International Conference on Computers, Systems and Signal Processing*, Bangalore, India, Dec. 1984, pp. 175–179. [Online]. Available: https://www.karlin.mff.cuni.cz/~holub/soubory/BB84original.pdf

[3] A. Broadbent, J. Fitzsimons, and E. Kashefi, "Universal Blind Quantum Computation," in *FOCS*. IEEE, 2009, pp. 517–526.

[4] A. M. Childs, "Secure Assisted Quantum Computation," *Quantum Inf. Comput.*, vol. 5, no. 6, pp. 456–466, 2005.

[5] M. Ben-Or and A. Hassidim, "Fast Quantum Byzantine Agreement," in *STOC*. ACM, 2005, pp. 481–485.

[6] A. Poremba, "Quantum proofs of deletion for learning with errors."

[7] P. A. Guerin, A. Feix, M. Araujo, and C. Brukner, "Exponential communication complexity advantage from quantum superposition of the direction of communication," vol. 117, no. 10, p. 100502, publisher: APS.

[8] B. van der Vecht, A. T. Yücel, H. Jirovská, and S. Wehner, "Qoala: an application execution environment for quantum internet nodes," *arXiv preprint arXiv:2502.17296*, 2025.

[9] C. Delle Donne *et al.*, "An operating system for executing applications on quantum network nodes," *Nature*, vol. 639, no. 8054, pp. 321–328, 2025.

[10] A. Fog, *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*, Technical University of Denmark, November 2022, last updated 2022-11-04. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[11] C. Bovy, H. Mertodimedjo, H. Uijterwaal, P. Mieghem, and G. Hooghiemstra, "Analysis of end-to-end delay measurements in internet," 01 2002.

[12] A. Dahlberg, B. van der Vecht, C. Delle Donne, M. Skrzypczyk, I. te Raa, W. Kozlowski, and S. Wehner, "NetQASM—A Low-Level Instruction Set Architecture for Hybrid Quantum–Classical Programs in a Quantum Internet," *Quantum Science and Technology*, vol. 7, no. 3, p. 035023, 2022.

[13] T. Northup, B. Lanyon, V. Krutyanskiy, A. Sorensen, and S. Maiti, "private communications," 2025.

[14] T. R. Beauchamp, H. Jirovská, S. Gauthier, and S. Wehner, "A modular quantum network architecture for integrating network scheduling with local program execution," 2025. [Online]. Available: https://arxiv.org/abs/2503.12582

[15] M. Skrzypczyk and S. Wehner, "An Architecture for Meeting Quality-of-Service Requirements in Multi-User Quantum Networks," 2021.

[16] "Qoala simulator implementation," available from https://github.com/QuTech-Delft/qoala-sim.

[17] C. Lab, "Quantum random number generator (qrng)," accessed: Jan 27, 2025. [Online]. Available: https://camacholab.byu.edu/qrng

[18] G. Avis, F. Ferreira da Silva, T. Coopmans, A. Dahlberg, H. Jirovská, D. Maier, J. Rabbie, A. Torres-Knoop, and S. Wehner, "Requirements for a processing-node quantum repeater on a real-world fiber grid," *npj Quantum Information*, vol. 9, no. 1, Oct. 2023. [Online]. Available: http://dx.doi.org/10.1038/s41534-023-00765-x

[19] V. Krutyanskiy, M. Canteri, M. Meraner, J. Bate, V. Krcmarsky, J. Schupp, N. Sangouard, and B. P. Lanyon, "Telecom-wavelength quantum repeater node based on a trapped-ion processor," *Physical Review Letters*, vol. 130, no. 21, May 2023. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.130.213601

[20] J. Rabbie, K. Chakraborty, G. Avis, and S. Wehner, "Designing quantum networks using preexisting infrastructure," *npj Quantum Information*, vol. 8, no. 1, Jan. 2022. [Online]. Available: http://dx.doi.org/10.1038/s41534-021-00501-3

[21] J. Rabbie and G. Avis, "Repalloc," 2020, Available from: https://github.com/jtrabbie/RepAlloc. [Online]. Available: https://github.com/jtrabbie/RepAlloc

[22] S. Barz, E. Kashefi, A. Broadbent, J. F. Fitzsimons, A. Zeilinger, and P. Walther, "Demonstration of blind quantum computing," *Science*, vol. 335, no. 6066, p. 303–308, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1126/science.1214707

[23] P. Drmota *et al.*, "Verifiable blind quantum computing with trapped ions and single photons," *Physical Review Letters*, vol. 132, no. 15, Apr. 2024. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.132.150604

[24] P.-J. Stas *et al.*, "Distributed blind quantum computing with a two-node quantum network," in *Quantum 2.0 Conference and Exhibition*. Optica Publishing Group, 2024, p. QTu4B.5. [Online]. Available: https://opg.optica.org/abstract.cfm?URI=QUANTUM-2024-QTu4B.5

[25] S. Oslovich, B. van der Vecht, and S. S. Wehner, "Data and code for compilation strategies for quantum network programs using qoala," 2025. [Online]. Available: https://doi.org/10.4121/3fa9be82-ff87-42bd-b00f-f1211a690001

[26] E. J. Maroun, H. E. Hansen, A. T. Kristensen, and M. Schoeberl, "Time-predictable synchronization support with a shared scratchpad memory," *Microprocessors and Microsystems*, vol. 64, pp. 34–42, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933118300905

[27] I. N. Bozkurt, W. Aqeel, D. Bhattacherjee, B. Chandrasekaran, P. B. Godfrey, G. Laughlin, B. M. Maggs, and A. Singla, "Dissecting latency in the internet's fiber infrastructure," *CoRR*, vol. abs/1811.10737, 2018. [Online]. Available: http://arxiv.org/abs/1811.10737

[28] T. Hruby, T. Crivat, H. Bos, and A. S. Tanenbaum, "On sockets and system calls minimizing context switches for the socket api," in *Proceedings of the 2014 International Conference on Timely Results in Operating Systems*, ser. TRIOS'14. USA: USENIX Association, 2014, p. 8.

Appendix

A. Parameter Description

The Qoala simulator [16] allows for low level details when configuring experiments. Below we describe each configurable parameter and the default values we chose for our experiments.

- **Dephasing time** (T_2) is a measure of the amount of dephasing noise a qubit in memory experiences. This value varies for different hardware platforms. We use a T_2 time of 10s for our simulations [13].
- **Gate duration** is the amount of time that it takes to execute a quantum gate (excluding the instruction execution time). These values can be unique for different hardware setups and for different types of gates (e.g. X gate vs. H gate). In our work we make the simplifying assumption that gates operating on the same number of qubits have identical gate durations. For our trapped ion hardware configuration we set the single qubit gate duration to 26.6μs [18] and the two qubit gate duration to 107μs [19].
- **Quantum instruction execution time** is the amount of time that it takes the quantum processing unit (QPU) to execute an instruction (excluding the gate duration). In the QNodeOS experiment [9], the average instruction execution time was 50μs.
- **Gate fidelity** measures how accurately a quantum gate performs its intended operation compared to the ideal operation. The values for gate fidelity can be unique for different hardware platforms and for different gates. In our work we make the simplifying assumption that gates operating on the same number of qubits have identical gate fidelities. For our trapped ion hardware configuration we set the single qubit gate fidelity to 0.99 [18] and the two qubit gate fidelity to 0.95 [19].
- **Network Schedule** specifies sequential time slots when each application being executed in a quantum network is

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.

allowed to generate entanglement. These time slots are referred to as *time bins*, and the duration of each slot is referred to as the *bin length*. In this work we assume that the network schedule follows a fixed repeating pattern. For example when considering a network schedule for three applications, (A,B,C), a pattern may be bin_A, bin_B, bin_C, bin_A, bin_B, ...

- **Bin Length** is how long a time bin in the network schedule lasts. In this work we assume all time bins have identical lengths. We use bin lengths that are an integer multiple of the expected time to generate entanglement.
- **Entanglement generation** is modeled as stochastic process in our simulator. Our process follows a geometric distribution, where each entanglement generation attempt succeeds with some probability, p_{succ} and each attempt takes a fixed amount of time t_{cycle}. These vary for different hardware setups. When attempting to generate an entangled pair, a node will repeatedly execute entanglement generation attempts until an attempt succeeds, or the end of the time bin is reached.
  Below we describe the model and values used in our trapped-ion hardware configuration [13].

$$p_{succ} = \frac{1}{2}\eta_{penalty}\left(\eta_{ion}\eta_{FC}^{ion\to telecom}\eta_{det}^{telecom}\right)^2 * 10^{-(\alpha/10)(d/2)} \quad (1)$$

$$t_{cycle} = t_{class} + t_{prep}^{ion} \quad (2)$$

  - α = 0.2: Attenuation factor at telecom frequency.
  - η_{ion}: The efficiency for emitting and collecting a photon. Anticipated value: 0.5 / 0.87.
  - $\eta_{FC}^{ion\to telecom}$: Frequency conversion efficiency from ion to telecom. Anticipated value: 0.7.
  - $\eta_{det}^{telecom}$: Detection efficiency for photons at telecom frequency. Anticipated value: 0.9.
  - $\eta_{penalty}$: A penalty paid to truncate the detection window, increasing the fidelity. Anticipated value: 0.2 with fidelity 0.95.
  - d: The distance (km) between nodes.
  - t_{cycle}: The average time per generation attempt.
  - t_{class}: Classical communication time.
  - t_{prep}^{ion}: Time to prepare the ion. Anticipated value $200\mu s$.

  The expected time to generate an entangled pair is t_{cycle}/p_{succ}.
- **Classical instruction execution time** is how long it takes the classical processing unit (CPS) to execute an instruction. In the QNodeOS experiment a desktop was used for the CPS (4 Intel 3.20GHz cores, 32 GB RAM, Ubuntu 18.04) [9]. Each clock cycle takes 0.3125ns. Looking at the instruction set for an Intel Pentium, the most expensive single instruction (in terms of clock cycles) took 46 clock cycles [10]. This would take roughly 15ns to execute on a single core of the CPS used in the QNodeOS experiment.
- **Node scheduling messaging processing time** is the communication time between the CPS/QPS schedulers and the node scheduler. As envisioned in the Qoala extension [8], we expect this to be implemented with shared memory. We use a value of 60ns derived from shared memory access taking 192 clock cycles [26].
- **Classical communication latency** is the time it takes to send a classical message from one node to another over the network (e.g. via TCP). We calculate the classical communication latency (CCL) as a function of distance (d) and hop count (h) using the following formula [11], [27].

$$\text{CCL} = h \cdot 244\mu s + 155\mu s + \frac{d}{200,000km/s}$$

- **Classical communication processing time** is the additional time that it takes the CPS to receive a classical message. This requires an additional 478 clock cycles for a nonblocking sysrecvfrom call [28]. This would take the QPS used in the QNodeOS experiment roughly 150ns.
- **SURFnet** is an educational and research network provider in The Netherlands. We use the SURF network topology for the distance and number of hops between nodes [20], [21]. For entanglement generation, we assume that the nodes are directly connected, with the heralding stations centered directly at the midpoint of the link. For classical communication, we assume that each intermediate node adds a hop, resulting in additional classical communication delay.

Authorized licensed use limited to: TU Delft Library. Downloaded on May 07,2026 at 13:10:40 UTC from IEEE Xplore. Restrictions apply.