Programming and executing applications on quantum network nodes

van der Vecht, B.

DOI
10.4233/uuid:6a7acab0-c27f-4dc6-a77f-9996e7bb3a41

Publication date
2025

Document Version
Final published version

Citation (APA)
van der Vecht, B. (2025). *Programming and executing applications on quantum network nodes*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:6a7acab0-c27f-4dc6-a77f-9996e7bb3a41

Programming and Executing Applications on Quantum Network Nodes

Bart van der Vecht

Programming and executing applications on quantum network nodes

Programming and executing applications on quantum network nodes

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. T. H. J. J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
dinsdag 15 april 2025 om 12:30 uur

door

Bart VAN DER VECHT

Master of Science in Computer Science and Engineering,
Technische Universiteit Eindhoven, Nederland,
geboren te Eindhoven, Nederland.

Dit proefschrift is goedgekeurd door de promotoren.

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. S. D. C. Wehner,	promotor
Prof. dr. A. van Deursen,	promotor
onafhankelijke leden:	
Prof. dr. M. Amoretti,	University of Parma, Italië
Dr. T. J. Coopmans,	Technische Universiteit Delft
Prof. dr. ir. R. E. Kooij,	Technische Universiteit Delft
Prof. dr. R. Van Meter,	Keio University, Japan
Prof. dr. K. G. Langendoen,	Technische Universiteit Delft , reservelid

Cover:	Bart van der Vecht, met hulp van Recraft.ai
Style:	TU Delft House Style, met aanpassingen van Moritz Beller `https://github.com/Inventitech/phd-thesis-template`
Gedrukt door:	Ridderprint | `www.ridderprint.nl`

Een digitale versie van dit proefschrift is beschikbaar op `https://repository.tudelft.nl/`.

Contents

Summary

Quantum networks consist of interconnected quantum computers, similar to how 'normal' (*classical*) computers and devices are connected together in networks like the internet. By using quantum mechanical phenomena like superposition and entanglement, quantum networks enable applications that are impossible with classical computers and networks, like extra secure communication and doing quantum computations in the cloud. So far, only simple 'proof of concepts' have been demonstrated on small-scale quantum networks that were optimized specifically for those demonstrations. To improve usage of quantum networks, and accelerate adoption, it is necessary to use (software) abstractions and tools that allow for flexibly programming and executing new applications on the *nodes*—the quantum devices and computers. Here it is important that (software) developers are able to write *computer programs* without requiring knowledge of the underlying (quantum) mechanisms and that nodes possess an *operating system* that is able to execute these programs. Such abstractions and tools do however not yet exist for quantum networks.

In this thesis, we therefore present new system and software architectures that enable, for the first time, programming and execution of arbitrary quantum network programs. This involves a number of challenges. We would like to stay independent of specific quantum hardware; we need to deal with the fact that quantum memory quality decreases (very quickly) over time; and a question is how exactly we should represent and execute the mix of classical and quantum operations.

We present a series of architectures that build on top of each other. First we introduce NetQASM — an *instruction set* for quantum network programs that contains instructions for making entanglement with other quantum devices in the network. Moreover we present a *software development kit* — a toolbox for software developers to program quantum network applications without having to deal with underlying quantum hardware.

Then, we present an *operating system* for nodes — QNodeOS — that is able to execute arbitrary programs that have been programmed using NetQASM. We implement QNodeOS and test it successfully on a real quantum network in the lab consisting of two small quantum computers. Furthermore we show that QNodeOS is able to *multitask* which leads to more efficient usage of the hardware.

We proceed by investigating how to improve the quality of applications by focusing on both *scheduling* and *compilation* of programs. This leads to a new design — Qoala, building on top of QNodeOS — in which a scheduler manages both classical and quantum tasks, and where a compiler can optimize program code and translate it to executable instructions.

Samenvatting

Kwantumnetwerken bestaan uit kwantumcomputers die met elkaar verbonden zijn, net zoals 'normale' (*klassieke*) computers en apparaten verbonden zijn in netwerken zoals het internet. Door gebruik te maken van kwantummechanische verschijnselen als superpositie en verstrengeling, bieden kwantumnetwerken toepassingen die onmogelijk zijn met klassieke computers en netwerken, zoals extra beveiligde communicatie en het doen van kwantumberekeningen in de cloud. Tot nu toe zijn er slechts simpele 'proof of concepts' gedemonstreerd op kleinschalige kwantumnetwerken die speciaal voor die demonstratie waren geoptimaliseerd. Om gebruik van kwantumnetwerken te bevorderen, en daarmee adoptie te versnellen, is het nodig om (software)abstracties en tools te gebruiken waardoor het mogelijk is om verschillende, nieuwe toepassingen flexibel te programmeren en uit te voeren op de *nodes*—de kwantumapparaten- en computers. Daarbij is het belangrijk dat (software)ontwikkelaars *computerprogramma's* kunnen schrijven zonder verstand te hoeven hebben van de onderliggende (kwantum)mechanismes, en dat nodes een *besturingssysteem* hebben dat in staat is zulke programma's uit te voeren. Dit soort abstracties en tools bestaan op het moment echter nog niet voor kwantumnetwerken.

In dit proefschrift presenteren wij daarom nieuwe systeem- en softwarearchitecturen die het voor het eerst mogelijk maken om willekeurige kwantumnetwerkprogramma's te programmeren en uit te voeren. Hierbij komt een aantal uitdagingen kijken. Zo willen we graag onafhankelijk blijven van specifieke kwantumhardware, moeten we rekening houden met het feit dat de kwaliteit van kwantumgeheugen met de tijd (erg snel) afneemt, en is het de vraag hoe we precies de mix van klassieke- en kwantumoperaties moeten representeren en uitvoeren.

We presenteren een reeks architecturen die op elkaar voortbouwen. Eerst introduceren we NetQASM — een *instructieset* voor kwantumnetwerkprogramma's dat instructies bevat voor het maken van verstrengeling met andere kwantumapparaten in het netwerk. Ook presenteren we een *software development kit* — een gereedschapskist voor softwareontwikkelaars om kwantumnetwerktoepassingen te programmeren zonder rekening te hoeven houden met de onderliggende kwantumhardware.

Vervolgens presenteren we een *besturingssysteem* voor nodes — QNodeOS — dat in staat is om willekeurige programma's uit te voeren die zijn geprogrammeerd met NetQASM. We implementeren QNodeOS en testen het met succes op een echt kwantumnetwerk in het lab bestaande uit twee kleine kwantumcomputers. We laten ook zien dat QNodeOS kan *multitasken* waardoor we efficiënter gebruik maken van de hardware.

Daarna onderzoeken we hoe we de kwaliteit van toepassingen kunnen verbeteren door te focussen op zowel *scheduling* (taakplanning) als *compilatie* (vertaling) van programma's. Dit leidt tot een nieuw ontwerp — Qoala, dat voortbouwt op QNodeOS — waarin een *scheduler* het overzicht houdt over zowel klassieke- als kwantumtaken en waarin een *compiler* programmeercode kan optimaliseren en vertalen naar uitvoerbare instructies.

Acknowledgments

This thesis is the result of more than four years of research and development, and would not have been possible without the help of many people. During my PhD at QuTech, I have had the privilege of being surrounded by a diverse group of smart, interesting, and hard-working people. All people who I've interacted with have shaped, either directly or more indirectly, the way I've worked through, sometimes battled through, my PhD.

First of all, many thanks to my promotor **Stephanie Wehner**. It was an honor to have been a part of working towards your vision for a quantum internet. I admire your ability to keep up many, many balls (although I'm not jealous of your busy calendar) and I enjoyed the balance you gave me between freedom and responsibility as well as giving very valuable feedback and guidance when needed. Although I've at times felt more an engineer than a scientist, you have always been supportive and motivated me about continuing the PhD, even when I had serious doubts. I look forward to see where your ambition for the quantum internet leads to, and I'm glad I'll remain a part of the wider endeavour.

I would also like to thank my committee members for taking the time to read through my thesis and subject me to critical questions during my defense: **Arie van Deursen**, acting as second promotor; **Rodney van Meter**, joining online all the way from Japan; **Michele Amoretti**, who I've had the pleasure of interacting with in various QIA- and compiler-discussion-related settings; **Tim Coopmans**, who was a fellow PhD student at QuTech when I joined and I have the honor to have him in this new role; **Rob Kooij**, who specifically moved his calendar around so he could join the defense; and **Koen Langendoen**, for being available as a reserve member.

Thanks to **Soham Chakraborty** and **Johannes Borregaard** for being in my Go/No Go commitee.

Of course many thanks to my wonderful paranymphs, **Carlo** and **Francisco**. Carlo, it has been a pleasure sharing an office for a long time. You really have a genuine interest in other people. Francisco, I admire your sharp wits! Looking forward to all the use cases you'll cook up for the quantum internet.

Special thanks to **Axel** who, in the chaotic situation of the initial lockdown and a supervisor on maternity leave, managed to on-board me and inspire me in my first months of the PhD. Your work has been a great starting point for my own PhD.

Other people in my group that I've had the pleasure of interacting with during the beginning of my PhD include **Aram**, **Matt**, **Guus** and **David**.

Álvaro, **Bethany**, **Scarlett**, you're all nearing the end of your PhD, and you can all be proud of what you've achieved! **Félix**, **Janice**, **Jeroen**, **Kaushik**, **Luca**, **Margrete**, **Soubhadra**, **Sounak**, **Thomas**, **Tzula**, thanks for being awesome group members, and I'm expecting great things from all of you. **Diego**, with your enthousiasm for code development you have been a great addition to our group. **Sam** and **Sacha**: I'm looking forward to the cool things you're going to do with Qoala!

Not only my own group but also other people at QuTech have inspired me. **Eric**, **Fenglei**, **Hemant**, **Kenneth**, **Sébastian**, **Siddanth**, thanks for being either great office neighbours, good chess opponents, or just nice colleagues!

Special thanks to **Mariagrazia**, I really enjoyed doing the QNodeOS experiments together and we can be very proud about the result! Also thanks to **Przemek**, **Tracy** and all other co-authors of the QNodeOS paper.

I also feel very privileged for all the trips abroad that I was allowed to make, for conferences and research visits. From Bad Honnef to Barcelona, from Austin to Paris, and from Bali to Seattle, those trips always left me inspired and with renewed energy. Special thanks to **Carmina** and **Anabel** for hosting me in Valencia. **Aritra**, **Gayane**, **Luise**, **Matt**, **Medina**, **Pablo**, and **Sebastian**, I had a lot of fun with you in Seattle.

Thanks to the master students and interns that I supervised - you all helped me in my own work. **Bob**, your project really helped kickstart the Qoala project. **Atak**, it was a pleasure working with you and you contributed greatly to the Qoala simulator. **Hana**, thanks for being such a great master student that I actually didn't have to do much myself! I'm looking forward to working more with you in the Stack Team.

Speaking about the Stack Team, **Guilherme**, **Ingmar**, **Thom** and **Wojciech**, you all have been crucial in the development of QNodeOS, and I'm glad I can keep working with some of you in the future!

Thanks also to all other software engineers at QuTech that have contributed to projects I've been involved in. **Michal**, thanks for taking over the SquidASM development! It's in very capable hands. Thanks to **Animesh**, **Bobby**, **Fer**, **Giuliano**, **Ivo**, **Mark**, **Olaf**, **Ravi V** and other software engineers that have helped with QNE. **Paul** and **Ravi B**, we haven't worked on any projects together, but I enjoyed hanging out with you in Leuven.

A big shout-out to the **QuTech band (T2 stars, formerly Q2)**. Thanks for the great rehearsals every Wednesday and all the awesome performances that we did. Over the years there have been too many members to list them all here, but one thing is for sure: you're all insanely talented and I immensely enjoy playing with you.

This PhD would also not have been possible without great ways to spend time outside of QuTech. Special thanks to my fellow **Tiramisu** band members, **David**, **Floris** and **Pim**. I can always really unwind by making music and having fun together with you. We'll make that album one day! **Grace**, also with you I really enjoyed playing piano together—especially in the lockdown period this was a great way to distract myself from the PhD. Shout-out to the **DSBA-USSR badminton club** for keeping me fit and helping me meet new friends. Thanks also to **Arminius** for the inspiring events as well as cultural trips to Romania and Suriname. I'd also like to mention **Boris**, **Dirk**, **Jonas**, **Luke**, **Marianne**, **Mark** and **Shuham** for intellectual (but also just very fun) philosophical discussions in our book club.

Elke, **Jeroen**, **Rob**, **Sjoerd** en **Tom**, ik ben blij dat we nog steeds onze vriendengroep hebben sinds de middelbare school! Uiteraard noem ik ook mijn familie, **Papa**, **Mama** en **Peter**, op wie ik altijd kan terugvallen. Ook vond ik altijd steun bij **Jan**, **Annet**, **Opa**, **Oma**, **Lense**, **Machteld**, **Hugo**, en **Martijn**. En als laatste natuurlijk mijn lieve **Anne**, mijn grote steun, met jouw liefde kan ik alles aan.

Curriculum Vitæ

Bart van der Vecht

2020 – 2025 Ph.D. in Quantum Computer Science
QuTech, Delft University of Technology, The Netherlands
Thesis: *Programming and executing applications on quantum network nodes*
Promotors: Prof. dr. S. D. C. Wehner, Prof. dr. A. van Deursen

2016 – 2020 M.Sc. in Computer Science and Engineering
Eindhoven University of Technology, The Netherlands
with an exchange at Universitá della Svizzera Italiana, Switzerland
Thesis: *Secure delegated storage with quantum protocols*
Advisors: Dr. B. Škorić, Dr. S. Wolf

2013 – 2016 B.Sc. in Software Science
Eindhoven University of Technology, The Netherlands
Software End Project: *Udapt*
Advisors: Dr. ir. T. Verhoeff, Dr. L. Somers

1996/01/02 Born in Eindhoven, The Netherlands

List of Publications

6. **B. van der Vecht***, S. Oslovich*, D. Rivera, and S. Wehner. *Qoala Compiler.* In preparation.
   This article is included in this thesis as Chapter 6.

5. **B. van der Vecht***, A. T. Yücel, H. Jirovská, and S. Wehner. *Qoala: an Application Execution Environment for Quantum Internet Nodes.* arXiv preprint:2502.17296. DOI: https://doi.org/10.48550/arXiv.2502.17296.
   This article is included in this thesis as Chapter 5.

4. C. D. Donne*, M. Iuliano*, **B. van der Vecht***, G. M. Ferreira, H. Jirovská, T. van der Steenhoven, A. Dahlberg, M. Skrzypczyk, D. Fioretto, M. Teller, P. Filippov, A. R.-P. Montblanch, J. Fischer, B. van Ommen, N. Demetriou, D. Leichtle, L. Music, H. Ollivier, I. t. Raa, W. Kozlowski, T. Taminiau, P. Pawełczak, T. Northup, R. Hanson, and S. Wehner. *An operating system for executing applications on quantum network nodes.* In: Nature **639** 321—328 (2025). DOI: https://doi.org/10.1038/s41586-025-08704-w.
   This article is included in this thesis as Chapter 4.

3. M. Pompili*, C. Delle Donne*, I. te Raa, **B. van der Vecht**, M. Skrzypczyk, G. M. Ferreira, L. de Kluijver, A. J. Stolk, S. L. N. Hermans, P. Pawełczak, W. Kozlowski, R. Hanson, and S. Wehner. *Experimental Demonstration of Entanglement Delivery Using a Quantum Network Stack.* In: Nature Partner Journal (npj) Quantum Information 8.1 (2022), p. 121. DOI: https://doi.org/10.1038/s41534-022-00631-2.

2. QuTech. *Quantum Network Explorer Software Development Kit (QNE SDK).* Non-scientific publication. https://www.quantum-network.com. 2021. This work relates to NetQASM (Chapter 3).

1. 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.* In: Quantum Science and Technology 7.3 (2022), p. 035023. DOI: https://doi.org/10.1088/2058-9565/ac753f.
   This article is included in this thesis as Chapter 3.

*Main author

1

Introduction

A world without computers and computer networks can hardly be imagined in our modern day. From basic tasks like communication and information retrieval to more complex functions such as running businesses, managing global logistics, and advancing scientific research, we rely on these computers and networks to work efficiently and stay connected. Computers are *programmable*, meaning that one can give them an arbitrary list of instructions — a recipe if you will — according to which they perform a large variety of functions. When connected through networks, computers can share data, resources, and applications across the world, amplifying their power and functionality. This combination of programmability and networking has given rise to technologies like the internet, cloud computing, and distributed systems, all of which form the foundation of today's digital world and has had a transformative impact on our society [7].

Quantum mechanics may seem, at first glance, unrelated to computers and networks, as it primarily describes how nature behaves at the sub-atomic level. However, quantum computers and quantum networks present opportunities for applications that are impossible on non-quantum (or *classical*) computers and networks. While classical, digital, computers and networks represent data as 0s and 1s (bits), quantum computers use quantum bits (qubits) that can exist in multiple states simultaneously, thanks to the principles of superposition and entanglement. This allows quantum computers to perform certain calculations much faster than classical computers, such as simulation of molecular structures, fast searching, optimization algorithms, and machine learning [15, 34]. Quantum networks use quantum entanglement to realize quantum connections between its *nodes*, which are quantum (computing) devices. Using such remote entanglement as a quantum connection, quantum networks enable applications [46] including data consistency in the cloud [3], privacy-enhancing proofs of deletion [37], exponential savings in communication [20], or secure quantum computing in the cloud [4, 9]. Scaling up of quantum networks is expected to result in a global *quantum internet*, in which, similar to the classical internet, arbitrary quantum devices can participate and communicate with each other.

To transform a theoretical idea for a new application or use-case into an actual working implementation on computers and networks, a set of abstractions is essential. These abstractions simplify complex software and hardware operations and allow developers to

focus on designing functionality rather than dealing with low-level details. These software abstractions include: programming languages to write the code, compilers to translate that code into machine-readable instructions, and runtime environments (such as operating systems) to execute the application. These tools, which are widespread for classical computing and networking [1, 23, 42], enable the efficient use of computing and networking resources, turning high-level ideas into executable software.

For quantum computers, similar abstractions have been developed, especially in the last decade. These include low-level instruction sets [28] for quantum circuits, higher-level languages for representing hybrid classical-quantum code [21], compilers [10] and full-stack runtimes for hybrid classical-quantum execution [2]. Without such abstractions and tools, it would be nearly impossible to realize and scale innovative applications, whether in classical or quantum computing and networking.

1.1 Problem statement

For quantum networks and for a future quantum internet, however, abstractions like languages, compilers and runtimes are at present virtually non-existing. Indeed, it is currently not possible for developers to program and execute arbitrary quantum network applications without having to deal with hardware-specific details.

Although small-scale quantum networks linking multiple quantum computing devices have recently been realized as physics experiments in laboratories [22, 24, 27, 32, 36, 39, 43, 44] and fiber networks [25, 29, 45], these demonstrations so far relied either on ad-hoc software, or chose to establish that hardware parameters were in principle good enough to support a given quantum network application, although the application itself was not realized [30, 35, 47]. These experiments are highly technical, requiring manual manipulation of microwave pulses, lasers, and extensive knowledge of quantum physics.

In order to achieve, for quantum networks, the same level of programmability as for classical computers, classical networks and quantum computers, what is missing is hence a framework that enables (1) programming quantum network application logic in a high--level, hardware-agnostic way (2) compilation of application code into low-level executable (quantum network node) machine code, and (3) execution of arbitrary applications on quantum network nodes on different kinds of hardware.

It is true that one can re-use existing abstractions from quantum computing, as well as from classical computing and networking, in order to realize the above framework. However, quantum network applications present unique challenges that are not addressed by existing solutions.

Challenge 1. First, at the time of starting this work there simply did not exist a representation for quantum network applications, high-level nor low-level. An initial approach at addressing this was presented with the CQC interface [14], but this is more a protocol for communication between classical and quantum devices rather than a unified programming model. Existing quantum computing languages and compilers [21, 28] do not support instructions for creating entanglement with remote nodes, while these are crucial for quantum network applications.

Challenge 2. Furthermore, there is not yet a hardware-agnostic framework for quantum network programming. Just as with classical and quantum computing, such an abstraction is needed to shield developers from low-level hardware details (especially since

there are multiple promising technologies including color centers in diamond [40] and trapped ions [33, 38]), enabling them to focus on application logic.

We should first highlight the difference between the quantum network applications we consider in this thesis and *distributed quantum computing* (DQC) [6]. The latter is a model of computation where multiple quantum computers (nodes in a network) work together to perform a quantum computation. In this model, a quantum program (such as a single quantum circuit) is distributed (for example using circuit cutting [8]) over separate quantum computers, which use entanglement between each other in order to jointly realize the original computation. Typically, there is a single entity that manages the circuit distribution as well as scheduling of entanglement between nodes. For DQC there have in the recent years been developments in the creation of frameworks to program and compile such distributed quantum computations [8, 11, 18].

By contrast, the quantum network applications we consider in this work involve independent nodes, each of which determines which programs to run, how to compile them, and how to schedule tasks. This is similar to the classical internet, where nodes run applications autonomously and decide when and how to interact with others. Indeed, our model focuses more on applications for a future quantum internet, rather than on DQC. With this distinction clear, we can consider other challenges in realizing the aforementioned needed framework for quantum network (or internet) applications.

Challenge 3. Quantum network applications consist of a hybrid of classical and quantum code segments. While such hybrid code is also seen in quantum computing — such as *variational quantum eigensolvers (VQE)* [16, 31] or *quantum approximate optimization algorithms (QAOA)* [17] — these typically alternate between classical code and quantum circuits, never leaving quantum memory 'live' while doing extensive classical computations. By contrast, quantum network applications are more *interactive*: classical and quantum code segments may run concurrently, communicating and influencing each other in real-time. For instance, a quantum circuit may "pause" halfway, keeping quantum states in memory, and wait for a value from a classical segment (like a classical message from a remote node) before continuing. Given the limited lifetime of quantum memories (quantum states decoherence — decrease in quality over time) [41], scheduling and synchronizing these interdependent quantum and classical segments is needed in order to achieve adequate application execution.

Challenge 4. Another crucial part of quantum network applications is entanglement generation between nodes. Research is being done on how to realize a *quantum network stack* [5, 13, 26], which organizes entanglement generation in networks, including timing synchronization, routing, and serving application requests [19]; however, integration of applications with such a stack has not yet been implemented and studied. Moreover, this integration requires handling both local application instructions and network-wide communication, which introduces a large span of time scales, a challenge for designing a software architecture for quantum network nodes. Indeed, entanglement generation requires very precise (at least nanosecond) timing synchronization between the network nodes [12], while application logic, including classical messaging between nodes, may be on the order of milliseconds.

Challenge 5. Finally, quantum network applications often have moments in which they are idle since they have to wait for a message to arrive from another node. Such idle

1

times present an opportunity to do *multitasking* — running multiple applications concurrently — on a node, in order to make more efficient use of quantum hardware.

1.2 Research objective

In this thesis, we address the gap that exists for quantum networks: namely that there is no programming and execution framework for quantum network (or internet) applications. The main goal is to *enable programming and execution of arbitrary quantum network applications in a hardware-agnostic way while optimizing runtime performance.* We work towards this goal by addressing the following research questions:

Q1. **How should quantum network applications be programmed?** We first address the question of how one should represent quantum network applications as programming code. The goal is to develop a model similar to that in classical and quantum computing, where applications have both a high-level, human-friendly representation and a lower-level, execution-oriented one.
One challenge comes from the hybrid nature of quantum network applications, which require the integration of both classical and quantum code. Additionally, we aim to support a variety of hardware types. For the high-level representation, programmers should be able to focus on application logic without needing to understand specific hardware constraints. For the lower-level representation, we aim for flexibility, allowing the integration of emerging hardware platforms as they are developed. We do this given the uncertainty around which quantum hardware platforms will ultimately prove viable.
Finally, the representation should be suitable for execution on quantum network nodes, leading to the next research question.

Q2. **How should a quantum network node execute arbitrary applications?** Programming an application is one step; executing it on real quantum network nodes is another. We tackle the challenge of executing arbitrary applications — anything a programmer might write using our representation (Q1) — on quantum network nodes. Again, the hybrid nature of these applications presents a challenge: how can we effectively control the execution of both classical and quantum code? Additionally, we must consider how application logic (classical or quantum) should interact with the networking code. Another difficulty is managing and integrating the range of timescales mentioned above. Finally, in order to make optimal use of hardware and to increase throughput, we investigate how to enable multitasking of applications.

Q3. **How can we improve performance of application execution?** A framework for programming and executing applications does not by itself guarantee optimal runtime performance. Application performance may be measured using classical metrics like execution time, and quantum metrics like success probability (see Chapter 2). In general, the runtime performance of applications depends on ahead-of-time *compilation* and runtime *scheduling*. We investigate how we can perform compilation and scheduling in order to increase performance of applications. Especially in a multitasking scenario, scheduling has a large effect on performance. However, even without multitasking, the question remains of how to schedule both application code and networking tasks.

1.3 Thesis outline

With this thesis, we achieve the goal of enabling programming and execution of quantum network applications. We do this by presenting new tools and system and software architectures. Figure 1.1 visualizes how these fit together.

In Chapter 2, we first provide more background information about quantum internet applications. It introduces concepts and terminology that are used in the following chapters.

In Chapter 3, we present NetQASM: the first programming representation for quantum network applications, addressing Q1. This representation includes a new low-level instruction set architecture tailored to quantum network applications, but it also contains a high-level software development kit, enabling developers to express their application logic. We purposefully make NetQASM hardware-independent and extendible. We also introduce a first model for execution of applications programmed using NetQASM, addressing Q2. We evaluate our design choices in simulation.

In Chapter 4 we fully focus on the question of application execution (Q2). Building on top of our execution model from Chapter 3, we present a detailed full-stack system architecture — QNodeOS — for executing arbitrary applications on quantum network nodes. This architecture is the first of its kind. We implement QNodeOS on a setup with two real physical quantum network nodes, and show that our architecture can successfully execute quantum network applications. We report on the performance of our architecture by looking at application throughput and success probability.

Based on what we learned from our QNodeOS implementation and evaluation, we propose an improved architecture — Qoala — for executing applications on quantum network nodes in Chapter 5. Qoala addresses the compilation and scheduling challenges found in QNodeOS, by allowing hybrid classical-quantum compilation and scheduling. We show how this architecture enables strategies to achieve better application performance, addressing Q3.

In Chapter 6 we discuss in more detail how Qoala can be used for improved compilation strategies. Finally, in Chapter 7 we conclude and reflect upon future directions.

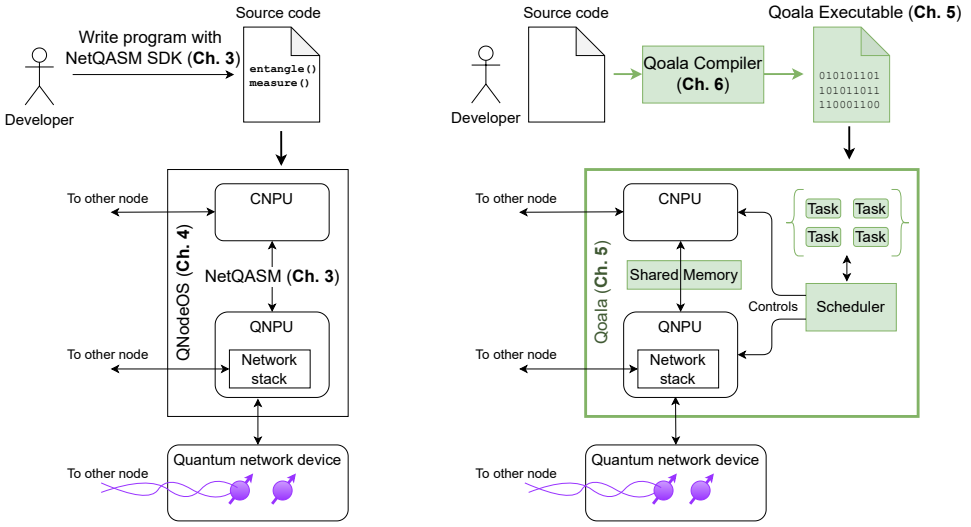


Figure 1.1: Visualization of how the architectures presented in this thesis fit together. Left: high-level schematic of a quantum network node stack. A developer writes a program using the NetQASM SDK (Chapter 3), which is executed by QNodeOS (Chapter 4). QNodeOS internally uses NetQASM (Chapter 3) to communicate between the classical and quantum systems (CNPU, QNPU, see Chapter 4). QNodeOS controls the quantum network device containing quantum memory (qubits, purple circles, some of which may be entangled with qubits in other nodes).
Right: Same quantum network node stack but with updates from Chapters 5 and 6: Qoala is an updated execution framework, still using the CNPU and QNPU, but adding a scheduler and shared memory (Chapter 5). Moreover, a compiler (Chapter 6) first converts source code into an executable, which is then executed by Qoala.

References

[1] A. Aho, J. Ullman, R. Sethi, and M. Lam. *Compilers: Principles, Techniques, and Tools*. 2nd edition. Boston Munich: Addison Wesley, Aug. 31, 2006. 1040 pp. ISBN: 978-0-321-48681-3.

[2] M. Bandic, S. Feld, and C. G. Almudever. "Full-stack quantum computing systems in the NISQ era: algorithm-driven and hardware-aware compilation techniques". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE). ISSN: 1558-1101. Mar. 2022, pp. 1–6. DOI: 10.23919/DATE54114.2022.9774643. URL: https://ieeexplore.ieee.org/abstract/document/9774643 (visited on Aug. 8, 2024).

[3] M. Ben-Or and A. Hassidim. "Fast Quantum Byzantine Agreement". In: *STOC*. ACM, 2005, pp. 481–485. DOI: 10.1145/1060590.1060662.

[4] A. Broadbent, J. Fitzsimons, and E. Kashefi. "Universal Blind Quantum Computation". In: *FOCS*. IEEE, 2009, pp. 517–526. DOI: 10.1109/FOCS.2009.36.

[5] M. Caleffi. "Optimal Routing for Quantum Networks". In: *IEEE Access* 5 (2017), pp. 22299–22312. DOI: 10.1109/ACCESS.2017.2763325.

[6] M. Caleffi, M. Amoretti, D. Ferrari, J. Illiano, A. Manzalini, and A. S. Cacciapuoti. "Distributed quantum computing: A survey". In: *Computer Networks* 254 (Dec. 1, 2024), p. 110672. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2024.110672. URL: https://www.sciencedirect.com/science/article/pii/S1389128624005048 (visited on Sept. 13, 2024).

[7] M. Castells. "The Impact of the Internet on Society: A Global Perspective". In: *Ch@nge: 19 Key Essays on How the Internet Is Changing Our Lives*. Madrid: BBVA, 2013.

[8] T. Chatterjee, A. Das, S. I. Mohtashim, A. Saha, and A. Chakrabarti. "Qurzon: A Prototype for a Divide and Conquer-Based Quantum Compiler for Distributed Quantum Systems". In: *SN Computer Science* 3.4 (June 10, 2022), p. 323. ISSN: 2661-8907. DOI: 10.1007/s42979-022-01207-9. URL: https://doi.org/10.1007/s42979-022-01207-9 (visited on Aug. 13, 2024).

[9] A. M. Childs. "Secure Assisted Quantum Computation". In: *Quantum Inf. Comput.* 5.6 (2005), pp. 456–466. DOI: 10.26421/QIC5.6-4.

[10] F. T. Chong, D. Franklin, and M. Martonosi. "Programming languages and compiler design for realistic quantum hardware". In: *Nature* 549.7671 (Sept. 2017). Publisher: Nature Publishing Group, pp. 180–187. ISSN: 1476-4687. DOI: 10.1038/nature23459. URL: https://www.nature.com/articles/nature23459 (visited on Sept. 13, 2024).

[11] D. Cuomo, M. Caleffi, K. Krsulich, F. Tramonto, G. Agliardi, E. Prati, and A. S. Cacciapuoti. "Optimized Compiler for Distributed Quantum Computing". In: *ACM Transactions on Quantum Computing* 4.2 (Feb. 24, 2023), 15:1–15:29. DOI: 10.1145/3579367. URL: https://dl.acm.org/doi/10.1145/3579367 (visited on Aug. 13, 2024).

[12] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveira Filho, R. Hanson, and S. Wehner. "A link layer protocol for quantum networks". In: *Proceedings of the ACM special interest group on data communication*. 2019, pp. 159–173. DOI: 10.1145/3341302.3342070.

[13] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveria Filho, R. Hanson, and S. Wehner. "A Link Layer Protocol for Quantum Networks". In: *ACM SIGCOMM 2019 Conference*. SIGCOMM '19. Beijing, China: ACM, 2019, p. 15. ISBN: 978-1-4503-5956-6/19/09. DOI: 10.1145/3341302.3342070. URL: http://doi.acm.org/10.1145/3341302.3342070.

[14] A. Dahlberg and S. Wehner. "SimulaQron—a simulator for developing quantum internet software". In: *Quantum Science and Technology* 4.1 (Sept. 2018), p. 015001. DOI: 10.1088/2058-9565/aad56e. URL: https://doi.org/10.1088%2F2058-9565%2Faad56e.

[15] A. M. Dalzell, S. McArdle, M. Berta, P. Bienias, C.-F. Chen, A. Gilyén, C. T. Hann, M. J. Kastoryano, E. T. Khabiboulline, A. Kubica, G. Salton, S. Wang, and F. G. S. L. Brandão. *Quantum algorithms: A survey of applications and end-to-end complexities*. Oct. 4, 2023. DOI: 10.48550/arXiv.2310.03011. arXiv: 2310.03011[quant-ph]. URL: http://arxiv.org/abs/2310.03011 (visited on Sept. 13, 2024).

[16] S. DiAdamo, M. Ghibaudi, and J. Cruise. "Distributed quantum computing and network control for accelerated vqe". In: *IEEE Transactions on Quantum Engineering* 2 (2021), pp. 1–21. DOI: 10.1109/TQE.2021.3057908.

[17] E. Farhi, J. Goldstone, and S. Gutmann. "A quantum approximate optimization algorithm". In: *arXiv preprint arXiv:1411.4028* (2014). DOI: 10.48550/arXiv.1411.4028.

[18] D. Ferrari, S. Carretta, and M. Amoretti. "A Modular Quantum Compilation Framework for Distributed Quantum Computing". In: *IEEE Transactions on Quantum Engineering* 4 (2023), pp. 1–13. ISSN: 2689-1808. DOI: 10.1109/TQE.2023.3303935. arXiv: 2305.02969[quant-ph]. URL: http://arxiv.org/abs/2305.02969 (visited on Sept. 13, 2024).

[19] S. Gauthier, G. Vardoyan, and S. Wehner. "A Control Architecture for Entanglement Generation Switches in Quantum Networks". In: *Proceedings of the 1st Workshop on Quantum Networks and Distributed Quantum Computing*. QuNet '23. New York, NY, USA: Association for Computing Machinery, Sept. 10, 2023, pp. 38–44. ISBN: 9798400703065. DOI: 10.1145/3610251.3610552. URL: https://dl.acm.org/doi/10.1145/3610251.3610552 (visited on Oct. 13, 2024).

[20] P. A. Guérin, A. Feix, M. Araújo, and Č. Brukner. "Exponential communication complexity advantage from quantum superposition of the direction of communication". In: *Physical review letters* 117.10 (2016). Publisher: APS, p. 100502. DOI: 10.1103/PhysRevLett.117.100502.

[21] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore. "Quantum programming languages". In: *Nature Reviews Physics* 2.12 (Dec. 2020). Publisher: Nature Publishing Group, pp. 709–722. ISSN: 2522-5820. DOI: 10.1038/s42254-020-00245-7. URL: https://www.nature.com/articles/s42254-020-00245-7 (visited on Aug. 13, 2024).

[22] J. Hofmann, M. Krug, N. Ortegel, L. Gérard, M. Weber, W. Rosenfeld, and H. Weinfurter. "Heralded Entanglement Between Widely Separated Atoms". In: *Science* 337.6090 (2012), pp. 72–75. DOI: 10.1126/science.1221856.

[23] K. James and R. Keith. *Computer Networking: A Top-Down Approach*. Boston Munich, June 7, 2016. 864 pp. ISBN: 978-0-13-359414-0.

[24] B. Jing, X.-J. Wang, Y. Yu, P.-F. Sun, Y. Jiang, S.-J. Yang, W.-H. Jiang, X.-Y. Luo, J. Zhang, X. Jiang, et al. "Entanglement of three quantum memories via interference of three single photons". In: *Nature Photonics* 13.3 (2019), pp. 210–213. DOI: 10.1038/s41566-018-0342-x.

[25] C. Knaut, A. Suleymanzade, Y.-C. Wei, D. Assumpcao, P.-J. Stas, Y. Huan, B. Machielse, E. Knall, M. Sutula, G. Baranes, et al. "Entanglement of nanophotonic quantum memory nodes in a telecom network". In: *Nature* 629.8012 (2024), pp. 573–578. DOI: 10.1038/s41586-024-07252-z.

[26] W. Kozlowski, A. Dahlberg, and S. Wehner. "Designing a Quantum Network Protocol". In: *arXiv preprint arXiv:2010.02575* (2020).

[27] V. Krutyanskiy, M. Galli, V. Krcmarsky, S. Baier, D. Fioretto, Y. Pu, A. Mazloom, P. Sekatski, M. Canteri, M. Teller, et al. "Entanglement of trapped-ion qubits separated by 230 meters". In: *Physical Review Letters* 130.5 (2023). Publisher: APS, p. 050803. DOI: 10.1103/PhysRevLett.130.050803.

[28] R. LaRose. "Overview and Comparison of Gate Level Quantum Software Platforms". In: *Quantum* 3 (Mar. 25, 2019). Publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, p. 130. DOI: 10.22331/q-2019-03-25-130. URL: https://quantum-journal.org/papers/q-2019-03-25-130/ (visited on Sept. 13, 2024).

[29] J.-L. Liu, X.-Y. Luo, Y. Yu, C.-Y. Wang, B. Wang, Y. Hu, J. Li, M.-Y. Zheng, B. Yao, Z. Yan, et al. "Creation of memory–memory entanglement in a metropolitan quantum network". In: *Nature* 629.8012 (2024), pp. 579–585. DOI: 10.1038/s41586-024-07308-0.

[30] W.-Z. Liu, Y.-Z. Zhang, Y.-Z. Zhen, M.-H. Li, Y. Liu, J. Fan, F. Xu, Q. Zhang, and J.-W. Pan. "Toward a Photonic Demonstration of Device-Independent Quantum Key Distribution". In: *Phys. Rev. Lett.* 129.5 (2022), p. 050502. DOI: 10.1103/PhysRevLett.129.050502.

[31] X. Liu, A. Angone, R. Shaydulin, I. Safro, Y. Alexeev, and L. Cincio. "Layer VQE: A variational approach for combinatorial optimization on noisy quantum computers". In: *IEEE Transactions on Quantum Engineering* 3 (2022), pp. 1–20. DOI: 10.1109/TQE.2021.3140190.

[32] D. L. Moehring, P. Maunz, S. Olmschenk, K. C. Younge, D. N. Matsukevich, L.-M. Duan, and C. Monroe. "Entanglement of Single-Atom Quantum Bits at a Distance". In: *Nature* 449 (2007), pp. 68–71. DOI: 10.1038/nature06118.

[33] C. Monroe, W. Campbell, C. Cao, T. Choi, S. Clark, S. Debnath, C. Figgatt, D. Hayes, D. Hucul, V. Inlek, R. Islam, S. Korenblit, K. Johnson, A. Manning, J. Mizrahi, B. Neyenhuis, A. Lee, P. Richerme, C. Senko, J. Smith, and K. Wright. "Quantum Networks with Atoms and Photons". In: *Journal of Physics: Conference Series* 467.1 (Dec. 2013), p. 012008. ISSN: 1742-6596. DOI: 10.1088/1742-6596/467/1/012008. URL: https://dx.doi.org/10.1088/1742-6596/467/1/012008 (visited on Oct. 9, 2024).

[34] A. Montanaro. "Quantum algorithms: an overview". In: *npj Quantum Information* 2.1 (Jan. 12, 2016). Publisher: Nature Publishing Group, pp. 1–8. ISSN: 2056-6387. DOI: 10.1038/npjqi.2015.23. URL: https://www.nature.com/articles/npjqi201523 (visited on Aug. 19, 2024).

[35] D. Nadlinger. "Device-independent key distribution between trapped-ion quantum network nodes". PhD thesis. University of Oxford, 2022.

[36] M. Pompili, S. L. N. Hermans, S. Baier, H. K. C. Beukers, P. C. Humphreys, R. N. Schouten, R. F. L. Vermeulen, M. J. Tiggelman, L. dos Santos Martins, B. Dirkse, S. Wehner, and R. Hanson. "Realization of a Multinode Quantum Network of Remote Solid-State Qubits". In: *Science* 372.6539 (2021), pp. 259–264. DOI: 10.1126/science.abg1919.

[37] A. Poremba. "Quantum proofs of deletion for learning with errors". In: *arXiv preprint arXiv:2203.01610* (2022). DOI: 10.48550/arXiv.2203.01610.

[38] A. Reiserer and G. Rempe. "Cavity-based quantum networks with single atoms and optical photons". In: *Reviews of Modern Physics* 87.4 (Dec. 1, 2015). Publisher: American Physical Society, pp. 1379–1418. DOI: 10.1103/RevModPhys.87.1379. URL: https://link.aps.org/doi/10.1103/RevModPhys.87.1379 (visited on Oct. 9, 2024).

[39] S. Ritter, C. Nölleke, C. Hahn, A. Reiserer, A. Neuzner, M. Uphoff, M. Mücke, E. Figueroa, J. Bochmann, and G. Rempe. "An Elementary Quantum Network of Single Atoms in Optical Cavities". In: *Nature* 484.7393 (2012), pp. 195–200. DOI: 10.1038/nature11023.

[40] M. Ruf, N. H. Wan, H. Choi, D. Englund, and R. Hanson. "Quantum networks based on color centers in diamond". In: *Journal of Applied Physics* 130.7 (Aug. 16, 2021), p. 070901. ISSN: 0021-8979. DOI: 10.1063/5.0056534. URL: https://doi.org/10.1063/5.0056534 (visited on Oct. 9, 2024).

[41] P. W. Shor. "Scheme for reducing decoherence in quantum computer memory". In: *Physical Review A* 52.4 (Oct. 1, 1995). Publisher: American Physical Society, R2493–R2496. DOI: 10.1103/PhysRevA.52.R2493. URL: https://link.aps.org/doi/10.1103/PhysRevA.52.R2493 (visited on Oct. 9, 2024).

[42] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating Systems Concepts (Ninth Edition)*. Wiley, 2014.

[43] L. J. Stephenson, D. P. Nadlinger, B. C. Nichol, S. An, P. Drmota, T. G. Ballance, K. Thirumalai, J. F. Goodwin, D. M. Lucas, and C. J. Ballance. "High-Rate, High-Fidelity Entanglement of Qubits Across an Elementary Quantum Network". In: *Phys. Rev. Lett.* 124.11 (2020), p. 110501. DOI: 10.1103/PhysRevLett.124.110501.

[44] R. Stockill, M. J. Stanley, L. Huthmacher, E. Clarke, M. Hugues, A. J. Miller, C. Matthiesen, C. Le Gall, and M. Atatüre. "Phase-Tuned Entangled State Generation between Distant Spin Qubits". In: *Phys. Rev. Lett.* 119.1 (2017), p. 010503. DOI: 10.1103/PhysRevLett.119.010503.

[45] A. J. Stolk, K. L. van der Enden, M.-C. Slater, I. t. Raa-Derckx, P. Botma, J. van Rantwijk, B. Biemond, R. A. Hagen, R. W. Herfst, W. D. Koek, et al. "Metropolitan-scale heralded entanglement of solid-state qubits". In: *arXiv preprint arXiv:2404.03723* (2024). DOI: 10.48550/arXiv.2404.03723.

[46] S. Wehner, D. Elkouss, and R. Hanson. "Quantum Internet: A Vision for the Road Ahead". In: *Science* 362.6412 (2018), pp. 1–9. DOI: 10.1126/science.aam9288.

[47] W. Zhang, T. van Leent, K. Redeker, R. Garthoff, R. Schwonnek, F. Fertig, S. Eppelt, W. Rosenfeld, V. Scarani, C. C.-W. Lim, et al. “A Device-Independent Quantum Key Distribution System for Distant Users”. In: *Nature* 607.7920 (2022), pp. 687–691. DOI: 10.1038/s41586-022-04891-y.

2 Preliminaries

In this chapter we discuss some of the concepts that are relevant for understanding the rest of the thesis. Although most of these concepts are explained again in the following chapters, the aim here is to already familiarize the reader with the overall context of this work.

2.1 Quantum networks

A quantum network consists of devices that are connected together and that can establish entanglement between separate devices in that network. A global network of quantum networks, envisioned to be realized in the future, may be called a *quantum internet*. More specifically, we assume a quantum network to consist of *nodes* that are connected by *classical channels* and *quantum channels*. Classical channels enable classical communication between nodes, while quantum channels are used for *entanglement* generation (Section 2.1.2) between nodes. Within a quantum network, one can distinguish between two main types of nodes: First, there are *end nodes* [36], with which users execute quantum network applications (Section 2.2). In classical networks, end nodes are laptops, phones or other devices. In the quantum domain, end nodes may be simple photonic devices that can only create or measure quantum states, or they may be quantum processors capable of arbitrary qubit operations and storage of information within a quantum memory. The type of end node dictates what applications are possible [36], and we have chosen to focus on the most general form of an end node, namely, a quantum processor including quantum memory. So, our goal is to enable programming and execution of arbitrary quantum network applications on end nodes that are quantum processors. For the remainder of this thesis, we will thus always take end nodes to be quantum-processor end nodes.

Second, a quantum network can include *intermediate nodes* that perform routines necessary to connect two or more end nodes (Figure 2.1). We refer the reader with a background in computer science to [35] for a gentle introduction to quantum networks. Intermediate nodes, such as quantum-repeater nodes, are used to establish long-distance entanglement between remote end nodes. These intermediate nodes may employ protocols such as entanglement swapping and entanglement distillation in order to realize end-to-end links with sufficiently high fidelity (quality) for network applications. These protocols

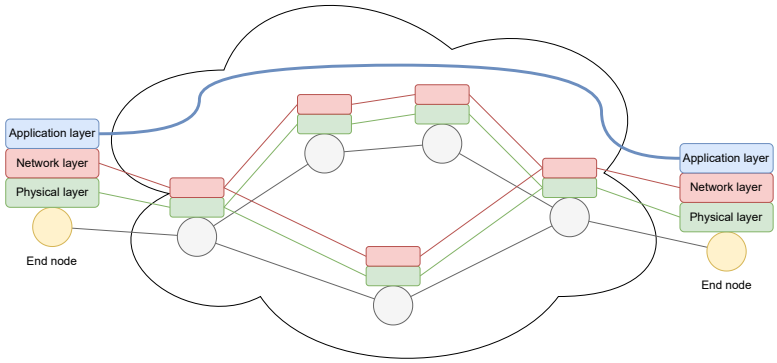


Figure 2.1: **Schematic overview of a quantum network.** A quantum network consists of nodes (yellow and grey circles) that are connected by classical and quantum communication channels (grey lines). Each node implements a physical layer (green boxes and lines) that enables entanglement generation with neighboring nodes. Each node also implements a network stack, including a network layer (red boxes and lines, which may be subdivided into a separate link layer and a network layer [9, 21]). This layer realizes long-distance entanglement creation between nodes and may include protocols such as entanglement swapping and distillation.
We emphasize however that the focus of this work is to program and execute applications on the end nodes, i.e. enabling the application layer in networking terms. Only *end nodes* (yellow circles) implement an additional application layer (blue boxes and line), which executes arbitrary user applications. From the perspective of this layer, end nodes are logically directly connected (blue line), and this layer is hence independent from implementations and protocols in the network layer and is only dependent on the service provided by the network layer. Logically directly connected means that the application layer relies on the service of the network layer to enable end-to-end entanglement generation between end nodes and does not concern itself with how the entanglement is generated. This abstraction is a key element enabled by a quantum network stack such as [9] and exactly analogous to abstractions used in classical networking, where e.g. a web browser can be executed on a laptop independently of how the internet connection between the laptop and a web server is realized.

are handled by a network stack (see, e.g. [9]) that exists at each node. The network stack includes a link layer, a network layer, a control plane, and other networking functions; it is responsible for entanglement generation.

Intermediate nodes do not execute user applications (i.e. the applications we focus on in this thesis), which is done only by end nodes. Therefore, only end nodes need to have an additional stack implementing the application layer in a network, which is referred to as an *application stack* (see Figure 2.1). The application stack is responsible for the execution of arbitrary user applications, and integrates with the network stack for entanglement generation over the network. We remark that it is the purpose of a network layer [9, 20] to provide a service to the application layer that allows entanglement generation with remote end nodes. Importantly, this service should not require the application layer to have any knowledge about the connectivity of the network.

2.1.1 End nodes

As mentioned above, end nodes in a quantum network possess a quantum processor acting on quantum memory. Quantum memory consists of individual quantum bits (*qubits*), each of which can have a quantum state, such as $|0\rangle, |1\rangle$ or $|+\rangle$ (see [26] for a an extensive

introduction to qubits and quantum states). The quantum network can deliver entangled pairs (Section 2.1.2) to end nodes, such that end nodes can obtain qubits in their quantum memory that are entangled with qubits in the quantum memory of other end nodes. 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. These abilities (classical and quantum processing, as well as classical and quantum communication, the latter being entanglement generation) are all required for the execution of quantum network applications (Section 2.2).

Quantum memory. 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 create an entangled state with another node. These qubits are called *communication qubits* [10], in contrast to *storage qubits* which can only directly interact with other qubits part of the same local node. 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. Some hardware implementations only have a single communication qubit and multiple storage qubits [5], whereas others can have multiple communication qubits [18].

There are various quantum hardware implementations for quantum network processors, such as nitrogen-vacancy centers in diamond [28], trapped ions [22], and neutral atoms [17, 30], which all have different capabilities and gates that can be performed.

Noise and decoherence. All current quantum processor implementations are in the so-called Noisy Intermediate-Scale Quantum (NISQ) stage, meaning that they have a limited number of qubits (typically in the order of tens or a few hundred for (non-network) quantum computers, and only a handful of qubits for quantum network processors), and these qubits are susceptible to *noise* and errors due to their limited *coherence* (lifetime) and imperfect control. Limited lifetime means that the quality of qubits decreases (called *decoherence*) over time, eventually rendering them unreliable (computations produce random or incorrect results). Decoherence may also happen by applying gates. Gates transform the state of qubits, but typically also induce some noise, leading to decoherence of the qubit.

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, and indirectly on the performance of applications.

Throughout this thesis, when we say 'qubit', these may be physical qubits, but may also be logical qubits (multiple physical qubits together representing one more robust usable qubit) in case the end node does error correction [24].

2.1.2 Entanglement generation

Entanglement is a phenomenon in quantum physics where two or more particles (qubits) are correlated in a way that is not possible classically. In a quantum network, such entangled qubits may be established across separate nodes, realizing a quantum connection between those nodes. Entanglement can be used as a resource in order to realize applications [36] that are impossible with classical networks, including applications such as

data consistency in the cloud [1], privacy-enhancing proofs of deletion [29], exponential savings in communication [15], or secure quantum computing in the cloud [6, 8].

In order for two neighboring quantum network nodes to produce entanglement between them, they need to simultaneously perform an action to trigger entanglement generation (at the physical layer, synchronized to nanosecond precision). This means neighboring quantum network nodes need to perform a network operation (entanglement generation) in a *very specific* time slot in which they make an attempt to generate entanglement. Such time slots are generally aggregated into larger time bins, corresponding to making batches of attempts in time slots synchronized at the physical layer. We refer to e.g. [27] for background information on the physical layer of entanglement generation in quantum networks, and the readers with a background in computer science to e.g. [9] for a detailed explanation of scheduling of entanglement generation in quantum networks.

The time bins cannot be determined by the quantum node itself. Instead, selection of time bins for a specific quantum operation requires agreement with the neighboring node [9] (and more generally with the quantum network when the end-to-end entanglement is made via intermediate network nodes, Figure 2.1) by means of a network schedule, e.g. determined by a (logically) centralized controller, see [31].

2.2 Programs and applications

In this section we discuss what (quantum network) applications and programs are, and how they are represented.

2.2.1 Quantum computing

First, let us briefly discuss quantum *computing* (rather than networking). A quantum computing program runs on a single quantum processor, which may be an individual node in a quantum network, but may also just be a standalone quantum computer. Such a local (non-network) program consists of performing operations on qubits, and is typically represented as a *quantum circuit*. A circuit describes the program's quantum memory — as individual qubits — and the gates that are applied on these qubits. When visualized, like in Figure 2.2, qubits are represented by horizontal lines and operations by boxes on those lines. Such circuits must be read from left to right: gates on the left are applied first, then the ones to the right and so on.

Operations include quantum *gates* (such as rotation gates or the Hadamard gate), initialization, and measurement (readout). Quantum gates may be applied on a single qubit, or on multiple qubits. For example, the CNOT gate used in the example of Figure 2.2 is a 2-qubit gate. The code, or the 'recipe' of quantum programs is hence classical, while the information and memory that the program manipulates is quantum. Program developers may describe their programs as circuits, but often also write them as program code (Figure 2.2), using existing languages like Python (e.g. [33]) or using custom quantum languages (such as e.g. [34]). We refer to Sections 3.1.2 and 6.3.3 for more examples of programming languages and frameworks. Besides quantum operations, there may be limited classical control, such as a gate being executed depending on a measurement outcome. Typically, a quantum circuit is executed in one go, on a very small timescale. Quantum memory used for quantum computing [23] often only stays coherent (alive and useful) for

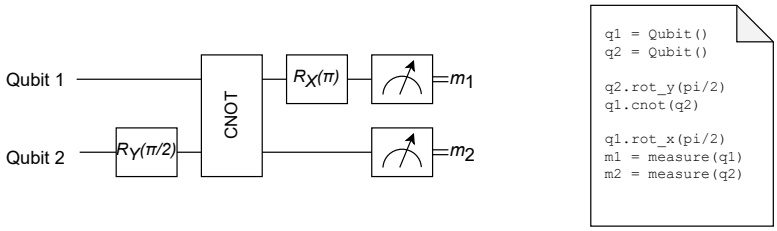


Figure 2.2: Two representations of an example quantum program using two qubits. This program can run on a quantum computer, which may be a node in a quantum network. Left: circuit model of the program. A horizontal line represents a single qubit. Blocks are quantum gates applied on qubits; there are single-qubit gates (such as a rotation gate around axis X with angle π ($R_X(\pi)$) and multi-qubit gates (such as a CNOT; note that a CNOT gate is also sometimes depicted as a vertical line and an XOR symbol on one of the qubits). Time goes from left to right, so the gates are applied in the order (left to right) they are depicted. A measurement 'gate' destroys the qubit and produces a classical bit, depicted by a double line. Right: the same program but represented in a programming language (language shown is fictional and for illustration purposes only; in Chapter 3 we present a real, detailed language). This is what a developer might write when programming ('coding') the program.

microseconds. For more information on quantum computing, see e.g. [26].

Quantum computing does not necessarily involve only quantum circuit execution with limited classical control. *Hybrid classical-quantum* programs consist of an interleaving of purely classical computation and quantum circuit execution. For example, in Variational Quantum Eigensolvers (VQE) [11, 25] and Quantum Approximate Optimization Algorithms (QAOA) [13], upon completion of a quantum circuit, the classical results are processed, resulting in a new quantum circuit that is then executed; this process repeats multiple times.

2.2.2 Quantum network (or internet) applications

Quantum network (or internet) applications, also called *protocols*, are multi-partite programs that involve entanglement generation and classical communication between different end nodes, as well as local computation. The local computation includes arbitrary classical computation as well as local quantum operations (like in quantum computing circuits, see above). Examples include Quantum Key Distribution (QKD) [2, 12], leader election protocols [14, 19], and Blind Quantum Computation (BQC) [37]. In this thesis, we consider quantum network applications in the quantum memory stage [36] and above. That is, applications that require the use of a quantum processor that can manipulate and store qubits. For simpler applications in the prepare-and-measure and entanglement generation stages [36], e.g. quantum key distribution [3, 12], where the quantum states are immediately measured by the nodes, it would be sufficient to realize a system implementing a quantum network stack and classical processing only.

2.2.3 Programs

Throughout this thesis, we will use the following terminology. *Applications* refer to multi-node protocols or use-cases of quantum networks, such as QKD and BQC. *Programs* refer to the code that is run on individual quantum network nodes. Applications are realized by the joint execution of programs on their respective nodes.

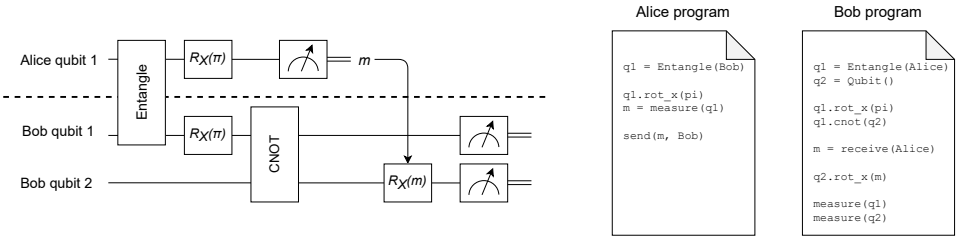


Figure 2.3: Two representations of an example two-node quantum network application. The application is run on the nodes Alice (possessing one qubit) and Bob (possessing two qubits). Left: circuit model of the application. The Entangle operation is depicted here as a two-qubit gate; however since it acts on qubits in different nodes (Alice and Bob), this is not as trivial as a local two-qubit gate such as the CNOT, and instead requires coordination between the two nodes. Furthermore, classical communication may happen in the form of sending a message. In this case, after measuring her qubit, Alice sends the classical outcome bit m to Bob. Bob then uses this in his local X-rotation gate. Right: the same application but represented as two separate programs written in a programming language. Since Alice and Bob are separate nodes, they each program their own local code. This code may however include external operations, such as entanglement creation with the other node, and sending or receiving messages. Note that these operations must match — something they are responsible for themselves, by e.g. following some pre-established protocol.

A multi-node quantum network application is hence partitioned into separate single-node programs that run concurrently on different network end nodes (e.g. in BQC: a client program on a client node and a server program on a server node, or in secret sharing [16]: a program each on N nodes). Each of these programs runs independently, and may also be independently programmed (and compiled), as shown in Figure 2.3. This highlights the difference with distributed quantum computing (see e.g. [7]), where all nodes can be accessed and controlled by a single program.

Program ingredients. The single-node programs that constitute a quantum network application are hybrid in nature (see Figure 2.4): First, they contain quantum operations, such as local quantum gates and measurements (e.g. to perform a server computation in BQC), and entanglement generation (e.g. to produce key in QKD). Second, programs need to perform classical operations, such as message passing (e.g., a BQC client program sending desired measurement bases to the BQC server), and local classical processing (e.g., post-processing measurement outcomes in QKD). Programs may also involve asynchronous operations (e.g. a server awaiting entanglement with multiple clients).

Quantum network applications may be represented as circuits (such as in Figure 2.3), although this may be less feasible for more complex applications if classical processing is more elaborate.

Interactivity. Classical blocks of code may depend on quantum ones via classical variables generated during the quantum execution (such as measurement results, notification of entanglement generation, and information on the state of the quantum system such as the availability of qubits). Similarly, quantum blocks may depend on variables set by the classical blocks (such as messages received from remote network nodes). Finally, quantum blocks may themselves depend on other quantum blocks via qubits in the quantum memory.

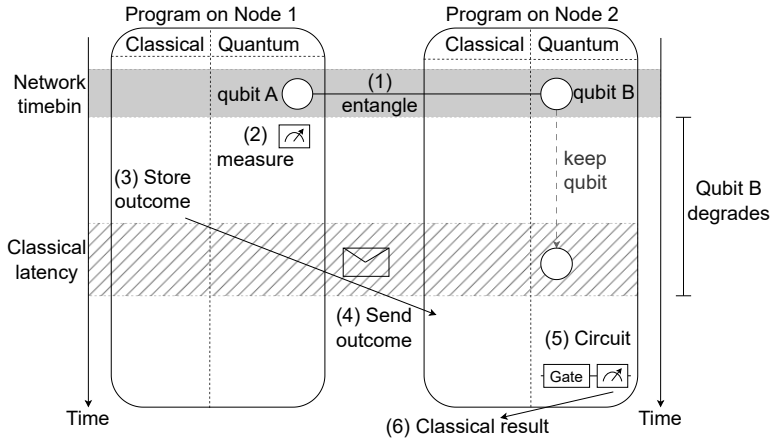


Figure 2.4: Another visualization of a quantum network application, focusing on the different types of operations and the role of time and latencies. This application consists of two hybrid classical-quantum programs (on Nodes 1 and 2) including (1) Entanglement generation between two qubits (circles) in a synchronized time slot (defined by a network controller). (2) A local measurement of qubit A at Node 1 resulting in a classical outcome bit (destroying the qubit). (4) Communication of the classical bit from Node 1 to Node 2 (taking non-deterministic time). (5) Execution of a quantum circuit on qubit B at Node 2 depending on the classical bit. The quality of qubit B has degraded during the time elapsed since (1). (6) Node 2 measures qubit B and outputs the classical result.

2.2.4 Application execution

Mode of Execution. There exist quantum applications and functionalities, where one pair of programs is executed only once, e.g. a simple example of quantum teleportation [4]. As in quantum computing, however, some quantum network applications [36] are expected to succeed only with a specific *success probability* p_{succ} when executed once. This may be either since the application itself is non-deterministic in nature, or (and this holds for all applications) because noise (see above) introduces errors. Applications are hence typically executed many times in succession, where outcome statistics are computed in order to validate successful execution (e.g. by majority of outcomes).

Performance metrics. Performance of application execution on quantum network nodes can be measured by several metrics. In this thesis we consider both (1) metrics that apply to a single application that one executes on the quantum network, and (2) metrics that apply to a node in the network that executes one or more programs.

For an application, we consider *makespan* as a classical metric and *success probability* as a quantum metric. Makespan is the time it takes to execute (all repetitions of) the application. The success probability is the one mentioned above. It is typically related to quantum fidelity $F \in [0,1]$, which is a measure of closeness of a quantum state to some ideal quantum state. Since present-day quantum systems are *noisy* (see above), quantum memory is non-perfect ($F < 1$) which negatively affects application success probability. For quantum network nodes, we consider common classical metrics [32]: utility (fraction of time that a node is doing useful things), throughput (amount of application executions per time unit) and latency (which may be between internal components of a node, or between nodes).

Multitasking. 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, explained in more detail in the following chapters.

References

[1] M. Ben-Or and A. Hassidim. "Fast Quantum Byzantine Agreement". In: *STOC*. ACM, 2005, pp. 481–485. DOI: 10.1145/1060590.1060662.

[2] C. H. Bennett and G. Brassard. "Quantum Cryptography: Public Key Distribution, and Coin-Tossing". In: *Proc. 1984 IEEE International Conference on Computers, Systems, and Signal Processing*. 1984, pp. 175–179. ISBN: doi:10.1016/j.tcs.2011.08.039. DOI: 10.1016/j.tcs.2011.08.039.

[3] 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. URL: https://www.karlin.mff.cuni.cz/~holub/soubory/BB84original.pdf.

[4] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. "Teleporting an Unknown Quantum State via Dual Classical and Einstein-Podolsky-Rosen Channels". In: *Phys. Rev. Lett.* 70.13 (1993), pp. 1895–1899. DOI: 10.1103/PhysRevLett.70.1895.

[5] H. Bernien. "Control, measurement and entanglement of remote quantum spin registers in diamond". PhD Thesis. TU Delft, 2014. DOI: 10.4233/uuid:75130c37-edb5-4a34-ac2f-c156d377ca55.

[6] A. Broadbent, J. Fitzsimons, and E. Kashefi. "Universal Blind Quantum Computation". In: *FOCS*. IEEE, 2009, pp. 517–526. DOI: 10.1109/FOCS.2009.36.

[7] A. S. Cacciapuoti, M. Caleffi, F. Tafuri, F. S. Cataliotti, S. Gherardini, and G. Bianchi. "Quantum internet: networking challenges in distributed quantum computing". In: *IEEE Network* 34.1 (2019), pp. 137–143. DOI: 10.1109/MNET.001.1900092.

[8] A. M. Childs. "Secure Assisted Quantum Computation". In: *Quantum Inf. Comput.* 5.6 (2005), pp. 456–466. DOI: 10.26421/QIC5.6-4.

[9] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveira Filho, R. Hanson, and S. Wehner. "A Link Layer Protocol for Quantum Networks". In: *SIGCOMM*. ACM, 2019, pp. 159–173. DOI: 10.1145/3341302.3342070.

[10] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveria Filho, R. Hanson, and S. Wehner. "A Link Layer Protocol for Quantum Networks". In: *ACM SIGCOMM 2019 Conference*. SIGCOMM '19. Beijing, China: ACM, 2019, p. 15. ISBN: 978-1-4503-5956-6/19/09. DOI: 10.1145/3341302.3342070. URL: http://doi.acm.org/10.1145/3341302.3342070.

[11] S. DiAdamo, M. Ghibaudi, and J. Cruise. "Distributed quantum computing and network control for accelerated vqe". In: *IEEE Transactions on Quantum Engineering* 2 (2021), pp. 1–21. DOI: 10.1109/TQE.2021.3057908.

[12] A. K. Ekert. "Quantum Cryptography Based on Bell's Theorem". In: *Phys. Rev. Lett.* 67.6 (1991), pp. 661–663. DOI: 10.1103/PhysRevLett.67.661.

[13] E. Farhi, J. Goldstone, and S. Gutmann. "A quantum approximate optimization algorithm". In: *arXiv preprint arXiv:1411.4028* (2014). DOI: 10.48550/arXiv.1411.4028.

[14] M. Ganz. "Quantum leader election". In: *arXiv preprint arXiv:0910.4952* (2009).

[15] P. A. Guérin, A. Feix, M. Araújo, and Č. Brukner. "Exponential communication complexity advantage from quantum superposition of the direction of communication". In: *Physical review letters* 117.10 (2016). Publisher: APS, p. 100502. DOI: 10.1103/PhysRevLett.117.100502.

[16] M. Hillery, V. Bužek, and A. Berthiaume. "Quantum secret sharing". In: *Physical Review A* 59.3 (1999), p. 1829. DOI: 10.1103/PhysRevA.59.1829.

[17] J. Hofmann, M. Krug, N. Ortegel, L. Gérard, M. Weber, W. Rosenfeld, and H. Weinfurter. "Heralded Entanglement Between Widely Separated Atoms". In: *Science* 337.6090 (2012), pp. 72–75. ISSN: 0036-8075. DOI: 10.1126/science.1221856. eprint: https://science.sciencemag.org/content/337/6090/72.full.pdf. URL: https://science.sciencemag.org/content/337/6090/72.

[18] I. V. Inlek, C. Crocker, M. Lichtman, K. Sosnova, and C. Monroe. "Multispecies Trapped-Ion Node for Quantum Networking". In: *Physical Review Letters* 118.25 (2017), pp. 1–5. ISSN: 10797114. DOI: 10.1103/PhysRevLett.118.250502. arXiv: 1702.01062.

[19] H. Kobayashi, K. Matsumoto, and S. Tani. "Simpler exact leader election via quantum reduction". In: *Chicago Journal of Theoretical Computer Science* 10 (2014), p. 2014.

[20] W. Kozlowski, A. Dahlberg, and S. Wehner. "Designing a Quantum Network Protocol". In: *CoNEXT*. ACM, 2020, pp. 1–16. DOI: 10.1145/3386367.3431293.

[21] W. Kozlowski and S. Wehner. "Towards Large-Scale Quantum Networks". In: *NANOCOM*. ACM, 2019, pp. 1–7. DOI: 10.1145/3345312.3345497.

[22] V. Krutyanskiy, M. Galli, V. Krcmarsky, S. Baier, D. Fioretto, Y. Pu, A. Mazloom, P. Sekatski, M. Canteri, M. Teller, J. Schupp, J. Bate, M. Meraner, N. Sangouard, B. Lanyon, and T. Northup. "Entanglement of trapped-ion qubits separated by 230 meters". In: *Physical Review Letters* 130.5 (2023), p. 050803. DOI: 10.1103/PhysRevLett.130.050803.

[23] N. P. de Leon, K. M. Itoh, D. Kim, K. K. Mehta, T. E. Northup, H. Paik, B. S. Palmer, N. Samarth, S. Sangtawesin, and D. W. Steuerman. "Materials challenges and opportunities for quantum computing hardware". In: *Science* 372.6539 (Apr. 16, 2021). Publisher: American Association for the Advancement of Science, eabb2823. DOI: 10.1126/science.abb2823. URL: https://www.science.org/doi/full/10.1126/science.abb2823 (visited on Oct. 25, 2024).

[24] D. A. Lidar and T. A. Brun. *Quantum Error Correction*. Cambridge University Press, 2013.

[25] X. Liu, A. Angone, R. Shaydulin, I. Safro, Y. Alexeev, and L. Cincio. "Layer VQE: A variational approach for combinatorial optimization on noisy quantum computers". In: *IEEE Transactions on Quantum Engineering* 3 (2022), pp. 1–20. DOI: 10.1109/TQE.2021.3140190.

[26] M. A. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. American Association of Physics Teachers, 2002.

[27] M. Pompili, C. Delle Donne, I. te Raa, B. van der Vecht, M. Skrzypczyk, G. M. Ferreira, L. de Kluijver, A. J. Stolk, S. L. N. Hermans, P. Pawełczak, W. Kozlowski, R. Hanson, and S. Wehner. "Experimental Demonstration of Entanglement Delivery Using a Quantum Network Stack". In: *npj Quantum Information* 8.1 (2022), p. 121. DOI: 10.1038/s41534-022-00631-2.

[28] M. Pompili, S. Hermans, S. Baier, H. Beukers, P. Humphreys, R. Schouten, R. Vermeulen, M. Tiggelman, L. dos Santos Martins, B. Dirkse, S. Wehner, and R. Hanson. "Realization of a multinode quantum network of remote solid-state qubits". In: *Science* 372.6539 (2021), pp. 259–264. DOI: 10.1126/science.abg1919.

[29] A. Poremba. "Quantum proofs of deletion for learning with errors". In: *arXiv preprint arXiv:2203.01610* (2022). DOI: 10.48550/arXiv.2203.01610.

[30] S. Ritter, C. Nölleke, C. Hahn, A. Reiserer, A. Neuzner, M. Uphoff, M. Mücke, E. Figueroa, J. Bochmann, and G. Rempe. "An elementary quantum network of single atoms in optical cavities". In: *Nature* 484.7393 (2012), p. 195.

[31] M. Skrzypczyk and S. Wehner. "An Architecture for Meeting Quality-of-Service Requirements in Multi-User Quantum Networks". 2021. arXiv: 2111.13124.

[32] R. Stankiewicz, P. Chołda, and A. Jajszczyk. "QoX: What is It Really?" In: *IEEE Communications Magazine* 49.4 (2011), pp. 148–158. DOI: 10.1109/MCOM.2011.5741159.

[33] D. S. Steiger, T. Häner, and M. Troyer. "ProjectQ: an open source software framework for quantum computing". In: *Quantum* 2 (Jan. 31, 2018). Publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, p. 49. DOI: 10.22331/q-2018-01-31-49. URL: https://quantum-journal.org/papers/q-2018-01-31-49/ (visited on Sept. 13, 2024).

[34] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler. "Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL". In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*. RWDSL2018. New York, NY, USA: Association for Computing Machinery, Feb. 24, 2018, pp. 1–10. ISBN: 978-1-4503-6355-6. DOI: 10.1145/3183895.3183901. URL: https://dl.acm.org/doi/10.1145/3183895.3183901 (visited on Oct. 9, 2024).

[35] R. van Meter. *Quantum Networking*. John Wiley and Sons, Ltd, 2014. DOI: 10.1002/9781118648919.

[36] S. Wehner, D. Elkouss, and R. Hanson. "Quantum Internet: A Vision for the Road Ahead". In: *Science* 362.6412 (2018), pp. 1–9. DOI: 10.1126/science.aam9288.

[37] S. Wehner, D. Elkouss, and R. Hanson. "Quantum internet: A vision for the road ahead". In: *Science* 362.6412 (Oct. 2018), eaam9288. ISSN: 0036-8075. DOI: 10.1126/science.aam9288. URL: http://www.sciencemag.org/lookup/doi/10.1126/science.aam9288.

3

NetQASM: A low-level instruction set architecture for hybrid quantum-classical programs in a quantum internet

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 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.

3.1 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

This chapter is based on the publication: 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". In: *Quantum Science and Technology* 7.3 (2022), p. 035023. DOI: 10.1088/2058-9565/ac753f.

A. Dahlberg and **B. van der Vecht** contributed equally to this work.

only classical communication. There are many applications [72] 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 [4] and secure blind quantum computing [12] or (2) allowing something to be done faster or more efficiently such as exponential savings in communication [11] and extending the baseline of telescopes [36]. 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 [40, 42, 43, 45, 47, 56, 65]. 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. (There have been examples of experiments with some simple logic but only with a very limited number of pre-loaded decision-branches.)

As explained in Chapter 2, quantum networks consist of *nodes* that are connected by *channels* (Figure 2.1). 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 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 [1]; 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.

As also explained in Chapter 2, 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) [4, 23], leader election protocols [28, 51], and Blind Quantum Computation (BQC) [72]. 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 3.1. These programs,

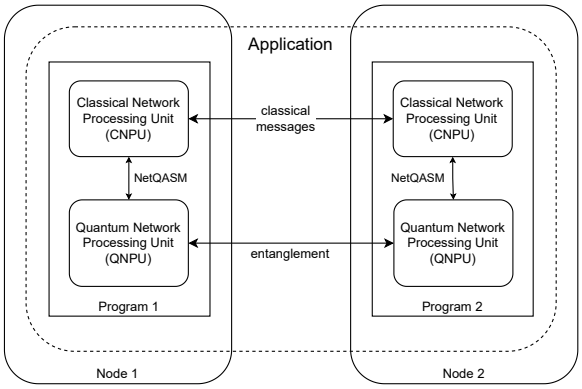


Figure 3.1: A quantum network application consists of a program for each of the nodes involved in the application. Each program is locally executed by the node. Program execution on each node is split into execution in a Classical Network Processing Unit (CNPU), which can send and receive classical messages, and a Quantum Network Processing Unit (QNPU), 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.

running on different nodes, are in principle programmed and executed by independent actors (such as a client that programs and executes a BQC client program, which communicates with a completely independent entity that acts as a BQC server and programs and executes its own BQC server program). We also note that programs often need to *wait* for some event to happen (such as arrival of a message from a remote node), which presents a motivation for *multitasking* (where waiting times may be filled by execution of other programs). We refer back to Chapter 2 for more information about quantum network programs.

3.1.1 Contribution

In this chapter we introduce an abstract model—including a Quantum Network Processing Unit (QNPU)— for end-nodes in a quantum network, which we define in Section 3.2. We then propose Quantum Network Assembly Language (NetQASM), an instruction set architecture that can be used to run arbitrary programs (of the form described in Figure 3.2) 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 CNPU 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

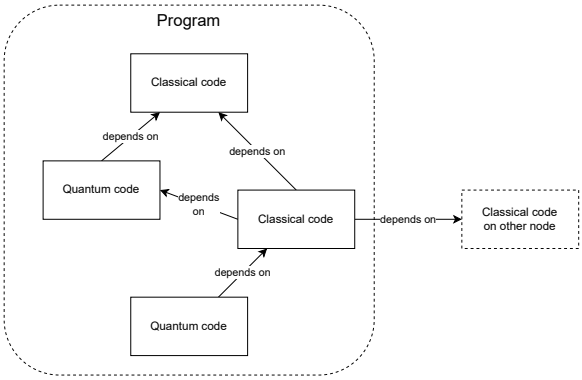


Figure 3.2: A program on a single node consists of different blocks of code, which can be quantum (pure quantum instructions with classical control in between), or classical (no quantum operations at all). These blocks may depend on each other in various ways. For example, the outcome of a measurement 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.

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 Nitrogen-Vacancy (NV) hardware, a promising platform for quantum network end-nodes [39, 70].

In our model, application-specific classical communication only happens at the CNPU (Figure 3.1). 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 [17, 52], (2) for the close interaction between classical and quantum operations, we use a shared-memory model for sharing classical data between the CNPU 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 3.4)—we make use of virtualized quantum memory, similar to virtual mem-

ory in classical computing [3], (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 [30], as part of our Quantum Network Explorer [63]. Through the SDK we have also enabled the quantum network simulators NetSquid [14] and SimulaQron [19] to run any application programmed in NetQASM.

We have evaluated NetQASM by simulating the execution of a teleportation application and a blind quantum computation 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 [61]. Furthermore, in Chapter 4, we present a full stack implementation that uses NetQASM.

3.1.2 Related work

In the field of quantum computing, a substantial amount of progress has been made related to developing architectures (e.g. [2, 8, 27, 37, 50, 57, 69, 71]), instruction sets (e.g. [16, 26, 35, 44, 46, 49, 54, 55, 67]) and compilers [21, 22, 33, 34, 38, 41, 53, 59, 60, 62, 66, 68, 73, 74]. 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 [16], `cQASM` [49], `eQASM` [26], `f-QASM` [54]. 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` [67], `Qiskit` [44], `Cirq` [35], `Q#` [55], `QuEST` [46].

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 [19], we introduced the Classical Quantum Combiner (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 3.4),

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 [17, 52]. 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 [15] 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:

1. Instructions for generating entanglement between remote nodes in the network need to be handled and integrated with the rest of the application, see Section 3.2 below.
2. 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.

3.1.3 Outline

In Section 3.2 we define relevant concepts and introduce the model of end-nodes that we use, including the QNPU. In Section 3.3 we discuss use-cases of a quantum network which NetQASM should support. In Section 3.4 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 3.5. In Section 3.6 and Section 3.7 we describe details about the NetQASM language and associated SDK. In Section 3.8 we quantitatively evaluate some of the design decision of NetQASM by benchmarking quality of execution using the quantum network simulator NetSquid [14, 64]. We conclude in Section 3.10.

3.2 Quantum node model

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 a Classical Network Processing Unit (CNPU) and a Quantum Network Processing Unit (QNPU). The CNPU 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 CNPU, 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 CNPU 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 CNPU and the QNPU is that the CNPU can do heavy and elaborate computation, while we assume the QNPU to be limited in processing power.

The CNPU can interact with the QNPU by for example sending instructions to do certain operations. The CNPU and the QNPU are logical components and may or may not be the same 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 CNPU and the QNPU is that the CNPU 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 3.3:

- **Processor:** The processor controls the other components of the QNPU and understands how to execute the operations specified by the CNPU. 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 Chapter 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 [17, 52].

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.

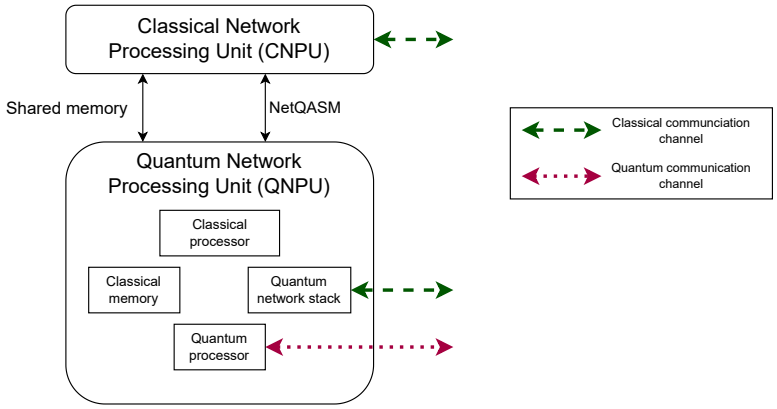


Figure 3.3: Overview of QNPU components and interfaces. The CNPU talks to the QNPU using NetQASM. The processor inside the QNPU can interact with all other components. Channels are connecting components with corresponding components in adjacent nodes in the network.

3.2.1 Applications and programs

As also mentioned in Section 2.2, 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 this thesis.

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

3.3 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 [16] 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 *blind quantum computation*, 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 [25]. 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 (MD) [17] type generation is required, where entangled state is measured directly, without storing in memory, to obtain correlated classical bits, such as in Quantum Key Distribution (QKD). However, in many cases a Create-Keep (CK) [17] 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 blind quantum computation, 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.

3.4 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 3.3). 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 3.2).
  Further requirements arise in platforms with a limited number of communication qubits. The extreme case is nitrogen-vacancy 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 [17, 52], 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 3.3 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 [15]. 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 3.2, we expect end-nodes to consist of a CNPU and a QNPU. These two components have different capabilities. For example, only the CNPU 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 CNPU and the QNPU (see Section 3.2). Indeed, only the QNPU can do quantum-related operations, whereas the CNPU needs to do classical communication. In order to make these work together, the two components have to share data somehow. This includes the CNPU requesting operations on the QNPU, and sending the following from the QNPU to the CNPU: (1) measurement outcomes of qubits, (2) information about entanglement generation, in particular a way to identify entangled pairs. This communication between CNPU and QNPU needs to be done during runtime of the program. This is in contrast to local quantum computa-

tion, 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 CNPU 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 3.1). The challenge is therefore to come up with an architecture for the interaction between the CNPU and the QNPU, as well as a way to let QNPU execution not take too long.
- **Platform-independence:** As explained in Section 3.1, 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 3.3 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.

3.5 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 3.6.

3.5.1 Interface between CNPU and QNPU

Execution model

As described in Section 3.2, and also in Section 3.4 program execution is split across the CNPU and the QNPU. Since the QNPU is assumed to have limited processing power (Section 3.2), our design lets the CNPU do most of the classical processing. The program blocks (Figure 3.2) are hence spread over two separate systems: blocks of purely classical code are executed by the CNPU, 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 CNPU, and we do not impose a restriction to its format. In our implementation (Section 3.7), we use Python. This classical code on the CNPU also handles all application-level classical communication between nodes, since it cannot be done on the QNPU.

We let the CNPU initiate a program. Whenever quantum code needs to be executed, the CNPU delegates this to the QNPU. Since processing delay should be minimized (Section 3.4), the communication between CNPU 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 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 3.2.

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 CNPU. Before receiving subroutines, the CNPU first *registers* a program at the QNPU. The QNPU then sets up the classical and quantum memories (see below) for this program. Then, the CNPU may send subroutines to the QNPU for execution.

Shared classical memory

Since classical and quantum blocks in the code (as per Figure 3.2) can depend on each other, the CNPU 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 CNPU 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 3.4).

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 CNPU 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 CNPU 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), 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

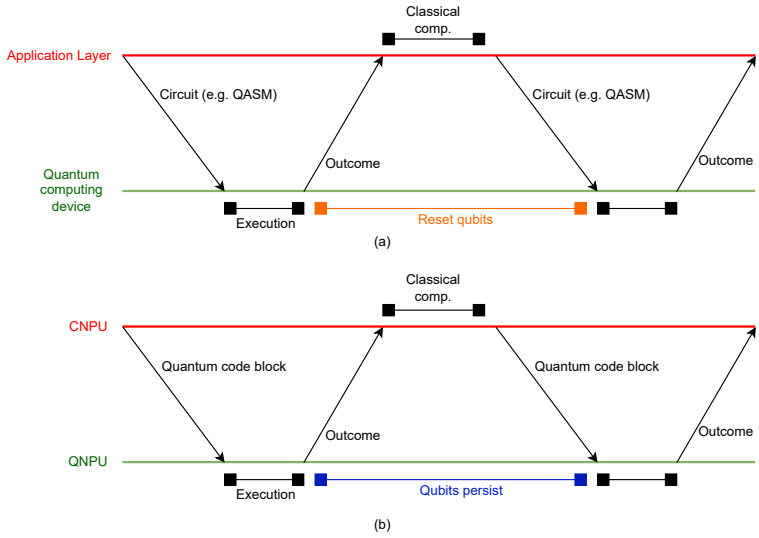


Figure 3.4: Program interaction between the CNPU and a quantum device in both the case of *hybrid-quantum computing* (a) and quantum networks (b). In the case of hybrid-quantum computing, qubits are reset in between circuits (in e.g. QASM). For quantum internet programs the qubits should on the other hand be kept in memory, since they might be entangled with another node and intended to be used further.

multiple measurement outcomes when doing multiple entangle-and-measure operations. The CNPU 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.

Unit modules

In order to support systems with multitasking (Section 3.4), 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 3.5). 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.

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 CNPU to do

processing while qubits are in memory. The following sequence of operations provides an example. (1) The CNPU 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 CNPU about it. There is now a qubit in the program's memory that is entangled with some qubit in R. (3) The CNPU does some classical processing and waits for a classical message from (the CNPU of) R. (4) Based on the contents of the message, the CNPU 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 3.4).

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 3.4). We use an approach that is similar to virtual memory in classical systems [3]. 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 3.4). 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.

3.5.2 NetQASM language

Instructions

As explained in Section 3.5.1, the CNPU 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

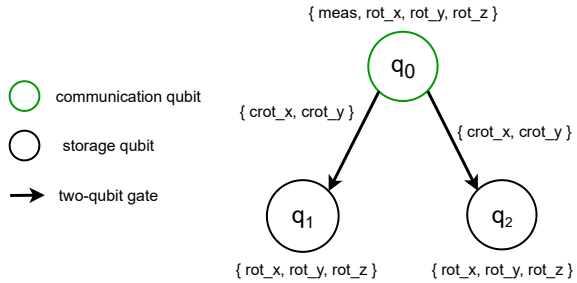


Figure 3.5: Example of a unit-module topology on a platform using nitrogen-vacancy centers in diamond. A unit-module is a hypergraph [6], with associated information on both nodes and edges. Each node represents a virtual qubit, containing information about (1) its qubit type (communication or storage), (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.

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 CNPU 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 3.4). We note the obvious performance gain by being able to do control logic without having to go back to the CNPU. 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 3.4). Therefore, NetQASM uses *flavors* of quantum instructions (Section 3.5.2). 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 [57]. See Appendix A.6 for more details on NetQASM instructions.

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 en-

```
array 10 @0   // array for writing EPR results to
array 1 @1    // array with virtual IDs for entangled qubits to
  generate
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 // wait until results for first pair are
  available
set Q0 0
meas Q0 M0    // measure the entangled qubit, store result in M0
qfree Q0      // free the qubit
ret_reg M0    // return measurement outcome
```

Figure 3.6: 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.

tanglement, 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 [17], including the network layer from [52]. 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 [17, 52]. NetQASM provides a way for programs to open such connections in the form of *EPR sockets*. The CNPU 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.

Flavors

We want to keep NetQASM platform-independent. However, we also want the potential for platform-specific optimization (Section 3.4). 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 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 CNPU 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.

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 3.7. 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. Examples are given in Section 3.7 and the Appendix.

3.6 Implementation

3.6.1 Interface between CNPU and QNPU

Here we explain the flow of messages between the CNPU and the QNPU. The CNPU starts by declaring the registration of an application, including resource-requirements for the application. After this, the CNPU sends some number of subroutines for the QNPU to execute before declaring the application is finished. See Figure 3.7 for a sequence diagram and below for a definition of the messages. In Section 3.6.2 we will describe in more details the content of the subroutines and the format of instructions. The QNPU returns to the CNPU an assigned application ID for the registered application and returns data based on the subroutines executed.

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

Figure 3.7 shows an example of a message exchange between the CNPU and the QNPU. The content of these messages is further detailed in Appendix A.1.

3.6.2 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 [16, 26, 49, 54].

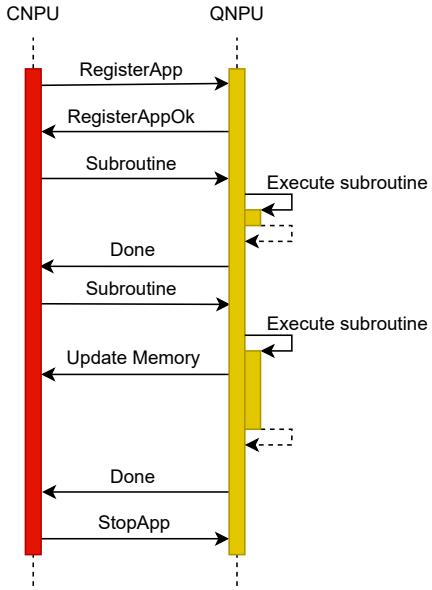


Figure 3.7: Flow of messages between the CNPU and the QNPU.

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 A.2.

3.6.3 Instructions

There are eight groups of instructions in the **core** of NetQASM. Also summarized in Figure 3.8, 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.

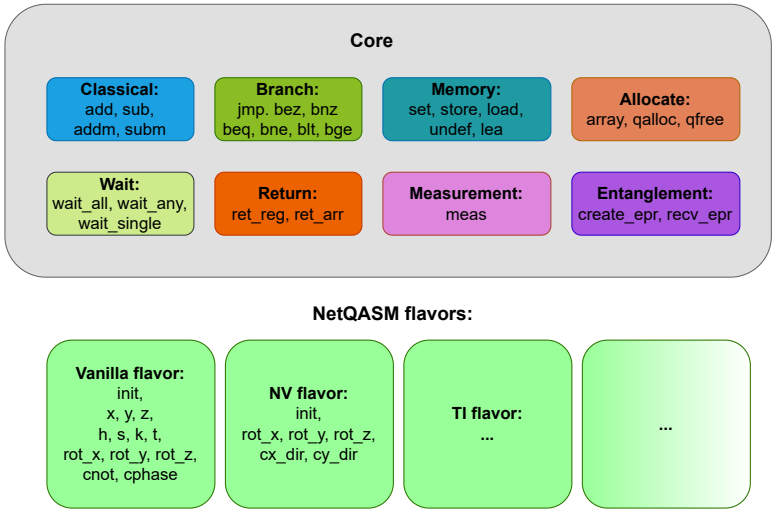


Figure 3.8: The **core** of NetQASM consists of eight groups of instructions. The quantum gates are defined as a set of software-visible gates part of a NetQASM **flavor**. The **vanilla flavor** is the unique platform-independent NetQASM **flavor** of NetQASM, which can be used by a compiler.

- **Return:** Returning classical values from the QNPU to the CNPU. In our implementation we implement this by having the QNPU write to the shared memory so that the CNPU 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 3.4. There is a single platform-independent NetQASM **flavor** which we call the **vanilla flavor**, see Figure 3.8. The **vanilla flavor** can be used as an intermediate representation for a compiler.

3.6.4 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 3.9. 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 3.7.

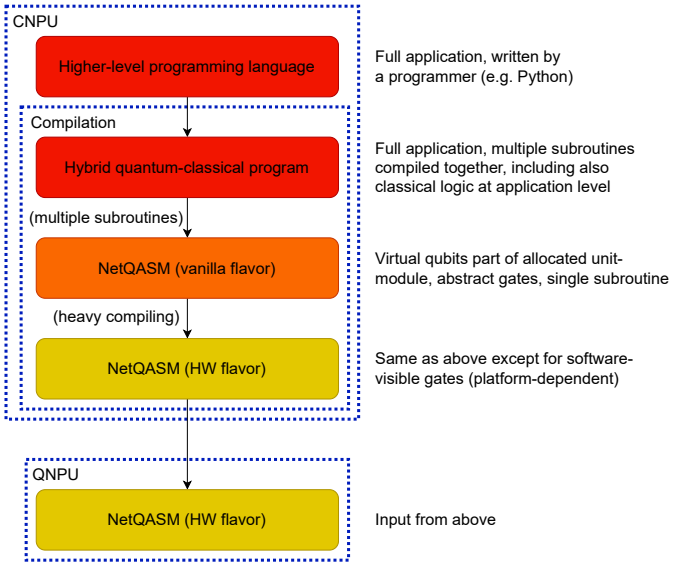


Figure 3.9: Compilation steps from higher-level programming language, to the NetQASM **flavor** exposed by the specific platform. What is contained at each level is further specified to the right of the diagram.

3.7 Python SDK

We implemented NetQASM by developing a Software Development Kit (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 3.8 we use this SDK to evaluate some of the design decisions of NetQASM.

We refer to the docs at [30] 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 A.8.2 we provide a few more examples of applications in the SDK and their corresponding NetQASM subroutines.

All code can be found at [30] and [31], 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 [14, 64] and SimulaQron [19], and (4) implemented applications in NetQASM, including: anonymous transmission [13], BB84 [4], blind quantum computing [10, 24], CHSH game [48], performing a distributed CNOT [20], magic square game [9], teleportation [5].

SDK. The SDK of NetQASM uses a similar framework to the SDK used by the predecessor CQC [29]. 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 CNPU does with the QNPU. More information about supported backends can be found below in Section 3.7. 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 CNPU.

```
# 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
```

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 [31] (using NetSquid [14, 64]) and SimulaQron [19]. A physical implementation of QNPU running on quantum hardware is being worked on at the time of writing. Using the SDK provided at [30], 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 [30].

3.8 Evaluation

We evaluate two of the design choices that we made for NetQASM: (1) exposing unit-modules to the CNPU 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 blind quantum computation application. These examples also showcase the ability of NetQASM to express general quantum internet applications.

We have implemented a simulator, called SquidASM [31], that simulates a network in which end-nodes have the internal architecture as described in Section 3.2, that is, with an CNPU and a QNPU. The simulator internally uses NetSquid [64], which was made specifically for the simulation of quantum networks. SquidASM executes programs written using the SDK (Section 3.7), 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 [32].

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 blind quantum computation* [24]. 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 nitrogen-vacancies (NV) in diamond, that can do heralded entanglement generation between each other. The simulated hardware uses noise models that are also used in [14]. For more details, see Section 3.9.

A note on how we chose what to evaluate and what not. We listed several design considerations in Section 3.4. We addressed these in our design decisions (Section 3.5). For some of these, it is straightforward to see how they address a certain consideration, such as conditionals allowing for fast runtime feedback, and unit modules for allowing

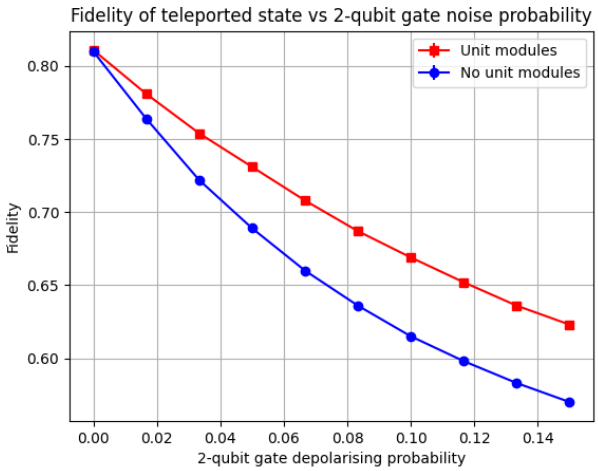


Figure 3.10: Average fidelity between the original state at the sender and the final state at the server, as a function of the depolarizing noise of the native two-qubit gate of the NV-platform, both for the case of performing step 6 after (**No unit modules**) and before (**Unit modules**) step 4 and 5. Execution time of the native two-qubit gate is set to 0.5 ms. The rest of the parameters used are listed in Section 3.9. Each point is the average over each of the six Pauli states as initial state, repeated 100 times.

multitasking, as explained in Section 3.5. 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 Nitrogen-Vacancy hardware for our nodes. This has two reasons. First, it is a promising hardware platform for quantum network nodes [70] 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 [61]. 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.

3.8.1 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 CNPU 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 CNPU 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 CNPU 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 nitrogen-vacancy centers in diamond (NV) and use well-established models for both the noise and operations supported on such platforms, see Section 3.9. 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 3.7), 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 implementing 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's 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 :

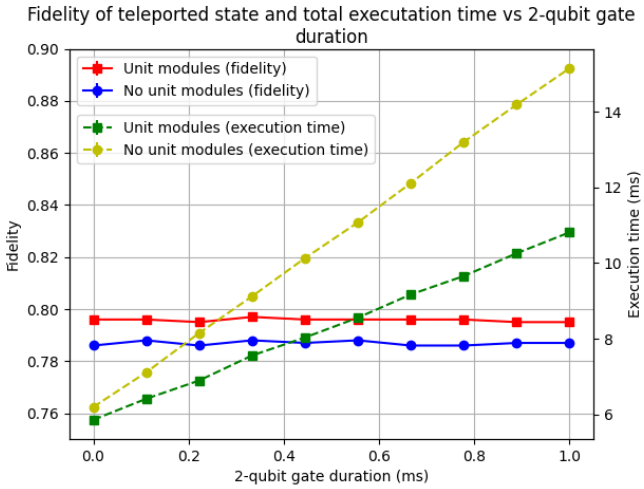


Figure 3.11: Average fidelity of the teleported state (left y-axis, solid lines) and total execution time of the teleportation application (right y-axis, dashed lines) as a function of the execution time of the native two-qubit gate of the NV-platform, both for the case of performing step 6 after (**No unit modules**) and before (**Unit modules**) step 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 Section 3.9. 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.

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 step 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's 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 3.5. 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 [43]) 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 [7] or [17].

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 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 CNPU, 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 (**Unit modules (UM)**) or not (**No unit-modules (NUM)**), we check the average fidelity of the teleported state as a function of the gate noise (Figure 3.10), as well as the average fidelity and execution time as a function of gate duration (Figure 3.10), 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 noise 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 3.10 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.

3.8.2 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 3.12, 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.

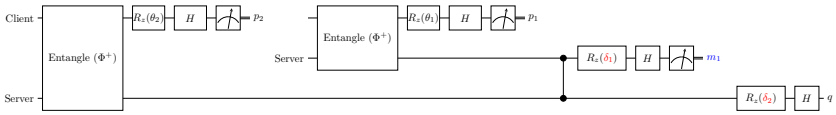


Figure 3.12: Circuit representation of the simulated BQC application. The client remotely prepares two qubits on the server, by twice creating an entangled pair with the server followed by a local measurement. The server locally entangles its two qubits (cphase gate). Then, the client and server use 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.

3

- **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 CNPUs, so these operations are not compiled to NetQASM subroutines. More details about the simulation can be found in Section 3.9.

The protocol is a verifiable blind quantum protocol [24], which means that the circuit in Figure 3.12 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 3.13. 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 3.12, the client remotely prepares two states on the server by twice creating entanglement and measuring its own half of the EPR pair. In Figure 3.14 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.

3.8.3 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 CNPU) has also been evaluated in [57]. 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.

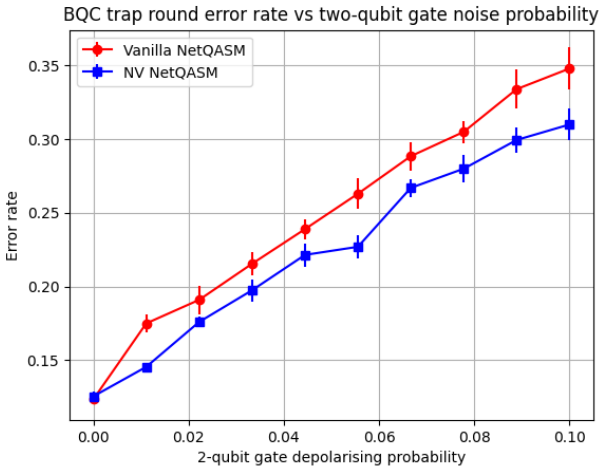


Figure 3.13: Average error rate of trap rounds for the circuit of Figure 3.12. Each point is the average over four combinations of θ_1 and θ_2, each used in 500 trap rounds. It can be seen that using the vanilla (platform-independent) NetQASM flavor results in a worse (higher) error rate on average.

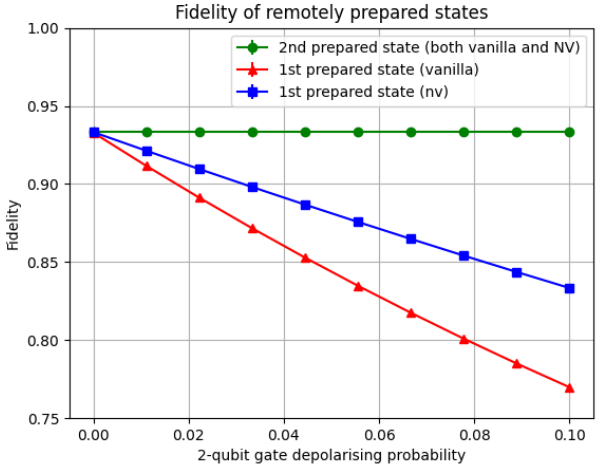


Figure 3.14: Fidelity of the two remotely prepared states on the server in the BQC application. To remotely prepare a state, the client and server first create an EPR pair, and the client then measures its half in a specific basis while the server keeps its half stored in the communication qubit. This 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 3.13. Error bars are negligible.

3.9 Simulation details

In this section we detail how simulations in Section 3.8 were performed and what models and parameters were used. All simulations used the NetQASM SDK [30], using NetSquid [14, 64] as the underlying simulator. All code used in these simulations can also be found at [31].

3.9.1 Noise model

In both the teleportation and the blind quantum computing scenario we used the same model for nitrogen-vacancy centres in diamonds as was used in [17] and [14]. All gates specified by the application in the SDK were translated to NV-specific gates, see Table 3.1, using a simple compiler without any optimization. The parameters used in the model from [17] are listed in Tables 3.1 and 3.2, 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

$$U_{\text{ec}_x}(\alpha) = \begin{pmatrix} R_x(\alpha) & 0 \\ 0 & R_x(-\alpha) \end{pmatrix} \tag{3.1}$$

$$U_{\text{ec}_y}(\alpha) = \begin{pmatrix} R_y(\alpha) & 0 \\ 0 & R_y(-\alpha) \end{pmatrix} \tag{3.2}$$

$$\tag{3.3}$$

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 3.2 are applied as depolarising noise by applying the perfect operation, producing the state ρ_{ideal}, and mapping this to

$$\rho_{\text{noisy}} = (1-p)\rho_{\text{ideal}} + \frac{p}{3} X\rho_{\text{ideal}} X + \frac{p}{3} Y\rho_{\text{ideal}} Y + \frac{p}{3} Z\rho_{\text{ideal}} Z \tag{3.4}$$

where X, Y and Z are the Pauli operators in Equations (A.1) to (A.3), $p = \frac{4}{3}(1-F)$, with F being the value specific in Table 3.2. Decoherence noise is specific as T_1 (energy/thermal relaxation time) and T_2 (dephasing time) [58].

3.9.2 BQC application and flavors

In Section 3.8.2 we simulated the blind quantum computation (BQC) application from Figure 3.12. The code for this is available at [31].

In the scenario when the application code was compiled to subroutines with the vanilla flavor, 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:

Gate	Durations (ns)	Explanation
`electron_init`	2e3	Initialize a comm. qubit (electron) to $\|0\rangle$
`electron_rot`	5	Single-qubit rotation on communication qubit (electron)
`measure`	3.7e3	Measure communication qubit (electron)
`carbon_init`	3.1e5	Initialize a storage qubit (carbon) to $\|0\rangle$
`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 (Equations (3.1) and (3.2))

Table 3.1: Gate durations for scenario **B** of Section 3.8. t is the value being swept in Figure 3.11. All values are from [17].

Parameter	Value	Explanation
`electron_T1`	1 hour	T_1 of communication qubit (electron)
`electron_T2`	1.46 seconds	T_2 of communication qubit (electron)
`electron_init`	0.99)	Fidelity to initialize comm. qubit (electron)
`electron_rot`	1.0	Fidelity for Z-rotation on communication qubit (electron)
`carbon_T1`	10 hours	T_1 of storage qubit (carbon)
`carbon_T2`	1 second	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.

Table 3.2: Noise parameters for used in the simulations of Section 3.8. f is the value being swept in Figure 3.10 and Figure 3.13. All fidelities are realized by a applying depolarising noise as in Equation (3.4). All values are from [14], 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.

```
rot_y PI_OVER_2
rot_x PI
```

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:

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

3.10 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 blind quantum computation, have already been implemented. A simple compiler has been implemented that can translate code written in the higher-level SDK into NetQASM.

3.11 Data availability

The data that support the findings of this study are openly available at the following DOI: `https://doi.org/10.4121/21355329`. The software packages created for this work (and

used for the simulation) are https://github.com/QuTech-Delft/netqasm (NetQASM) and https://github.com/QuTech-Delft/squidasm (SquidASM). These packages are also part of the Quantum Network Explorer (QNE) SDK found at https://quantum-network.com.

References

[1] M. H. Abobeih, J. Cramer, M. A. Bakker, N. Kalb, M. Markham, D. J. Twitchen, and T. H. Taminiau. "One-second coherence for a single electron spin coupled to a multi-qubit nuclear-spin environment". In: *Nature Communications* 9.1 (Dec. 2018), p. 2552. ISSN: 2041-1723. DOI: 10.1038/s41467-018-04916-z. arXiv: 1801.01196. URL: http://www.nature.com/articles/s41467-018-04916-z.

[2] M. Amy and V. Gheorghiu. *staq – A full-stack quantum processing toolkit*. 2019. arXiv: 1912.06070 [quant-ph].

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[4] C. H. Bennett and G. Brassard. "Quantum Cryptography: Public Key Distribution, and Coin-Tossing". In: *Proc. 1984 IEEE International Conference on Computers, Systems, and Signal Processing*. 1984, pp. 175–179. ISBN: doi:10.1016/j.tcs.2011.08.039. DOI: 10.1016/j.tcs.2011.08.039.

[5] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. "Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels". In: *Physical Review Letters* 70.13 (1993), p. 1895.

[6] C. Berge. *Hypergraphs: Combinatorics of Finite Sets*. North-Holland Mathematical Library. Elsevier Science, 1984. ISBN: 9780080880235. URL: https://books.google.nl/books?id=jEyfse-EKf8C.

[7] H. Bernien. "Control, measurement and entanglement of remote quantum spin registers in diamond". PhD Thesis. TU Delft, 2014. DOI: 10.4233/uuid:75130c37-edb5-4a34-ac2f-c156d377ca55.

[8] J. E. Bourassa, R. N. Alexander, M. Vasmer, A. Patil, I. Tzitrin, T. Matsuura, D. Su, B. Q. Baragiola, S. Guha, G. Dauphinais, K. K. Sabapathy, N. C. Menicucci, and I. Dhand. "Blueprint for a Scalable Photonic Fault-Tolerant Quantum Computer". In: *arXiv preprint arXiv:2010.02905* (2020).

[9] G. Brassard, R. Cleve, and A. Tapp. "Cost of Exactly Simulating Quantum Entanglement with Classical Communication". In: *Phys. Rev. Lett.* 83 (9 Aug. 1999), pp. 1874–1877. DOI: 10.1103/PhysRevLett.83.1874. URL: https://link.aps.org/doi/10.1103/PhysRevLett.83.1874.

[10] A. Broadbent, J. Fitzsimons, and E. Kashefi. "Universal Blind Quantum Computation". In: *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '09. Washington, DC, USA: IEEE, Oct. 2009, pp. 517–526. ISBN: 978-1-4244-5116-6. DOI: 10.1109/FOCS.2009.36. arXiv: 0807.4154. URL: http://ieeexplore.ieee.org/document/5438603/.

[11] H. Buhrman, R. Cleve, S. Massar, and R. de Wolf. "Nonlocality and communication complexity". In: *Reviews of Modern Physics* 82.1 (Mar. 2010), pp. 665–698. ISSN: 0034-6861. DOI: 10.1103/RevModPhys.82.665. arXiv: 0907.3584. URL: https://link.aps.org/doi/10.1103/RevModPhys.82.665.

[12] A. M. Childs. "Secure Assisted Quantum Computation". In: *Quantum Info. Comput.* 5.6 (Sept. 2005), pp. 456–466. ISSN: 1533-7146. arXiv: 0111046 [quant-ph]. URL: http://dl.acm.org/citation.cfm?id=2011670.2011674.

[13] M. Christandl and S. Wehner. "Quantum Anonymous Transmissions". In: *Advances in Cryptology - ASIACRYPT 2005*. Ed. by B. Roy. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 217–235. ISBN: 978-3-540-32267-2.

[14] T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J. de Oliveira Filho, M. Papendrecht, J. Rabbie, F. Rozpędek, M. Skrzypczyk, et al. "Netsquid, a network simulator for quantum information using discrete events". In: *Communications Physics* 4.1 (2021), pp. 1–15.

[15] A. W. Cross, A. Javadi-Abhari, T. Alexander, N. de Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, J. Smolin, J. M. Gambetta, and B. R. Johnson. "OpenQASM 3: A broader and deeper quantum assembly language". In: *arXiv preprint arXiv:2104.14722* (2021).

[16] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. "Open Quantum Assembly Language". In: *arXiv preprint arXiv:1707.03429* (2017).

[17] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveria Filho, R. Hanson, and S. Wehner. "A Link Layer Protocol for Quantum Networks". In: *ACM SIGCOMM 2019 Conference*. SIGCOMM '19. Beijing, China: ACM, 2019, p. 15. ISBN: 978-1-4503-5956-6/19/09. DOI: 10.1145/3341302.3342070. URL: http://doi.acm.org/10.1145/3341302.3342070.

[19] A. Dahlberg and S. Wehner. "SimulaQron—a simulator for developing quantum internet software". In: *Quantum Science and Technology* 4.1 (Sept. 2018), p. 015001. DOI: 10.1088/2058-9565/aad56e. URL: https://doi.org/10.1088%2F2058-9565%2Faad56e.

[20] V. S. Denchev and G. Pandurangan. "Distributed quantum computing: A new frontier in distributed systems or science fiction?" In: *ACM SIGACT News* 39.3 (2008), pp. 77–95. ISSN: 01635700. DOI: 10.1145/1412700.1412718.

[21] Y. Ding, X.-C. Wu, A. Holmes, A. Wiseth, D. Franklin, M. Martonosi, and F. T. Chong. *SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation*. 2020. arXiv: 2004.08539 [quant-ph].

[22] B. Dury and O. Di Matteo. "A QUBO Formulation for Qubit Allocation". In: *arXiv preprint arXiv:2009.00140* (2020).

[23] A. K. Ekert. "Quantum cryptography based on Bell's theorem". In: *Physical Review Letters* 67.6 (1991), p. 661.

[24] J. F. Fitzsimons and E. Kashefi. "Unconditionally verifiable blind quantum computation". In: *Physical Review A* 96.1 (2017), p. 012303.

[25] J. F. Fitzsimons. "Private quantum computation: an introduction to blind quantum computing and related protocols". In: *npj Quantum Information* 3.1 (June 2017), p. 23. ISSN: 2056-6387. DOI: 10.1038/s41534-017-0025-3. URL: https://doi.org/10.1038/s41534-017-0025-3.

[26] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. "eQASM: An Executable Quantum Instruction Set Architecture". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 224–237. DOI: 10.1109/HPCA.2019.00040.

[27] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. "An Experimental Microarchitecture for a Superconducting Quantum Processor". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: Association for Computing Machinery, 2017, pp. 813–825. ISBN: 9781450349529. DOI: 10.1145/3123939.3123952. URL: https://doi.org/10.1145/3123939.3123952.

[28] M. Ganz. "Quantum leader election". In: *arXiv preprint arXiv:0910.4952* (2009).

[29] *Git repository with code for CQC*. https://github.com/SoftwareQuTech/CQC-Python. 2021.

[30] *Git repository with code for NetQASM*. https://github.com/QuTech-Delft/netqasm. 2021.

[31] *Git repository with code for SquidASM*. https://github.com/QuTech-Delft/squidasm. 2021.

[32] *Git repository with simulation code and data used for the evaluation in this paper.* https://github.com/QuTech-Delft/netqasm-paper-data. 2022.

[33] P. Gokhale, A. Javadi-Abhari, N. Earnest, Y. Shi, and F. T. Chong. *Optimized Quantum Compilation for Near-Term Algorithms with OpenPulse*. 2020. arXiv: 2004.11205 [quant-ph].

[34] P. Gokhale, S. Koretsky, S. Huang, S. Majumder, A. Drucker, K. R. Brown, and F. T. Chong. "Quantum Fan-out: Circuit Optimizations and Technology Modeling". In: *arXiv preprint arXiv:2007.04246* (2020).

[35] Google. *Cirq*. https://cirq.readthedocs.io/en/stable/. 2020.

[36] D. Gottesman, T. Jennewein, and S. Croke. "Longer-baseline telescopes using quantum repeaters". In: *Physical Review Letters* 109.7 (2012), p. 070503.

[37] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. "Quipper: a scalable quantum programming language". In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 333–342. URL: https://dl.acm.org/doi/pdf/10.1145/2491956.2462177.

[38] T. Häner, D. S. Steiger, K. Svore, and M. Troyer. "A software methodology for compiling quantum programs". In: *Quantum Science and Technology* 3.2 (2018), p. 020501. URL: https://iopscience.iop.org/article/10.1088/2058-9565/aaa5cc/pdf.

[39] R. Hanson. "Realization of a multi-node quantum network of remote solid-state qubits". In: *Photonics for Quantum*. Vol. 11844. International Society for Optics and Photonics. 2021, p. 1184402.

[40] B. Hensen, H. Bernien, A. E. Dréau, A. Reiserer, N. Kalb, M. S. Blok, J. Ruitenberg, R. F. L. Vermeulen, R. N. Schouten, C. Abellán, W. Amaya, V. Pruneri, M. W. Mitchell, M. Markham, D. J. Twitchen, D. Elkouss, S. Wehner, T. H. Taminiau, and R. Hanson. "Loophole-free Bell inequality violation using electron spins separated by 1.3 kilometres". In: *Nature* 526.7575 (Oct. 2015), pp. 682–686. ISSN: 0028-0836. DOI: 10.1038/nature15759. arXiv: 1508.05949. URL: http://www.nature.com/articles/nature15759.

[41] K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks. "A Verified Optimizer for Quantum Circuits". In: *arXiv preprint arXiv:1912.02250* (2019).

[42] J. Hofmann, M. Krug, N. Ortegel, L. Gérard, M. Weber, W. Rosenfeld, and H. Weinfurter. "Heralded Entanglement Between Widely Separated Atoms". In: *Science* 337.6090 (2012), pp. 72–75. ISSN: 0036-8075. DOI: 10.1126/science.1221856. eprint: https://science.sciencemag.org/content/337/6090/72.full.pdf. URL: https://science.sciencemag.org/content/337/6090/72.

[43] P. C. Humphreys, N. Kalb, J. P. Morits, R. N. Schouten, R. F. Vermeulen, D. J. Twitchen, M. Markham, and R. Hanson. "Deterministic delivery of remote entanglement on a quantum network". In: *Nature* 558 (2018), pp. 268–273. ISSN: 14764687. DOI: 10.1038/s41586-018-0200-5.

[44] IBM. *Qiskit*. https://qiskit.org/. 2020.

[45] I. V. Inlek, C. Crocker, M. Lichtman, K. Sosnova, and C. Monroe. "Multispecies Trapped-Ion Node for Quantum Networking". In: *Physical Review Letters* 118.25 (2017), pp. 1–5. ISSN: 10797114. DOI: 10.1103/PhysRevLett.118.250502. arXiv: 1702.01062.

[46] T. Jones, A. Brown, I. Bush, and S. C. Benjamin. "QuEST and High Performance Simulation of Quantum Computers". In: *Scientific Reports* 9.1 (July 2019), p. 10736. ISSN: 2045-2322. DOI: 10.1038/s41598-019-47174-9. URL: https://doi.org/10.1038/s41598-019-47174-9.

[47] N. Kalb, A. A. Reiserer, P. C. Humphreys, J. J. W. Bakermans, S. J. Kamerling, N. H. Nickerson, S. C. Benjamin, D. J. Twitchen, M. Markham, and R. Hanson. "Entanglement distillation between solid-state quantum network nodes". In: *Science* 356.6341 (June 2017), pp. 928–932. ISSN: 0036-8075. DOI: 10.1126/science.aan0070. arXiv: 1703.03244. URL: http://www.sciencemag.org/lookup/doi/10.1126/science.aan0070.

[48] J. Kaniewski and S. Wehner. "Device-independent two-party cryptography secure against sequential attacks". In: *New Journal of Physics* 18.5 (2016). ISSN: 13672630. DOI: 10.1088/1367-2630/18/5/055004. arXiv: 1601.06752.

[49] N. Khammassi, G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. "cQASM v1.0: Towards a Common Quantum Assembly Language". In: *arXiv preprint arXiv:1805.09607* (2018).

[50] N. Khammassi, I. Ashraf, J. v. Someren, R. Nane, A. Krol, M. A. Rol, L. Lao, K. Bertels, and C. G. Almudever. "OpenQL: A portable quantum programming framework for quantum accelerators". In: *arXiv preprint arXiv:2005.13283* (2020). URL: https://arxiv.org/pdf/2005.13283.pdf.

[51] H. Kobayashi, K. Matsumoto, and S. Tani. "Simpler exact leader election via quantum reduction". In: *Chicago Journal of Theoretical Computer Science* 10 (2014), p. 2014.

[52] W. Kozlowski, A. Dahlberg, and S. Wehner. "Designing a Quantum Network Protocol". In: *arXiv preprint arXiv:2010.02575* (2020).

[53] L. Liu and X. Dou. *A New Qubits Mapping Mechanism for Multi-programming Quantum Computing*. 2020. arXiv: 2004.12854 [cs.DC].

[54] S. Liu, X. Wang, L. Zhou, J. Guan, Y. Li, Y. He, R. Duan, and M. Ying. "$Q|SI\rangle$: A Quantum Programming Environment". In: *arXiv preprint arXiv:1710.09500* (2017).

[55] Microsoft. *Q#*. https://docs.microsoft.com/en-us/quantum/. 2020.

[56] D. L. Moehring, P. Maunz, S. Olmschenk, K. C. Younge, D. N. Matsukevich, L.-M. Duan, and C. Monroe. "Entanglement of single-atom quantum bits at a distance". In: *Nature* 449.7158 (Sept. 2007), pp. 68–71. ISSN: 1476-4687. DOI: 10.1038/nature06118. URL: https://doi.org/10.1038/nature06118.

[57] P. Murali, N. M. Linke, M. Martonosi, A. J. Abhari, N. H. Nguyen, and C. H. Alderete. "Full-Stack, Real-System Quantum Computer Studies: Architectural Comparisons and Design Insights". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 527–540. ISBN: 9781450366694. DOI: 10.1145/3307650.3322273. URL: https://doi.org/10.1145/3307650.3322273.

[58] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. 10th Anniversary Edition. Cambridge: Cambridge University Press, 2010. ISBN: 9780511976667. DOI: 10.1017/CBO9780511976667. URL: http://ebooks.cambridge.org/ref/id/CBO9780511976667.

[59] S. Nishio, Y. Pan, T. Satoh, H. Amano, and R. V. Meter. "Extracting Success from IBM's 20-Qubit Machines Using Error-Aware Compilation". In: *ACM Journal on Emerging Technologies in Computing Systems* 16.3 (July 2020), pp. 1–25. ISSN: 1550-4840. DOI: 10.1145/3386162. URL: http://dx.doi.org/10.1145/3386162.

[60] S. Niu, A. Suau, G. Staffelbach, and A. Todri-Sanial. "A Hardware-Aware Heuristic for the Qubit Mapping Problem in the NISQ Era". In: *arXiv preprint arXiv:2010.03397* (2020). URL: https://arxiv.org/pdf/2010.03397.pdf.

[61] M. Pompili, C. D. Donne, I. t. Raa, B. van der Vecht, M. Skrzypczyk, G. Ferreira, L. de Kluijver, A. J. Stolk, S. L. Hermans, P. Pawełczak, et al. "Experimental demonstration of entanglement delivery using a quantum network stack". In: *arXiv preprint arXiv:2111.11332* (2021).

[62] M. G. Pozzi, S. J. Herbert, A. Sengupta, and R. D. Mullins. *Using Reinforcement Learning to Perform Qubit Routing in Quantum Compilers*. 2020. arXiv: 2007.15957 [quant-ph].

[63] *Quantum Network Explorer*. https://www.quantum-network.com. 2021.

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

[65] N. Sangouard, C. Simon, H. De Riedmatten, and N. Gisin. "Quantum repeaters based on atomic ensembles and linear optics". In: *Reviews of Modern Physics* 83.1 (2011), p. 33.

[66] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan. "t|ket⟩: A retargetable compiler for NISQ devices". In: *Quantum Science and Technology* (Apr. 2020). ISSN: 2058-9565. DOI: 10.1088/2058-9565/ab8e92. URL: http://dx.doi.org/10.1088/2058-9565/ab8e92.

[67] R. S. Smith, M. J. Curtis, and W. J. Zeng. "A Practical Quantum Instruction Set Architecture". In: *arXiv preprint arXiv:1608.03355* (2016).

[68] R. S. Smith, E. C. Peterson, M. G. Skilbeck, and E. J. Davis. *An Open-Source, Industrial-Strength Optimizing Compiler for Quantum Programs*. 2020. arXiv: 2003.13961 [quant-ph].

[69] D. S. Steiger, T. Häner, and M. Troyer. "ProjectQ: An Open Source Software Framework for Quantum Computing". In: *arXiv preprint* (Dec. 2016). arXiv: 1612.08091. URL: http://arxiv.org/abs/1612.08091.

[70] T. H. Taminiau, J. Cramer, T. van der Sar, V. V. Dobrovitski, and R. Hanson. "Universal control and error correction in multi-qubit spin registers in diamond". In: *Nature nanotechnology* 9.3 (Mar. 2014), pp. 171–176. DOI: 10.1038/nnano.2014.2.

[71] D. Wecker and K. M. Svore. "LIQUi|>: A software design architecture and domain-specific language for quantum computing". In: *arXiv preprint arXiv:1402.4467* (2014). URL: https://arxiv.org/pdf/1402.4467.pdf.

[72] S. Wehner, D. Elkouss, and R. Hanson. "Quantum internet: A vision for the road ahead". In: *Science* 362.6412 (Oct. 2018), eaam9288. ISSN: 0036-8075. DOI: 10.1126/science.aam9288. URL: http://www.sciencemag.org/lookup/doi/10.1126/science.aam9288.

[73] Y. Zhang, H. Deng, and Q. Li. "Context-Sensitive and Duration-Aware Qubit Mapping for Various NISQ Devices". In: *arXiv preprint arXiv:2001.06887* (2020).

[74] A. Zulehner and R. Wille. "Compiling SU (4) quantum circuits to IBM QX architectures". In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 2019, pp. 185–190. URL: https://dl.acm.org/doi/pdf/10.1145/3287624.3287704.

4

QNodeOS: An operating system for executing applications on quantum network nodes

The goal of future quantum networks is to enable new internet applications that are impossible to achieve using solely classical communication[51, 95, 99]. Up to now, demonstrations of quantum network applications[5, 33, 72] and functionalities[21, 42, 47, 58, 67, 75] on quantum processors have been performed in ad-hoc software that was specific to the experimental setup, programmed to perform one single task (the application experiment) directly into low-level control devices using expertise in experimental physics. Here, we report on the design and implementation of the first architecture capable of executing quantum network applications on quantum processors in platform-independent high-level software. We demonstrate the architecture's capability to execute applications in high-level software, by implementing it as a quantum network operating system — QNodeOS — and executing test programs including a delegated computation from a client to a server[14] on two quantum network nodes based on nitrogen-vacancy (NV) centers in diamond[19, 31]. We show how our architecture allows us to maximize the use of quantum network hardware, by multitasking different applications on a quantum network for the first time. Our architecture can be used to execute programs on any quantum processor platform corresponding to our system model, which we illustrate by demonstrating an additional driver for QNodeOS for a trapped-ion quantum network node based on a single $^{40}Ca^{+}$ atom[36]. Our architecture lays the groundwork for computer science research in the domain of quantum network programming, and paves the way for the development of software that can bring quantum network technology to society.

This chapter is based on the article: C. D. Donne, M. Iuliano, **B. van der Vecht**, G. M. Ferreira, H. Jirovská, T. van der Steenhoven, A. Dahlberg, M. Skrzypczyk, D. Fioretto, M. Teller, P. Filippov, A. R.-P. Montblanch, J. Fischer, B. van Ommen, N. Demetriou, D. Leichtle, L. Music, H. Ollivier, I. t. Raa, W. Kozlowski, T. Taminiau, P. Pawełczak, T. Northup, R. Hanson, and S. Wehner. "An operating system for executing applications on quantum network nodes". In: *Nature* 639 (Mar. 12, 2025), pp. 321–328. DOI: 10.1038/s41586-025-08704-w.

C. Delle Donne, M. Iuliano and **B. van der Vecht** contributed equally to this work.

4.1 Introduction

The first quantum networks linking multiple quantum processors as end nodes have recently been realized as physics experiments in laboratories [43, 56, 69, 77, 82, 87, 88] and fiber networks [52, 63, 89], opening the tantalizing possibility of realizing advanced quantum network applications [99] such as data consistency in the cloud [7], privacy-enhancing proofs of deletion [79], exponential savings in communication [40], or secure quantum computing in the cloud [14, 20]. Demonstrations relied either on ad-hoc software, or chose to establish that hardware parameters were in principle good enough to support a given quantum network application, although the application itself was not realized [64, 72, 100]. While quantum nodes have been linked at the hardware level [43, 48, 52, 56, 60, 69, 77, 82, 87, 88, 89], including the design [2, 25, 54, 76] and realization [70, 78] of network stacks to manage entanglement generation, a critical innovation required to make quantum networks useful is lacking: an architecture enabling the execution of quantum applications.

It is a major challenge to design and implement an architecture that can enable the execution of arbitrary quantum network applications on quantum processor end nodes (Figure 4.1), while enabling programming in high-level software that neither depends on the underlying quantum hardware, nor requires the programmer to understand the physics of the underlying devices. In the domain of the conventional internet, the possibility of programming arbitrary internet applications in high-level software has led to the realization of radically new communication applications by diverse communities, which had a transformative impact on our society [17]. What's more, the advent of programmable hardware and new application areas sparked novel fields of computer science research and guided further hardware development (e.g. network programming and protocols, distributed systems, internet of things, and more). A similar development is underway in quantum computing, where the availability of high-level programming tools allows a broad participation in developing applications [80].

In realizing the first such system architecture we overcome all challenges below, including both fundamental challenges that are inherent to quantum network applications at any scale, as well as technological challenges that arise from the current state of the art of quantum network hardware.

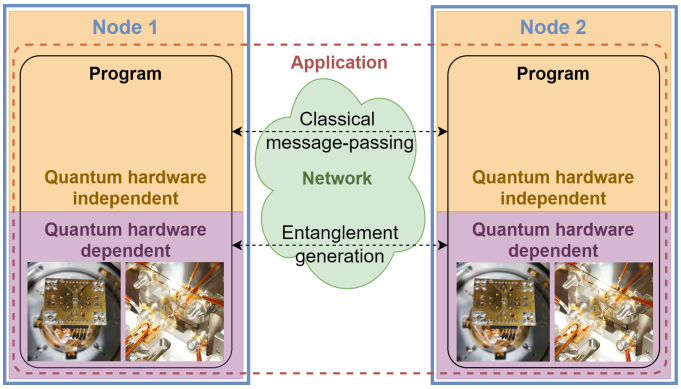


Figure 4.1: **Application Paradigm.** A quantum networking application consists of multiple programs, each running on one of the end nodes [26]. An end node is a device in a quantum network that executes user applications. A network stack enables entanglement generation between end nodes over a quantum network (Figure 4.6). The distinct programs at each end node can only interact via (1) quantum communication (e.g. entanglement generation) and (2) classical communication. This allows a programmer to realize security-sensitive applications, but prohibits a global orchestration of the quantum execution, like one might do in (distributed) quantum computing [16] in which a single quantum program is executed on multiple nodes. Our architecture allows programs to be written in high-level quantum hardware independent software, and executed on a quantum hardware independent system that controls a hardware dependent system (QDevice, Figure 4.2) such as a nitrogen-vacancy (NV) center node with a diamond chip (photo taken by authors, left images) or a trapped-ion quantum node [93] (right images). These platforms constitute physically very different QDevice systems, but can both be programmed by our architecture.

4.2 Design considerations and challenges

Interactive Classical-Quantum Execution. The execution of quantum network applications requires a continuing interaction between the quantum and classical parts of the execution, including interactions between different programs (Figure 4.1). For example, during secure quantum computing in the cloud [14, 65], the program on the server is waiting for classical messages from a remote client program before continuing the quantum execution at the server. This is in sharp contrast to quantum computing applications, where a quantum application is a single program that can be executed in one batch, without the need to keep quantum states live while waiting for input from other programs. In quantum computing, only relatively low-level and predictable interactions between classical and quantum processing are realized, such as in quantum error correction [61], or mid-circuit measurements [13]. Higher-level classical-quantum interactions in quantum computing [11] do not keep qubits live in memory.

We assume that the programs are divided into classical and quantum blocks of instructions (by a programmer or a compiler). Classical blocks consist of local classical operations executed on a conventional classical processor, as well as networked classical operations (i.e. sending messages to remote nodes) executed using network devices. Quantum blocks consist of local quantum operations (gates, measurements, classical control logic), as well as networked quantum operations (entanglement generation) executed on quantum hardware. A single quantum block, in essence, corresponds to a program in quantum computing, and may contain simple classical control logic, such as for the purpose of mid-circuit

measurements [13].

Different Hardware Platforms. Interfacing with different hardware platforms presents technological challenges: currently, a clear line between software and hardware has not been defined, and the low-level control of present-day quantum processor hardware has been built to conduct physics experiments. Early microarchitectures [10, 37] and operating systems [39, 53] for quantum computing do not address the execution of quantum network applications. We thus have to define a hardware abstraction layer (HAL), capable of interfacing with quantum network processors, including present-day setups.

Timescales. It is a fundamental challenge that different parts of such a system operate at vastly different timescales. For nodes separated by hundreds of kilometers, the duration of network operations is in the millisecond (ms) regime, and some applications [99] need significant local classical processing (ms). In contrast, the time to execute quantum operations on processing nodes is in the regime of microseconds (μs), and the low-level control (including timing synchronization between neighboring nodes to generate entanglement [44]) requires nanosecond (ns) precision.

Memory Lifetimes. Present-day quantum network nodes have short coherence times, posing a technological challenge to ensure operations are executed within the timeframe allowed by the quantum memory.

Scheduling Local and Network Operations. Entanglement is a key resource for quantum network applications [99]. In contrast to classical networking, entanglement is a form of stateful connection already at the physical layer where both nodes hold one qubit. Our architecture should allow for the execution of applications at end nodes, and these may be separated by a large quantum network that facilitates entanglement generation between them. This can be achieved by the implementation of a network stack [25, 54]. A technological challenge arises in the integration of such a network stack with the application stack of QNodeOS, when employing heralded entanglement generation at the physical layer [25] (as done in all current demonstrations linking quantum processors [78, 89]). Heralded entanglement generation requires agreement between neighboring network nodes to trigger entanglement generation in precise time-bins [25], organized into a network schedule [85] that dictates when nodes make entanglement. Such a schedule could be set by a centralized controller [85], or by a distributed protocol (see e.g. [25]).

It is a technological challenge to manage the interdependencies between the schedule of local operations, and the networked operations, since in all current processing node implementations [34, 77], entanglement generation cannot be performed simultaneously with local operations [57, 77]. While interdependencies may be mitigated in the future [97], this implies that we cannot schedule (i.e. decide when to execute) the execution of local quantum operations independently of the network schedule.

Multitasking. When executing (quantum) network applications, one node is typically idle while waiting for the other node before it can continue execution. For example, a client program may need to wait for a server to finish processing and send a message. It is hence a fundamental challenge how we can increase the utility of the system by performing multitasking [29, 68], that is, allowing the concurrent execution of several programs at once to make use of idle times. Consequently, there is a need for managing state

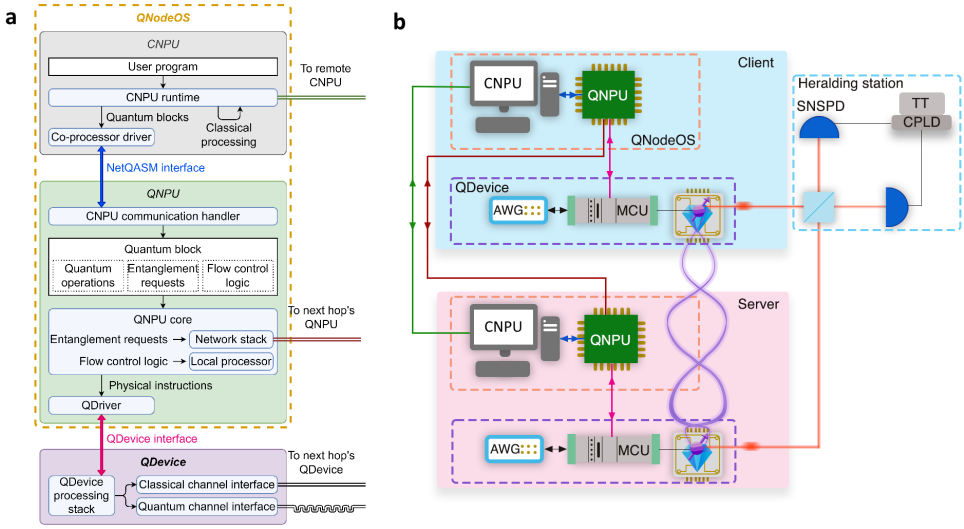


Figure 4.2: **QNodeOS architecture. (a)** QNodeOS consists of a Classical Network Processing Unit (CNPU) and a Quantum Network Processing Unit (QNPU, classical system). QNodeOS controls a QDevice (quantum hardware and low-level classical control). **(b)** Schematic of our implementation of QNodeOS on a two-node setup where both QDevices control a single qubit in a diamond nitrogen-vacancy (NV) center. The CNPU is implemented on a general-purpose PC, and the QNPU on an embedded system, connected via Gigabit Ethernet (blue). The QNPU connects to its QDevice via a serial peripheral interface (SPI, pink). The two QNPUs (brown), and the two CNPUs (green) connect to each other via Gigabit Ethernet. The setup is based on [78] with two QDevices (including arbitrary waveform generators (AWG) and microcontroller units (MCU); QDevices communicating over a classical DIO interface) and a heralding station composed by a balanced 50:50 beam-splitter (whose output ports are connected to superconducting nanowire single-photon detectors (SNSPD) via optical fibers (red)), a TimeTagger (TT), and a Complex Programmable Logic Device (CPLD) that heralds the entanglement generation between QDevices and sends a classical message to the MCU.

and resources for multiple independent programs, including processes, quantum memory management, and entanglement requests.

In Section 4.8 we provide more in-depth design considerations and challenges.

4.3 Architecture

We divide the architecture logically into three main components (Figure 4.2, Section 4.6): The Classical Network Processing Unit (CNPU) is the logical element responsible for starting the execution of programs, and the execution of classical code blocks; the Quantum Network Processing Unit (QNPU, realized on classical hardware) is the logical element responsible for governing the execution of the quantum code blocks; The CNPU and QNPU together form QNodeOS and control the QDevice, which is responsible for executing any quantum operations (gates, measurements, entanglement generation at the physical layer [25]) on the quantum hardware. Upon starting the execution, the CNPU creates a process (a well-known concept in classical operating systems [30, 90]) on the CNPU (a CNPU process), registers the program on the QNPU (via the QNPU's end node application programming interface (API), Section 4.9.2), which, in turn, creates its own associated

QNPU process (including context such as process owner, ID, process state and priority). QNodeOS also defines kernel processes on the QNPU, which are similar to user processes, but are created when the system starts (on boot). The CNPU sends quantum blocks to the QNPU in the form of NetQASM subroutines [26] (see also Chapter 3). Classical control logic in quantum blocks is executed by the QNPU processor. Quantum gates and measurements (from any QNPU process) and entanglement instructions (from the network process) are delegated to the QDevice by submitting physical instructions (Section 4.6), after which the QDevice responds back to the QNPU with the result of the instruction (Section 4.9.6).

To enable different hardware platforms, we introduce a QDriver realizing the HAL for any hardware corresponding to our minimal QDevice system model (Section 4.6). The QDriver is the only hardware-dependent element of the architecture, and is responsible for translating quantum operations, expressed in NetQASM [26], into platform dependent (streams of) physical instructions to the underlying QDevice. We realize a QDriver for the trapped-ion system of [92, 93], and one for NV centers in diamond based on the system of [42, 77, 78]. We validate the trapped-ion QDriver (Figure 4.5) by implementing and verifying a set of single-qubit gate operations (Section 4.6), and the QDriver on the NV system as part of the full stack system evaluation (see below). To allow for different timescales, we logically divide the architecture into CNPU, QNPU and QDevice which can thus be realized at different timing scale granularities. In our proof-of-concept implementation, we realize the CNPU and QNPU on different devices, reflecting the ms timescales of communication between distant nodes (Section 4.6).

Ensuring the necessary interactivity requires architectural as well as implementation choices: as programs may depend on messages from remote nodes, the architecture needs to be able to dynamically handle both classical and quantum blocks, even if not known at runtime. Consequently, it is not possible to preload all quantum blocks of the program into the low-level controller of the QDevice ahead of time as done in previous physics experiments. Instead, in our system model the QDevice is capable of executing individual physical instructions similar to a classical CPU. Consequently, the QNPU is continuously ready to receive new NetQASM subroutines from the CNPU, and the QDevice can continuously receive and respond to physical instructions from the QNPU (Section 4.6).

In our NV QDevice implementation, we address the challenge of interactivity by interleaving specific user-requested pulse sequences (realizing physical instructions sent from QNodeOS) and dynamical decoupling (DD) sequences (protecting quantum memory from decoherence) in an arbitrary waveform generator (AWG) [46]. The DD sequences extend qubit coherence times up to $T_{\text{coh}} = 13(2)$ ms, while arbitrary physical instructions can be handled by triggering the corresponding pulse sequence, without knowing them in advance (Section 4.6).

To integrate local operations with the network schedule, our architecture first introduces a QNPU scheduler that can choose which of the ready processes is assigned to the local processor (Figure 4.2, Section 4.6) and QDevice. This allows interleaving the execution of different processes directly on the QNPU without incurring delays on the timescale of the CNPU (ms), addressing the challenge of short coherence times. In our implementation, we choose to schedule QNPU processes using a priority based non-preemptive scheduler [62], due to limited quantum memory lifetimes, which make it undesirable to

pre-empt and temporarily store quantum states while halting the execution. Second, we realize a network process as a kernel process, which manages entanglement generation using the network stack [25, 54] (implemented in [78] without the ability to execute network applications). While our architecture can work with any way of setting a network schedule, in our implementation we choose to determine the schedule using a time-division multiple access (TDMA) controller [85], allowing the schedule to be centrally optimized to mitigate present-day memory decoherence. The network process handles entanglement requests submitted by user processes, coordinates entanglement generation with the rest of the network via the TDMA controller, interacts with the QDevice and eventually returns entangled qubits to user processes. User processes enter the waiting state when they need entanglement, and become ready again once entanglement was delivered. The network process has the highest scheduling priority, and is consequently given precedence over the execution of any local quantum operations. We remark local operations may still be executed during time-bins already occupied by the network schedule, if a running non-preemptable user process prevents the network process from running, as we indeed observe in our evaluation.

To increase utility, QNodeOS allows multiple programs to be run concurrently, using the QNPU scheduler from above to enable multitasking [29, 68] user processes on the QNPU itself. The QNPU hence needs to keep context for each process, including a virtual quantum memory space (as in classical operating systems [27]). Similar to classical memory management systems [74], a quantum memory management unit (QMMU) on the QNPU manages qubit allocations from processes, and translates virtual qubit addresses in NetQASM subroutines to physical addresses in the QDevice. This allows flexibility in translating a virtual qubit address to: (1) a different physical qubit address over time, allowing qubits to be rearranged transparently in the physical memory in the future, or (2) a logical qubit address, when QNodeOS is executed on top of a processor employing quantum error correction [61] (Section 4.6). Entanglement generation between different pairs of processes at remote nodes are distinguished by Entanglement Request (ER) sockets, inspired by classical sockets [18, 59], which are established once a user process requests entanglement from the network process. In our implementation, processes of the same priority are scheduled first-come-first-served [74], where the total schedule of the program in our implementation is dependent both on the schedule on the CNPU as well as the QNPU (Section 4.6).

In Section 4.9 we provide more details about the QNodeOS design and implementation.

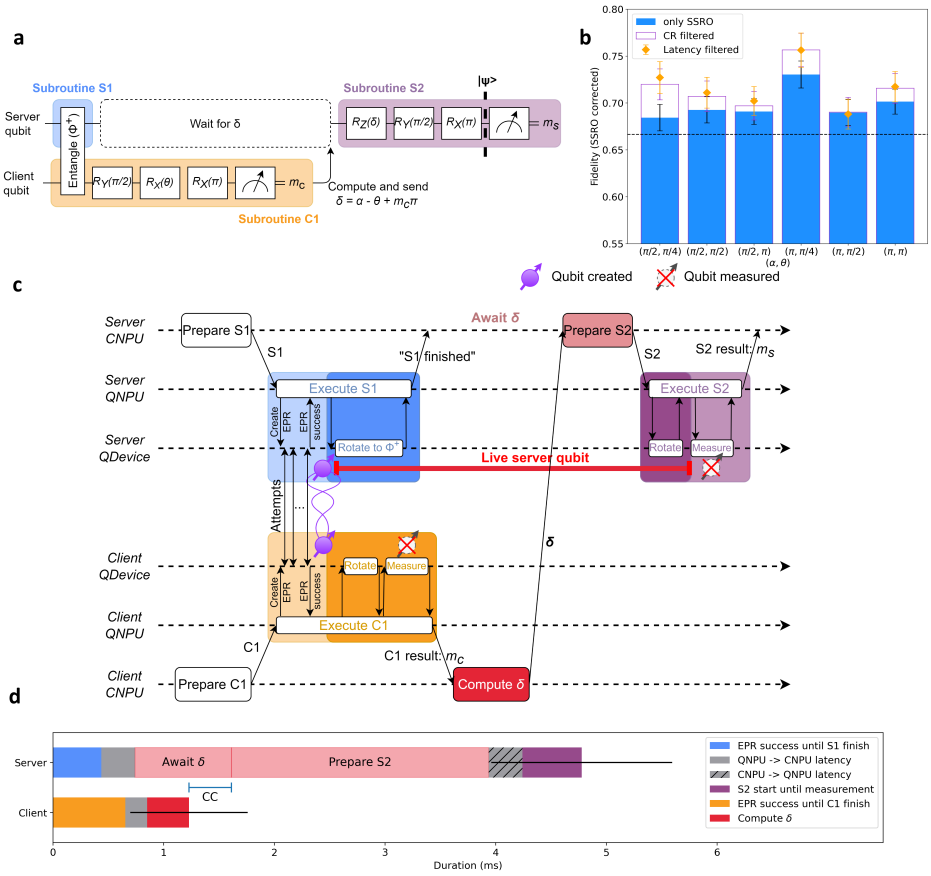


Figure 4.3: **Delegated computation between two NV center nodes using QNodeOS. (a)** Delegated Quantum Computation (DQC) circuit (effective computation: single-qubit rotation $R_Z(\alpha)$, Section 4.6). The DQC application consists of k repetitions of this circuit (varying measurement bases for tomography on $|\psi\rangle$) realized by two programs: the DQC-client program (client node, repeating the sequence "quantum block (C1, orange) – classical block (computing δ)" k times), and the DQC-server program (server node, repeating "quantum block (S1, blue) - classical block (receiving δ) –quantum block (S2, purple)" k times). Client and server produce an entangled pair $|\Phi^+\rangle = (|00\rangle + |11\rangle)/\sqrt{2}$ (S1 and first part of C1). The client performs local gates and a measurement ("destroying" qubit), resulting in outcome bit m_c (rest of C1). Client computes δ as function of m_c and DQC parameters $\alpha \in [0, 2\pi)$ and $\theta \in [0, 2\pi)$, and sends δ to server (classical message). Meanwhile the server keeps its qubit coherent (alive). Upon receiving δ, the server applies gates depending on δ, resulting in single-qubit state $|\psi\rangle$ (S2) depending only on α and θ. **(b)** Experimental results of executing DQC for 6 different sets of (α, θ) parameters (k = 1200, i.e. 7200 executions of the circuit of Figure 4.3a). The fidelity of the resulting server state to the target state $|\psi\rangle$ is estimated using single-qubit tomography (1200 measurement results per data point), and corrected for known tomography errors (SSRO, blue), and post-selected for Charge-Resonance (CR) check validation (purple), and post-selected for latencies (orange) (Section 4.6). **(c)** Sequence diagram including the interaction CNPU-QNPU-QDevice for one execution of the DQC circuit of Figure 4.3a on QNodeOS (repeated k = 1200 times in each experiment) (time flows to the right; not to scale). CNPUs prepare NetQASM subroutines (C1, S1, S2), and send them to their respective QNPUs. CNPUs also do classical computation (computing δ) and communication (message containing δ). QNPUs execute subroutines, sending physical instructions to their QDevices. Entanglement is generated by QDevices doing a batch of attempts, resulting in the heralding of a two-qubit entangled state (Bell pair) rotated to $|\Phi^+\rangle$ by the server. **(d)** Processing times and latencies while server qubit is live (time frame red line 3c, averaged over all 7200 circuit executions except executions with latency spikes, see Section 4.6), including CNPU-QNPU communication latencies, CNPU processing on both nodes and client-server communication latency (CC) (average total of ~ 4.8(±0.8) ms, error bars for the sum of individual segments (variance per segment in Section 4.11.6).

4.4 Demonstrations

Delegated Computation. We first validate our architecture and implementation by the first successful execution of an arbitrary — i.e. not preloaded — execution of a quantum network application in high-level software on quantum processors. We implement QNodeOS on a two-node setup of NV centers using one qubit per node (Figure 4.2, Section 4.6). We choose to execute an elementary form of delegated quantum computation (DQC) [14] from a client to a server, because the client and server programs jointly realize repetitions of a circuit (Figure 4.3a) that triggers all parts of our system (Figure 4.3c). We first verify that the quantum result (fidelity) was found to be above the classical bound [66] > 2/3, which verifies that QNodeOS can successfully handle interactive applications consisting of entanglement generation, millisecond-scale memory lifetimes, and classical message passing. The non-perfect fidelity (Figure 4.3b) comes mainly from two sources: a noisy entangled state with fidelity 0.72(2) (quantum hardware limitation), and decoherence in the server qubit (depending on T_{coh}) due to waiting for several milliseconds (classical software latencies, Figure 4.3d). We proceed to characterize latencies. As expected, we find that the duration that the server qubit must remain alive is dominated (> 50%) by processing in the CNPU, which could be improved by caching the preparation of S2, and implementing the CNPU and QNPU on one board (Outlook). We observe that CNPU processing time varies significantly (standard deviation 30%, Section 4.11.6), due to limited scheduling control over CNPU processes (Section 4.6). Using an a priori estimate of what delays lead to too low a quality of execution (i.e. delays that are too long for the server qubit to be stored with sufficiently high quality), we discard application iterations in which the CNPU latencies spiked by more than 8.95 ms. This lead to discarding of 2% of iterations in post-processing (Section 4.6).

Demonstration of Multitasking. We also validate QNodeOS's multitasking capability by the first concurrent execution of two quantum applications on a quantum network: the DQC application, and a single-node local gate tomography (LGT) application on the client (Figure 4.4a). The two programs for the client are started in the CNPU at the same time (two CNPU processes, subject to CNPU scheduler), which means that the QNPU continuously receives subroutines for both programs from the CNPU (two QNPU processes and corresponding subroutines, subject to QNPU scheduler). This leads to a multitasking challenge directly on the QNPU to schedule the different subroutines received (Figure 4.4b). Since the client has only one qubit, the multitasking of DQC and LGT never results in both programs having a quantum state alive on the client; therefore, multitasking should not affect the fidelity of LGT. We observe interleaved execution of DQC quantum blocks and LGT quantum blocks on the client node (Figure 4.4b). The LGT application produces a quantum result (fidelity, Figure 4.4c) equal to that in the scenario where we run LGT on its own (not interleaved by DQC circuit executions), as expected.

We further test multitasking by scaling up the number of programs executed concurrently on the client node, up to 5 DQC and 5 LGT programs running at the same time on the client. The interleaved execution of subroutines of different programs increases device utilization (fraction of time spent on executing physical instructions) on the client QDevice compared to the same scenario but with multitasking disabled (Figure 4.4d). As expected, we observe that LGT subroutines were scheduled to be executed in between DQC subroutines, resulting in lower client QDevice idle time. When multitasking 1 DQC and 1 LGT

program, we observe 1 or 2 subroutines in between DQC iterations in most cases (LGT subroutine duration 2.4 ms, Section 4.12.3). We observe cases where both server and client QDevice remain idle, which could be improved in part by smarter CNPU-QNPU scheduling algorithms: (1) both the client and server wait until the start of the next network schedule time-bin (time-bin length 10 ms) (2) the client QNPU finishes a subroutine for user process P, but must wait until the CNPU sends the next subroutine for P (up to 150 ms for 1 DQC and 1 LGT program, but less (up to only 8 ms) when more applications are running, since there are more CNPU processes independently submitting subroutines), (3) the client is ready to perform entanglement generation for DQC, but the next time-bin starts only at some future time t, preventing activation of the network process. The scheduler activates a user process which runs a LGT circuit, which completes at some time $> t$, delaying the start of the DQC network process, even though the server node was ready at t.

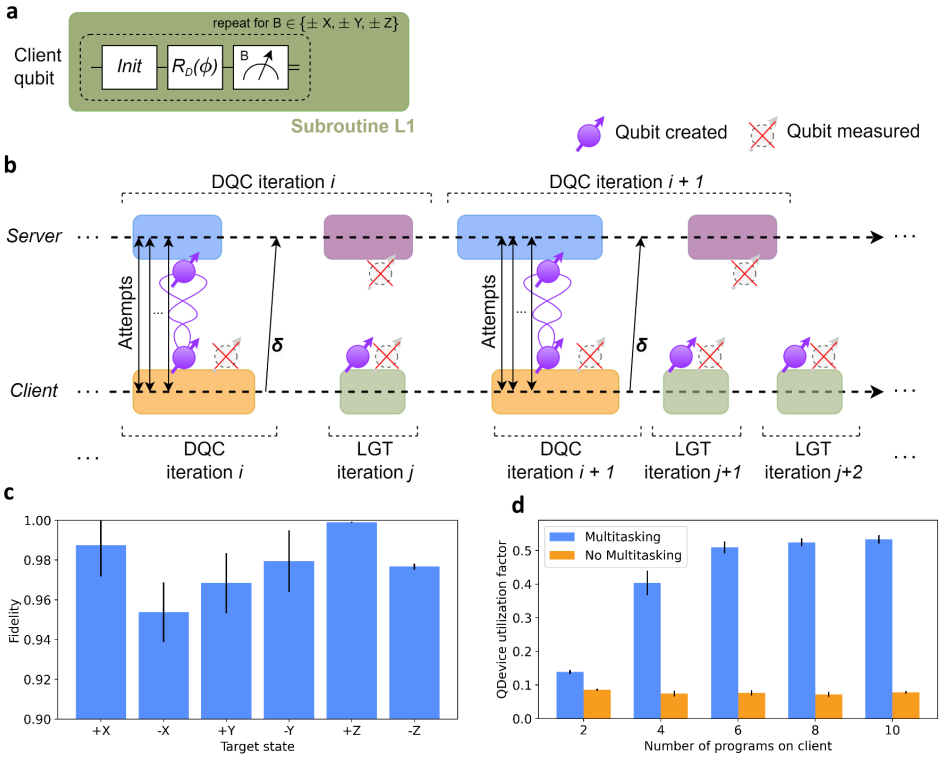


Figure 4.4: **Multitasking experiment on two NV centers with QNodeOS**. **(a)** Local Gate Tomography (LGT) Circuit. A single NetQASM subroutine (L1) executes the following 6 times for different bases $B \in \{\pm X, \pm Y, \pm Z\}$: initialize qubit to $|0\rangle$, rotate around fixed axis $D \in \{X, Y\}$ by angle $|\phi\rangle$, measure in B. The LGT application consists of a single LGT program, which submits subroutine L1 for execution to the QNPU (fixed D and ϕ) k times in succession. **(b)** Example sequence diagram illustrating concurrent execution (multitasking) of the DQC application (Figure 4.3) and the LGT program on the client: time slice in which two DQC circuit repetitions (Figure 4.3a) are realized (2 subroutines on the client (orange), 4 on the server (blue and purple)), and three LGT circuit repetitions (3 subroutines, green). The client QNPU receives subroutines for both the DQC program and the LGT program, which the QNPU scheduler can interleave: While the server executes S2 (purple), the client cannot yet execute the next S1 (orange) since it involves joint entanglement generation. In this idle time, the client can execute a number of LGT subroutines (number can vary). **(c)** Results of multitasking LGT (client) and DQC (on both server and client). For each input pair $(D, \phi) \in \{(X,0),(X,\pi),(Y,pi/2),(Y,-\pi/2),(X,-\pi/2),(X,\pi/2)\}$ (6 cardinal states $\{\pm X, \pm Y, \pm Z\}$), the following experiment was performed: simultaneously (1) a single LGT program was initiated on the client (k = 1000), (2) a single DQC-client program was initiated on the client (k = 200 successive subroutines), and (3) a single DQC-server program was initiated at the server (k = 200, i.e. 400 successive subroutines). This resulted in a total of 6000 LGT subroutine executions and 36000 LGT measurement results, yielding plotted fidelity estimates for the LGT quantum state before measurement. Results are the same as running LGT on its own (no multitasking with DQC), as expected (Section 4.12.2). **(d)** Scaling number of programs on the client. For $N \in \{1,2,3,4,5\}$, we initiate at the same time: (1) N LGT programs (each using k = 100) on the client, (2) N DQC-client programs on the client (each using k = 60), and (3) N DQC-server programs on the server (each using k = 60). This results in $2N$ programs active at the same time on the client, each continuously submitting subroutines from the CNPU to the QNPU, where the QNPU scheduler chooses which process to execute when. Each experiment was repeated but with multitasking disabled on the client. Plot shows the utilization factor of the QDevice (fraction of time spent executing instructions), corrected for variable entanglement generation duration (Section 4.6), with (blue) and without (orange) multitasking, showing that multitasking can increase device utilization.

4.5 Outlook

We designed and implemented the first architecture allowing high-level programming and execution of quantum network applications. Our architecture does not depend on the distance or connectivity between the end nodes, as long as the network stack enables the use of a quantum network to generate entanglement between them. To deploy our system onto nodes separated by several kms, one possible improvement to the implementation of our architecture would be to realize the CNPU and the QNPU on two devices on a single system board, ideally with mutual access to a shared memory to avoid ms delays in their communication. Such a merge would also allow the definition of a joint classical-quantum executable and processes, opening further doors to reduce latencies by a better scheduling control. Our architecture could also be used to distribute a quantum computing program over multiple quantum processors by submitting jobs as NetQASM subroutines to the QNPU of each node.

Our work provides a framework for a new domain of computer science research into programming quantum network applications on quantum processors including: novel real-time [81] scheduling algorithms for classical-quantum processes, compile methods for quantum network applications [26], or novel programming language concepts including entanglement to make software development even easier, thus advancing the vision to make quantum network technology broadly available.

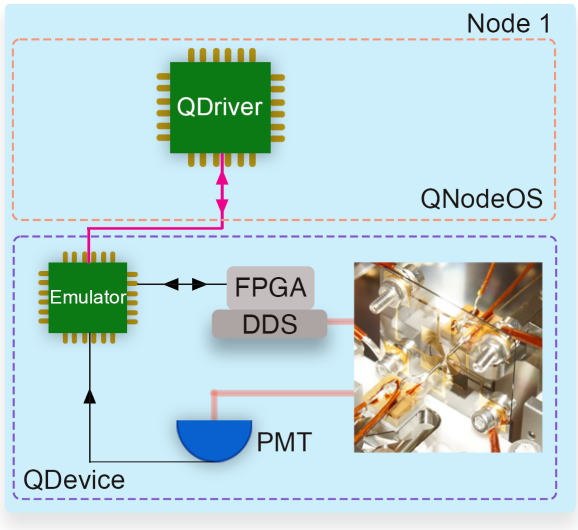


Figure 4.5: **Trapped-ion QDevice implementation.** Schematic of our implementation of QNodeOS on a single-node setup in which the QDevice contains a single trapped-ion qubit. The QNPU QDriver is implemented on a field-programmable gate array (FPGA) that connects to its QDevice via a serial peripheral interface (SPI) (Section 4.6). The setup consists of an emulator that translates between SPI messages and TTL signals, experimental control hardware that includes an FPGA and direct digital synthesis (DDS) modules, a trapped-ion qubit [93] under ultra-high vacuum (Figure 4.1), and a photomultiplier tube (PMT) that registers atomic fluorescence.

4.6 Methods

QDevice Model. The QDevice includes a physical quantum device, which can initialize and store quantum bits (qubits) which are individually identified by a physical address, apply quantum gates, measure qubits, and create entanglement with QDevices on other nodes (either entangle-and-measure, or entangle-and-keep [25]), either another end node, or an intermediary node in the network. We remark that the ability for two end node QDevices that are not immediate neighbors in the quantum network (but that are separated by other network nodes) to generate entanglement between them relies on the architecture implementing a network layer protocol as part of a network stack [25]. Qubits thereby refers to any possible realization of qubits, including logical qubits realized by error-correction. The QDevice exposes the following interface to QNodeOS (Section 4.9.6): number of qubits available, and the supported physical instructions that QNodeOS may send. Physical instructions include qubit initialization, single- and two-qubit gates, measurement, entanglement creation, and a 'no-op' for do nothing. Each instruction has a corresponding response (including entanglement success or failure, or a measurement outcome) that the QDevice sends back to QNodeOS.

QNodeOS and the QDevice interact by passing messages back and forth on clock ticks at a fixed rate (100 kHz in our NV implementation, 50 kHz in the trapped-ion implementation). During each tick, at the same time (1) QNodeOS sends physical instruction to QDevice, (2) QDevice can send a response (for a previous instruction). Upon receiving an instruction, the QDevice performs the appropriate (sequence of) operations (e.g. a par-

ticular pulse sequence in the AWG). An instruction may take multiple ticks to complete, where the QDevice returns the response (success, fail, outcome) during the first clock tick following completion. The QDevice handles an entanglement instruction by performing (a batch of) entanglement generation attempts [78] (synchronized by the QDevice with the neighboring node's QDevice).

QNodeOS Architecture. QNodeOS consists of two layers: CNPU and QNPU (Figure 4.2a, Section 4.3, Section 4.9). Processes on the QNPU are managed by the Process Manager, and executed by the local processor. Executing a user process means executing NetQASM (see [26] and Chapter 3) subroutines (quantum blocks) for that process, which involves running classical instructions (including flow control logic) on the QNPU's local processor, sending entanglement requests to the network stack, and handling local quantum operations by sending physical instructions to the QDriver (Figure 4.2a). Executing the network process means asking the network stack which request (if any) to handle and sending the appropriate (entanglement generation) instructions to the QDevice.

A QNPU process can be in the following states (see Figure 4.8 for state diagram): idle, ready, running and waiting. A QNPU process is running when the QNPU processor is assigned to it. The network process becomes ready when a network schedule time-bin starts; it becomes waiting when it finished executing and waits for the next time-bin; it is never idle. A user process is ready when there is at least one NetQASM subroutine pending to be executed; it is idle otherwise; it goes into the waiting state when it requests entanglement from the network stack (using NetQASM entanglement instructions [26]) and is made ready again when the requested entangled qubit(s) are delivered.

The QNPU scheduler oversees all processes (user and network) on the QNPU, and chooses which ready process is assigned to the QNPU processor. CNPU processes can run concurrently, and their execution (order) is handled by the CNPU scheduler. The QNPU scheduler operates independently and only acts on QNPU processes. CNPU processes can only communicate with their corresponding QNPU processes. Since multiple programs can run concurrently on QNodeOS, the QNPU may have multiple user processes that have subroutines waiting to be executed at the same time. This hence requires scheduling on the QNPU.

Processes allocate qubits through the Quantum Memory Management Unit (QMMU), which manages virtual qubit address spaces for each process, and translates virtual addresses to physical addresses in the QDevice. The QMMU can also transfer ownership of qubits between processes, for example from the network process (having just created an entangled qubit), to a user process that requested this entanglement. The Network Stack uses Entanglement Request (ER) sockets (opened by user programs through QNPU API once execution starts) to represent quantum connections with programs on other nodes. The Entanglement Management Unit (EMU) maintains all ER sockets and makes sure that entangled qubits are moved to the correct process.

NV QDevice Implementation. The two-node network employed in this work includes the nodes "Bob" (server) and "Charlie" (client) (separated by 3 meters) described in [42, 77, 78]. For the QDevice, we replicated the setup used by [78], which mainly consists of: an Adwin-Pro II [101] acting as the main orchestrator of the setup; a series of subordinate devices responsible for qubit control, including laser pulse generators, optical readout circuits and an arbitrary waveform generator (Zurich Instruments HDAWG [46]).

The quantum physical device, based on NV centers, counts one qubit for each node. The two QDevices share a common 1 MHz clock for high-level communication and their AWGs are synchronized at sub-nanosecond level for entanglement attempts.

We address the challenge of limited memory lifetimes by employing dynamical decoupling (DD). While waiting for further physical instructions to be issued, DD sequences are used to preserve the coherence of the electron spin qubit [28]. DD sequences for NV-centers can prolong the coherence time (T_{coh}) up to hundreds of ms [42] or even seconds [1]. In our specific case, we measured T_{coh}=13(2) ms for the server node, corresponding to 1300 DD pulses. The discrepancy to the state-of-the-art for similar setups is due to several factors. To achieve such long T_{coh}, a thorough investigation of the nuclear spin environment is necessary to avoid unwanted interactions during long DD sequences, resulting in an even more accurate choice of interpulse delay. Other noise sources include unwanted laser fields, the quality of microwave pulses and electrical noise along the microwave line.

A specific challenge arises at the intersection of extending memory lifetimes using DD, and the need for interactivity: to realize individual physical instructions, many waveforms are uploaded to the Arbitrary Waveform Generator (AWG), where the QDevice decodes instructions sent by QNodeOS into specific preloaded pulse sequences. This results in a waveform table, containing 170 entries. The efficiency of the waveforms is limited by the AWG's waveform granularity that corresponds to steps that are multiples of 6.66 ns, having a direct impact on the T_{coh}. We are able to partially overcome this limitation through the methods described in [23]. Namely, each preloaded waveform, corresponding to one single instruction, has to be uploaded 16 times in order to be executed with sample precision. To not fill up the waveform memory of the device, we apply the methods in [23] only to the DD pulses that are played while the QDevice waits for an instruction from the QNPU, whereas the instructed waveforms (gate/operation + first block of XY8 DD sequence) are padded according to the granularity, if necessary. The physical instructions supported by our NV QDevice is given in Section 4.10.1.

NV QNPU Implementation. The QNPUs for both nodes are implemented in C++ on top of FreeRTOS [103], a real-time operating system for microcontrollers. The stack runs on a dedicated MicroZed [104]—an off-the-shelf platform based on the Zynq-7000 SoC, which hosts two ARM Cortex-A9 processing cores, of which only one is used, clocked at 667 MHz. The QNPU was implemented on top of FreeRTOS to avoid re-implementing standard OS primitives like threads and network communication. FreeRTOS provides basic OS abstractions like tasks, inter-task message passing, and the TCP/IP stack. The FreeRTOS kernel—like any other standard OS—cannot however directly manage the quantum resources (qubits, entanglement requests and entangled pairs), and hence its task scheduler cannot take decisions based on such resources. The QNPU scheduler adds these capabilities (Section 4.9.5).

The QNPU connects to peer QNPU devices via TCP/IP over a Gigabit Ethernet interface (IEEE 802.3 over full-duplex Cat 5e). The communication goes via two network switches (Netgear JGS524PE, one per node). The two QNPUs are time-synchronized through their respective QDevices (granularity 10 μs), since these already are synchronized at the μs-level (common 1Mhz clock).

The QNPU device interfaces with the QDevice's ADwin-Pro II through a 12.5 MHz SPI interface, used to exchange 4-byte control messages at a rate of 100 kHz.

NV CNPU Implementation. The CNPUs for both nodes are a Python runtime executing on a general-purpose desktop machine (4 Intel 3.20 GHz cores, 32 GB RAM, Ubuntu 18.04). The choice of using a high-level system was made as the communication between distant nodes would ultimately be in the ms-timescales, and this allows for ease of programming the application. The CNPU machine connects to the QNPU device via TCP over a Gigabit Ethernet interface (IEEE 802.3 over full-duplex Cat 8, average ping RTT of 0.1 ms), via the same single network switch as mentioned above (one per node), and sends application registration requests and NetQASM subroutines over this interface (10 to 1000 bytes, depending on the length of the subroutine). CNPUs communicate with each other through the same two network switches.

Scheduler Implementation. We use a single Linux process (Python) for executing programs on the CNPU. CNPU 'processes' are realized as threads created within this single Python process. Python was chosen since the NetQASM SDK is implemented in Python. When running multiple programs concurrently, a pool of such threads is used. Scheduling of the Python process and its threads is handled by the Linux OS. Each thread establishes a TCP connection with the QNPU in order to use the QNPU API (including sending subroutines and receiving their results) and executes the classical blocks for its corresponding program. Both the CNPU and QNPU maintain processes for running programs. The CNPU scheduler (standard Linux scheduler, see above) schedules CNPU processes, which indirectly controls in which order subroutines from different programs arrive at the QNPU. The QNPU scheduler handles subroutines of the same process priority on a first-come-first-served (FCFS) basis, leading however to executions of QNPU processes not in the order submitted by the CNPU (Section 4.12.3).

Using only the CNPU scheduler is not sufficient since (1) we want to avoid millisecond delays needed to communicate scheduling instructions across CNPU and QNPU, (2) user processes need to be scheduled in conjunction with the network process (meeting the challenge of scheduling both local and network operations), which is only running on the QNPU, and (3) QNPU user processes need to be scheduled with respect to each other, (e.g. a user process is waiting after having requested entanglement, allowing another user process to be run; as observed in the multitasking demonstration).

Sockets and the Network Schedule. In an ER Socket, one node is a 'creator' and the other a 'receiver'. As long as an ER socket is open between the nodes, an entanglement request from only the creator suffices for the network stack to handle it in the next corresponding time-bin, i.e. the 'receiver' can comply with entanglement generation even if no request has (yet) been made to its network stack.

Trapped-ion Implementation. The experimental system used for the trapped-ion implementation is discussed in [92, 93] and is described in detail in [91]. The implementation itself is described in [36]. We confine a single $^{40}Ca^{+}$ion in a linear Paul trap; the trap is based on a 300 μm thick diamond wafer on which gold electrodes have been sputtered. The ion trap is integrated with an optical microcavity composed of two fiber-based mirrors, but the microcavity is not used here. The physical-layer control infrastructure consists of C++ software; Python scripts; a pulse sequencer that translates Python commands to a

hardware description language for a field-programmable gate array (FPGA); and hardware that includes the FPGA, input triggers, direct digital synthesis (DDS) modules, and output logic.

QNodeOS provides physical instructions through a development FPGA board (Texas Instruments, LAUNCHXL2-RM57L75) that uses a serial peripheral interface (SPI). We programmed an additional board (Cypress, CY8CKIT-14376) that translates SPI messages into TTL signals compatible with the input triggers of our experimental hardware. The implementation consisted of sequences composed of seven physical instructions: initialization, $R_x(\pi)$, $R_y(\pi)$, $R_x(\pi/2)$, $R_y(\pi/2)$, $R_y(-\pi/2)$, and measurement. First, we confirmed that message exchange occurred at the rate of 50 kHz as designed. Next, we confirmed that we could trigger the physical-layer hardware. Finally, we implemented seven different sequences. Each sequence was repeated 10^4 times, which allowed us to acquire sufficient statistics to confirm that our QDriver results are consistent with operation in the absence of the higher layers of QNodeOS.

Metrics. Both classical and quantum metrics are relevant in the performance evaluation: The quantum performance of our test programs is measured by the fidelity $F(\rho, |\tau\rangle)$ of an experimentally obtained quantum state ρ to a target state $|\tau\rangle$ where $F(\rho, |\tau\rangle) = \langle\tau|\rho|\tau\rangle$, estimated by quantum tomography [73]. Classical performance metrics include device utilization $T_{\text{util}} = 1 - T_{\text{idle}}/T_{\text{total}}$ where T_{idle} is the total time that the QDevice is not executing any physical instruction, and T_{total} is the duration of the whole experiment excluding time spent on entanglement attempts (see below).

Experiment Procedure NV Demonstration. Applications are written in Python using the NetQASM SDK [26] (code in Appendix B), with a compiler targeting the NV flavor [26], as it includes quantum instructions that can be easily mapped to the physical instructions supported by the NV QDevice. The client and server nodes independently start execution of their programs by invoking a Python script on their own CNPU, which then spawns the threads for each program. During application execution, the CNPUs have background processes running, including QDevice monitoring software.

A fixed network schedule is installed in the two QNPUs, with consecutive time-bins (all assigned to the client-server node pair) with a length of 10 ms (chosen to be equal to 1000 communication cycles between QNodeOS and QDevice as in Ref. [78]) to assess the performance without introducing a dependence on a changing network schedule. During execution, the CNPUs and QNPUs record events including their timestamps. After execution, corrections are applied to the results (see below) and event traces are used to compute latencies.

Delegated Quantum Computation. Our demonstration of DQC (Figure 4.3) implements the effective single-qubit computation $|\psi\rangle = H \circ R_z(\alpha) \circ |+\rangle$ on the server, as a simple form of blind quantum computing (BQC) that hides the rotation angle α from the server, when executed with randomly chosen θ, and not performing tomography. The remote entanglement protocol utilized is the single-photon protocol [12, 15, 41] (Section 4.10.1).

Filtering. Results, with no post-selection, are presented including known errors that occur during the tomography single-shot readout (SSRO) process (Figure 4.3b, blue) (details on the correction Supplementary of [77]). We also report the post-selected results in which

data are filtered based on the outcome of the Charge-Resonance check [83] after one application iteration (Figure 4.3b, purple). This filter enables the elimination of false events, specifically when the emitter of one of the two nodes is not in the right charge state (ionization) or the optical resonances are not correctly addressed by the laser fields after the execution of one iteration of DQC.

Additional filtering (Figure 4.3b latency filter) is done on those iterations that showed latency not compatible with the combination of T_{coh} of the server and the average entangled state fidelity. For this filter, a simulation (using a depolarizing model, based on the measured value T_{coh}, Section 4.11.4) was used to estimate the single qubit fidelity (given the entanglement fidelity measured above) as a function of the duration the server qubit stays live in memory in a single execution of the DQC circuit (Figure 4.3a). This gives a conservative upper bound of the duration as 8.95 ms, to obtain a fidelity of at least 0.667. All measurement results corresponding to circuit executions exceeding 8.95 ms duration were discarded (146 out of 7200 data points).

Other main sources of infidelity, that are not considered in this analysis of the outcome, include, for instance, the non-zero probability of double excitation for the NV center [41]. During entanglement generation, the NV center can be re-excited, leading to the emission of two photons that lower the heralded entanglement fidelity. The error can be corrected by discarding those events that registered, in the entanglement time-window, a photon at the heralding station (resonant Zero-Phonon Line photon) and another one locally at the node (off-resonant Phonon-Side Band photon).

Finally, the dataset presented in Figure 4.3b (not shown chronologically) was taken in "one shot" to prove the robustness of the physical layer, therefore no calibration of relevant experimental parameters was performed in between, leading to possible degradation of the overall performance of the NV-based setup.

The single qubit fidelity is calculated with the same methods as in [47], measuring in the state $|i\rangle$ and in its orthogonal state $|-i\rangle$, provided that we expect the outcome $|i\rangle$, whereas the two-qubit state fidelity is computed taking into account only the same positive-basis correlators (XX, YY, ZZ).

Multitasking: Delegated Computation and Local Gate Tomography. In the first multitasking evaluation, we concurrently execute two programs on the client: a DQC-client program (interacting with a DQC-server program on the server) and a Local Gate Tomography (LGT) program (on the client only) (Figure 4.4). The client CNPU runtime executes the threads executing the two different programs concurrently. The client QNPU has two active user processes, each continuously receiving new subroutines from the CNPU, which are scheduled with respect to each other and the network process.

Estimates of the fidelity (Figure 4.4b) include same corrections as in the Supplementary of [77] To assess the quantum performance of the LGT application, we used a mocked entanglement generation process on the QDevices (executing entanglement actions without entanglement) to simplify the test: weak-coherent pulses on resonance with the NV transitions, that follow the regular optical path, are employed to trigger the CPLD in the entanglement heralding time-window. This results in comparable application behavior for DQC (comparable rates and latencies, Section 4.12.1) with respect to multitasking on QNodeOS.

Multitasking: QDevice Utilization when scaling number of programs. We scale the number of programs being multitasked (Figure 4.4d): We observe how the client QNPU scheduler chooses the execution order of the subroutines submitted by the CNPU. DQC subroutines each have an entanglement instruction, causing the corresponding user process to go into the waiting state when executed (waiting for entanglement from the network process). The QNPU scheduler schedules another process [(56%, 81%, 99%) for (N=1, N=2, N>2)] of the times that a DQC process is put into the waiting state (demonstrating that the QNPU schedules independently from the order in which the CNPU submits subroutines). The number of consecutive LGT subroutines (of any LGT process; LGT block execution time 2.4 ms) that is executed in between DQC subroutines is 0.83 for N=1, increasing for each higher N until 1.65 for N = 5, showing that indeed idle times during DQC are partially filled by LGT blocks (Section 4.12.3).

Device utilization (see Metrics above) quantifies only the utilization factor in between entanglement generation time windows to fairly compare the multitasking and the non-multitasking scenario. In both scenarios, the same entanglement generation processes are performed, which hence have the same probabilistic durations in both cases. To avoid inaccurate results due to this probabilistic nature, we exclude the entanglement generation time windows in both cases.

4.7 Data availability

The datasets that support this manuscript and the software to analyze them are available at `https://doi.org/10.4121/6aa42f05-6823-4848-b235-3ea19e39f4ae`. The application software development kit used for writing program code is open-sourced on GitHub [105]. The QNodeOS source code is not currently open source.

4.8 Detailed design considerations and challenges

We provide additional information for some of the design considerations and challenges for an operating system for executing applications on a quantum network node.

4.8.1 Quantum networks

Node Types. Within a quantum network, one can distinguish between two main types of nodes: First, there are *end nodes* [99], with which users execute quantum network applications. Classically, such end nodes are laptops, phones or other devices. In the quantum domain, end nodes may be simple photonic devices that can only create or measure quantum states, or they may be quantum processors capable of arbitrary qubit operations and storage of information within a quantum memory. The type of end node dictates what applications are possible [99], and we have chosen to focus on the most general form of an end node, namely, a quantum processor. More precisely, our goal is to enable programming and execution of arbitrary quantum network applications on end nodes that are quantum processors. For the remainder of this text, we will thus always take end nodes to be quantum-processor end nodes.

Second, a quantum network can include *intermediate nodes* that perform routines necessary to connect two or more end nodes (Figure 4.6). We refer the reader with a background in computer science to [95] for a gentle introduction to quantum networks. In-

termediate nodes, such as quantum-repeater nodes, are used to establish long-distance entanglement between remote end nodes. These intermediate nodes may employ protocols such as entanglement swapping and entanglement distillation in order to realize end-to-end links with sufficiently high fidelity for network applications. These protocols are handled by a network stack (see, e.g. [25]) that exists at each node. The network stack includes a link layer, a network layer, a control plane, and other networking functions; it is responsible for entanglement generation.

Intermediate nodes do not execute user applications, which is done only by end nodes. Therefore, only end nodes need to have an additional stack implementing the application layer in a network, which is referred to as an *application stack* (see Figure 4.6). The application stack is responsible for the execution of arbitrary user applications, and integrates with the network stack for entanglement generation over the network. We remark that it is the purpose of a network layer [25, 54] to provide a service to the application layer that allows entanglement generation with remote end nodes. Importantly, this service should not require the application layer to have any knowledge about the connectivity of the network. While QNodeOS can in principle also be run on intermediate nodes as it already implements a network stack, we remark that it is designed primarily to enable the execution of applications on end nodes, which is the focus of this work.

End Node Quantum Memory. A quantum processor end nodes possesses quantum memory in the form of qubits, and allows gates and measurements to be executed on such qubits (see Section 4.6, 'QDevice Model'), which can be used to realize user applications. These qubits may be physical qubits, but may also be logical qubits (multiple physical qubits together representing one more robust usable qubit) in case the end node does error correction [61]. For our architecture it does matter how qubits are realized in the QDevice (which may internally employ quantum error correction), as long as the QDevice follows our system model.

Entanglement. Entanglement is a phenomenon in quantum physics where two or more particles are correlated in a way that is not possible classically. In a quantum network, such entangled particles may be established across separate nodes, realizing a quantum connection between those nodes. Entanglement can be used by quantum network applications as a resource in order to realize applications [99] that are impossible with classical networks, including applications such as data consistency in the cloud [7], privacy-enhancing proofs of deletion [79], exponential savings in communication [40], or secure quantum computing in the cloud [14, 20].

4.8.2 Application paradigm

Our architecture is primarily meant to *enable the execution of quantum network applications in the quantum memory stage [99] and above.* That is, applications that require the use of a quantum processor that can manipulate and store quantum bits (qubits). For simpler applications in the prepare-and-measure and entanglement generation stages [99], e.g. quantum key distribution [8, 35], where the quantum states are immediately measured by the nodes, our system can also be used, but it would be sufficient to realize a system implementing a quantum network stack and classical processing only.

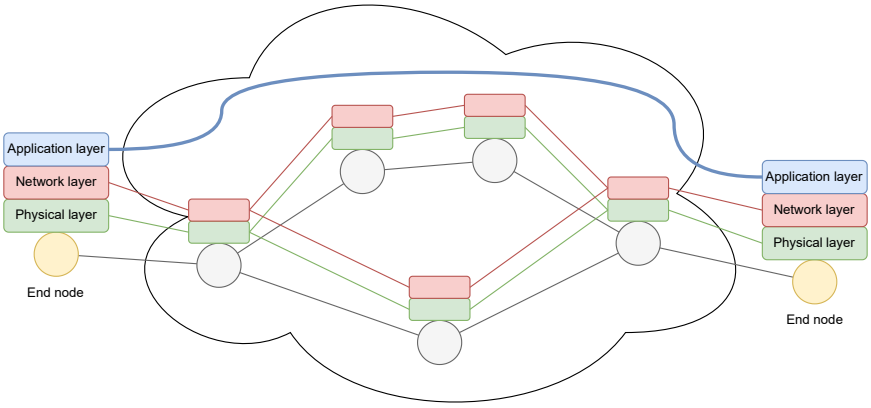


Figure 4.6: **Schematic overview of a quantum network.** A quantum network consists of nodes (yellow and grey circles) that are connected by classical and quantum communication channels (grey lines). Each node implements a physical layer (green boxes and lines) that enables entanglement generation with neighboring nodes. The physical layer is the domain of the QDevice. Each node also implements a network stack, including a network layer (red boxes and lines, which may be subdivided into a separate link layer and a network layer [25, 55]). This layer realizes long-distance entanglement creation between nodes and may include protocols such as entanglement swapping and distillation. As QNodeOS implements a network stack, it could also be deployed on intermediate nodes in a network, where e.g. entanglement distillation could be added to the protocol realizing the network layer service implemented by QNodeOS.
We emphasize however that the focus of this work is to program and execute applications on the end nodes, i.e. enabling the application layer in networking terms. Only *end nodes* (yellow circles) implement an additional application layer (blue boxes and line), which executes arbitrary user applications. From the perspective of this layer, end nodes are logically directly connected (blue line), and this layer is hence independent from implementations and protocols in the network layer and is only dependent on the service provided by the network layer. Logically directly connected means that the application layer relies on the service of the network layer to enable end-to-end entanglement generation between end nodes and does not concern itself with how the entanglement is generated. This abstraction is a key element enabled by a quantum network stack such as [25] and exactly analogous to abstractions used in classical networking, where e.g. a web browser can be executed on a laptop independently of how the internet connection between the laptop and a web server is realized. In the same way, QNodeOS could operate on end nodes separated by a large quantum network of the future, in which many intermediary nodes may lie on the path connecting the end nodes.

Separated programs. Recall that a quantum networking application consists of multiple programs, each running on one of the end nodes, where for ease of explanation we will assume we are executing an application between two nodes, i.e. a client and a server. Each node in the network runs its own independent *Quantum Network Operating System (QNodeOS)*, on which the node's program is executed. The two programs may interact with each other via message passing and entanglement generation, where both types of interactions are managed by the node's QNodeOS. Next to interaction via the programs, the nodes may exchange additional classical messages which are not part of the program itself, for example, in order to enable the realization of a network stack [25] managing entanglement generation between the nodes.

Classical blocks of code consist of instructions for local classical operations and classical message passing. Quantum *blocks of code* consists of (1) quantum operations (initialization, quantum gates, measurement), (2) low-level classical control logic (branching on classical variables and loops), as well as (3) instructions to make entanglement between remote nodes. We remark that classical and quantum instructions may require many actions by the underlying QNodeOS (and quantum system controlled by it) in order to be fulfilled: it is the goal of such instructions to abstract away aspects of the underlying system.

Classical blocks of code may depend on quantum ones via classical variables generated during the quantum execution (such as measurement results, notification of entanglement generation, and information on the state of the quantum system such as the availability of qubits). Similarly, quantum blocks may depend on variables set by the classical blocks (such as messages received from remote network nodes). Finally, quantum blocks may themselves depend on other quantum blocks via qubits in the quantum memory.

Performance metrics. Next to classical metrics, such as utility (see 'Methods, Metrics'), throughput or latency [86], the successful execution of quantum network applications is governed by quantum metrics, which are unique to quantum networks and not present in classical networks. Such quantum network-specific metrics include fidelity (see 'Methods, Metrics'), or the probability of success in executing an application, where the latter depends directly on the fidelity of the quantum states prepared.

Mode of Execution. There exist quantum applications and functionalities, where one pair of programs is executed only once, e.g. a simple example of quantum teleportation [9]. As in quantum computing, however, some quantum network applications [99] are expected to succeed only with a specific *probability of success* p_{succ} when executed once. The application is then typically executed many times in succession in order to gather statistics (for example to amplify p_{succ}). A common use case for executing the same application repeatedly also occurs when evaluating the performance of a system (as we do here), where the goal is to estimate quantum performance metrics, such as the probability of success or the fidelity (see 'Methods, Metrics'). When executing the same application multiple times, the programmer can choose to launch many instances of the same program at once if multitasking is possible (see below), or to write one program which repeatedly executes the node's part of the program, asking for a successive execution of the application.

4.8.3 Interactive classical-quantum execution

Let us elaborate further on the relation, and differences, between the execution of quantum network applications, and the execution of quantum computing applications: One could envision building a system for executing quantum network applications on top of a simpler system for the execution of quantum computing programs, as long as the latter can be *augmented with networking instructions to generate entanglement*: in essence one quantum block can be seen as one quantum computing program. Such a block may realize mid-circuit measurements by the classical control logic allowed within one quantum block, or error correction. Error correction could in this paradigm be realized both by classical control logic allowed within one quantum block, or by considering the error correction itself as part of the Quantum Device (QDevice) (see Section 4.9.1) which then only exposes logical qubits and operations to QNodeOS, instead of physical qubits and operations. In that sense, one may think of the interactivity required between classical and quantum operations as taking place not only at a higher level, but also stemming from the fact that classical messages are used to create a new interaction between *separate* quantum programs, while in quantum computing we have only one single program.

4.8.4 Different hardware platforms

Platform Independence

We provide further background on the concept of platform, i.e. hardware, independence. It is the goal of our architecture to be platform-independent, including a standard interface to a driver for different hardware platforms. The driver is thus the only part that is platform-dependent in order to steer the underlying hardware platform. Such an interface is known as a hardware abstraction layer Hardware Abstraction Layer (HAL) that allows interfacing with different (quantum) platforms. To restate, in the context of "classical" operating systems, a HAL is a core component that existed in many operating systems (like Windows 7 [84, Section 19.3.1]) and continues to be used extensively to this day in operating systems for a broad set of computing platforms, including mobile ones [**android_hal**]. A HAL allows the operating system kernel to interact with the device hardware (drivers) through standardized programming interfaces, instead of relying on interfaces written specifically for each available hardware (which would, for example, necessitate to know how to configure specific memory bus or access specific type of memory for a specific hardware running on a device that the operating system has to control). Therefore, a HAL allows for an ultimate portability of the operating system, making it platform-independent above the HAL, and simplifies the architecture of the operating system.

QDevice

We consider as a quantum processor system the QDevice model (see Section 4.6, 'QDevice Model'), exposing a set of physical instructions addressing specific qubits (see Section 4.9.6). These physical instructions may be dependent on the type of quantum hardware, e.g., Nitrogen Vacancy (NV) in diamond, or trapped ions, and (1) include instructions for initializing and measuring qubits on the chip, (2) moving the state of a qubit to another location in the quantum memory (3) performing quantum gates, as well as (4) to make attempts at entanglement generation at the physical layer [78].

Quantum processors in general offer two types of qubits (see e.g. [25]): *communication qubits* which can be used to generate entanglement with remote nodes next to other quantum operations, as well as *storage qubits* which cannot be used to generate entanglement and only for implementing local quantum operations. We remark that on near-term quantum processors, the types of operations also depends on the connectivity of the qubits. That is, not all (pairs of) qubits may allow the same set of quantum operations to be performed on them.

To later enable compile time optimization, it is desirable that quantum hardware furthermore exposes the capabilities of the quantum chip: (1) the number of qubits, (2) the type of each qubit, (3) the memory lifetime of the qubits, (4) the physical instructions that can be performed on on the qubit(s) and (5) the average quality of these instructions.

4.8.5 Timescales

Quantum network programs are meant to be executed between distant nodes, meaning the communication times between them are in the *millisecond* regime. We remark that the same is *not true for networked or distributed quantum computing* : if the goal is to combine several less powerful quantum processors via a network into one more powerful quantum computing cluster, then it is advisable to place the individual processors as close to each other as possible, in order to minimize the time needed to (1) exchange messages, and (2) generate entanglement between processors. Thus, apart from the execution of applications following a different paradigm (see Figure 4.1), the case of distributed quantum computing also has different timescales than quantum networking. Of course, it is conceivable that in the future, one may also link distant quantum computers into more powerful quantum computing clusters via quantum internet infrastructure.

4.8.6 Scheduling network operations

In order for two neighboring quantum network nodes to produce heralded entanglement between them, they need to simultaneously perform an action to trigger entanglement generation (at the physical layer, *synchronized to nanosecond precision*). This means neighboring quantum network nodes need to perform a network operation (entanglement generation) in a *very specific* time slot in which they make an attempt to generate entanglement. Such time slots are generally aggregated into larger time bins, corresponding to making batches of attempts in time slots synchronized at the physical layer. We refer to e.g. Ref. [78] for background information on the physical layer of entanglement generation in quantum networks, and the readers with a background in computer science to e.g. Ref. [25] for a detailed explanation of scheduling of entanglement generation in quantum networks.

In short, network operations in quantum networks need to be executed by the node at very specific time bins. These time bins cannot be determined by the quantum node itself. Instead selection of time bins for a specific quantum operation require agreement with the neighboring node [25] (and more generally with the quantum network when the end-to-end entanglement is made via intermediary network nodes) by means of a network schedule, e.g. determined by a (logically) centralized controller, see Ref. [85].

4.8.7 Scheduling local operations versus scheduling network operations

For computer scientists, we provide further information on the inability to execute at the same time both local as well as networked quantum operations on present-day quantum processors. At a high-level, present-day quantum processor can be seen as both a quantum Central Processing Unit (CPU)/memory, as well as network device at the same time. Physical properties of the device and its control at the level of experimental physics, prohibits the usage of the quantum processors for both network and CPU/memory functions at the same time. A good example is given by the system of NV centers in diamond [44, 49]: the communication qubit, i.e. the network device, of the NV quantum processor system is given by its electron spin. Further storage qubits may be available by the surrounding nuclear spins in the diamond material. However, such nuclear spins cannot easily be addressed without involving the electron spin, prohibiting their use as a separate processor that is independent from the use as a network device.

It is conceivable that in the future, two devices could be used [97]: one *quantum processor as a network device* (but not as a device for execution of general quantum gates and measurements), and a another *quantum processor performing only local quantum operations* (but not as a device for long-distance networking). The network device could produce entanglement with distant quantum nodes (which may be taking many *milliseconds* to conclude successfully), and only once such entanglement is ready inject it into the second quantum processor. The latter may still involve short-distance entanglement generation between the network device and the second quantum processor, which however is very fast at short distances. This way the time that the second quantum processor would be blocked by networking operations would diminish significantly.

4.8.8 Multitasking

When executing quantum network applications, multitasking is well motivated in order to increase the utility of the system. Multitasking (or time sharing) is a well-established concept in classical operating systems (see e.g. [84, Section 1.4]) that allows the concurrent execution of multiple programs. For the reader from physics, we summarize some of these concepts in order to give context, and then reflect on what these imply in our setting.

In order to allow for multitasking, operating systems typically employ a notion of processes (or threads [84, Chapter 4], or tasks [84, Section 3.1]), where a process is created whenever a program starts, and the process forms an instance of the program being executed on the system. Multitasking (time sharing) thus refers to the concurrent execution of multiple processes at once, where it is possible to have multiple processes for the same program, corresponding to the execution of several instances of the program in parallel. We remark that the term concurrent thereby refers to the fact that the processes are existing in the system at the same time, while—due to the fact that they need to share limited resources (e.g. a CPU or other devices)—not all of them may be running at the same time.

Allowing multitasking requires the system to include a number of additional features:

Managing Processes

At a high-level, multitasking requires the system to keep track of the currently running processes, which means that when program starts executing, a process must be registered

in the system. Since the system needs to decide which process can be executed at what time, i.e., which process can be given access to the necessary resources to allow its execution, the system needs to keep track of the state of the process, which typically includes (1) whether it is ready for execution, (2) currently running, or (3) whether it cannot currently be executed since it is waiting e.g. for other processes.

In the case of executing quantum network applications, different parts of the application require different resources in order to run: classical blocks need the classical processor (CPU) and potentially network device present in a CNPU, while quantum blocks require the quantum processor (QDevice). It is desirable for our system that both resources can be used concurrently. That is, two different processes should be able to execute a classical block (on the CNPU), and a quantum block (on the QNPU) at the same time.

Memory Management Unit

A program typically relies on the ability to store classical variables (in a classical memory), as well as quantum variables (the state of qubits in a quantum memory). Such variables are stored in a classical and quantum memory device (here, the quantum processor), respectively. In order to allow multiple concurrent processes at the same time, the system needs to keep track of which part of the classical and quantum memory is assigned to which process. This concept is known broadly as memory management [84, Section 1.7] in classical operating systems.

In order to allow multitasking of quantum network applications, we thus require a Quantum Memory Management Unit (QMMU) (next to standard ways of performing classical memory management). The QMMU is responsible for the following tasks:

Qubit information handling. A QMMU has knowledge of the physical qubits available on the underlying quantum hardware, and may keep any other information about said qubits, such as the qubit type (communication or storage qubit) and qubit lifetime. Physical qubits thereby refer to both qubits realized at the device level, e.g. in the electron spin states of the NV center in diamond, or at a logical level where quantum error correction [61] is used to protect the quantum memory, i.e. one logical qubit is created by performing error-correcting using many device level qubits. A QMMU should allow such physical qubits to be assigned to different owners, i.e. different processes, or the operating system itself.

Transfer of qubit ownership. The QMMU may also allow a transfer of ownership of the qubits from one owner to the other, such as for example from a network process which makes entanglement to a user process.

Quantum memory virtualization. A QMMU may also provide abstractions familiar to classical computing such as a virtual address space, where the applications refer to virtual qubit addresses that are then translated to physical qubit addresses. This virtual address space avoids the situation in which physical qubit addresses must be bound at compile time, particularly limiting when allowing multiple applications to concurrently run on the same node. This would allow the transparent moving of qubits in a quantum memory in the future (for example moving them from a processor to a memory-only device while the process is waiting, e.g., for a message from a remote node). We remark however that the noise in present-day quantum devices means that any such move introduces a significant

amount of additional noise to the quantum state that may prevent the successful execution of the application.

Qubit memory lifetime management. Advanced forms of a QMMU may also cater to the limitations of near term quantum devices, by matching memory lifetime requirements specified by the application code to the capabilities of the underlying qubits, as well their topology, i.e., taking into account which two qubits allow two-qubit gates to be performed on them directly. While one cannot measure the decoherence of a qubit during a general program execution on the quantum level, the QMMU could also take into account additional information from the classical control system to signal to the application that a qubit has become invalid.

Scheduler

When multitasking, we need to decide which process should be executed at what time. This concept is referred to as scheduling in classical operating systems [90, Section 2.4], [84, Section 3.2]. We first discuss design considerations for scheduling when executing quantum network applications, and then reflect on how scheduling may be realized at different levels of the operating system for the quantum network nodes.

General considerations. We first provide three general considerations for completeness, which are not specific to the execution of quantum network applications but apply to all system in which several resources (such as the QDevice and a classical CPU) can be used (largely) independently of each other:

1. *Local quantum computation*: in addition to quantum networking, a node's resources must also be reserved for local quantum gates, which are integral parts of quantum network applications.
2. *Multitasking*: for a node to be shared by multiple users, the scheduler should not allocate all the available resources to a single application indefinitely, and instead it should be aware of the presence of multiple applications.
3. *Inter-block dependencies*: quantum and classical processing blocks of an application may depend on results originating from other blocks, and thus cannot be scheduled independently.

Quantum network considerations. Two specific considerations stand out in the domain of quantum networking:

1. *Synchronized network schedule*: due to the bilateral nature of entanglement, each node needs to have its quantum networking activity synchronized with its immediate neighbors. This means that while the scheduler at each QNodeOS node runs independently of each other, nodes must take into account the network schedule which defines when the node needs to perform networking actions with its neighboring node.
2. *Limited memory lifetimes*: the performance of quantum networking applications depends on both classical as well as quantum metrics. Once qubits are initialized, or entanglement has been created, the limited lifetime of present-day quantum memories implies that execution must be completed by a specific time in order to achieve a desired level of quantum performance.

Quantum/classical performance metrics trade-off. The best quantum performance is reached when the entire quantum network system (all nodes) are reserved for the execution of one single quantum network application. That is, programs are executed in a serial fashion and no multitasking is performed that could introduce delays which negatively impact the quantum performance. However, this approach does not in general achieve the best utilization of the system.

While our implementation makes use of a simple priority based scheduler, we remark that our work opens the door to apply more advanced forms of schedulers in the future. In particular, the fact that execution quality degrades over time suggests using forms of real-time schedulers for quantum network applications (taking inspiration from the extensive work on this topic in classical systems, see e.g. Ref. [62]). We remark that a programmer (or compiler) could provide advise on such (soft) deadlines, for example in the form of a lookup table that includes suggestions for deadlines for a desired level of quantum performance, based on the capabilities provides by the underlying hardware systems (e.g. memory lifetimes, expected execution time of quantum blocks), and the network (e.g. rate and quality (fidelity) of the entanglement that can be delivered). This advise could then be used by the scheduler to inform its scheduling decisions.

We remark that determining precise deadlines (e.g. when too much time has elapsed for the qubits to yield a specific probability of success) is in general a computationally expensive procedure, sometimes estimated in practice by a repeated simulation of the execution. It is an interesting open question to find (heuristic) efficient methods to approximate a performance prediction. We remark that there is no way in quantum mechanics to measure the current quality of a qubit or operation during the ongoing execution, and such qualities are determined by performing estimates independently of the program execution itself. Of course, QNodeOS could itself engage in such estimates when idling in order to update its knowledge of the capabilities of the quantum hardware.

To allow for potentially time-consuming classical pre- and post-processing, it is natural to apply such deadlines not for the entirety of the application, but for the period between initializing the qubits and terminating the quantum part of the execution. While outside the scope of this work, we remark that this type of scheduling offers to inspire new work in a form of "quantum soft-real time" scheduling, where deadlines may occasionally be missed at the expense of reduced application performance (success probability), to maximize the overall (averaged) performance of the system in which applications are typically executed repeatedly.

Scheduling at different system levels. Above we discussed scheduling at the level of processes, corresponding to executions of program instances. A system may realize scheduling at different levels, including

1. *Classical versus quantum processes*: The system may sub-divide processes into classical processes (executing classical code blocks), and quantum user processes (executing quantum code blocks). In this case, these can be scheduled independently (provided inter-dependencies are taken into account).
2. *Scheduling of quantum blocks*: The system may further sub-divide quantum processes into smaller units to allow different quantum code blocks of the same process to be scheduled independently.

3. *Scheduling of individual operations*: The level of operating systems is not typically concerned with the scheduling of individual operations, which is instead taken care of by the underlying CPU. We remark that while we do not envision this type of scheduling to be part of such a system in the future, but rather be relegated to control hardware in a microarchitecture for quantum nodes as e.g. in Fu et al. [38], our current realization of QNodeOS achieves a simple form of instruction schedule by populating an instruction queue in software due to the absence of a suitable low-level microarchitecture.

4.9 QNodeOS design and implementation

We proceed with a more detailed description of the QNodeOS architecture and implementation, where (for the reader's convenience) we include some information already found in Sections 4.3 and 4.6. Recall, that a quantum network application is realized by running separate programs, one on each end node of the quantum network that takes part in the quantum application. The individual programs interact with each other only via classical message passing and entanglement generation. Each program itself consists of classical and quantum blocks of code (see Section 4.8.2) which require execution in the quantum memory for the application to succeed.

4.9.1 QNodeOS architecture

Quantum Network Node System

We remark that QNodeOS—a real-time system for quantum network nodes—is designed to be deployed on end and intermediary nodes (Section 4.8.1), where QNodeOS use on intermediary nodes can be restricted to facilitate entanglement generation over the network via a (series) of intermediate nodes. As the focus of this work is the execution of quantum network application, we focus here on running QNodeOS on end nodes.

In our model, as depicted in Figure 4.2a, we divide the functions of a node into three high-level components:

- a *CNPU*, on which classical blocks of code are executed. The *CNPU* is required at end nodes, and requires classical computing hardware (including a classical CPU), as well as a classical network device to allow the exchange of messages with the CNPU of remote nodes. While quantum networking programs can in principle be developed and compiled outside of the CNPU), the *CNPU* may also realize a user environment where quantum networking programs (refer to Section 4.9.1) are developed and compiled, and where program results are stored;
- a *QNPU*, which receives quantum blocks from the CNPU and entanglement generation requests from peer nodes, and manages execution on the quantum physical device;
- a *QDevice*—the quantum physical device—consisting of a quantum processor, a quantum network device, and a quantum memory, where actual quantum computations and communications take place. In present-day quantum hardware implementations, the same device acts as a quantum processor, a network device and a memory.

In summary, in our design a quantum network program starts on the general-purpose Operating System (OS), i.e. a CNPU, which runs classical code blocks internally, and offloads quantum code blocks to the QNPU. The QNPU runs the quantum code blocks, relying on the underlying quantum device, i.e., QDevice, to execute the actual quantum operations.

CNPU and QNPU—while both being capable of performing non-quantum operations—are conceptually separate components, with the main difference being that the QNPU is expected to meet real-time requirements (to enable entanglement generation) and perform its arbitration tasks within set deadlines, whereas the CNPU does not need to provide such guarantees. This is because the QNPU should adhere to a network schedule which imposes real-time requirements. CNPU, QNPU and QDevice have a classical connection to their

counterpart at the remote node, where the QDevice also has an additional optical fiber connection to the quantum network to perform quantum operations.

An implementation of the quantum network node could have these three top-level components (CNPU, QNPU and QDevice) deployed on three physically distinct environments, or group some of them on the same chip or board. Furthermore, classical and quantum code blocks can be run on a single system, provided that this system has a connection to the quantum device to execute the actual quantum instructions. However, in the interest of a simpler implementation, where each system has a scoped responsibility, we opted to map classical and quantum blocks onto two distinct environments. Classical blocks are run on a system that features a fully-fledged OS (here, Linux), with access to high level programming languages (like C++ and Python) and libraries. Quantum blocks are delegated to the *QNPU*, which is a system capable of interpreting quantum code blocks and managing the resources of a quantum device.

We note that the QNPU itself is an entirely classical system that interacts with the quantum hardware (the QDevice). At the moment, our implementation of the QNPU is fully software, including the instruction processor. In general, the system may be implemented entirely in software running on a classical CPU, or parts of its functionality may be implemented in classical hardware, e.g. Field Programmable Gate Array (FPGA) (see the description of the trapped-ion platform implementation in Section 4.10.2) or Application-Specific Integrated Circuit (ASIC).

Quantum Network Programs

A quantum networking user program is what a programmer writes on the CNPU, in a high-level language, through the use of some Software Development Kit (SDK). Classical code blocks can in principle be programmed in any language yielding an executable suitable to run on the CNPU. Fully-classical code blocks—which include local processing and communication with other end nodes—often produce input data for the next quantum code blocks. That is, a classical code block typically precedes a quantum code block whose instructions depend on external data coming from a remote end node. In the future, quantum blocks could include real-time execution constraints, for example, a deadline by which execution should be completed in order to reach a specific application performance while the quantum memory has a limited memory lifetime.

NetQASM. To express quantum code blocks, we make use of *NetQASM* (see [26] and Chapter 3) as an instruction set for quantum network programs, which is described in detail in [26]. Before this work, NetQASM has only ever been used to execute quantum network programs on simulated quantum network nodes, and has never been realized on hardware to execute quantum network applications.

The instruction set used in NetQASM for the quantum code blocks is similar to other Quantum Assembly Language (QASM) languages (see e.g. Refs. [24, 37, 50]), but it is extended to include instructions for quantum networking. We emphasize that NetQASM is not a strict requirement of QNodeOS, and other ways to express quantum code blocks could be used in other implementations. The instruction set of this language should support both computational and networking quantum instructions, as well as simple classical arithmetic and branching instructions to be used for real-time processing on the QNPU. It is the compiler's task to transform high-level blocks for the QNPU into NetQASM blocks.

NetQASM defines a notion of NetQASM subroutines, where each subroutine corresponds to a quantum block of code, specified by the compiler or programmer. We therefore use the term *quantum block* to refer to a *NetQASM subroutine* in the remainder of this text. A full list of operands that can appear in a NetQASM subroutine is given in [26, Appendix B]. NetQASM assumed subroutines would be executed on a form of QNPU (without specifying an architecture for the QNPU), potentially using a form of shared memory with CNPU. In the absence of a shared memory, NetQASM allowed classical variables inside subroutines to be kept on the QNPU, and accessed read-only by the CNPU via the NetQASM interface (see below). The CNPU can also specify classical constants for the use inside subroutines, as part of submitting a subroutine to the QNPU.

We use here the NetQASM SDK [105] to write programs, where the SDK compiles a quantum network program, written in Python, into a series of classical and quantum code blocks. This SDK was previously used to express programs on a simulated quantum network [107].

NetQASM Interface. Our interface between the CNPU and the QNPU (Section 4.9.2) includes the NetQASM interface defined in [26, Appendix A]. This interface in particular allows the CNPU to register a program on the QNPU, submit NetQASM subroutines, and access the results of said subroutines.

Program Processing Pipeline

CNPU Processing. When a program start execution on the CNPU, a new CNPU process is created. As we separate the CNPU from the QNPU in our implementation, it is natural to rely on the properties of an existing classical operating system to take care of this function. In our implementation, we start a single program on the CNPU which then creates a thread (using standard Linux thread library [106] for each CNPU process. The classical blocks belonging to the CNPU program are executed locally on the CNPU. These may involve some form of coordination with the remote CNPU of the user program, as well as pre- or post-processing of the results coming from NetQASM subroutines. While this can also be done later, when the program starts it will typically also establish a TCP/IP connection with the program running on the remote CNPU leading to the establishment of a TCP/IP socket that will be used for classical application level communication between the CNPUs.

The CNPU then registers the program on the QNPU. Later, NetQASM subroutines of these programs are sent from the CNPU to the QNPU through the *NetQASM interface*.

QNPU Processing. When a program is registered with the QNPU by the CNPU, the QNPU creates a user process to store program data and execution state. The QNPU also keeps track of NetQASM subroutines belonging to the user process, which may be submitted only later, as well as other run-time data analogous to what a typical process control block contains, useful for the execution of the program. As depicted in Figure 4.2a, a subroutine can, in general, be composed of three classes of instructions:

- *Quantum operations*: quantum physical operations, to be performed on the underlying quantum device;

- *Classical logic*: arithmetic and branch instructions, to be executed in-between quantum operations, useful to store results of quantum operations and to perform responsive decision-making;
- *Entanglement requests*: requests to generate an entangled qubit pair with a remote node in the network.

Classical logic is processed locally on the QNPU, and potentially results in the update of a process's data. This data includes NetQASM variables capturing measurement results, for example, that may latter be conveyed to the CNPU.

When the user process starts on the QNPU an Entanglement Request (ER) socket (see Section 4.9.3) is established with the remote QNPU that is used to associate later entanglement requests with the specific user process. Entanglement requests contained in the NetQASM subroutines are forwarded to the quantum network stack, which stores them together with other requests coming from network peers. Entanglement generation requests coming from other nodes in the network are received on the quantum network stack through the *Quantum Network Stack (QNetStack) interface*.

Quantum instructions are sent to the QDevice through the QDevice Driver (QDriver), which provides an abstraction of the QDevice interface. The QDriver translates NetQASM instructions into physical instructions suitable to the underlying physical platform.

QDevice Processing. Physical instructions are executed on the QDevice, the quantum processing and networking unit. The QDevice processing stack heavily depends on the underlying physical platform—for instance, NV centers in diamond, or Trapped Ions.

As we remarked in Section 4.9.1, a QDevice has two communication channels with its direct neighbors: a *classical channel*, used for low-level synchronization of the entanglement generation procedure and other configuration routines, and a *quantum channel*, typically an optical fiber, through which qubits can travel.

4.9.2 QNPU stack

QNodeOS is a system consisting of multiple abstraction layers, as depicted in Figure 4.7. It is designed to be platform-independent, i.e., independent of the underlying quantum physical platform (quantum hardware) controlled by QNodeOS, where connections to different realizations of QDevice are captured by a platform-dependent QDriver. The implementation of QNodeOS itself is of course dependent on the classical physical platform(s) on which QNodeOS is implemented, including the physical interface between the CNPU and QNPU.

QNPU API

At the center of the stack lie the *the QNPU API handler* layer and the *the QNPU core* layer. The API handler is responsible for listening to system calls made to the the QNPU API, and to relay these calls to the appropriate component inside of the core layer. Such system calls may originate from the CNPU via the *CNPU communication handler*, see again Figure 4.7.

The QNPU API is the central engine for managing the execution of local quantum operations and entanglement requests, and manages the hardware resources of the QDevice. The QNPU API exposes services to:

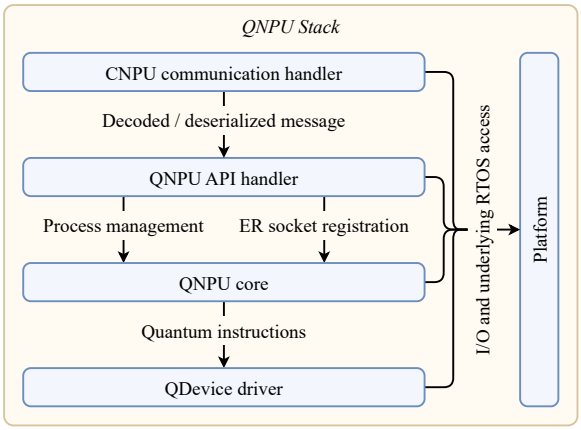


Figure 4.7: QNPU stack. The *QNPU API handler* and *the QNPU core* form the central processing layers, and are independent of the underlying quantum physical platform and of the device where QNodeOS runs. The *CNPU communication handler* translates protocol-specific messages from the CNPU into API calls. The *QDevice driver* (or QDriver) abstracts the QDevice hardware. The *Platform* layer abstracts the hardware where QNodeOS runs, and is accessible to all other layers. Note that three other Application Programming Interface (API) types are implemented, i.e. *control*, *management*, and *operations*. Control API is used for the network schedule, while management and operations API are for operational purposes.

- *Register and deregister a program on the QNPU*; This is part of the NetQASM interface (see Section 4.9.1).
- *Add a quantum block (subroutine) for a user process*; This is again part of the NetQASM interface.
- *Open an ER socket with a remote node* (NetQASM interface).
- *Control to configure the quantum network stack*, i.e., to configure the network schedule; This is used for the interaction with a network controller that sets network-wide entanglement schedules, as presented in Ref. [85].
- *Perform management and operations functions.*

The topmost horizontal layer is the *CNPU communication handler*, which implements a *protocol wrapper* around NetQASM. We implement this wrapper protocol using EmbeddedRPC [102] for the on-the-wire definition of the messages (including (de-)serialization)). The communication handler translates protocol-specific messages into API calls for the QNPU. EmbeddedRPC allows to decouple the interface definition and (de-)serialization from the underlying transport layer. We note that only the transport layer is implementation-specific, which depends on the devices where CNPU and QNPU are implemented and on what the physical interface between them looks like.[1]

The *QDevice driver* (QDriver) layer, at the bottom of the stack, provides an abstraction of the QDevice hardware, and its implementation depends on the nature of the QDevice

[1]TCP/IP for now, shared memory in the future.

itself, and on the physical communication interface between QNPU and QDevice. Two QDevice implementations may differ in a variety of factors, including what quantum physical platform they feature and what digital controller interfaces with the QNPU.

Lastly, the vertical *Platform* layer provides System on a Chip (SoC)-specific abstractions for the QNPU to access the physical resources of the platform it is implemented on, including I/O peripherals, interrupts controllers and timers. Additionally, if the QNPU is implemented on top of a lower-level operating system, this layer gives access to system calls to the underlying OS. The Platform layer is vertical to indicate that it can be accessed by all other QNPU layers.

Porting the QNPU to a different SoC (or similar hardware) boils down to implementing a new platform layer. Deploying the QNPU on a different QDevice, instead, requires both a new QDriver and a compiler—on the CNPU—that emits quantum instructions supported by the specific QDevice.

4.9.3 Processes

A quantum network program starts on the CNPU—there, the CNPU environment compiles it into classical and quantum code blocks, and creates a new process associated with the program. In the future, an optimized compilation ahead of execution could produce an executable that includes further information (such as execution deadlines depending on the device's memory lifetimes, as mentioned at the beginning of Section 4.9.1). The CNPU then registers the program with the QNPU (through the QNPU's end node API, see Section 4.9.2), which, in turn, creates its own process associated with the registered program. The process on the CNPU is a standard OS process, which executes the classical code blocks and interacts, (that is: communicating NetQASM subroutines and their results between CNPU and QNPU), with the counterpart process on the QNPU. This interaction can be done by means of a shared memory (and when no shared memory is physically realized: by an exchange of messages [26]). On the QNPU, a process encapsulates the execution of quantum code blocks of a program with associated context information, such as process owner, process number (ID), process state, and process priority.

In the near-term test applications we execute, the execution time of a program is typically dominated by that of quantum blocks, as entanglement generation is a time-consuming operation. Without advanced quantum repeaters [3], its duration grows exponentially with the distance between the nodes. For this reason, we focus on the scheduling of quantum blocks only, and thus we only discuss QNPU processes (also referred to as *user processes*) from this point onward. Again, this does not exclude that in a future iteration of the design CNPU and QNPU could be merged into one system, and therefore classical and quantum blocks would be scheduled jointly.

Process Types and Their Interaction

QNPU user processes. The QNPU allocates a new *user process* to each quantum network program registered by the CNPU. A user process is the program's execution context, and consists of NetQASM blocks and other context information—the process control block—including process number (ID), process owner, process state, process scheduling priority, program counter, and pointers to process data structures. Process state and priority determine how processes are scheduled on the QNPU. A user process becomes active (ready to

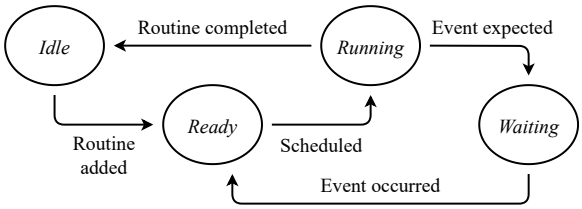


Figure 4.8: Process state diagram. An *idle* process becomes *ready* when a block for that process is loaded onto the QNPU (from the CNPU). A ready process becomes *running* when it is scheduled. A running process goes back to idle if all blocks are completed, or transitions to *waiting* if it expects an event to occur before it can proceed. A waiting process becomes ready again when the expected event occurs.

be scheduled) as soon as the QNPU receives a quantum code block from the CNPU. Multiple user processes—relative to different CNPU programs—can be concurrently active on the QNPU, but only one can be running at any time. A running user process executes its quantum code block directly, except for entanglement requests, which are instead submitted to the quantum network stack and executed asynchronously.

QNodeOS network process. The QNPU also defines *kernel processes*, which are similar to user processes, but are created when the system starts (on boot) and have different priority values. Currently, the only existing kernel process is the *network process*. The network process, owned by the QNetStack, handles entanglement requests submitted by user processes, coordinates entanglement generation with the rest of the network, and eventually returns entangled qubits to user processes. The activation of the network process is dictated by a network-wide entanglement generation schedule. Such a schedule defines when a particular entanglement generation request can be processed, and therefore it has intersecting entries on adjacent nodes (given that entanglement is a two-party process). The schedule can be computed by a centralized network controller [85] or by a distributed protocol [25]. In our design, the network process follows a *time division multiple access schedule*, computed by a centralized network controller (as originally proposed by Skrzypczyk and Wehner [85]) and installed on each QNodeOS node (see Section 4.9.3).

QNPU process states. A QNPU process can be in any of the following states: (1) *Idle*: when the CNPU has registered a program and the QNPU has spawned a process, but it has not received a block yet; (2) *Ready*: when it has (at least) one block, sent from the CNPU, and can be scheduled and run; (3) *Running*: when it is running on the QNPU and has the quantum processor and the quantum network device assigned to it; (4) *Waiting*: when it is waiting for some event to occur. Figure 4.8 shows the possible process states and the valid state transitions. A process transitions from idle to ready when one block gets added. A ready process transitions to running when the the QNPU scheduler assigns it to the processor. A running process transitions to waiting when it has to wait for an event to occur, and transitions from waiting to ready when the event occurs—for instance, a process could be waiting for an Entanglement Pair Request (EPR) pair to be generated, and become ready again when the pair is established. Finally, a process goes back to the idle state when all its blocks have been completed.

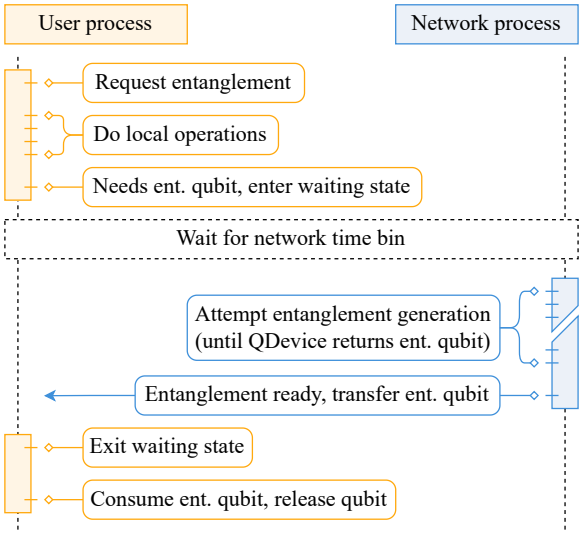


Figure 4.9: Flow of execution between a user process requesting entanglement and the network process responsible for generating entanglement. The user process starts by asynchronously issuing an entanglement request. Once issued, it is free to continue with other local operations or classical processing. Once it reaches a point in its execution where entanglement is required the process enters the waiting state. The network process is scheduled once the appropriate time bin (as determined by the network schedule) starts. Once running, it attempts entanglement generation until entanglement success (or until a set timeout). The entangled qubit is then transferred to the user process. This unblocks the process which consumes the entanglement and releases the qubit. In our experiments, the process always immediately waits after requesting entanglement (no local operations are done in between).

Inter-process communication. At the moment, the QNPU does not allow for any explicit inter-process communication. The only indirect primitive available to processes to interact with one another is *qubit ownership transfer*, used when a process produces a qubit state which is to be consumed by another process. Most notably, the quantum network stack kernel process transfers ownership of the entangled qubits that it produces to the process which requested the EPR pairs.

Process concurrency. The strict separation between local quantum processing and quantum networking is a key design decision in QNodeOS, as it helps us address the scheduling challenge, see Section 4.8. A user process can continue executing local instructions even after it has requested entanglement. Conversely, networking instructions can execute asynchronously of local quantum instructions. This is important in a quantum network, since entanglement generation must be synchronized with the neighboring node (and possibly the rest of the network [85]). Additionally, separating user programs into user processes also allows QNodeOS to schedule several programs *concurrently*.

Process flow. Figure 4.9 illustrates the typical control flow between a user process and the network process. User processes are free to execute any non-networked instructions independently of the network process and other user processes. Once the program reaches a point in its execution where an entangled qubit is required, the process enters the waiting

state and is flagged as waiting for entanglement. When the network process is scheduled, it issues network instructions and generates entanglement as requested by the user process. Once an entangled pair is generated by the network process, the qubit is handed over to the waiting user process. When all the entangled pairs that the user process was waiting for are delivered, the user process becomes ready and can start running again.

Process Scheduling

At present, the the QNPU scheduler does not give any guarantees on when a process is scheduled—for that, one would need to define concrete real-time constraints to feed to the scheduler. Instead, the current version of the QNPU implements a best-effort scheduler, which selects processes on the basis of their *priority*, and does not allow preemption. In particular, the network process is assigned the highest priority, and is activated whenever the network schedule specifies entanglement should be made in the next time-bin [85].

As already mentioned, QNodeOS defines the concept of *user* processes and *kernel* processes, with the QNetStack process being the only kernel process at the moment. User processes are released (i.e., they become ready) asynchronously—when a process block is loaded, or when they leave the waiting state—while the QNetStack process is released periodically—at the beginning of each time bin of the network schedule (although the period of time bins can vary). Given that generating an EPR pair on a link requires that both nodes attempt entanglement simultaneously, the QNPU assigns the QNetStack process a priority higher than any user process. This ensures that, at the beginning of each time bin of the network schedule, the *priority-based* process scheduler can assign the QNetStack process as soon as the processor is available, and thus a node can start attempting entanglement with its neighbor as soon as possible and minimize wasted attempts on the neighbor node.

Figure 4.10 exemplifies a snapshot of a hypothetical execution of a user process and the QNetStack process. The latter is activated at the beginning of a time-bin assigned to networking, and is scheduled as soon as the processor is available—for instance, at times 0 and 4 it is scheduled immediately, while at time 8 it is scheduled after one time unit, as soon as the running process yields. The user process becomes ready at time 0—at which point the QNetStack process is ready as well and has highest priority, meaning the network process is scheduled; then it is scheduled at time 2, as soon as the QNetStack process completes; then it goes into waiting state at time 3 because the user process requested entanglement and it waits for the entanglement to be established; finally it becomes ready again at time 7—and it is scheduled immediately given that no other processes are running.

To avoid context switching overhead, potentially leading to degraded fidelity, the QNPU scheduler is *cooperative*. That is, once a process is scheduled, it gets to run until it either completes all of its instructions or it blocks waiting for entanglement. Allowing process preemption would need a definition of critical section and could potentially impact the quality of the affected qubit states. Moreover, the lack of a preemption mechanism could potentially result in low-priority user processes hogging the processor at the expense of high-priority entanglement generation attempts. On the other hand, if entanglement instructions always consume the entirety of the time bin, the QNetStack process would be immediately assigned the processor each time it relinquishes it, causing low-priority user processes to starve. To at least mitigate the second issue, we made sure that

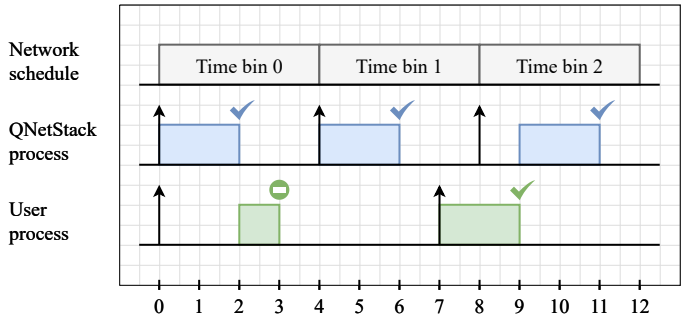


Figure 4.10: Snapshot of a hypothetical execution of a user process and the QNetStack process. The higher-priority QNetStack process is activated at the start of each time bin of the network schedule, and it is assigned to the processor as soon as it is available. The lower-priority user process gives precedence to the QNetStack process when they become ready at the same time, but, when it is running on the processor, it is not preempted if the QNetStack process becomes ready while the user process is running. Black arrows represents a moment where the process goes into the ready state and the green stop sign (at time 3) represents a process going into the waiting state.

the number of consecutive entanglement attempts performed by the QDevice within one single entanglement instruction is always less than how many would fit in a time bin, so as to leave some slack for low-priority user processes to run.

Networking

The network stack QNetStack is based on the existing stack [25], including the link layer Quantum Entanglement Generation Protocol (QEGP) [25]. However the main difference between the QNetStack implemented on the QNPU and the original design of the protocols lies in how the QEGP processes the outstanding entanglement requests. QEGP [25] employed the concept of a distributed queue to sort and schedule entanglement requests on one node by coordinating with the counterpart node on the other end of the link, to ensure that both nodes would be servicing the same entanglement request at any given time. This synchronization is necessary because different entanglement requests may require different EPR pair fidelities, in which case QEGP would issue different QDevice entanglement instructions. However, link-local request scheduling becomes more complicated if nodes have more than just one link. In that case, entanglement requests would be better scheduled at a level where network-wide request schedules are known.

Network Schedule. The QEGP protocol implemented on the QNPU transitioned [78] from scheduling entanglement requests via a pairwise agreed upon distributed queue, to deferring this task to a *logically centralized control plane*, whereby a node's schedule can be computed on the basis of the whole network's needs by a (logically) centralized controller (see e.g. [85]). This means that the network stack of the nodes convey their demands for end-to-end entanglement generation to the central controller, who then makes a *network schedule*, which is communicated back to the nodes.

All nodes divide time into time-bins, where the central controller employs a scheduling algorithm to assign either network actions (or no actions) to time-bins. That is, the term

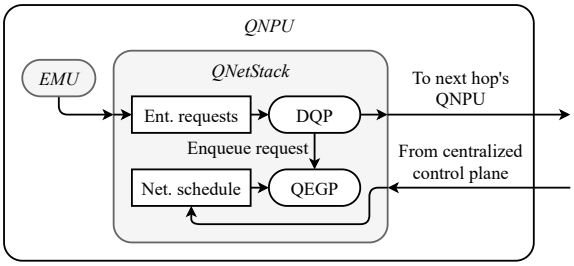


Figure 4.11: Internal components and data structures of the Quantum Network Stack (QNetStack). Entanglement requests are received through the Entanglement Management Unit (EMU), while the network schedule is installed by a centralized control plane. Quantum Entanglement Generation Protocol (QEGP) maps such requests onto the network schedule to produce the correct entanglement instructions. While not needed on our 2 node implementation, a Distributed Queue Protocol (DQP) (which is a simplified version of the DQP presented in [25, Section 5.2.1]) could forward entanglement requests to the next hop's Quantum Network Processing Unit (QNPU) to realize a network layer protocol such as [54].

.

network schedule refers to a schedule, i.e. allocation of resources over time, of time-bins at the nodes, where a time-bin may be marked for networking activities (entanglement generation) or be left empty (to be used arbitrarily to execute local operations). Given that entanglement generation requires a non-deterministic amount of attempts and time, time bins are computed to be large enough to accommodate the average run time of an entanglement generation instruction. We remark that the node functions internally as a higher timing granularity than a time-bin allocated by the network scheduler, that is, it can execute other operations (such as for example local quantum operations) also within a time-bin allocated by the network schedule, provided entanglement is made early.

Once the node received the network schedule from the controller, the network schedule is used to satisfy all outstanding end-to-end entanglement requests, and is used by QEGP to produce the correct QDevice instructions at any point in time. Whenever a time bin is assigned to networking to two neighboring nodes, the nodes attempt entanglement generation over their shared link in order to realize the QEGP link layer protocol. Figure 4.11 shows internal components and data structures of the QNetStack as it is implemented on the QNPU. Entanglement requests received by the Entanglement Management Unit (EMU) are forwarded by Quantum Network Protocol (QNP) to the next hop's QNPU system. Entanglement requests and network schedule—the latter installed by a logically centralized control plane—are used by QEGP to produce the correct entanglement instructions to populate the QNetStack process's block at each activation of the process.

ER Socket. The concept of an ER socket is inspired by that of a classical network socket, in that it defines the endpoint of an entanglement generation request, and is used by the QNPU's quantum network stack to set up network tables and to establish connections with its peers. We remark that the current realization of the ER Socket (see below) is a proof of concept implementation opening future computer science research, and does not aim to prevent misuse if different users had access to the same node. A program can request from QNodeOS the opening of an ER socket with a program on a remote node.

An ER socket is identified by the tuple (`node_id, er_socket_id, remote_node_id, remote_er_socket_id`). The other program (on the other node) must open its own corresponding ER socket (i.e with values (`remote_node_id, remote_er_socket_id, node_id, er_socket_id`)) on its own QNodeOS. A request for opening an ER socket is executed by the CNPU, by asking the QNPU (through the QNPU API) to open the socket. The QNPU then registers the ER socket with the quantum network stack (provided it did not yet exist), and the CNPU also keeps a reference using the tuple as an identifier. The program can then use this socket for requests. The network stack only handles requests for entanglement between two nodes if the corresponding ER sockets are opened on both nodes.

Programs are themselves responsible for coordinating the ER socket IDs. Using these IDs allow the same node pair to open multiple pairs of ER sockets, which may be used by different applications or inside the same application. Socket IDs must be unique within the node. ER sockets are typically opened at the start of a program, and live (and may be used multiple times) until the program finishes.

Programs use the ER socket to submit entanglement requests to the network stack. This is done through NetQASM instructions (`create_epr` and `recv_epr`) that refer to the ER socket in their operands. One program must execute a `create_epr` instruction and the other a `recv_epr` instruction (to be coordinated by the programs themselves). The program executing the `create_epr` instruction is treated by the network stack as the *initiator* and the program executing `recv_epr` the *receiver*. Upon receiving an entanglement request, the network stacks of the two nodes communicate between each other in order to coordinate entanglement generation. The initiator node always initiates this communication. The receiver node always accepts the entanglement initiative as long as the corresponding ER socket is open. This means that the receiver node agrees with entanglement generation as soon as the initiator node has submitted an entanglement request (through its `create_epr`), even if the receiver node itself has not yet submitted its corresponding request (through its `recv_epr`). On the receiver node, the generated entangled qubit will remain in memory until it gets asked for by a user process executing this `recv_epr`.

Multitasking

Multi-tasking forms an essential element of our architecture already at the level of scheduling the network process in relation to any user process, to address the challenges inherent in the way entanglement is produced at the physical layer, requiring agreement on a network schedule (see Section 4.9.3). For this important reason, the QNPU is designed to arbitrate between these two processes (see Figure 4.9), and to manage the resources being used by each of them. *Multitasking*, hence, is a fundamental requirement for a system managing the hardware of a quantum network node, especially while such hardware has only limited resources available.

To further increase the utility of the system, we also allow the multi-tasking of user processes. Like in most operating systems, these tasks, which on the QNPU are encapsulated into processes, can sometimes necessitate a resource which is not immediately available—for instance, a free qubit, or a qubit entangled with a remote one. Maximizing the utilization of the quantum device is also one of the goals of QNodeOS, whose design allows multiple processes, user and kernel, to be active concurrently, so that whenever one is in a waiting state, another one can potentially be scheduled to use the quantum

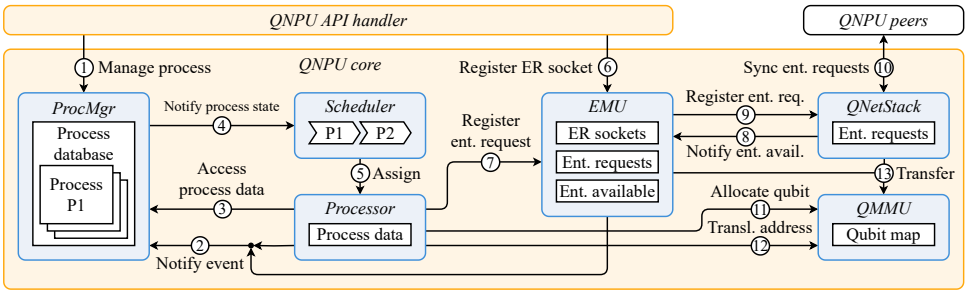


Figure 4.12: Quantum Network Processing Unit (QNPU) core components and internal interfaces. The core layer includes: (1) a *process manager* (ProcMgr), which owns and manages access to QNPU processes; (2) a *scheduler*, responsible for selecting the next process to be run; (3) a *processor*, which processes blocks' instructions; (4) an *EMU*, which keeps a list of entanglement requests and available entangled qubits; (5) a *QNetStack*, whose responsibility is to coordinate with peer nodes to schedule quantum networking instructions; (6) a *QMMU*, which keeps a record of allocated qubits.

device. This design aspect is relevant for quantum networking nodes, as the execution of the local program is often waiting, both for classical messages from remote nodes, as well as the generation of entanglement.

Lastly, multitasking is an important feature for systems that are to be shared by multiple users, and that offer each user the possibility to run multiple programs concurrently. The multitasking capabilities of QNodeOS are also aimed at improving the average throughput and latency of user programs.

4.9.4 QNodeOS components and interfaces

We provide here additional details on the components of the QNPU architecture and their interfaces. Figure 4.12 gives an overview of all the components of the QNPU. The *process manager* marshals accesses to all user and kernel processes. The *scheduler* assigns ready processes to the *processor*, which runs quantum instructions through the underlying QDevice, processes classical NetQASM instructions locally, and registers entanglement requests with the *EMU*. The EMU maintains a list of ER sockets and entanglement requests, forwards the latter to the *quantum network stack*, which, in turn, registers available entangled qubits with the EMU. Finally, the *QMMU* keeps track of used qubits, and transfers qubit ownership across processes when requested.

Process Manager

The process manager owns QNPU processes and marshals accesses to those. Creating a process, adding a block to it and accessing the process's data must be done through the process manager. Additionally, the process manager is used by other components to notify *events* that occur inside the QNPU, upon which the state of one of more processes is updated. Process state updates result in a notification to the scheduler.

Interfaces. The process manager exposes interfaces for three services:

- Process management (interface 1 in Figure 4.12): to create and remove processes, and to add quantum blocks to them. When the user registers a program, the the QNPU API Handler uses the process manager to create a QNPU user process. The returned process ID can be later used to add a block to that process, or to remove the process once all its blocks are fully processed.
- Event notification (interface 2 in Figure 4.12): to notify an event occurred inside the QNPU, including the addition of a block, the completion of a block, the scheduling of the process, the hitting of a *Waiting* condition (see Figure 4.8), and the generation of an entangled qubit destined to the process. Some events trigger follow-up actions—for instance, when a process that was waiting for an event becomes ready, it gets added to the queue of ready processes maintained by the scheduler.
- Process data access (interface 3 in Figure 4.12): to access a process's blocks and its classical memory space, mostly used while running the process (through the processor).

Scheduler
The QNPU scheduler registers processes that are ready to be scheduled, and assigns them to the QNPU processor when the latter is available. Ready processes are stored in a *prioritized ready queue*, and processes of the same priority are scheduled with a first-come-first-served policy.

Interfaces. The scheduler only exposes one interface for process state notifications (interface 4 in Figure 4.12), used by the process manager to signal when a process transitions to a new state. When a QNPU process transitions to the ready state, it is directly added to the scheduler's prioritized ready queue. When a process becomes idle, or is waiting for an event to happen, the scheduler simply registers that the processor has become available.

Processor
The QNPU processor handles the execution of QNPU user and kernel processes, by running classical instructions locally and issuing quantum instructions to the QDriver. It is also responsible for multitasking by means of process manager. While executing a process, the processor reads its blocks and accesses (reads and writes) its classical memory. The processor implements a specific instruction set architecture dictated by the NetQASM language of choice.

Interfaces. The processor exposes one interface for processor assignment (interface 5 in Figure 4.12), used by the QNPU scheduler to activate the processor, when it is idling, and assign it to a QNPU process.

Entanglement Management Unit
The Entanglement Management Unit (EMU) contains a list of open *ER sockets* and a list of *entanglement requests*, and keeps track of the *available entangled qubits* produced by the quantum network stack. Received entanglement requests are considered valid only if an ER socket associated to such requests exists. Valid requests are forwarded to the quantum network stack. Entangled qubit generations are notified as events to the process manager.

Interfaces. The EMU exposes interfaces for three services:

- ER socket registration (interface 6 in Figure 4.12): to register and open ER sockets belonging to a program, and to set up internal classical network tables and to establish classical network connection.
- ER registration (interface 7 in Figure 4.12): to add entanglement requests to the list of existing ones, to be used when matching produced entangled qubits with a process that requested them.
- Entanglement notification (interface 8 in Figure 4.12): to register the availability of an entangled qubit, produced by the quantum network stack, and to link it to an existing entanglement request.

Quantum Network Stack

The quantum network stack on the QNPU closely follows the model presented by Dahlberg et al. [25] which is based on the classical Open Systems Interconnect (OSI) network stack model for the purpose of the separation of responsibilities. In particular, *data link layer* is part of the quantum network stack on the QNPU. The *physical layer* is implemented on the QDevice, the *application layer* is part of the CNPU, and all remaining layers are not currently part of the stack.

The quantum network stack component has an associated *QNPU kernel process*, created statically on the QNPU. However, this process's block is dynamic: the instructions to be executed on the processor depend on the outstanding entanglement generation requests received from EMU and network peers.

Interfaces. The quantum network stack exposes interfaces for two services:

- Entanglement request registration (interface 9 in Figure 4.12): to add entanglement requests coming from the EMU to the list of existing ones, which are used to fill in the quantum network stack process's block with the correct quantum instructions to execute.
- Entanglement request synchronization (interface 10 in Figure 4.12): similar to the entanglement request registration interface, but to be used to synchronize (send and receive) requests with QNodeOS network peers.

Quantum Memory Management Unit

Quantum Memory Management Unit (QMMU) receives requests for *qubit allocations* from QNPU processes, and manages the subsequent usage of those. It also translates NetQASM *virtual qubit addresses* into physical addresses for the QDevice, and keeps track of which process is using which qubit at a given time. In general, a QMMU should take into account that the topology of a quantum memory determines what operations can be performed on which qubits, and thus allow processes to allocate qubits of a specific type upon request. An advanced QMMU could also feature algorithms to move qubits in the background—that is, without an explicit instruction from a process's block—to accommodate a program's topology requirements while not trashing the qubits being used by other QNPU processes. Such a feature could prove crucial to increase the number of processes that can be using the quantum memory at the same time, and to enhance multitasking performances.

Interfaces. The QMMU exposes interfaces for three services:

- Qubit allocation and de-allocation (interface 11 in Figure 4.12): a running process can ask for one or more qubits, which, if available, are allocated by the QMMU, and the physical addresses of those are mapped to the virtual addresses provided by the requesting process.
- Virtual address translation (interface 12 in Figure 4.12): before sending quantum instructions to the QDriver, the processor uses virtual qubit addresses specified in NetQASM to retrieve physical addresses from the QMMU, and then replaces virtual addresses with physical addresses in the instructions for the QDriver.
- Qubit ownership transfer (interface 13 in Figure 4.12): qubits are only visible to the process that allocates them. However, in some cases, a process may wish to transfer some if its qubits to another one. A notable example is the quantum network process transferring an entangled qubit to the process that will use it.

4.9.5 QNPU implementation: scheduler

The QNPU scheduler is an important component of our QNodeOS architecture, and deals with scheduling of QNPU processes. The QNPU is implemented on FreeRTOS [103], which itself includes a scheduler. On FreeRTOS, code is organized into tasks, which can be seen as separate threads or processes. These tasks are scheduled concurrently by FreeRTOS based on priority. In our implementation, we realize QNPU components and interfaces (hence including the QNPU scheduler) as FreeRTOS tasks. We configured task priorities such that the components with tight interaction with the QDevice (QDriver, quantum network stack, QNPU processor) have highest priority. We stress the difference between the FreeRTOS scheduler and our QNPU process scheduler. The QNPU scheduler schedules QNPU processes based on their status and priorities, which are independent of the priorities assigned by the FreeRTOS scheduler. The FreeRTOS hence runs on a different layer: it makes sure the QNPU components (including QNPU scheduler, processor, QDriver) run concurrently. The QNPU scheduler runs on the level of QNPU processes. Whenever the FreeRTOS scheduler activates the FreeRTOS task realizing the QNPU scheduler, the QNPU scheduler then schedules the process with the highest priority on a first come first serve basis, by adding it to the processing queue of the relevant resource (e.g. QNPU processor) and generating an interrupt leading to the execution of the QNPU processor task on FreeRTOS (and consequently the execution of the process).

4.9.6 QDevice interface

The implementation of a QDevice depends on a number of factors. Most importantly, the physical signals that are fed to the quantum processing and networking device (and those that are output from the device) are specific to the nature of the device itself. Different qubit realizations require different digital and analog control. For instance, manipulating the state of a spin-based qubit (e.g., in a NV center processor) and that of an atom qubit (e.g., in a trapped ion processor) are two physical processes that vastly differ in a number of complex ways.

For QNodeOS to be portable to a diverse set of quantum physical platforms, there needs to be a common *QDevice interface* that QNodeOS can rely on, and that each QDevice in-

stance can implement as it is most convenient for the underlying quantum device. This interface (1) needs to be *general*, (2) to be able to *express all quantum operations* that different quantum devices might be capable of performing, and (3) *abstract*, so that two different implementations of a well-defined qubit manipulation operation can be expressed with the same instruction on QNodeOS. Nevertheless, an interface that is too general could result in a high implementation complexity on the QDevice, as it might have to transform high-level instructions in a series of native operations on the fly. Other than complexity of implementation, a very high-level set of QDevice instructions might compromise the compiler's ability to optimize a program for a certain physical platform, as reported by Murali et al. [71].

Design Choices

Defining a set of instructions to express abstract quantum operations as close as possible to what different quantum physical platforms can natively perform is—to some extent—an open problem. Nonetheless, we have made an effort to specify an interface which is a good compromise between generality and expressiveness. The QDevice interface is essentially a set of instructions that QNodeOS expects a QDevice to implement. To be precise, a QDevice might implement a subset of the interface, according to what native physical operations it can perform. The CNPU compiler must then have knowledge about the set of instructions implemented by the underlying QDevice, so that it can decompose instructions that are not natively supported.

Even though this interface does not impose any formal timing constraints, it is important to note that a QDevice implementation that tries to guarantee more or less deterministic instruction processing latencies can prove more beneficial to the real-time requirements of the QNPU. Particularly, it would be advisable to time-bound the processing time of operations whose duration is by nature probabilistic—most notably, those involving entanglement generation. Creating an EPR pair may involve a varying number of attempts. Sometimes, if the remote node becomes unresponsive for some time, the number of necessary attempts can increase by a large amount. Capping the number of attempts could, for instance, provide a more deterministic maximum processing latency for entanglement instructions, which in turn might help QNodeOS react more timely to temporary failures or downtime periods of remote nodes. Not to mention that unbounded entanglement attempts affect the state of other qubits in memory, because of both passive decoherence and cross-qubit noise.

QDevice synchronization

The QDevice receives physical instructions from QNodeOS, acts on them, and returns a response. For entanglement instructions, the QDevice must first synchronize with the QDevice on the other node (using classical communication). If the other QDevice is busy, (e.g. it is still trying to pass the CR check, see Section 4.10.1 and [78]), synchronization fails, and an `ENT_SYNC_FAIL` response is returned (see Table 4.2).

Instructions and Operands

Table 4.1 lists the complete set of instructions defined in the QDevice interface. Instructions can have operands, whose range of valid values depends on the underlying QDevice. For instance, an operand that specifies which qubit to apply an operation to can only have

Instruction	Description
INI	Initialize a qubit to default state
SQG	Perform a single-qubit gate
TQG	Perform a two-qubit gate
AQG	Perform a gate on all qubits
MSR	Measure a qubit in a specified basis
ENT	Attempt entanglement generation
ENM	Attempt entanglement and measure qubit
MOV	Move qubit state to another qubit
SWP	Swap the state of two qubits
ESW	Swap qubits belonging to two EPR pairs
PMG	Set pre-measurement gates

Table 4.1: Summary of QDevice instructions defined in the QDevice interface. A specific QDevice might implement a subset of these, depending on the underlying quantum physical device and on other design constraints.

as many valid values as there are physical qubits in memory. Details for each instruction and its operands are given below.

Qubit Initialization (INI). The INI instruction brings a qubit to the $|0\rangle$ state. On some physical platforms, single-qubit initialization is not possible, thus this instruction initializes all qubits to the $|0\rangle$ state.

Operand	Description
qubit	Physical address of the qubit to initialize, ignored on platforms where single-qubit initialization is not possible

Single-Qubit Gate (SQG). The SQG instruction manipulates the state of one qubit. The gate is expressed as a rotation in the Bloch sphere.

Operand	Description
qubit	Physical address of the qubit to manipulate
axis	Rotation axis, can be X, Y, Z or H (support is QDevice-dependent)
angle	Rotation angle (granularity and range are QDevice-dependent)

Two-Qubit Gate (TQG). The TQG instruction manipulates the state of two qubits. The gate is expressed as a controlled rotation, with one qubit being the control and the other one being the target.

Operand	Description
qub_c	Physical address of the control qubit
qub_t	Physical address of the target qubit
axis	Rotation axis, can be X, Y, Z or H (support is QDevice-dependent)
angle	Rotation angle (granularity and range are QDevice-dependent)

All-Qubit Gate (`AQG`). The `AQG` instruction manipulates the state of all available qubits. The gate is expressed as a rotation in the Bloch sphere.

Operand	Description
`axis`	Rotation axis, can be X, Y, Z or H (support is QDevice-dependent)
`angle`	Rotation angle (granularity and range are QDevice-dependent)

Qubit Measurement (`MSR`). The `MSR` instruction measures the state of one qubit in a specified basis. A qubit measurement is destructive—that is—the qubit has to be reinitialized before it can be used again.

Operand	Description
`qubit`	Physical address of the qubit to measure
`basis`	Measurement basis, can be X, Y, Z, H (support is QDevice-dependent)

Entanglement Generation (`ENT`). The `ENT` instruction performs a series of entanglement generation attempts, until one succeeds, or until a maximum number of attempts is reached (the behavior is QDevice-dependent).

Operand	Description
`nghbr`	Neighboring node to attempt entanglement with, if the local QDevice has multiple quantum links
`fid`	Target entanglement fidelity (granularity and range are QDevice-dependent)

Entanglement Generation With Qubit Measurement (`ENM`). The `ENM` instruction performs a series of entanglement generation attempts followed by an immediate measurement of the local qubit, until one succeeds, or until a maximum number of attempts is reached (the behavior is QDevice-dependent).

Operand	Description
`nghbr`	Neighboring node to attempt entanglement with, if the local QDevice has multiple quantum links
`fid`	Target entanglement fidelity (granularity and range are QDevice-dependent)
`basis`	Measurement basis, can be X, Y, Z, H (support is QDevice-dependent)

Qubit Move (`MOV`). The `MOV` instruction moves the state of one qubit to another qubit. A qubit move renders the state of the source qubit undefined, and the qubit has to be reinitialized before it can be used again.

Operand	Description
`qub_s`	Physical address of the source qubit
`qub_d`	Physical address of the destination qubit

Qubit Swap (SWP). The SWP instruction swaps the state of two qubits.

Operand	Description
qub_1	Physical address of the first qubit
qub_2	Physical address of the second qubit

Entanglement Swap (ESW). The ESW instruction results in two qubits belonging to two EPR pairs to have their roles swapped.

Operand	Description
qub_1	Physical address of the first qubit
qub_2	Physical address of the second qubit

Pre-Measurement Gates Setting (PMG). The PMG instruction allows for a set of (up to) 3 rotations to be performed before a qubit measurement (MSR or ENM). If the axis of the second rotation is orthogonal to the axis of the first and the third rotation, these gates can be used to perform a qubit measurement in an arbitrary basis, given that most likely a QDevice can natively measure in a limited set of bases.

Operand	Description
axes	Combination of orthogonal axes to use for the three successive rotations, can be X–Y–X, Y–Z–Y and Z–X–Z (support is QDevice-dependent)
ang_1	Rotation angle of the first gate, relative to the first axis in axes (granularity and range are QDevice-dependent)
ang_2	Rotation angle of the second gate, relative to the second axis in axes (granularity and range are QDevice-dependent)
ang_3	Rotation angle of the third gate, relative to the third axis in axes (granularity and range are QDevice-dependent)

No operation (NOP). The NOP instruction does not result in any operation on the QDevice.

Return values

Table 4.2 lists the possible return values that the QDevice sends back to QNodeOS as a response to a physical instruction.

Physical Instruction	Return values	Description
INI, SQG, TQG, AQG, PMG	SUCCESS	Always successful
MSR	SUCCESS_0 or SUCCESS_1	Measurement outcome is 0 or 1*
ENT	SUCCESS_<state>	Entanglement generation was successful; the state is <state>†
ENM	SUCCESS_<state>_<outcome>	Entanglement generation was successful; state was <state>† and outcome is <outcome> (0 or 1)
ENT, ENM	ENT_FAILURE	Entanglement generation was attempted and failed
ENT, ENM	ENT_SYNC_FAILURE	Entanglement generation was not attempted since synchronization failed (other node is busy)

Table 4.2: *Measurements are always in the Z basis, where outcome 0 corresponds to $|0\rangle$ and outcome 1 to $|1\rangle$. † Possible states depend on the implementation. For NV these are PSI_PLUS and PSI_MINUS, see Section 4.10.1.

4.10 QDevice implementations

4.10.1 NV center platform

The QDevice employed for the benchmark experiments is constituted by an NV center processor. We use the NV center in its negatively charged state (called NV^-) for quantum information processing. NV^- is a spin-1 system, whose ground states are non-degenerate in the presence of an external magnetic field, see Figure 4.13 [31]. We employ the $m_s = 0$ as our $|0\rangle$ state for the qubit, while for the $|1\rangle$ we can choose one of $m_s = \pm 1$. Details on how the choice is made will follow in the next section. The NV can be optically excited resonantly (637 nm) and off-resonantly (typically 532 nm), and it emits in 3% of the cases single photons (Zero-Phonon Line (ZPL) photons), while the remaining part is constituted by the emission of a photon and a phonon (Phonon-Side Band (PSB)). The optical transitions are spin-selective, as shown in Figure 4.13. In the presence of lateral strain and external DC field (Stark effect), the excited states of the NV split apart, maintaining their spin-selective properties. In this work, we use a natural lateral strain between 2 GHz and 5 GHz. The cycling transition denoted as Readout (RO) in Figure 4.13 is used to emit single photons (ZPL) for entanglement generation and to read out the state of the qubit (fluorescence in the PSB). From the excited states, the NV can also decay through metastable states (not shown in Figure 4.13). The preferable decay from such metastable states is the $m_s = 0$ state. In this way, it is possible to optically initialize the qubit state to $|0\rangle$ (dashed line in Figure 4.13), with fidelity above 99%, when on-resonantly exciting the Spinpump (SP) transition and averaging for long enough to ensure a spin-flip. In our experiments, we apply a laser field on resonance with the SP transition at 500 nW for 1.5 μs for fast initialization during entanglement attempts, whereas a slow initialization (10 nW for 100 μs) is used for single-qubit gates experiments (like local tomography). On the other hand, while exciting the RO transition, decays to $m_s = \pm 1$ are also possible, but they present longer cyclicity. This feature is relevant for the optical read-out of the qubit state, which can be done in a single shot and is discussed in the following sections.

In our demonstration, the server has an external magnetic field of $B_z = 189$ mT aligned along the symmetry axis z of the NV, while the client experiences $B_z = 23$ mT. The magnetic field is applied via permanent magnets placed both inside and outside the high-vacuum chamber of our closed-cycle cryostats. Fluctuations of the magnetic field are observed on the order of nT on a timescale of hours, therefore they do not constitute a limitation for our demonstration. We also measured a perpendicular component of the permanent magnetic field for both setups of ~1 mT. Such misalignment becomes crucial for the coherence time of the electron spin qubit, as in the interaction with the surrounding nitrogen nuclear spins, the off-axis hyperfine interaction terms become non-negligible and the decoupling of the electron spin is harder [31]. Notably, the server node is in the regime of "high magnetic field". In the level structure depicted in Figure 4.13, this means that the $m_s = -1$ ground state crossed the $m_s = 0$ state (at ~ 100mT), and the optical transitions for the SP are well separated, such that a double laser field with proper detuning is necessary to correctly address both of them.

Single Node Operations

In this section, details on how to operate a single node for quantum information processing are given. The physical setup is the one employed for the demonstration in Ref. [78].

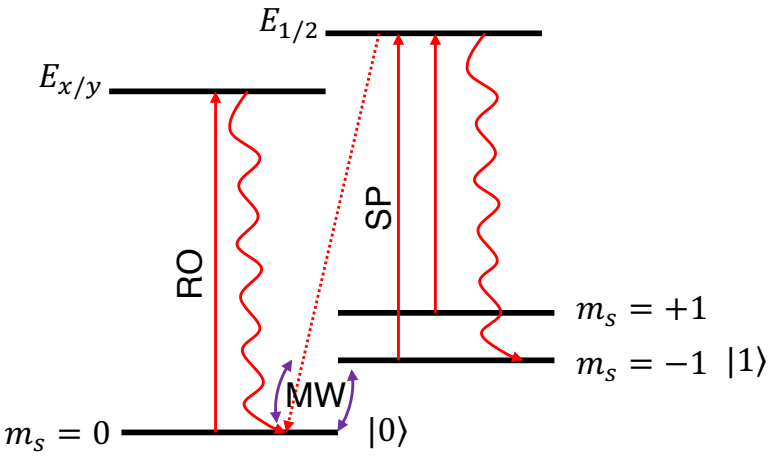


Figure 4.13: Energy structure of NV^- at 4 K. The ground state of the NV splits into three distinct levels (Zeeman splitting). The optical transitions are spin-selective. The excited states are represented as one, but they are non-degenerate when lateral strain is applied. We denote as Readout (RO) the transition $|0\rangle \longrightarrow E_{x/y}$ and as Spinpump (SP) the transition $|1\rangle \longrightarrow E_{1/2}$. The wiggly lines represent the photoluminescence when such transitions are excited, whereas the dashed lines represent the decay via metastable states that is used for initialization of the qubit state into $|0\rangle$. Microwave (MW) pulses enables the transfer of population between the two states of the qubit, allowing for quantum information processing.

Charge-resonance check. To use the NV as a processing node, it is necessary to guarantee that it is in the correct charge state and the laser fields are on resonance with the transitions. Before executing any instructions coming from the QNPU, both nodes go through the so-called Charge-Resonance (CR) check. We apply resonant fields for 100 μs on both the RO (1 nW) and SP (10 nW) transitions and we monitor the fluorescence. If the number of photons exceeds the threshold (25 for the client and 60 for the server for our experiments), the node is considered ready to accept instructions from the QNPU and can proceed with synchronization with the other node (for multinode instructions). The threshold is set considering the brightness of each NV. The success is considered valid for 100 ms. After this time, if no instructions arrive, the CR check is repeated. In case the number of photons is below the threshold, we distinguish two cases: (1) the counts are between the success threshold and a second threshold called Repump: we repeat the CR check and tune the frequency of the red lasers, as they might not address the transitions correctly; (2) the number of counts is below the Repump threshold (set at 15 for the client and 25 for the server): this means that the NV might be in the dark charge state (NV^0) due to ionization. To restore the charge state, in the next round of CR check we first illuminate with off-resonant green laser (20 μW for 50 μs), or, for the client node only, with yellow light (575 nm, 35 nW for 300 μs) on resonance with the ZPL transition of NV^0 [4]. This is necessary because we additionally apply an external DC field to the NV on the client node. We, indeed, exploit the Stark effect to tune the RO transition to be the same as the server's one [6]. In this way, we can ensure photon indistinguishability in frequency that is crucial for entanglement generation. The typical DC field used for this work is ~2V, modulated via an error signal that is computed on the CR check counts, acting as a

	Client	Server
Duration π rotation	200 *ns*	190 *ns*
Amplitude π rotation	0.78	0.89
Skewness π rotation	-1.5e^{-9}	-3.5e^{-9}
Duration $\pi/2$ rotation	150 *ns*	100 *ns*
Amplitude $\pi/2$ rotation	0.38	0.56
Skewness $\pi/2$ rotation	-1.2e^{-8}	-7.1e^{-9}
Power	42 W	42 W

Table 4.3: Characterizing values for the MW pulses. Other rotation angles have the same duration and skewness as the π pulse, and the amplitudes scale accordingly. The rotation axes are obtained by changing the phase of the pulse. With the current setup configuration, only rotations along $\hat{x}$ and $\hat{y}$ axes are feasible, so $\hat{z}$ rotations are compiled as combinations of gates along $\hat{x}$ and $\hat{y}$.

Proportional-Integral-Derivative (PID) loop.

The CR check is repeated after an experiment iteration. This round is utilized to validate the experiment and post-select the results based on success or failure of this procedure, as discussed in Section 4.6.

Single qubit gates. To manipulate the state of the electron spin qubit, microwave pulses are on resonance with the transition $|0\rangle \longrightarrow |1\rangle$ are employed. For the server node, the $m_s = -1$ state is used as $|1\rangle$ and the resonance frequency is 2.4 GHz. The client node utilizes the $m_s = +1$, with a resonance frequency of 3.5 GHz. The choice of the $|1\rangle$ is made based on the gate fidelity.

We use skewed-Hermite Microwave (MW) pulses [96, 98] with high Rabi frequency (~10 MHz), which generates an alternating magnetic field capable of manipulating the state of the qubit. The characterizing values for the two nodes are reported in Table 4.3. The measured infidelity on a single MW pulse is below 1%. Instructed by the QNPU, we performed local quantum tomography on both the server and the client, showing high fidelity. One example is reported in Figure 4.18.

Dynamical decoupling. Once MW pulses are set up with high fidelity, it is possible to implement Dynamical Decoupling (DD) sequences that increase the coherence time of the electron spin qubit. DD sequences are especially crucial in our experiments when the latency of the QNPU is long (milliseconds timescale), like in the Delegated Quantum Computation (DQC) demonstration. The characterizing parameter for a DD sequence is the time delay between the X and Y pulses. To optimize it, we swept the interpulse delay, at the sample precision of our Arbitrary Waveform Generator (0.42 *ns*, Zurich Instruments HDAWG), while playing the effective single-qubit computation of the DQC protocol instructed by the QNPU, as explained in Section 4.6, on both the client and the server. In doing so, we added an extra waiting time of 5 *ms* between the initialization of the qubit into the superposition state and the subsequent gates to mimic the real-case scenario of the DQC. In this way, we are able to set the optimal interpulse delay, obtaining a single-qubit fidelity of 0.96(2) for the server and 0.88(2) for the client.

Single-shot readout. When a measurement instruction arrives from the QNPU, this is translated by the physical layer as a Single-Shot Readout measurement. To assign a state

to the qubit, we can use the RO optical transition. The RO laser field is on for ~10 μs at 1 nW. This will produce fluorescence only if the NV is in the $|0\rangle$ state. If no photons are detected while the laser is on, the outcome is assigned to the $|1\rangle$ state. The fidelity of the measurement process is defined as $F = 1/2(F_{0|0} + F_{1|1})$, where $F_{0|0}$ ($F_{1|1}$) represents the fidelity of measuring $|0\rangle$ ($|1\rangle$) when the qubit is prepared in $|0\rangle$ ($|1\rangle$). For our experiments, we obtain 0.841(4) and 0.997(1) respectively for the client, and 0.912(3) and 0.995(1) for the server, achieving above 0.90 of process fidelity.

Entanglement generation

The entanglement request from the QNPU is translated into executing a single-photon protocol. The communication qubit on each node is initialized in the state $\sqrt{\eta}|0\rangle + \sqrt{1-\eta}|1\rangle$, where η represents the bright state population. For maximum state fidelity, the condition $\eta_C p_C \approx \eta_S p_S$ applies, where $\eta_{C(S)}$ is the bright state population of the client (server) and $p_{C(S)}$ is the photon detection probability of the client (server). In this work, $\eta_C = 0.07$ and $\eta_S = \eta_C \frac{p_C}{p_S} = 0.04$. The choice of η_C is a trade-off between entangled state fidelity and entanglement generation rate. The produced entangled state is (non-deterministically) one of two Bell states $|\Psi^{\pm}\rangle = \frac{1}{\sqrt{2}}(|01\rangle \pm e^{i\Delta\theta}|10\rangle)$, based on which detector clicked at the heralding station. The phase $\Delta\theta$ is actively stabilized [77] before the execution of the entanglement request, via a combination of homodyne interference, for the global phase of the network, and a heterodyne interference, to stabilize the local phase at each node. Pauli-correction gates, based on the state prepared, are issued from the server QNPU to its QDevice to obtain $|\Phi^+\rangle$: an X_π gate if the generated Bell-state is $|\Psi^+\rangle$ and an X_π gate followed by a Z_π gate (decomposed into X and Y gates) for $|\Psi^-\rangle$. As preparation for the experiment, we verified the entanglement generation, instructed by the QNPUs and using the same method as in Ref. [78], achieving a fidelity of 0.72(±0.02) for the $|\Phi^+\rangle$ state. The Bell corrections done through the server QNPU take up to 0.16 ms for $|\Psi^+\rangle$ and up to 0.49 ms for $|\Psi^-\rangle$. On the other hand, generating entanglement without the QNPU and with no Pauli correction, we achieve an average fidelity of 0.74(±0.03), with $\eta_C = 0.1$ and $\eta_S = 0.06$. The choice of different η values is due, in the first place, to speed up the rate of such a measurement. It shows, however, better performance with respect to the instructed version, which is due to the fact that the Pauli-correction instruction comes with a latency.

4.10.2 Trapped-ion platform

Setup

The trapped-ion QDevice implementation faces different challenges than the NV QDevice implementation. Trapped-ion state preparation, gate operations, and readout occur on longer timescales (between microseconds and milliseconds) than the corresponding operations for NV centers. As a result, latencies introduced by QNodeOS are insignificant, and we do not expect the use of QNodeOS to reduce fidelities of local gate operations or entangling operations on the trapped-ion QDevice. On the other hand, trapped ions are typically manipulated using control sequences that are compiled for a given set of parameters and uploaded to hardware. (Here, sequences consist of pulses of Transistor-Transistor Logic (TTL) signals, analog voltages, and radio frequency or microwave signals, some of which are phase-referenced to one another. These pulses typically control the laser and

microwave fields with which ions are manipulated.) The challenge here is that decision making within QNodeOS must take place further up the network stack and is not compatible with pre-compiled sequences.

We address this challenge by exploiting a triggering capability within our pre-compiled sequences (which are written as Python scripts and then translated to a hardware description language for a FPGA). A sequence can contain labels that act as memory pointers; at any point in a given sequence, a function can jump to one of these labels, at which point execution continues starting at that label. Thus, we can structure a sequence as a list of possible subsequences—each of which corresponds to a physical instruction or some part thereof. This list is preceded by a control subsequence. Input triggers from QNodeOS cause the control subsequence to jump to a certain subsequence representing a physical instruction. After the subsequence—that is, the physical instruction—is implemented, the sequence returns to the control subsequence, where it waits for another input trigger.

A second challenge is the compatibility between QNodeOS and physical-layer hardware. The QDriver for QNodeOS is implemented with a development FPGA board (Texas Instruments LAUNCHXL2-RM57L [94]) that sends messages via Serial Peripheral Interface (SPI). Our physical layer hardware, however, is not compatible with serial communication protocols. We bridge this gap with an emulator board (Cypress, CY8CKIT-14371 [45]). The emulator board requests and reads SPI messages from QNodeOS and, based on the message, generate TTL signals that are sent as input triggers to the physical layer hardware. The emulator also receives outputs from the physical layer hardware: it monitors whether the hardware is available for new commands or busy, and it collects measurement results and passes them back to QNodeOS. In this case, the measurement result consists of TTL signals from the Photomultiplier Tube (PMT) detecting ion fluorescence. When a counter value on the emulator board exceeds a certain preset threshold, the ion state is registered as the qubit state $|0\rangle$ and otherwise as $|1\rangle$.

Testing the QDriver

Tests were carried out using a trapped-ion setup designed for integration with a fiber-based cavity [92, 93]. The qubit states consisted of the $4^2S_{1/2}$ and $3^2D_{5/2}$ manifolds of $^{40}Ca^+$, hereafter referred to as $|0\rangle$ and $|1\rangle$. The cavity was not used in these tests, which focused on single-qubit operations. The cavity was designed to enable ion-photon entanglement, which we plan to implement in future work through physical instructions from QNodeOS. Our primary goal in these tests was to verify that the QNodeOS hardware, the emulator, and the physical-layer hardware could work together.

Initial tests confirmed that messages were being exchanged at the programmed clock rate of 50 kHz and that hardware pulses in the physical layer were triggered correctly via the emulator. Next, the following seven tests were implemented:

1. Initialization of the ion in a specific Zeeman state via Doppler cooling and optical pumping;
2. a bit flip around the X axis via a π pulse with phase 0 on the 729 nm quadrupole transition of $^{40}Ca^+$;
3. a bit flip around the Y axis via a π pulse with phase $\pi/2$ on the 729 nm transition;

4. preparation of a superposition state via a $\pi/2$ pulse with phase 0 on the 729 nm transition;
5. readout of a qubit eigenstate in the Y basis via a π rotation around X followed by a $\pi/2$ pulse with phase $\pi/2$ around X on the 729 nm transition;
6. readout of a superposition state in the X basis via a $\pi/2$ rotation around Y followed by a $\pi/2$ pulse around X on the 729 nm transition;
7. measurement of the ion's electronic state via fluorescence at 397 nm in the presence of an 866 nm repump, following preparation of a superposition state.

Operations are considered to be correctly realized from the point of QNodeOS, but do contain errors at the quantum level. Results for the tests (numbers above) were as follows:

1. The ion was detected in the target initial state $|0\rangle$ in 98.4% of trials;
2. Following the X-axis bit flip operation, the ion was detected in $|1\rangle$ in 96% of trials;
3. Following the Y-axis bit flip operation, the ion was detected in $|1\rangle$ in 95% of trials;
4. A projective measurement determined that the $^{40}Ca^{+}$ ion was in $|0\rangle$ 52% of the time and in $|1\rangle$ 48% of the time;
5. A projective measurement determined that the $^{40}Ca^{+}$ ion was in $|0\rangle$ 54% of the time and in $|1\rangle$ 46% of the time;
6. The ion was detected in $|0\rangle$ 93% of the time;
7. The ion population was found to be in $|0\rangle$ 37% of the time and in $|1\rangle$ 63% of the time.

These results were consistent with the performance of the physical-layer hardware in the absence of QNodeOS. (Note that Doppler cooling had not been optimized and that magnetic-field drifts at the time were not properly compensated for. Gate operations with much higher fidelities are typically achieved in trapped-ion experiments, but here our focus was on verifying the electronic signaling.) No problems or inconsistencies with the electronic signaling were identified.

A next step will be to implement a more sophisticated processing of PMT TTL signals by the emulator board in order to identify when the ion has been delocalized due to a background-gas collision; in that case, additional laser cooling will be implemented that returns the ion to Doppler-limited temperatures. Such a step is a typical part of compiled physical-layer sequences but should now be implemented within QNodeOS as part of the physical instruction for qubit initialization.

4.11 Delegated quantum computation (DQC) experiment on NV

4.11.1 Procedure

We execute the application in a tomography way to establish QNodeOS the quantum performance metric (Figure 4.3b, where we use P_c to refer to the client program, and P_s to the server program): The client CNPU initiates P_c with fixed (α, θ). This results in a single CNPU process, a single QNPU process, and opening of an ER socket (see Section 4.9.3) with the server node. At the same time, the server CNPU initiates P_s resulting in single CNPU process, a single QNPU process, and opening of an ER socket with the client node. The client and the server programs execute the subroutines in Figure 4.3c, looping 1200 times: both immediately start the second iteration once the first is completed. After the 1200th iteration, both client and server stop their respective CNPU and QNPU processes. Source code including compiled NetQASM subroutines is available in Appendix B. We repeat 6 times for $(\alpha, \theta) \in \{\pi/2, \pi\} \times \{\pi/4, \pi/2, \pi\}$ for a total of 7200 executions of the circuit depicted in Figure 4.3a. We expect $|\psi\rangle$ to be either $|-Y\rangle$ (for $\alpha = \pi/2$) or $|-Z\rangle$ (for $\alpha = \pi$). To estimate the resulting $|\psi\rangle$ per (α, θ), the contents of S2 (containing the server qubit measurement) in the server loop is was varied such that we obtained 600 measurement outcomes in basis $|+Y\rangle$ ($|+Z\rangle$) and 600 measurement outcomes in the corresponding orthogonal basis $|-Y\rangle$ ($|-Z\rangle$) for $\alpha = \pi/2$ (π).

Since our experiments are conducted on two NV nodes that are directly connected, we install a constant network schedule with time-bins of 10 ms in which all time-bins are assigned to networking. This allows us to assess the performance of executing quantum network applications without introducing a dependence on changing network schedules. This means the network process is made ready at the start of each such time-bin, although may not instruct the QDevice to make entanglement if no requests for entanglement have been made.

4.11.2 Definitions

The result of a single DQC circuit execution (Figure 4.3a) is a single-qubit state ρ_{DQC} on the server. The success of running DQC can be expressed as the fidelity of ρ_{DQC} compared to the expected state (in case of no noise) $|\psi\rangle$ (Figure 4.3a). In the following we will call this fidelity the DQC fidelity, or F_{DQC}.

The value of F_{DQC} is affected the most by (1) the fidelity F_{EPR} of the entangled pair created between the client and server, and (2) the *qubit memory time* t_{mem}, which is the time that the server qubit must remain in memory (from entanglement success until measurement). The latter depends on the time at which the client sends a message to the server (Figure 4.3). We refer to the two-qubit maximally entangled Bell states as $|\Phi^+\rangle = \big(|00\rangle + |11\rangle\big)$, and $|\Psi^\pm\rangle = \big(|01\rangle \pm |10\rangle\big)$, where $\Phi^+ = |\Phi^+\rangle\langle\Phi^+|$ and $\Psi^\pm = |\Psi^\pm\rangle\langle\Psi^\pm|$.

4.11.3 Post-selection based on latency

In our experiments, the server qubit memory time t_{mem} has a significant variance across executions of the DQC circuit. In some iterations, there were huge spikes in latencies, which skew the results significantly. An upper bound t_{max} (see Section 4.11.4) was used to filter out results from iterations in which t_{mem} was larger than t_{max}. This resulted in

filtering 146 out of 7200 data points. We note that for computing F_{DQC}, we applied the latency filter on top of the Single-Shot Readout (SSRO) and CR filters (see Methods). For the processing time analysis (below), however, we applied only the latency filter directly to all 7200 original data points.

4.11.4 Simulation

A simulation (using NetSquid [22]) of the DQC application was performed in order to estimate the expected F_{DQC} on our NV setup, and to establish a suitable value for t_{max} (used in latency post-selection).

We emphasize that this simulation is a heuristic to find t_{max}, and does not aim to predict the performance to full accuracy. All runs for which latencies were less than t_{max} were ultimately used to assess the performance from data, not using this simulation.

The simulation contains the following steps, where we used the model explained in Ref. [77]:

1. Start with a density matrix ρ_{EPR} describing the approximate state of the EPR pair just after entanglement success.
2. Apply operations representing the local gates on both the client and server, including the measurement on the client qubit. These operations are assumed to be perfect (no noise).
3. Apply depolarizing noise to the server qubit for a duration of t_{mem}, using the decoherence formula $e^{-(t_{\text{mem}}/T_{\text{coh}})^n}$ where T_{coh} was set to 13 ms and $n = 1.67$. These values are obtained via fitting experimental data from prior tests.
4. Calculate the fidelity between the final server qubit state and the expected state $|\psi\rangle$.

Based on the parameters of the setup when the DQC experiment was performed, ρ_{EPR} is set to

$$\begin{bmatrix} 0.049 & 0 & 0 & 0 \\ 0 & 0.437 & 0.284 & 0 \\ 0 & 0.284 & 0.454 & 0 \\ 0 & 0 & 0 & 0.061 \end{bmatrix}$$

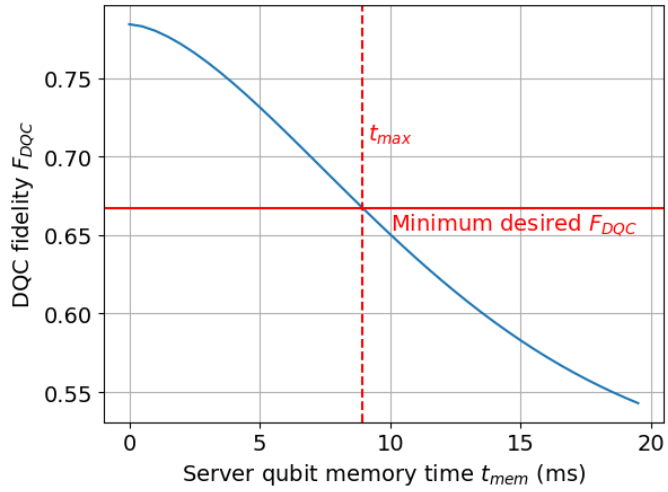
which has fidelity 0.729 to the perfect Ψ^+ state. The setup can also produce Ψ^- states but for simplicity we use only the Ψ^+ case here.

The simulation computes an estimate of F_{DQC} for a given server qubit memory time t_{mem}. Since the desired minimum value for F_{DQC} was 0.667, the latency threshold t_{max} was set to 8.95 ms (Figure 4.14a).

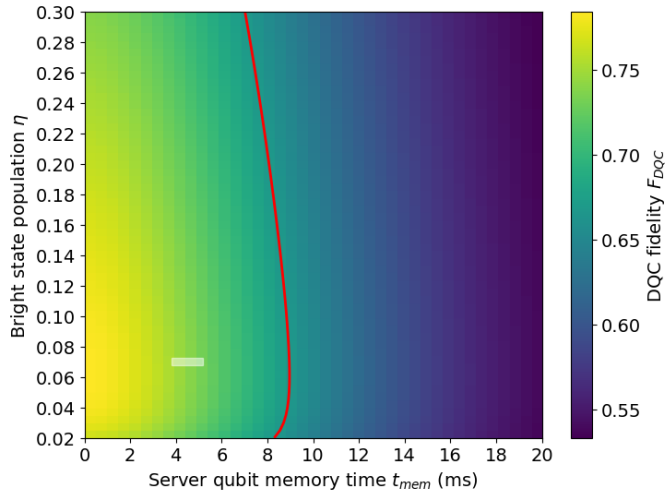
4.11.5 Sweep of qubit memory time and bright state population

As explained in Section 4.10.1, entanglement is created using the single-photon protocol using bright state population parameter η.[2] Using the simulation, we can estimate how F_{DQC} would change for different values of η and t_{mem}. Figure 4.14b shows the estimated F_{DQC} for different values of η and t_{mem}. It indicates that for the particular setup used,

[2] In most literature, the variable α is used for this parameter; here we use η to avoid confusion with the α parameter of the DQC application.



(a)



(b)

Figure 4.14: (a) Expected values (based on simulation, Section 4.11.4) of DQC fidelity F_{DQC} for different duration values that the server qubit must remain in memory (t_{mem}). The maximum allowed qubit memory time t_{max} is chosen such that application iterations that are expected to result in too low F_{DQC} (< 0.667) are filtered out. (b) Expected values (based on simulation) of DQC fidelity F_{DQC} for different values of the bright state population (η) in the single click protocol, and for different duration values that the server qubit must remain in memory (t_{mem}). The red line indicates the threshold of 0.667 for the target fidelity. The white box represents the experimentally obtained results (we fixed $\eta = 0.07$ and observed t_{mem} 4.8(8) ms, see Figure 4.3d).

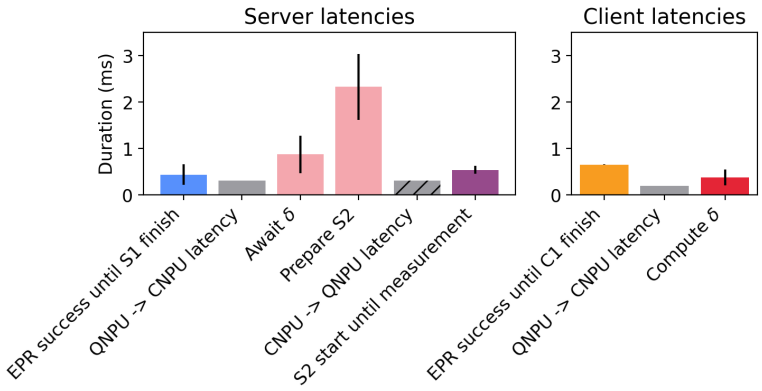


Figure 4.15: Average latency (duration) of each of the processes happening while the server qubit remains in memory in the DQC application. The QNPU to CNPU latency and CNPU to QNPU latency are estimated as explained in Section 4.11.6, and fixed to 0.305 ms (server) and 0.197 ms (client). The other latencies are the mean and variance of the corresponding processes averaged over all DQC circuit iterations that passed the latency filter.

increasing η has little effect, while reducing qubit memory time does. For the DQC experiment $\eta = 0.07$ was used.

4.11.6 Processing time and latencies

Here we provide a detailed breakdown of the duration of execution phases of the DQC application, in order to gain insights into the processing times and latencies of the system for the different components.

Server qubit memory time

Figure 4.3c shows the duration that the server qubit must remain in memory t_{mem} while waiting, averaged over all DQC circuit iterations that passed the latency filter. Figure 4.3d shows the breakdown of t_{mem} into individual segments of processing on both client and server. In Figure 4.15 we show the average duration and the variance of each of these segments. The largest time is spent on preparing S2, which involves running Python code on the CNPU and converting this (using Python) into a NetQASM subroutine. Caching of the preparation of the NetQASM subroutine could significantly speed up this process. In the future, further improvements could include an optimized ahead-of-time compilation step. The large variance is due to the fact that on the CNPU, other (background) processes run simultaneously with the DQC application process, and there is no precise control over the scheduling of these processes.

Tracing

The CNPU, QNPU, and QDevice all keep track of events happening in their system, by storing a tuple (t, e) where t is a timestamp and e the name of the event. The events that are traced on the CNPU and QNPU are listed in Section 4.14. A trace plot showing events in CNPU, QNPU, and QDevice during a single execution of the DQC circuit is also shown in Section 4.14.

Derived latency (fit)	Description	Value (ms)
$\Delta_{cS1} - \Delta_{qS1}$	Send S1 + receive S1 result	0.384
$\Delta_{qS12} - \Delta_{cS12}$	Receive S1 result + Send S2	0.609
$\Delta_{cS2} - \Delta_{qS2}$	Send S2 + receive S2 result	0.467
$\Delta_{cC1} - \Delta_{qC1}$	Send C1 + receive C1 result	0.394

Table 4.4: Derived values for CNPU-QNPU communication latencies. The Δ variables are observed timestamp differences on the CNPU or QNPU, per execution of the DQC circuit, as shown in Figure 4.16. Subtracting pairs of variables from each other produces sums of two CNPU-QNPU communication latencies. These sums of latencies highly fluctuate per execution of the DQC circuit, due to the inaccuracy of the CNPU timestamps. However, the data fits a constant value, which is shown in the table and used in further analysis.

The QNPU timestamp granularity is 10 μs, since that is the duration of a single QNPU clock cycle. This clock cycle is synchronized with the clock of the QDevice, which in turn is synchronized with the QDevice of the other node (see Section 4.6 and all paragraphs therein related to NV implementation). This results in the two QNPUs (of the two nodes in the experiment) having synchronized clocks with 10 μs precision. This means that the event indicating to the QNPUs that EPR generation has succeeded happens at the same clock cycle on both QNPUs.

The CNPU is not a real-time system (instead, it runs on a general purpose Linux OS) and records timestamps by consulting the system clock at μs precision. These timestamps are not synchronized to the QNPU timestamps. Furthermore, the CNPU timestamps obtained in this way are not as consistent as the real-time clock ticks on the QNPU. Therefore, the relative CNPU time compared to the QNPU time (on the same node) may fluctuate.

CNPU-QNPU communication latency

The latency of communication between the CNPU and QNPU can be calculated by looking at the time between CNPU events and QNPU events. However, since the CNPU timestamps are fluctuating compared to the QNPU timestamps, we cannot use a direct comparison between CNPU and QNPU timestamps. Instead, we look at time differences on the CNPU and compare them to time differences on the QNPU, given that we know the order in which events occur during the DQC application execution. Figure 4.16 shows a schematic overview of events happening on the CNPU and the QNPU during a single execution of the DQC circuit. By comparing, e.g., (1) the time difference on the CNPU between sending subroutine S1 and receiving its result with (2) the time difference on the QNPU between receiving subroutine S1 and finishing it, we can estimate the total latency of sending S1 from CNPU to QNPU and receiving its result. Using this technique, we can estimate the latencies for each communication between CNPU and QNPU, as listed in Table 4.4. Again, since the CNPU timestamps fluctuate compared to the QNPU timestamps, the derived latencies fluctuate and can even be negative. However, for all derived latencies, we found that a constant function best fit the data. This verifies that the actual latency is constant as expected, and that the variance is due to the inaccuracy of CNPU timestamps.

Using the result from Table 4.4, we can compute bounds on the four individual latency variables of the server (we have a system of three linear equations, and we know that all

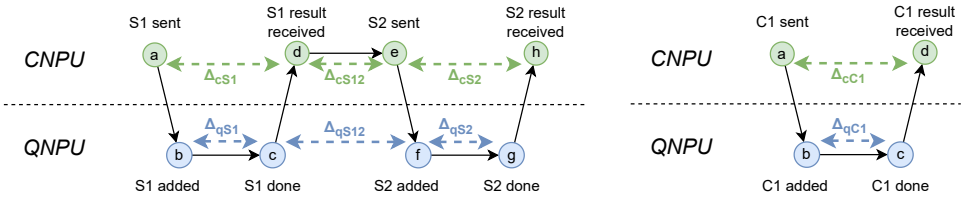


Figure 4.16: Schematic of events happening on the CNPU and QNPU during a single execution of DQC on the server (a) and the client (b). Time flows to the right. The Δ variables are the time differences between events, and are used to estimate CNPU-QNPU communication latencies ($a \longrightarrow b$, $c \longrightarrow d$, $e \longrightarrow f$, $g \longrightarrow h$ on the server and $a \longrightarrow b$, $c \longrightarrow d$ on the client).

4

latencies must be strictly non-negative):

- Sending S1 from CNPU to QNPU: < 0.242 ms.
- Receiving S1 result on CNPU from QNPU: between 0.142 and 0.384 ms.
- Sending S2 from CNPU to QNPU: between 0.225 and 0.467 ms.
- Receiving S2 result on CNPU from QNPU: < 0.242 ms.

In the latency breakdown of the server qubit memory time (see Section 4.11.6) we are only interested in the latencies that happen during the time that the server qubit is in memory. For the server these are the latencies for receiving the S1 result and sending S2. The sum of these two latencies is $\Delta_{qS12} - \Delta_{cS12} = 0.609$ ms (see Table 4.4). For simplicity, we say that both latencies constitute half of this time, as mentioned in the caption of Figure 4.15. Similarly, for the client we are only interested in the latency of receiving the C1 result. For simplicity we take this latency to be the same as that of sending C1, i.e. we use half of $\Delta_{cC1} - \Delta_{qC1}$.

Entanglement generation

An overview of all values discussed in this section is given in Table 4.5.

EPR generation happens by attempting entanglement repeatedly until success. The QNPU sends an `ENT` physical instruction (Table 4.1) to the QDevice, which starts a batch of physical attempts. Each attempt takes 3.95 μs and a batch contains 500 attempts. If a batch fails (no success after 500 attempts), the QNPU sends another `ENT` instruction. Table 4.5 lists the average success probability per attempt and per batch that we found in the DQC experiments. As explained in Section 4.10.1, the NV QDevice creates either a Ψ^+ or a Ψ^- state. Table 4.5 shows statistics on how often each of these states was created during our experiments.

Figure 4.17 shows the distribution of time it takes to generate an EPR pair in the DQC experiment, where the average duration of such is 439 ms. This is the duration between starting the network process and finishing it, which includes entanglement attempts until success on the QDevice and subsequent Bell state corrections to Φ^+ (see Section 4.10.1). This duration corresponds to a fitted rate of 2.28(3) created EPR pairs per second. If only

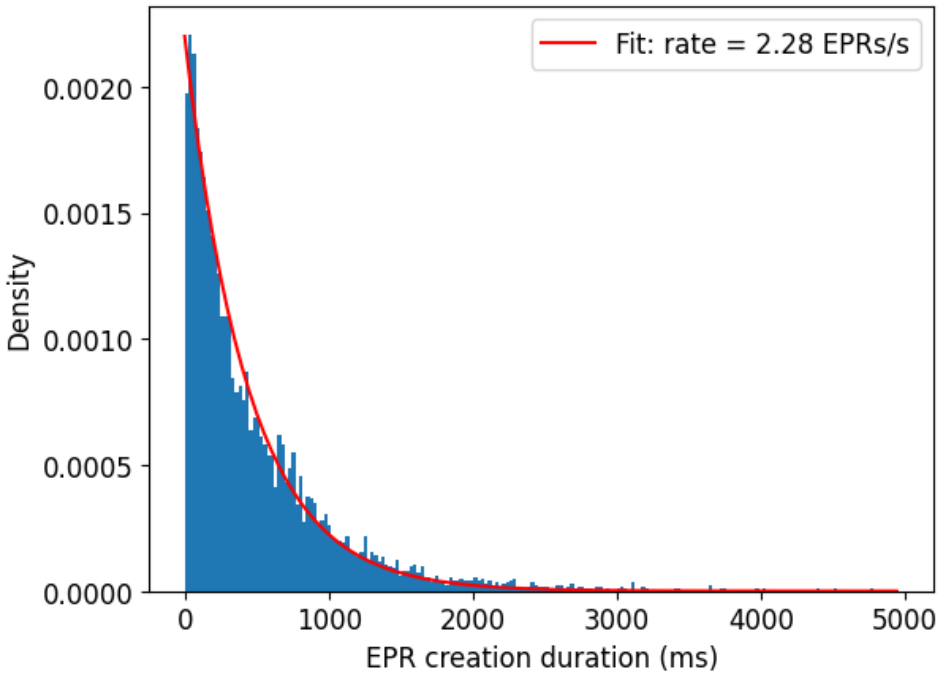


Figure 4.17: Histogram of EPR generation durations (time from first attempt until success) based on all EPR generations in the DQC experiment (using only latency-filtered data points, see Section 4.11.3). The histogram shows which fraction of all durations were in a particular duration window (window width: 25 ms). Expected EPR generation duration follows an exponential decay, with a rate parameter of 2.28(3) successes (EPR pairs) per second.

the QDevice entanglement generation is considered (i.e. without Bell state corrections and without QNPU processing overhead), this rate is 2.37(2) EPR pairs per second.

Local gate durations

As part of the DQC execution, the QNPU sends physical instructions to the NV QDevice for executing local quantum gates. In Table 4.6 we report on the observed durations of these gates from the perspective of the QNPU: these durations are from the time the physical instruction is sent to the QDevice until the corresponding result is received from the QDevice. We note that these durations are longer than these gates would take if they were executed directly on the QDevice (without QNodeOS, see Table 4.3) because of two reasons: (1) the limited granularity with which the QNPU and QDevice communicate (rounds of 10 μs) and (2) the fact the QDevice interleaves DD sequences in between sequences for the physical instruction itself, as explained in Section 4.6.

General experiment statistics

Table 4.7 lists statistics about the overall DQC experiment (all 7200 DQC circuit executions combined). We confirm our hypothesis that the overwhelming fraction of time is spent

Parameter	Value
Duration of a single entanglement attempt*	3.95 μs
Number of attempts per batch*	500
Average number of failed batches until success	144
Average success probability per batch	6.95×10^{-3}
Average success probability per attempt	1.39×10^{-5}
Number of Psi+ states generation	3187 (44.3%)
Number of Psi- states generation	4013 (55.7%)
EPR generation rate (fit) (QDevice)	2.37(2) EPRs/s
EPR generation rate (fit) (QNodeOS)	2.28(3) EPRs/s
Average fraction of EPR generation time spent on sync failure	0.18

Table 4.5: Overview of values derived from the DQC experiment analysis, based on all 7200 DQC circuit executions. Entries with an asterisk (*) are values that we fixed in our experiments. The other values are observed experimental results. Average success probabilities are derived from the number of failed batches until success. EPR generation rate is distinguished between QDevice and QNodeOS. For the QDevice, it indicates the fitted (to an exponential decay function) time between the first ENT physical instruction and the first entanglement success (see Section 4.9.6). For QNodeOS, it indicates the fitted time between the start of the network process and the end of the network process (i.e. when entanglement has been created and Bell state corrections have been applied, see Section 4.10.1). Entanglement sync failures happen when one QDevice (server or client) wants to attempt entanglement but the other QDevice is not ready (Section 4.9.6). Such sync failures were observed intermittently during a batch of entanglement attempts.

Physical instruction	Duration (client)	Duration (server)
Measure	130–160 μs	80 - 100 μs
X90	80–100 μs	50 - 130 μs
X180	80–100 μs	10 - 130 μs
-X90	—	50 - 130 μs
Y90	70–200 μs	50 - 130 μs
Y90	—	50 - 130 μs

Table 4.6: Duration of executing local quantum gates on the NV QDevice in the DQC experiment. Durations are from sending the physical instruction from QNPU to QDevice until receiving the QDevice response. The -X90 and Y90 gates were never executed in the client DQC program.

on the network process, namely generating EPR pairs. We also see that as expected, the server spends more time on user processes than the client does, since it does more local gates than the client (namely, the gates in subroutine S2).

4.11.7 QNPU network process analysis

In this section we focus on the execution of the network process in the QNPU as observed in the execution of DQC. The ER sockets (Section 4.9.3) are designed to facilitate the generation of entanglement belonging to a pair of user processes between two different QNPUs. In particular, the ER socket allows the QNPU to proceed with entanglement generation, while only one node may not have issued a request for entanglement yet.

During execution of the DQC application, the client QNPU has a single user process P_c for its DQC program and the server QNPU has a single user process P_s for its DQC

Value	Client	Server
Total experiment duration	4243 s	4065 s
Time spent executing network process	3840 s	3825 s
Time spent executing user processes	5.041 s	7.618 s

Table 4.7: Overall durations of the DQC experiment.

program. Both user processes realize the repeated execution of subroutines that jointly realize the DQC circuit (Figure 4.3a).

In each single repetition of the DQC circuit, P_s executes first S1 and then S2, and P_c executes C1. P_s (in S1) and P_c (in C1) execute a NetQASM instruction for creating an entangled pair, which results in an entanglement request that is submitted to the network stack. Then, P_c and P_s go into the waiting state (see Section 4.9.3) until the entangled pair is delivered by the network process.

P_c executes a `create_epr` instruction and P_s executes a `recv_epr` instruction (determined by program source code, see Appendix B. Therefore, the client is seen as the *initiator* (see Section 4.9.3). P_s and P_c open a pair of ER sockets with each other when they start and keep it open for the whole experiment. P_c and P_s, being on different nodes, operate independently, and may hit their entanglement request instruction at different times. Since the client is the initiator and the server the receiver, the server is always willing to handle an entanglement request with the client. So, the network stack on both client and server will handle a request for entanglement as soon as the client submitted it to its network stack, regardless of whether the server already executed the corresponding `recv_epr` in S1.

We observe that in 3245 out of all 7200 DQC circuit executions, the client submitted the corresponding entanglement request to its network stack (in C1) *before* the server submitted its entanglement request to its own network stack (in S1), but where the server still complied by starting the network process and handling the request.

Client waits for server

From our architecture, we expect that it can happen that the client must wait for the server. This can be the case in the following scenario: The client executes C1 for DQC circuit iteration i and submits the entanglement request. Then, the next network time bin starts and the client QNPU starts the network process. However, the server is at this time (the beginning of the time bin) still busy with executing S2 for iteration $i-1$ (in user process P_s). Therefore the server QNPU cannot yet activate its own network process. Since the ER socket with the server is open and the client is the 'initiator', the client will send entanglement physical instructions to the QDevice anyway, but the QDevice will not be able to do actual attempts because the server QDevice is not ready (Section 4.9.6). Only when the server QNPU completes S2, it can activate the network process, which then sends entanglement physical instructions to the QDevice. Only at this point the QDevices can start actual entanglement generation. We observe that it did indeed happen that the client had to wait for the server, although we observed this behavior in only in 60 out of 7200 DQC circuit executions.

Parameter	Value
Number of times server puts EPR request to network stack before client	1774/7200
Number of times server starts entanglement before putting in EPR request	3245/7200
Number of times submitted EPR request is handled in immediate next time bin	5523/7200
Average number of bins that pass before request is handled	2.33
Number of times server needs to wait for client	1323/7200
Number of times client needs to wait for server	60/7200
Number of times client network process starts > 100 μs after time bin starts	0
Number of times server network process starts > 100 μs after time bin starts	13

Table 4.8: Statistics on the QNPU network process behavior during the whole DQC experiment, i.e. totalled over all 7200 DQC circuit iterations.

Server waits for client

We expect that it can also happen that the server must wait for the client. This can be the case in the following scenario: The server executes S1 for DQC circuit iteration i and submits the entanglement request. Then, the next network time bin starts. However, the client did not yet hit the entanglement request in C1 for DQC iteration i, so there is nothing to do for the server network process. The server hence needs to wait for the next time-bin, and check again if by now the client has submitted its entanglement request. We observe that in 1323 out of 7200 DQC circuit executions, the server had to wait for the client.

Start of network process

We examine the start of the network process in relation to the start of a time bin. In particular, the start of the network process may be delayed if there is still a user process running.

The network process is only activated at the beginning of a time bin. In our experiment, a time bin starts every 10 ms and lasts 10 ms. In most cases when the network process is activated, this activation happens very quickly after the time bin start (within 100 μs, as some QNPU software processing is needed). For the client QNPU, the network process never starts more than 100 μs after a time bin start. For the server, in 13 out of 7200 DQC circuit executions, the network process starts more than 100 μs after a time bin starts, since in these cases there was still a user process running. In Table 4.8, an overview of all network process statistics is given.

4.12 Multitasking experiments on NV

The multitasking evaluation was done in two parts:

- **Quantum tomography while multitasking**: Executing a single DQC application (on client and server) and a single Local Gate Tomography (LGT) application (on client only) where it was verified that the LGT application produces expected quantum results (see Section 4.12.2).
- **Scaling the number of applications**: Executing N DQC applications and N LGT applications, where the classical device utilization metric was compared with a version of QNodeOS without multitasking, and where we investigated the behavior of the QNPU scheduler on the client in the context of multiple programs (see Section 4.12.3).

The network schedule was set as in the previous DQC experiment for direct comparison.

4.12.1 Mocked entanglement

For the multitasking evaluation, we focused on the behavior of QNodeOS, and opted not to use the standard entanglement generation procedure in our NV QDevices as done in the DQC experiments (Section 4.11) to allow for a simpler experiment. Instead, we used a mocked entanglement generation process on the QDevices (executing entanglement actions without entanglement): Weak-coherent pulses on resonance with the NV transitions, that follow the regular optical path, are employed to trigger the CPLD in the entanglement heralding time-window.

We stress that in our multitasking experiments, the exact same physical instructions are sent to the QDevice as would be done when using real entanglement, and the exact same responses are sent back. Therefore, QNodeOS needed to perform the same operations (including scheduling decisions) as it would have needed to do with real entanglement. Furthermore, we aimed to keep the rate of entanglement 'success' in the QDevices the same order of magnitude as that of the DQC experiments (10.14 EPRs/s compared to 2.37 EPRs/s in the DQC experiment) by keeping the mean-photon number of the weak-coherent pulse comparable to p_C and p_S (in the order of ~ 10^{-4}).

4.12.2 Tomography results

We perform tomography when not multi-tasking, in order to verify our expectation that multi-tasking should not affect the quantum performance of LGT: The tomography results of the LGT application in the multitasking scenario are given in Figure 4.4c. We also ran the same LGT application on the client in a non-multitasking scenario. In this case, the client ran the LGT application and there was no DQC application run at all (the server did nothing). The tomography results of LGT for the non-multitasking scenario are given in Figure 4.18. The results are slightly different since the multitasking experiment was done on a different day than the non-multitasking experiment. However, within error bars we verify that multitasking does not affect the quantum performance of the LGT application.

4.12.3 Scaling to more than two applications

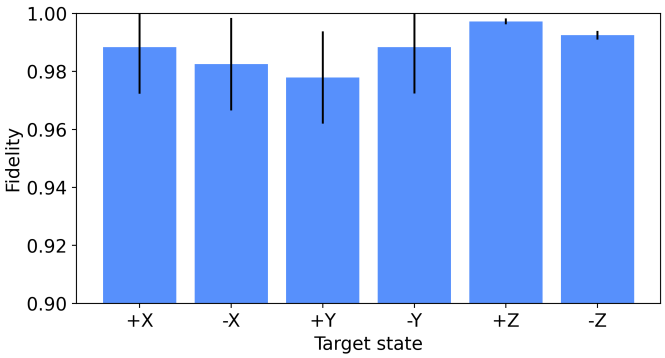


Figure 4.18: Local Gate Tomography results on the client node in a non-multitasking scenario.

QNPU processes and steps

For the scaling evaluation, we did an experiment for each $N \in \{1,2,3,4,5\}$. For each experiment, the client CNPU started N DQC-client programs and N LGT programs concurrently (pseudocode in Appendix B), and the server CNPU started N DQC-server programs. In this section we discuss the observed behavior of the client and server QNPUs during these experiments. The client QNPU has $2N$ user processes (N DQC user processes and N LGT user processes), each of which continuously receives quantum blocks in the form of NetQASM subroutines (C1 for DQC processes and L1 for LGT processes). These $2N$ user processes and the single client network process are scheduled by the client QNPU scheduler. The server has N user processes (all for DQC) which are scheduled together with the server network process by the server QNPU scheduler. Figure 4.19 shows a schematic diagram of the nominal (most often occurring) pattern of scheduling.

In both S1 and C1, there is a single `create_epr` NetQASM instruction (see [26] and Chapter 3) for creating entanglement with the other node, followed by a `wait_all` NetQASM instruction that waits until the request entangled qubit is delivered. The `create_epr` instruction is handled by the QNPU processor by sending the entanglement request to the network stack. Upon executing the `wait_all` instruction, the user process executing this subroutine (S1 or C1) goes into the waiting state (green stop sign in Figure 4.19). When the network process completes (having created the entangled qubit), the user process can be resumed, finishing the subroutine (C1 or S1).

On the server QNPU, for each DQC user process U the following sequence is repeated:

- U is in the *idle* state;
- NetQASM subroutine S1 is submitted by the CNPU to the QNPU, moving U to *ready*;
- U is activated; S1 is executed until it hits the `wait_all` instruction; U goes into the *waiting* state;
- The network process handles the entanglement request for S1 until EPR creation succeeds; U goes into *ready* again;
- U is activated; S1 is executed until completion; U goes to *idle*;

- NetQASM subroutine S2 is submitted by the CNPU; U goes to *ready*;
- U is activated; S2 is executed until completion; U goes to *idle*.

The above sequence is for one execution of the DQC circuit (Figure 4.3a), and is hence repeated many times.

On the client QNPU, for each DQC user process U the following sequence is repeated:

- U is in the *idle* state;
- NetQASM subroutine C1 is submitted by the CNPU, moving U to *ready*;
- U is activated; C1 is executed until it hits the `wait_all` instruction; U goes into the *waiting* state;
- the network process handles the entanglement request for C1 until EPR creation succeeds; U goes into *ready* again;
- U is activated; C1 is executed until completion; U goes to *idle*.

The above sequence is for one execution of the DQC circuit (Figure 4.3a), and is hence repeated many times.

On the client QNPU, for each LGT user process U the following sequence is repeated:

- U is in the *idle* state;
- NetQASM subroutine L1 is submitted by the CNPU, moving U to *ready*;
- U is activated; L1 is executed until completion; U goes to *idle*.

The above sequence is for one execution of the LGT circuit (Figure 4.4a), and is hence repeated many times.

For the above sequences for user processes, only the internal order is fixed; the time in between steps depends on the QNPU scheduler, as well as the time at which the CNPU submits subroutines. Furthermore, since there are multiple user processes at the same time (for the server, only for $N > 1$), the above steps happen for each user process U_i and the steps are interleaved. Figures 4.19 to 4.21 show examples of how these user processes can be interleaved on both client and server QNPU.

DQC and LGT interleaving

We investigate the degree of interleaving the execution of DQC and LGT, in particular how many LGT subroutines are executed when a DQC process is waiting: The client QNPU executes both DQC and LGT user processes. DQC user processes are often in the waiting state. This happens when their C1 subroutine is suspended, waiting for the network process to handle their entanglement request. The network process is only activated at the beginning of a time bin, which happens only every 10 ms, or when a user process finishes executing a subroutine, the latter not occurring very frequently for low number of programs N. Furthermore, DQC user processes can be in the idle state, namely when they completed execution of C1 for some iteration i of the DQC circuit, but are still waiting for the CNPU to send C1 for iteration $i+1$. In both these types of 'gaps' (waiting or idle), LGT subroutines can be executed (each taking $\approx$2.4 ms). Table 4.9 lists the maximum number of consecutive LGT subroutines that were executed in between DQC subroutines for both types of gaps.

Subroutine (Quantum block) execution order
We investigate whether the QNPU schedules quantum subroutines in a different order than they arrived from the CNPU. As expected, we find that this is the case. Although the QNPU handles subroutines from the CNPU first-come-first-served, some of these subroutines (in our experiments, precisely the DQC subroutines that wait for entanglement) are put into the waiting state. This allows the QNPU to schedule other subroutines (in our experiments, we observe LGT subroutines being executed), even if they arrived later from the CNPU than the waiting DQC subroutine. Schematic overviews of such scheduling that we observed are depicted in Section 4.13.

User process idle times
We examine the number of times, and the duration, that a user process is idle waiting for submission of a subroutine from the CNPU as a function of N: A user process is *idle* when there are currently no subroutines associated with the process pending to be executed. This means that the QNPU waits, at least for this user process, until the CNPU sends the next subroutine for the user process. Table 4.9 lists the number of times and durations of moments at which all client QNPU user processes are idle. This number and their durations decrease for larger values of N. This is expected since there are more active processes, and hence more subroutines being sent from the CNPU for different processes. In most cases, when finishing a subroutine for user process U, there is then another user process U' already waiting with another subroutine to execute.

Network process start delays
We examine the scheduling behavior of the network process in relation to user processes. We expect that due to the fact we use a non-preemptive scheduler, a network process may not be activated at the start of a network time bin, due to a user process still being executed. We investigate the occurrence of such events in our multi-tasking experiment, including the delay with which the network process is started in such a scenario (see Table 4.9): When a user process submits and entanglement request to the network stack, this request is handled at the earliest when the network process is activated. This happens either at the start of the next network time bin, or when a user process finishes a subroutine. Therefore, there is often some time in between submitting the request and the network process handling it. This waiting time is in most cases bounded by 10 ms, since that is the length of a time bin, and all time bins are assigned to networking in our experiment. However, in some cases the client may still be executing a LGT subroutine when a new time bin starts, delaying the start of the network process until this subroutine has finished. We expect however that in all cases, as soon as such an LGT subroutine finishes, the QNPU scheduler then immediately schedules the network process, and not another LGT subroutine. We found that the maximum difference between time bin start and network process start is 2.59 ms, which verifies that indeed at most one LGT subroutine is sometimes executed during a time bin start (LGT subroutine execution duration being ≈2.4 ms.)

We remark that with increasing N, the network process is delayed more frequently by a LGT subroutine. This is expected due to the fact more subroutines from different user processes await execution. Consequently, with increasing N it also happens more frequently that the client and server do not start execution of the network process in the same time-bin (see below).

Client waits for server and vice versa

In order to better understand the concurrent execution of multiple applications (here DQC and LGT) and corresponding programs, we investigate scenarios and times in which the client waits for the server (or vice versa).

The client and server open an ER socket at the beginning of each DQC application. So, during runtime, there are N ER sockets opened on the server QNPU (one for each DQC process) and N ER sockets opened on the client QNPU (one for each DQC process). In each DQC application, the client QNPU user process for that DQC application is the 'initiator' (see Section 4.9.3). This means that as soon as the client user process submits a request for entanglement (from within C1), both server and client QNPU start their network process to handle it (at the start of the next time bin, and provided the network process should not first handle a request from a user process from another DQC application).

It can happen that the client QNPU and server QNPU do not start their network process at the same time bin. This mostly happens when one of the nodes is still busy executing a user process subroutine when a time bin starts, as explained above. If this happens, the QNPU that did already start their network process sends entanglement instructions to their QDevice, but this will not result in physical entanglement attempts since the other QDevice is not available (leading to a entanglement sync failure, see Section 4.9.6). Table 4.9 lists the number of times that this happened.

For each of the N DQC applications that are running on client and server, and for each execution of the DQC circuit in those applications, there is a single entanglement request from the client (in C1) and a single entanglement request from the server (in S1). For each of these request pairs, the client at some point starts the network process and handles this request, and the server at some point starts the network process and handles its corresponding request. For each such pair of requests, the following scenarios can happen:

1. Client and server QNPU start their network process in the same time-bin (one of them may start a bit later than the start of the time-bin because it needs to complete a quantum subroutine).
2. The client starts its network process in time-bin k but the server starts it at some time-bin $> k$. This happens when the server still has a qubit in memory when time-bin k starts. Therefore, the server cannot activate its network process yet. A qubit still being in memory happens when the server QNPU has executed S1 for some DQC process (which produced an entangled qubit) but has not yet executed S2 (in which the qubit is measured and hence freed).
3. The server starts its network process in time-bin k but the client starts it at time-bin $k+1$. This happens (although rarely) when the client user process puts the entanglement request to the network stack just before the start of k. The server will immediately start attempts at k, but the client itself is still processing and 'misses' k; the client then only starts at time-bin $k+1$.

Table 4.9 lists how often the above scenarios happen for each N.

Parameter	N = 1	N = 2	N = 3	N = 4	N = 5
average no. LGT subroutines in between any DQC subroutines	0.83	1.42	1.59	1.65	1.65
max no. consecutive LGT subroutines in between DQC subroutines	3	4	6	7	8
max no. consecutive LGT subroutines when a DQC is in waiting state	2	3	4	4	4
% of times that ≥ 1 LGT subroutines fills time waiting for time bin	56	81	99	99	100
no. times that network process is delayed by a LGT subroutine	88/360	212/720	554/1080	940/1440	1170/1800
no. time windows in which all user processes are idle	399	56	4	1	0
Average length of idle time window (ms)	10	5.9	5.9	8.3	—
Maximum length of idle time window (ms)	152	31	15	8.3	—
% client and server start network process at same time bin	95.8	58.2	42.1	42.4	38.5
% server started network process 1 time bin after client	0.8	37.9	50.1	50.9	53.1
% server started network process > 1 time bins after client	0.0	3.3	7.7	6.6	8.4
% client started network process 1 time bin after server	3.3	0.6	0.1	0.1	0.0

Table 4.9: Overview of values derived from the multitasking experiments in which N DQC applications (on client and server) and N LGT applications (client only) were executed concurrently, for $N \in \{1, 2, 3, 4, 5\}$.

4.13 Multitasking scheduling patterns

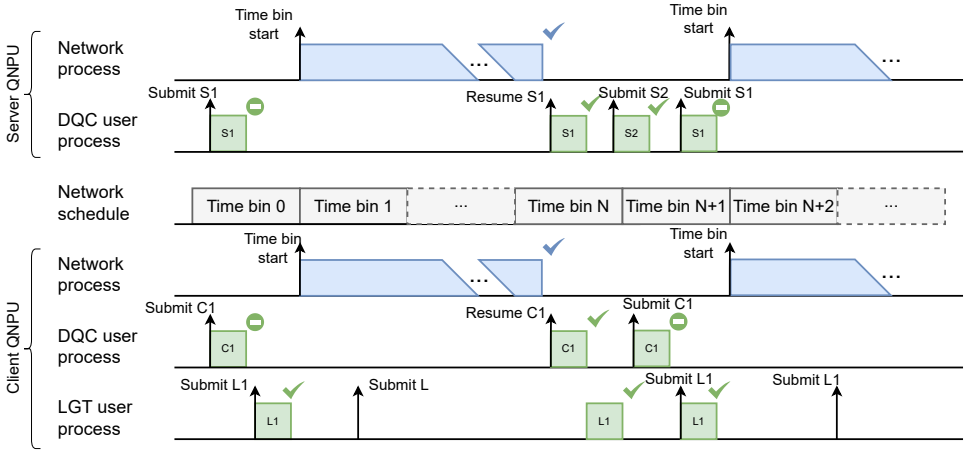


Figure 4.19: Nominal scheduling pattern on the client and server QNPUs when multitasking 1 DQC application (on client and server) and 1 LGT application (on client only). Pictured is a slice of time (moving to the right) in which a whole DQC circuit execution is realized, and 3 LGT circuit executions. Up-arrows indicate that the process becomes *ready* (either since a subroutine was submitted from the CNPU, or because a requested entangled qubit becomes available). Green blocks are NetQASM subroutines. Blue blocks are entanglement generation. Ticks indicate completion of a subroutine (user process) or entanglement request (network process). Stop sign means the user process goes into the *waiting* state. Time not to scale. Time bin length is 10 ms. Duration of L1 is ≈2.4 ms. Duration of entanglement generation is non-deterministic. On the server QNPU the following happens. S1 arrives from CNPU; DQC user process becomes *ready*. DQC user process is activated and executes S1. The entanglement instruction inside S1 is reached; entanglement request is sent to network stack; DQC user process becomes *waiting*. When time bin 1 starts, network process becomes *ready*. There is a pending entanglement request, so network process is activated; QDevice attempts entanglement until success (after non-deterministic number of time bins, blue tick). Requested entangled qubit is available: DQC user process becomes *ready* again; is activated; executes S1 until completion; becomes *idle*. QNPU receives subroutine S2 from CNPU; activates DQC user process; executes S2 until completion. At this point, the QNPU completed execution of the current repetition of the DQC circuit. QNPU then receives again a subroutine S1 (for the next DQC circuit iteration), and the same pattern repeats. On the client QNPU the following happens. C1 arrives from CNPU; DQC user process becomes *ready*. DQC user process is activated and executes C1. The entanglement instruction inside C1 is reached; entanglement request is sent to network stack; DQC user process becomes *waiting*. L1 arrives from CNPU; LGT user process becomes *ready*. LGT user process is activated; fully executes L1. When time bin 1 starts, network process becomes *ready*. There is a pending entanglement request, so network process is activated; QDevice attempts entanglement until success (blue tick). While network process is active, another L1 block arrives from CNPU (for next LGT circuit iteration) so LGT user process becomes *ready*. LGT user process is not activated since network process is still running. Upon entanglement success, requested qubit is available; DQC user process is activated to complete C1. QNPU has now completed execution of the current repetition of the DQC circuit. LGT user process is activated to execute L1 which was still pending. The same pattern repeats.

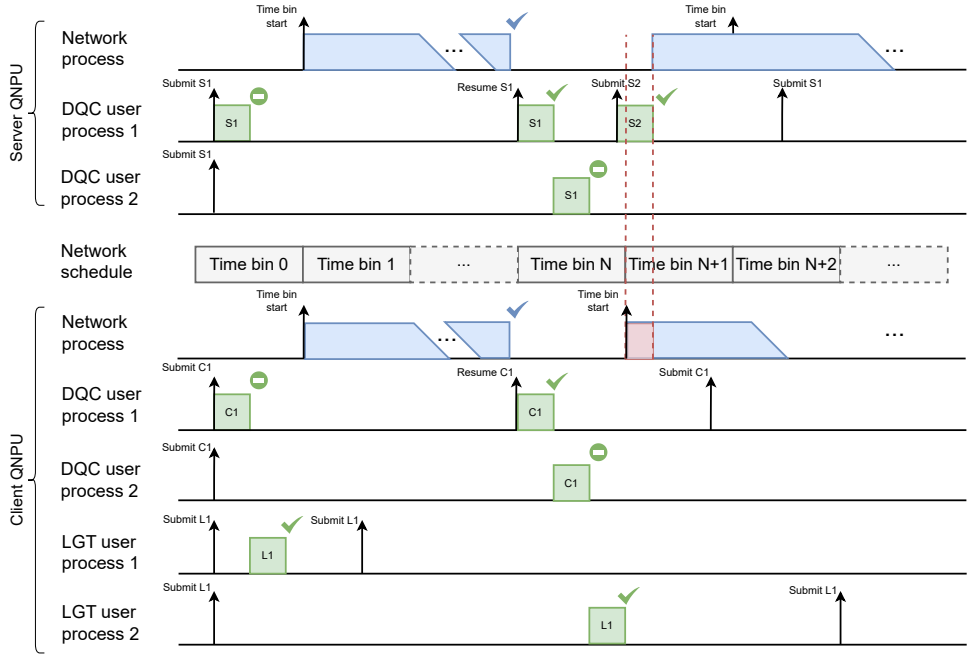


Figure 4.20: Example scheduling pattern of scenario with 2 DQC applications and 2 LGT applications (the symbol and color coding is the same as in Figure 4.19). In this case, the client needs to wait (red shaded area) for the server to finish S2 of DQC user process 1, before they can do entanglement generation for DQC user process 2. Scenario: 2 DQC applications (A1 and A2) are concurrently executed (A1: DQC-server program executed by server DQC user process 1 and DQC-client program executed by client DQC user process 1; A2: DQC-server program executed by server DQC user process 2 and DQC-client program executed by client DQC user process 2). Client and server successfully create entanglement for some DQC circuit execution i for A1 (just before time bin N starts). Client finishes C1 for user process 1, and meanwhile the server finishes S1 for user process 1. The client has completed its part of DQC circuit execution i for A1, but the server still needs to wait for S2 from the CNPU. Then, the client executes C1 for user process 2, which is the start of circuit execution j for A2; user process 2 becomes waiting. Meanwhile the server executes S1 for user process 2 which becomes waiting. The client needs to wait until the start of the next time bin ($N+1$) until it can activate the network process to handle the request. In the meantime, it can execute an L1 block. Time bin $N+1$ starts and the client handles the request. However, the server has received S2 for execution i of A1, and starts executing it just before the time bin starts. Only after finishing it, the server can start the network process, which picks up the request for A2. While S2 is executing, the client QDevice tries to do entanglement attempts, but gets entanglement sync failures (Section 4.9.6) since the server QDevice is busy with S2.

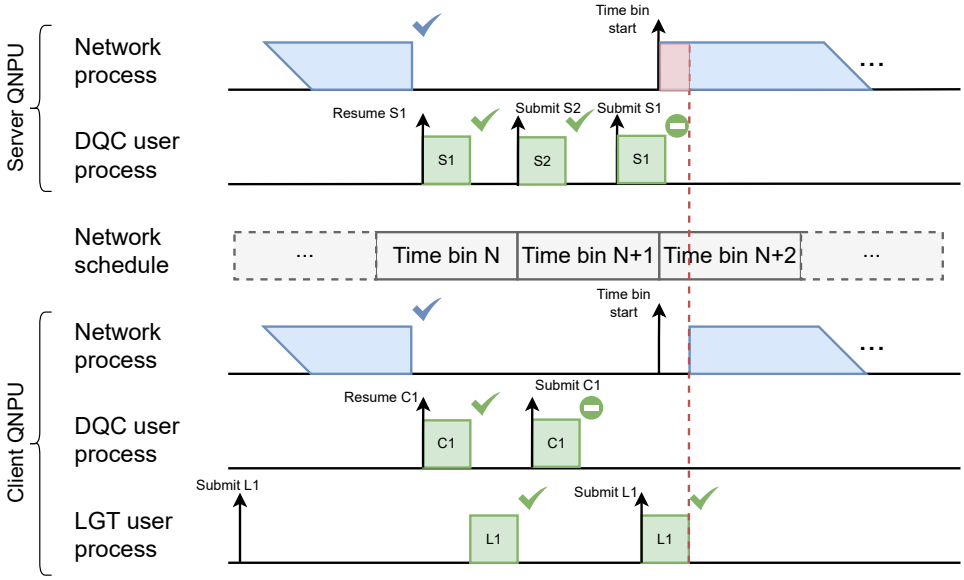


Figure 4.21: Example scheduling pattern of multitasking one DQC application (on client and server) and one LGT application (on client only), where the server must wait for client to finish its LGT user process (red area); the symbol and color coding is the same as in Figure 4.19. At the start of time bin $N+1$, the server activates the network process to handle the request that was put by the previous S1 execution. However the client only starts some time later during the time bin, since it first needs to finish executing L1 for the LGT user process.

4.14 Traces

In our NV experiments, the CNPU, QNPU and QDevice, on both client and server nodes, trace (i.e. record the timestamps of) events happening on their system. The events that are traced on the CNPU and QNPU are listed in Tables 4.10 and 4.11, respectively. The NV QDevice separately records messages received (physical instructions from the QNPU, see Table 4.1) and responses sent back to the QNPU(see Table 4.2).

Figure 4.22 shows a full-stack trace slice of a single execution of the DQC circuit. This particular sequence of events started at offset 60460 ms from the start of the experiment. The following events (among others) can be seen:

- At ≈ 60470 ms: client CNPU sends subroutine C1 to the QNPU; it is received slightly after on the QNPU (`PROCMGR_SUBROUTINE_ADDED_P0`).
- Slightly after 60470 ms: the QNPU starts the user process containing C1; it hits the entanglement instruction and moves the process to the waiting state (`PROCESSOR_WAIT_USER_PROCESS`).
- At 60480 ms: the first next time bin starts, starting the network process on both client and server. This results in `ENTANGLE` commands being sent to the QDevices by both client and server.
- Between 60480 and 60550 ms: the two QDevices repeatedly attempt entanglement but fail (each `ENTANGLE` instruction from the QNPU starts one batch; each `ENTANGLEMENT_FAILURE` return message indicates the batch failed).
- Meanwhile at 60485 ms, the server CNPU sends subroutine S1 to the QNPU.
- At ≈ 60552.5 ms, the QDevices succeed in entanglement generation, producing a $|\Psi^+\rangle$ Bell pair.
- After this, the client and server finish C1 and S1, respectively. The client sends instructions for local gates ending with a `MEASURE` physical instruction. The server starts S1, hits the `recv_epr` instruction, goes into the waiting state, gets immediately unblocked (since the entangled pair was already created) and sends a bell state correction gate to the QDevice (`X180`).
- At ≈ 60553.5 ms, the client CNPU receives the result of C1 (`RESULT_RCVD`), and sends the classical message δ to the server CNPU (`CLAS_MSG_SENT`).
- At ≈ 60554 ms, the server CNPU receives δ (`CLAS_MSG_RCVD`).
- At ≈ 60557 ms, the server CNPU sends S2 to the QNPU. The QNPU executes the user process containing S2 which involves sending local quantum instructions to the QDevice ending with a measurement.
- At ≈ 60558 ms, the QNPU sends the result of S2 to the CNPU.

Event name	Description
SUBROUTINE_SEND_ATTEMPT	Try to send subroutine to QNPU
SUBROUTINE_SENT	Subroutine sent to QNPU
RESULT_RCVD	Subroutine results received from QNPU
CLAS_MSG_SENT	Classical message sent to other node
CLAS_MSG_RCVD	Classical message received from other node

Table 4.10: CNPU events that are traced (recorded with their timestamps) during application execution.

Event name	Description
SCHEDULER_ARRIVE_USER_PROCESS	A user process goes to the Ready state
SCHEDULER_SCHEDULE_USER_PROCESS	A user process goes to the Running state
SCHEDULER_ARRIVE_NET_PROCESS	Network process goes to the Ready state
SCHEDULER_SCHEDULE_NET_PROCESS	Network process goes to the Running state
PROCMGR_SUBROUTINE_ADDED_P<i>	New subroutine received from CNPU for process <i>
PROCMGR_SUBROUTINE_DONE_P<i>	A subroutine for process <i> finished execution
PROCESSOR_START_USER_PROCESS	Processor starts or resumes executing a user process
PROCESSOR_WAIT_USER_PROCESS	Processor suspends a user process and puts it in the Waiting state
PROCESSOR_FINISH_USER_PROCESS	Processor stops executing a user process
PROCESSOR_START_NET_PROCESS	Processor starts or resumes executing the network process
PROCESSOR_FINISH_NET_PROCESS	Processor stops executing the network process
QDEVICE_PRODUCE_<cmd>_CMD	Processor prepares <cmd> command for the QDevice
QDEVICE_CONSUME_CMD	QDevice reads the next command from the QNPU
QDEVICE_PRODUCE_OUTCOME	QDevice sends result to the QNPU
PROCESSOR_CONSUME_OUTCOME	Processor reads QDevice result
QNETWORK_ENT_PULL	Network stack pulls instruction from the EGP
EGP_NEI_OK	QEGP notifies that EPR pair has been created

Table 4.11: QNPU events that are traced (recorded with their timestamps) during application execution. <i> can be any number from 0 to 9 ('subroutine added' and 'subroutine done' events are not traced for processes with ID 10 or larger). <cmd> can be any physical instruction.

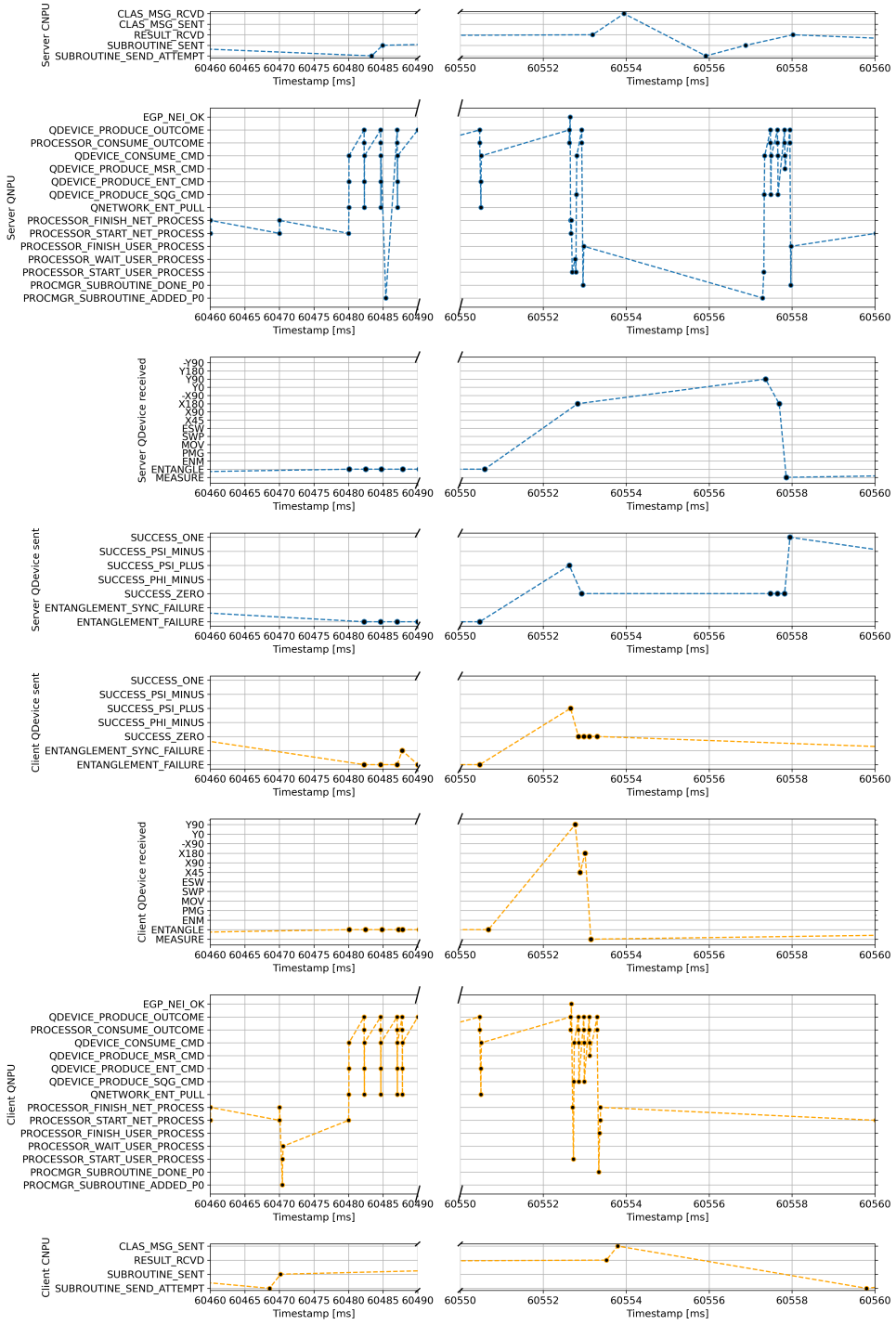


Figure 4.22: Full-stack event trace for one particular execution of the DQC circuit. Between timestamps 60490 and 60550 are more entanglement attempts which are cut out for the sake of clarity.

References

[1] M. H. Abobeih, J. Cramer, M. A. Bakker, N. Kalb, M. Markham, D. J. Twitchen, and T. H. Taminiau. "One-second Coherence for a Single Electron Spin Coupled to a Multi-qubit Nuclear-spin Environment". In: *Nature Commun.* 9.1 (2018), pp. 1–8. DOI: 10.1038/s41467-018-04916-z.

[2] *Aliro Security, Advanced Secure Networking*. URL: https://www.aliroquantum.com (visited on Oct. 30, 2024).

[3] K. Azuma, S. E. Economou, D. Elkouss, P. Hilaire, L. Jiang, H.-K. Lo, and I. Tzitrin. "Quantum repeaters: From Quantum Networks to the Quantum Internet". In: *Rev. Mod. Phys.* 95 (4 Oct. 2023), pp. 045006-1–045006-66. DOI: 10.1103/RevModPhys.95.045006.

[4] S. Baier, C. E. Bradley, T. Middelburg, V. V. Dobrovitski, T. H. Taminiau, and R. Hanson. "Orbital and Spin Dynamics of Single Neutrally-Charged Nitrogen-Vacancy Centers in Diamond". In: *Phys. Rev. Lett.* 125 (19 Nov. 2020), pp. 193601-1–193601-6. DOI: 10.1103/PhysRevLett.125.193601.

[5] S. Barz, E. Kashefi, A. Broadbent, J. F. Fitzsimons, A. Zeilinger, and P. Walther. "Demonstration of Blind Quantum Computing". In: *Science* 335.6066 (2012), pp. 303–308. DOI: 10.1126/science.1214707.

[6] L. C. Bassett, F. J. Heremans, C. G. Yale, B. B. Buckley, and D. D. Awschalom. "Electrical Tuning of Single Nitrogen-Vacancy Center Optical Transitions Enhanced by Photoinduced Fields". In: *Phys. Rev. Lett.* 107 (26 Dec. 2011), pp. 266403-1–266403-5. DOI: 10.1103/PhysRevLett.107.266403.

[7] M. Ben-Or and A. Hassidim. "Fast Quantum Byzantine Agreement". In: *STOC*. ACM, 2005, pp. 481–485. DOI: 10.1145/1060590.1060662.

[8] 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. URL: https://www.karlin.mff.cuni.cz/~holub/soubory/BB84original.pdf.

[9] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. "Teleporting an Unknown Quantum State via Dual Classical and Einstein-Podolsky-Rosen Channels". In: *Phys. Rev. Lett.* 70.13 (1993), pp. 1895–1899. DOI: 10.1103/PhysRevLett.70.1895.

[10] K. Bertels, A. Sarkar, T. Hubregtsen, M. Serrao, A. A. Mouedenne, A. Yadav, A. Krol, I. Ashraf, and C. G. Almudever. "Quantum computer architecture toward full-stack quantum accelerators". In: *IEEE Transactions on Quantum Engineering* 1 (2020). Publisher: IEEE, pp. 1–17. DOI: 10.1109/TQE.2020.2981074.

[11] K. Bharti, A. Cervera-Lierta, T. H. Kyaw, T. Haug, S. Alperin-Lea, A. Anand, M. Degroote, H. Heimonen, J. S. Kottmann, T. Menke, W.-K. Mok, S. Sim, L.-C. Kwek, and A. Aspuru-Guzik. "Noisy intermediate-scale quantum algorithms". In: *Reviews of Modern Physics* 94.1 (Feb. 15, 2022). Publisher: American Physical Society, p. 015004. DOI: 10.1103/RevModPhys.94.015004. URL: https://link.aps.org/doi/10.1103/RevModPhys.94.015004 (visited on July 10, 2024).

[12] S. Bose, P. Knight, M. Plenio, and V. Vedral. "Proposal for teleportation of an atomic state via cavity decay". In: *Physical Review Letters* 83.24 (1999), p. 5158. DOI: 10.1103/PhysRevLett.83.5158.

[13] L. Botelho, A. Glos, A. Kundu, J. A. Miszczak, Ö. Salehi, and Z. Zimborás. "Error mitigation for variational quantum algorithms through mid-circuit measurements". In: *Physical Review A* 105.2 (2022). Publisher: APS, p. 022441. DOI: 10.1103/PhysRevA.105.022441.

[14] A. Broadbent, J. Fitzsimons, and E. Kashefi. "Universal Blind Quantum Computation". In: *FOCS*. IEEE, 2009, pp. 517–526. DOI: 10.1109/FOCS.2009.36.

[15] C. Cabrillo, J. I. Cirac, P. Garcia-Fernandez, and P. Zoller. "Creation of entangled states of distant atoms by interference". In: *Physical Review A* 59.2 (1999), p. 1025. DOI: 10.1103/PhysRevA.59.1025.

[16] M. Caleffi, M. Amoretti, D. Ferrari, D. Cuomo, J. Illiano, A. Manzalini, and A. S. Cacciapuoti. "Distributed quantum computing: a survey". In: *arXiv preprint arXiv:2212.10609* (2022). DOI: 10.48550/arXiv.2212.10609.

[17] M. Castells. "The Impact of the Internet on Society: A Global Perspective". In: *Ch@nge: 19 Key Essays on How the Internet Is Changing Our Lives*. Madrid: BBVA, 2013.

[18] G. L. Chesson. "The network UNIX system". In: *ACM SIGOPS Operating Systems Review* 9.5 (1975). Publisher: ACM New York, NY, USA, pp. 60–66. DOI: 10.1145/1067629.806522.

[19] L. Childress and R. Hanson. "Diamond NV centers for quantum computing and quantum networks". In: *MRS bulletin* 38.2 (2013), pp. 134–138. DOI: 10.1557/mrs.2013.20.

[20] A. M. Childs. "Secure Assisted Quantum Computation". In: *Quantum Inf. Comput.* 5.6 (2005), pp. 456–466. DOI: 10.26421/QIC5.6-4.

[21] K. S. Chou, J. Z. Blumoff, C. S. Wang, P. C. Reinhold, C. J. Axline, Y. Y. Gao, L. Frunzio, M. Devoret, L. Jiang, and R. Schoelkopf. "Deterministic teleportation of a quantum gate between two logical qubits". In: *Nature* 561.7723 (2018). Publisher: Nature Publishing Group UK London, pp. 368–373. DOI: 10.1038/s41586-018-0470-y.

[22] T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J. Oliveira, M. Papendrecht, J. Rabbie, F. Rozpędek, M. Skrzypczyk, L. Wubben, W. de Jong, D. Podareanu, A. Torres-Knoop, D. Elkouss, and S. Wehner. "NetSquid, a NETwork Simulator for QUantum Information using Discrete events". In: *Communications Physics* 4.1 (2021), p. 164. DOI: 10.1038/s42005-021-00647-8.

[23] A. Corna. *Efficient Generation of Dynamic Pulses*. 2021. URL: https://www.zhinst.com/europe/en/blogs/efficient-generation-dynamic-pulses.

[24] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. "Open Quantum Assembly Language". 2017. arXiv: 1707.03429.

[25] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveira Filho, R. Hanson, and S. Wehner. "A Link Layer Protocol for Quantum Networks". In: *SIGCOMM*. ACM, 2019, pp. 159–173. DOI: 10.1145/3341302.3342070.

[26] 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". In: *Quantum Science and Technology* 7.3 (2022), p. 035023. DOI: 10.1088/2058-9565/ac753f.

[27] R. C. Daley and J. B. Dennis. "Virtual memory, processes, and sharing in Multics". In: *Communications of the ACM* 11.5 (1968). Publisher: ACM New York, NY, USA, pp. 306–312. DOI: 10.1145/363095.363139.

[28] G. De Lange, Z. Wang, D. Riste, V. Dobrovitski, and R. Hanson. "Universal dynamical decoupling of a single solid-state spin from a spin bath". In: *Science* 330.6000 (2010). Publisher: American Association for the Advancement of Science, pp. 60–63. DOI: 10.1126/science.1192739.

[29] J. B. Dennis. "Segmentation and the design of multiprogrammed computer systems". In: *Journal of the ACM (JACM)* 12.4 (1965). Publisher: ACM New York, NY, USA, pp. 589–602. DOI: 10.1145/321296.321310.

[30] J. B. Dennis and E. C. Van Horn. "Programming semantics for multiprogrammed computations". In: *Communications of the ACM* 9.3 (1966). Publisher: ACM New York, NY, USA, pp. 143–155. DOI: 10.1145/365230.365252.

[31] M. W. Doherty, N. B. Manson, P. Delaney, F. Jelezko, J. Wrachtrup, and L. C. Hollenberg. "The nitrogen-vacancy colour centre in diamond". In: *Physics Reports* 528 (2013). DOI: 10.1016/j.physrep.2013.02.001.

[33] P. Drmota, D. Nadlinger, D. Main, B. Nichol, E. Ainley, D. Leichtle, A. Mantri, E. Kashefi, R. Srinivas, G. Araneda, et al. "Verifiable blind quantum computing with trapped ions and single photons". In: *Physical Review Letters* 132.15 (2024). Publisher: APS, p. 150604. DOI: 10.1103/PhysRevLett.132.150604.

[34] P. Drmota, D. Main, D. Nadlinger, B. Nichol, M. Weber, E. Ainley, A. Agrawal, R. Srinivas, G. Araneda, C. Ballance, et al. "Robust quantum memory in a trapped-ion quantum network node". In: *Physical Review Letters* 130.9 (2023). Publisher: APS, p. 090803. DOI: 10.1103/PhysRevLett.130.090803.

[35] A. K. Ekert. "Quantum Cryptography Based on Bell's Theorem". In: *Phys. Rev. Lett.* 67.6 (1991), pp. 661–663. DOI: 10.1103/PhysRevLett.67.661.

[36] D. Fioretto. "Towards a flexible source for indistinguishable photons based on trapped ions and cavities". PhD thesis. University of Innsbruck, 2020.

[37] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. "eQASM: An Executable Quantum Instruction Set Architecture". In: *HPCA*. IEEE, 2019, pp. 224–237. DOI: 10.1109/HPCA.2019.00040.

[38] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. "An Experimental Microarchitecture for a Superconducting Quantum Processor". In: *MICRO*. ACM, 2017, pp. 813–825. DOI: 10.1145/3123939.3123952.

[39] E. Giortamis, F. Romão, N. Tornow, and P. Bhatotia. "QOS: A Quantum Operating System". In: *arXiv preprint arXiv:2406.19120* (2024). DOI: 10.48550/arXiv.2406.19120.

[40] P. A. Guérin, A. Feix, M. Araújo, and Č. Brukner. "Exponential communication complexity advantage from quantum superposition of the direction of communication". In: *Physical review letters* 117.10 (2016). Publisher: APS, p. 100502. DOI: 10.1103/PhysRevLett.117.100502.

[41] S. Hermans, M. Pompili, L. D. S. Martins, A. R. Montblanch, H. Beukers, S. Baier, J. Borregaard, and R. Hanson. "Entangling remote qubits using the single-photon protocol: an in-depth theoretical and experimental study". In: *New Journal of Physics* 25.1 (2023), p. 013011. DOI: 10.1088/1367-2630/acb004.

[42] S. Hermans, M. Pompili, H. Beukers, S. Baier, J. Borregaard, and R. Hanson. "Qubit teleportation between non-neighbouring nodes in a quantum network". In: *Nature* 605.7911 (2022), pp. 663–668. DOI: 10.1038/s41586-022-04697-y.

[43] J. Hofmann, M. Krug, N. Ortegel, L. Gérard, M. Weber, W. Rosenfeld, and H. Weinfurter. "Heralded Entanglement Between Widely Separated Atoms". In: *Science* 337.6090 (2012), pp. 72–75. DOI: 10.1126/science.1221856.

[44] P. C. Humphreys, N. Kalb, J. P. J. Morits, R. N. Schouten, R. F. L. Vermeulen, D. J. Twitchen, M. Markham, and R. Hanson. "Deterministic Delivery of Remote Entanglement on a Quantum Network". In: *Nature* 558.7709 (2018), pp. 268–273. DOI: 10.1038/s41586-018-0200-5.

[45] Infineon Technologies AG. *CY8CKIT-143A - Infineon Technologies*. en. Accessed: 2024-07-03. 2024. URL: https://www.infineon.com/cms/en/product/evaluation-boards/cy8ckit-143a/ (visited on July 3, 2024).

[46] Z. Instruments. *HDAWG 750 MHz Arbitrary Waveform Generator*. HDAWG 750 MHz Arbitrary Waveform Generator. 2019. URL: https://www.zhinst.com/europe/en/products/hdawg-arbitrary-waveform-generator.

[47] M. Iuliano, M.-C. Slater, A. J. Stolk, M. J. Weaver, T. Chakraborty, E. Loukiantchenko, G. C. d. Amaral, N. Alfasi, M. O. Sholkina, W. Tittel, et al. "Qubit teleportation between a memory-compatible photonic time-bin qubit and a solid-state quantum network node". In: *arXiv preprint arXiv:2403.18581* (2024). DOI: 10.48550/arXiv.2403.18581.

[48] B. Jing, X.-J. Wang, Y. Yu, P.-F. Sun, Y. Jiang, S.-J. Yang, W.-H. Jiang, X.-Y. Luo, J. Zhang, X. Jiang, et al. "Entanglement of three quantum memories via interference of three single photons". In: *Nature Photonics* 13.3 (2019), pp. 210–213. DOI: 10.1038/s41566-018-0342-x.

[49] N. Kalb, A. A. Reiserer, P. C. Humphreys, J. J. W. Bakermans, S. J. Kamerling, N. H. Nickerson, S. C. Benjamin, D. J. Twitchen, M. Markham, and R. Hanson. "Entanglement Distillation Between Solid-State Quantum Network Nodes". In: *Science* 356.6341 (2017), pp. 928–932. DOI: 10.1126/science.aan0070.

[50] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. "cQASM v1.0: Towards a Common Quantum Assembly Language ". 2018. arXiv: 1805.09607.

[51] H. J. Kimble. "The Quantum Internet". In: *Nature* 453.7198 (2008), pp. 1023–1030. DOI: 10.1038/nature07127.

[52] C. Knaut, A. Suleymanzade, Y.-C. Wei, D. Assumpcao, P.-J. Stas, Y. Huan, B. Machielse, E. Knall, M. Sutula, G. Baranes, et al. "Entanglement of nanophotonic quantum memory nodes in a telecom network". In: *Nature* 629.8012 (2024), pp. 573–578. DOI: 10.1038/s41586-024-07252-z.

[53] W. Kong, J. Wang, Y. Han, Y. Wu, Y. Zhang, M. Dou, Y. Fang, and G. Guo. "Origin Pilot: a Quantum Operating System for Effecient Usage of Quantum Resources". 2021. arXiv: 2105.10730. URL: https://arxiv.org/abs/2105.10730.

[54] W. Kozlowski, A. Dahlberg, and S. Wehner. "Designing a Quantum Network Protocol". In: *CoNEXT*. ACM, 2020, pp. 1–16. DOI: 10.1145/3386367.3431293.

[55] W. Kozlowski and S. Wehner. "Towards Large-Scale Quantum Networks". In: *NANOCOM*. ACM, 2019, pp. 1–7. DOI: 10.1145/3345312.3345497.

[56] V. Krutyanskiy, M. Galli, V. Krcmarsky, S. Baier, D. Fioretto, Y. Pu, A. Mazloom, P. Sekatski, M. Canteri, M. Teller, et al. "Entanglement of trapped-ion qubits separated by 230 meters". In: *Physical Review Letters* 130.5 (2023). Publisher: APS, p. 050803. DOI: 10.1103/PhysRevLett.130.050803.

[57] V. Krutyanskiy, M. Meraner, J. Schupp, V. Krcmarsky, H. Hainzer, and B. P. Lanyon. "Light-matter entanglement over 50 km of optical fibre". In: *npj Quantum Information* 5.1 (2019). Publisher: Nature Publishing Group UK London, p. 72. DOI: 10.1038/s41534-019-0186-3.

[58] S. Langenfeld, S. Welte, L. Hartung, S. Daiss, P. Thomas, O. Morin, E. Distante, and G. Rempe. "Quantum teleportation between remote qubit memories with only a single photon as a resource". In: *Physical Review Letters* 126.13 (2021). Publisher: APS, p. 130502. DOI: 10.1103/PhysRevLett.126.130502.

[59] P. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, and B. Stumpf. "The architecture of an integrated local network". In: *IEEE Journal on selected Areas in Communications* 1.5 (1983). Publisher: IEEE, pp. 842–857. DOI: 10.1109/JSAC.1983.1146002.

[60] T. van Leent, M. Bock, F. Fertig, R. Garthoff, S. Eppelt, Y. Zhou, P. Malik, M. Seubert, T. Bauer, W. Rosenfeld, W. Zhang, C. Becher, and H. Weinfurter. "Entangling single atoms over 33 km telecom fibre". In: *Nature* 607.7917 (July 2022). Publisher: Nature Publishing Group, pp. 69–73. ISSN: 1476-4687. DOI: 10.1038/s41586-022-04764-4. URL: https://www.nature.com/articles/s41586-022-04764-4 (visited on Oct. 30, 2024).

[61] D. A. Lidar and T. A. Brun. *Quantum Error Correction*. Cambridge University Press, 2013.

[62] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (1973), pp. 46–61. DOI: 10.1145/321738.321743.

[63] J.-L. Liu, X.-Y. Luo, Y. Yu, C.-Y. Wang, B. Wang, Y. Hu, J. Li, M.-Y. Zheng, B. Yao, Z. Yan, et al. "Creation of memory–memory entanglement in a metropolitan quantum network". In: *Nature* 629.8012 (2024), pp. 579–585. DOI: 10.1038/s41586-024-07308-0.

[64] W.-Z. Liu, Y.-Z. Zhang, Y.-Z. Zhen, M.-H. Li, Y. Liu, J. Fan, F. Xu, Q. Zhang, and J.-W. Pan. "Toward a Photonic Demonstration of Device-Independent Quantum Key Distribution". In: *Phys. Rev. Lett.* 129.5 (2022), p. 050502. DOI: 10.1103/PhysRevLett.129.050502.

[65] Y. Ma, E. Kashefi, M. Arapinis, K. Chakraborty, and M. Kaplan. "QEnclave-A practical solution for secure quantum cloud computing". In: *npj Quantum Information* 8.1 (2022). Publisher: Nature Publishing Group UK London, p. 128. DOI: 10.1038/s41534-022-00612-5.

[66] S. Massar and S. Popescu. "Optimal extraction of information from finite quantum ensembles". In: *Physical review letters* 74.8 (1995). Publisher: APS, p. 1259. DOI: 10.1103/PhysRevLett.74.1259.

[67] D. Matsukevich, P. Maunz, D. Hayes, L.-M. Duan, and C. Monroe. "Quantum teleportation between distant matter qubits". In: *Science* 323.5913 (2009). Publisher: American Association for the Advancement of Science, pp. 486–489. DOI: 10.1126/science.1167209.

[68] J. D. McCullough, K. H. Speierman, and F. W. Zurcher. "A design for a multiple user multiprocessing system". In: *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*. 1965, pp. 611–617. DOI: 10.1145/1463891.1463957.

[69] D. L. Moehring, P. Maunz, S. Olmschenk, K. C. Younge, D. N. Matsukevich, L.-M. Duan, and C. Monroe. "Entanglement of Single-Atom Quantum Bits at a Distance". In: *Nature* 449 (2007), pp. 68–71. DOI: 10.1038/nature06118.

[70] I. Monga, E. Saglamyurek, E. Kissel, H. Haffner, and W. Wu. "QUANT-NET: A testbed for quantum networking research over deployed fiber". In: *Proceedings of the 1st Workshop on Quantum Networks and Distributed Quantum Computing*. QuNet '23. New York, NY, USA: Association for Computing Machinery, Sept. 10, 2023, pp. 31–37. ISBN: 9798400703065. DOI: 10.1145/3610251.3610561. URL: https://dl.acm.org/doi/10.1145/3610251.3610561 (visited on Oct. 30, 2024).

[71] P. Murali, N. M. Linke, M. Martonosi, A. J. Abhari, N. H. Nguyen, and C. H. Alderete. "Full-Stack, Real-System Quantum Computer Studies: Architectural Comparisons and Design Insights". In: *ISCA*. ACM, 2019, pp. 527–540. DOI: 10.1145/3307650.3322273.

[72] D. Nadlinger. "Device-independent key distribution between trapped-ion quantum network nodes". PhD thesis. University of Oxford, 2022.

[73] M. Paris and J. Rehacek. *Quantum state estimation*. Vol. 649. Springer Science & Business Media, 2004.

[74] J. L. Peterson and A. Silberschatz. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1985. DOI: 10.5555/3526.

[75] W. Pfaff, B. J. Hensen, H. Bernien, S. B. van Dam, M. S. Blok, T. H. Taminiau, M. J. Tiggelman, R. N. Schouten, M. Markham, D. J. Twitchen, et al. "Unconditional quantum teleportation between distant solid-state quantum bits". In: *Science* 345.6196 (2014). Publisher: American Association for the Advancement of Science, pp. 532–535. DOI: 10.1126/science.1253512.

[76] A. Pirker and W. Dür. "A Quantum Network Stack and Protocols for Reliable Entanglement-Based Networks". In: *New Journal of Physics* 21.3 (2019), p. 033003. URL: 10.1088/1367-2630/ab05f7.

[77] M. Pompili, S. L. N. Hermans, S. Baier, H. K. C. Beukers, P. C. Humphreys, R. N. Schouten, R. F. L. Vermeulen, M. J. Tiggelman, L. dos Santos Martins, B. Dirkse, S. Wehner, and R. Hanson. "Realization of a Multinode Quantum Network of Remote Solid-State Qubits". In: *Science* 372.6539 (2021), pp. 259–264. DOI: 10.1126/science.abg1919.

[78] M. Pompili, C. Delle Donne, I. te Raa, B. van der Vecht, M. Skrzypczyk, G. M. Ferreira, L. de Kluijver, A. J. Stolk, S. L. N. Hermans, P. Pawełczak, W. Kozlowski, R. Hanson, and S. Wehner. "Experimental Demonstration of Entanglement Delivery Using a Quantum Network Stack". In: *npj Quantum Information* 8.1 (2022), p. 121. DOI: 10.1038/s41534-022-00631-2.

[79] A. Poremba. "Quantum proofs of deletion for learning with errors". In: *arXiv preprint arXiv:2203.01610* (2022). DOI: 10.48550/arXiv.2203.01610.

[80] *Quantum Software Development Kits in 2024*. AIMultiple: High Tech Use Cases & Tools to Grow Your Business. 2024. URL: https://research.aimultiple.com/quantum-sdk/ (visited on July 11, 2024).

[81] K. Ramamritham and J. A. Stankovic. "Scheduling algorithms and operating systems support for real-time systems". In: *Proceedings of the IEEE* 82.1 (1994). Publisher: IEEE, pp. 55–67. DOI: 10.1109/5.259426.

[82] S. Ritter, C. Nölleke, C. Hahn, A. Reiserer, A. Neuzner, M. Uphoff, M. Mücke, E. Figueroa, J. Bochmann, and G. Rempe. "An Elementary Quantum Network of Single Atoms in Optical Cavities". In: *Nature* 484.7393 (2012), pp. 195–200. DOI: 10.1038/nature11023.

[83] L. Robledo, H. Bernien, I. Van Weperen, and R. Hanson. "Control and Coherence of the Optical Transition of Single Nitrogen-Vacancy Centers in Diamond". In: *Physical review letters* 105.17 (2010), p. 177403. DOI: 10.1103/PhysRevLett.105.177403.

[84] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating Systems Concepts (Ninth Edition)*. Wiley, 2014.

[85] M. Skrzypczyk and S. Wehner. "An Architecture for Meeting Quality-of-Service Requirements in Multi-User Quantum Networks". 2021. arXiv: 2111.13124.

[86] R. Stankiewicz, P. Chołda, and A. Jajszczyk. "QoX: What is It Really?" In: *IEEE Communications Magazine* 49.4 (2011), pp. 148–158. DOI: 10.1109/MCOM.2011.5741159.

[87] L. J. Stephenson, D. P. Nadlinger, B. C. Nichol, S. An, P. Drmota, T. G. Ballance, K. Thirumalai, J. F. Goodwin, D. M. Lucas, and C. J. Ballance. "High-Rate, High-Fidelity Entanglement of Qubits Across an Elementary Quantum Network". In: *Phys. Rev. Lett.* 124.11 (2020), p. 110501. DOI: 10.1103/PhysRevLett.124.110501.

[88] R. Stockill, M. J. Stanley, L. Huthmacher, E. Clarke, M. Hugues, A. J. Miller, C. Matthiesen, C. Le Gall, and M. Atatüre. "Phase-Tuned Entangled State Generation between Distant Spin Qubits". In: *Phys. Rev. Lett.* 119.1 (2017), p. 010503. DOI: 10.1103/PhysRevLett.119.010503.

[89] A. J. Stolk, K. L. van der Enden, M.-C. Slater, I. t. Raa-Derckx, P. Botma, J. van Rantwijk, B. Biemond, R. A. Hagen, R. W. Herfst, W. D. Koek, et al. "Metropolitan-scale heralded entanglement of solid-state qubits". In: *arXiv preprint arXiv:2404.03723* (2024). DOI: 10.48550/arXiv.2404.03723.

[90] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. USA: Prentice-Hall, Inc., Nov. 2005. ISBN: 978-0-13-142938-3.

[91] M. Teller. "Measuring and modeling electric-field noise in an ion-cavity system (PhD thesis)". PhD thesis. University of Innsbruck, July 2021. URL: https://resolver.obvsg.at/urn:nbn:at:at-ubi:1-90870.

[92] M. Teller, D. A. Fioretto, P. C. Holz, P. Schindler, V. Messerer, K. Schüppert, Y. Zou, R. Blatt, J. Chiaverini, J. Sage, et al. "Heating of a trapped ion induced by dielectric materials". In: *Physical Review Letters* 126.23 (2021), p. 230505. DOI: 10.1103/PhysRevLett.126.230505.

[93] M. Teller, V. Messerer, K. Schüppert, Y. Zou, D. A. Fioretto, M. Galli, P. C. Holz, J. Reichel, and T. E. Northup. "Integrating a fiber cavity into a wheel trap for strong ion–cavity coupling". In: *AVS Quantum Science* 5.1 (2023). DOI: 10.1116/5.0121534.

[94] Texas Instruments. *LAUNCHXL2-RM57L Development Kit*. Accessed: 2024-07-03. 2024. URL: https://www.ti.com/tool/LAUNCHXL2-RM57L (visited on July 3, 2024).

[95] R. van Meter. *Quantum Networking*. John Wiley and Sons, Ltd, 2014. DOI: 10.1002/9781118648919.

[96] L. M. K. Vandersypen and I. L. Chuang. "NMR techniques for quantum control and computation". In: *Rev. Mod. Phys.* 76 (4 Jan. 2005), pp. 1037–1069. DOI: 10.1103/RevModPhys.76.1037. URL: https://link.aps.org/doi/10.1103/RevModPhys.76.1037.

[97] G. Vardoyan, M. Skrzypczyk, and S. Wehner. "On the Quantum Performance Evaluation of two Distributed Quantum Architectures". In: *Performance Evaluation* 153 (2022), pp. 1–26. DOI: 10.1016/j.peva.2021.102242.

[98] S. W. Warren. "Effects of arbitrary laser or NMR pulse shapes on population inversion and coherence". In: *The Journal of Chemical Physics* 81 (1984). DOI: 10.1063/1.447644. URL: https://doi.org/10.1063/1.447644.

[99] S. Wehner, D. Elkouss, and R. Hanson. "Quantum Internet: A Vision for the Road Ahead". In: *Science* 362.6412 (2018), pp. 1–9. DOI: 10.1126/science.aam9288.

[100] W. Zhang, T. van Leent, K. Redeker, R. Garthoff, R. Schwonnek, F. Fertig, S. Eppelt, W. Rosenfeld, V. Scarani, C. C.-W. Lim, et al. "A Device-Independent Quantum Key Distribution System for Distant Users". In: *Nature* 607.7920 (2022), pp. 687–691. DOI: 10.1038/s41586-022-04891-y.

[101] *ADwin-Pro II*. Jäger GmbH. URL: https://www.adwin.de/us/produkte/proII.html (visited on Feb. 28, 2023).

[102] *eRPC (Embedded RPC)*. Freescale Semiconductor, Inc. URL: https://github.com/EmbeddedRPC/erpc (visited on July 8, 2024).

[103] *FreeRTOS Real-Time Operating System for Microcontrollers*. Amazon Web Services. URL: https://www.freertos.org/ (visited on Feb. 28, 2023).

[104] *MicroZed Development Board*. Avnet. URL: https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/microzed/ (visited on Feb. 28, 2023).

[105] *NetQASM SDK*. QuTech. URL: https://github.com/QuTech-Delft/netqasm (visited on Feb. 28, 2023).

[106] *pthreads(7) - Linux manual page*. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html (visited on July 15, 2024).

[107] *SquidASM: a Simulator that Executes NetQASM Applications*. QuTech. URL: https://github.com/QuTech-Delft/squidasm (visited on July 14, 2024).

5

Qoala: an Application Execution Environment for Quantum Internet Nodes

In Chapter 4, we presented a first-of-its-kind operating system for programmable quantum network nodes, called QNodeOS. In this chapter, we present an extension of QNodeOS called Qoala, which introduces (1) a unified program format for hybrid interactive classical-quantum programs, providing a well-defined target for compilers, and (2) a runtime representation of a program that allows joint scheduling of the hybrid classical-quantum program, multitasking, and asynchronous program execution. Based on concrete design considerations, we put forward the architecture of Qoala, including the program structure and execution mechanism. We implement Qoala in the form of a modular and extendible simulator that is validated against real-world quantum network hardware (available online). However, Qoala is not meant to be purely a simulator, and implementation is planned on real hardware. We evaluate Qoala's effectiveness and performance sensitivity to latencies and network schedules using an extensive simulation study. Qoala provides a framework that opens the door for future computer science research into quantum network applications, including scheduling algorithms and compilation strategies that can now readily be explored using the framework and tools provided.

5.1 Introduction

Advances in quantum computing and quantum communication technologies are paving the way for a *quantum internet* [31, 55], where quantum applications are executed across multiple network nodes. Examples of such applications include quantum key distribution (QKD) [7, 22] and blind quantum computation (BQC) [2, 9] from a client to a quantum cloud server. A multi-node quantum internet application is partitioned into separate

This chapter is based on the preprint: **B. van der Vecht**, A. T. Yücel, H. Jirovská, and S. Wehner. "Qoala: an Application Execution Environment for Quantum Internet Nodes". In: *arXiv preprint arXiv:2502.17296* (2025). DOI: 10.48550/arXiv.2502.17296.

single-node *programs* (e.g. a client program and a server program in BQC) that run concurrently on different network nodes. To support security sensitive applications, each program performs local classical and quantum computations on its own private node, and programs interact with each other only via classical message passing and entanglement generation. This is in sharp contrast to distributed quantum computing (see e.g. [13]), where all nodes can be accessed and controlled by a single program.

The single-node programs that constitute a quantum internet application are hybrid in nature (see Figure 5.1): they can contain both classical and quantum operations, and these operations can be both local (executed fully on the node itself) or networked (interacting with another node in the network). Quantum operations include quantum gates and measurements, e.g. to perform a server computation in BQC, (*local quantum*), and entanglement generation, e.g. to produce classical bits for a secret key in QKD (*networked quantum*). Entanglement is a special property of two quantum bits (qubits) that forms a key resource for quantum internet applications. All quantum operations are executed on quantum processors that can store, manipulate and measure quantum information, where small networks including such processors have been realized using different quantum hardware platforms including, for example, nitrogen-vacancy (NV) centers in diamond [43], and trapped ions [33]. Programs also need to perform classical operations, such as message passing (*networked classical*, e.g. a BQC client program sending desired measurement bases to the BQC server), and local classical processing (*local classical*, e.g. post-processing measurement outcomes in QKD).

Realizing the execution of quantum internet applications presents unique challenges (see Section 5.3): First, a program for a quantum internet application is not merely a hybrid of classical and quantum code segments; these segments are also highly *interactive*: classical and quantum code may run concurrently, communicating and influencing each other. E.g., a quantum circuit (a series of local quantum gates) may "pause" halfway, keeping quantum states in memory, and wait for a value from a classical segment (e.g. a classical message from a remote node) before continuing. This interactivity makes arbitrary quantum network applications more complex than simple prepare-and-measure quantum network protocols that do not require this interactivity, such as QKD. Quantum memories have limited lifetimes, meaning qubits are subject to decoherence, degrading their quality over time. This introduces the need control the joint schedule of the classical and quantum segments of the program to reach desired levels of application performance.

Second, a compiler should be able to optimize the whole program including both classical and quantum code, as well as to provide information that can be used in our architecture to align and inform scheduling decisions.

Finally, we are faced with a mix of time scales: on the one hand, entanglement generation requires a very precise network schedule that is agreed ahead of time between the network nodes [16]. On the other hand, classical messages are exchanged asynchronously between the nodes without guaranteed message delivery times. This motivates an architecture in which different segments of the system may operate at different levels of timing precision. In Chapter 4, we presented QNodeOS, the first architecture for executing arbitrary programs on quantum network nodes. QNodeOS tackles the above challenges, but we suggested that there is room for improvements in the architecture, including enabling better support for compilation and gaining better scheduling control by putting compo-

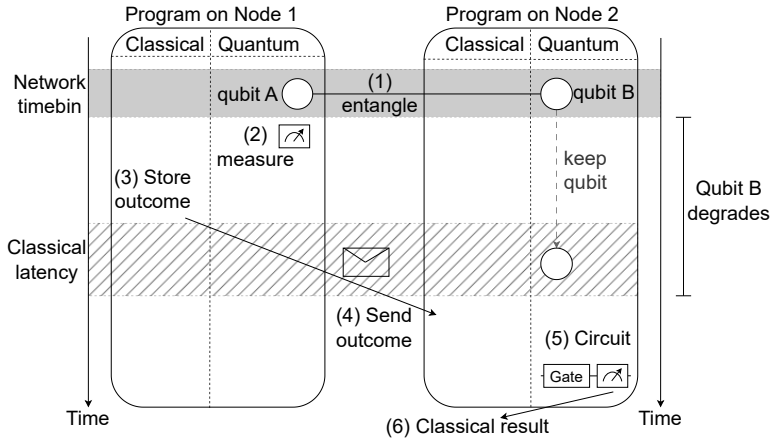


Figure 5.1: Example application consisting of two hybrid classical-quantum programs (on Nodes 1 and 2) including (1) Entanglement generation between two qubits (circles) in a synchronized time slot (defined by network controller). (2) A local measurement of qubit A at Node 1 resulting in a classical outcome bit (destroying the qubit). (3) Outcome bit is stored in classical memory. (4) Communication of the classical bit from Node 1 to Node 2 (taking non-deterministic time). (5) Execution of a quantum circuit on qubit B at Node 2 depending on the classical bit. The quality of qubit B has degraded during the time elapsed since (1). (6) Node 2 measures qubit B and outputs the classical result.

nents on the same board. In this chapter, we explore these improvements.

5.1.1 Main contributions

We propose an extension of the QNodeOS architecture for program execution on quantum network nodes, called Qoala, that addresses the above challenges. Qoala is an execution environment tailored to programmable quantum internet nodes, accommodating the **hybrid, interactive, networked, and asynchronous nature** of quantum internet applications.

Unified program format for hybrid-classical quantum programs: Qoala defines a unified program format for executables, encompassing classical and quantum (networked and local) code, and defining basic blocks. This format is suitable for arbitrary quantum network programs up to the most advanced stage [55]. This paves the way for a joint optimization of the classical and quantum code by a compiler.

Runtime representation allowing scheduling: Qoala separates the static unified program format from a runtime representation consisting of *tasks*. This paves the way to design and implement algorithms for scheduling the quantum program in order to meet deadlines imposed by decoherence of the quantum memory. To provide advice to the scheduler on deadlines to achieve a desired program performance, programs can specify advice for timing and prioritization depending on the quantum hardware capabilities of the node. The separation of a static program from its runtime tasks also allows for the programmer to define asynchronous code segments, the execution of which is decided by the scheduler alone. This is the first architecture that allows for effective scheduling control of hybrid interactive classical-quantum programs, thus addressing a critical issues

in the successful execution of quantum internet applications.

Integration with quantum network stack: Qoala integrates with the existing quantum network stack [16], also present in QNodeOS [42], for realizing entanglement generation between nodes. This opens the door for Qoala to be implemented on such networks.

Implementation in hardware validated simulation: We implement the proposed architecture as a **modular and composable simulator**, which enables the evaluation of different execution strategies and techniques. The simulation is validated against real-world quantum hardware implementations, opening the door to understand performance tradeoffs and requirements for Qoala's implementation. Specifically, the simulator allows configuring different hardware parameters, latencies, and software component organizations, to evaluate implementation choices of Qoala in simulation.

Using the implementation we demonstrate the effectiveness and feasibility of our proposed architecture on different types of quantum hardware, including its ability to schedule and multitask applications using a number of existing scheduling methods (EDF, FCFS). We continue to examine tradeoffs in the classical and quantum performance metrics of using different types of scheduling approaches. We examine Qoala's improvement over NetQASM ([17] and Chapter 3) in enabling hybrid classical-quantum compilation possibilities. Finally, we study trends in application performance when varying the amount of concurrency, and examine the impact of a network schedule for entanglement generation on the performance of Qoala.

We stress that Qoala is not just a simulator. Qoala is an architecture for executing quantum network programs, and is not tied to specific implementations. The simulator implementation of the architecture validates the design and opens possibilities for further research. However, Qoala is also planned to be implemented as (part of) an operating system running on real (quantum) hardware.

We highlight the role of Qoala in opening the door for computer science research. We make our simulator available as open source [44], paving the way for computer scientists to conduct further research, e.g., into the design of compilers, or schedulers that can readily be tested using the simulator.

The remainder of this chapter is structured as follows. Section 5.2 compares our work to related studies. In Section 5.3 we explain important context and terminology, followed by considerations that we used to design our architecture (Section 5.4). Section 5.5 discusses our implementation and Section 5.6 provides evaluation results using this implementation. We conclude and give suggestions on future work (Section 5.7).

5.2 Related work

Networks of quantum processors have been realized using different quantum hardware platforms including, for example, nitrogen-vacancy (NV) centers in diamond [43], and trapped ions [33]. A first operating system QNodeOS [20] (see also Chapter 4) including a network stack [42] has been designed and implemented on real quantum network nodes based on NV centers in diamond. QNodeOS makes use of the NetQASM execution framework [17] (see also Chapter 3), where a classical network processing unit (CNPU) dispatches NetQASM routines for execution by a quantum network processing unit (QNPU). Our work builds on top of ideas of QNodeOS and NetQASM, but addresses critical challenges that were not handled by these previous systems, including the ability to schedule

	QNodeOS	Qoala
Executes quantum internet applications	✓	✓
Interfaces with quantum network stack	✓	✓
Allows scheduler to meet memory lifetimes	✗	✓
Allows classical-quantum hybrid compilation	✗	✓

Figure 5.2: QNodeOS (Chapter 4) vs. Qoala capabilities.

hybrid programs and to optimize over the whole program code (see Figure 5.2 for a comparison). Building on the only such systems that have seen real world implementation on quantum hardware, opens the door for a later implementation of Qoala on quantum hardware by implementing an improved low-level classical control hardware architecture (Section 5.4).

Research has been done on related topics, such as distributed quantum computing, or hybrid (non-interactive) quantum computing. Hybrid classical-quantum programs have been extensively studied in quantum computing, e.g. in the context of *variational quantum eigensolvers (VQE)* [18, 39] or *quantum approximate optimization algorithms (QAOA)* [23]. However, they differ in two important aspects: although they are hybrid, they are not *interactive* during the quantum execution: (1) classical and quantum segments do not run concurrently, but quantum segments are executed in their entirety before returning to classical segments, i.e. no quantum state is kept in the processor between the execution of different quantum segments. (2) such hybrid programs lack network interoperability (entanglement generation and classical message-passing between nodes), and also do not have the same timing and flexibility requirements.

Distributed quantum computing [12] shares similarities with quantum internet applications but differs in several aspects. In the former, complete control is assumed over all participating nodes, such as an application distributed across multiple cores on a single chip [30, 40]. Generally, the capabilities of each core and the latencies between them are fully known, allowing for precise scheduling and orchestration of individual programs running on each core to optimize overall execution. In contrast, programs in quantum internet applications operate independently (and may even be running on different quantum hardware); therefore they have a degree of autonomy in their own scheduling, and are not fully aware of the actions or timing of other programs.

Entanglement distribution in networks is another related topic that has been extensively studied (see e.g. surveys [4, 56]). However, these works do not deal with executing network applications, and give only predictions for applications in which entanglement is immediately measured (e.g. QKD).

The concept of (soft) deadlines for program execution is of course well known from classical real-time systems that are often used in domains where deterministic and time-critical response is essential, such as automotive, aerospace and medical devices [11, 26, 38], including examples of systems with mixed timing precision [10]. We draw inspiration from this domain, and the present architecture opens the door to explore algorithms and concepts from this domain to be applied to the execution of quantum internet applications.

5.3 Design considerations

5.3.1 Background and context

We first revisit some of the relevant background and context (see also Chapter 2).

Quantum nodes. A quantum internet connects quantum nodes on which quantum programs may be executed. In their most general form, such nodes are *processing nodes* that have a quantum memory to store quantum bits (qubits) on which quantum operations (qubit initialization, quantum gates and measurements) can be performed. Pairs of nodes can establish *entanglement* between them over a quantum network. Entanglement is a special property of two qubits (an *entangled pair*), where one qubit is stored in the memory of each node. Nodes can also exchange classical messages (e.g. via dedicated classical links or the internet), where no guarantees are assumed on their message delivery times.

Programs. A program is a series of instructions to be executed by a node. Instructions can be categorized into four types: local classical processing, classical message-passing, quantum local processing (quantum operations), and remote entanglement generation. A program can keep classical variables in a classical memory, and quantum variables (qubits) in the node's quantum memory during the execution. Multiple programs, each running on their own node, together form an *application* (see Figure 5.1), e.g. QKD (two programs, one per node), or secret sharing [28] (a program each on many nodes). Programs may involve asynchronous operations (e.g. a server awaiting entanglement with multiple clients).

Network schedule. A quantum network stack has been proposed [16] and implemented [42] that turns entanglement generation into a robust service independent of the quantum hardware platform. Important for the design of an architecture for the execution of quantum internet applications is that in this stack, the nodes will establish a network schedule of time slots in which they will trigger entanglement generation (due to need to synchronize entanglement generation at the physical layer [16] at high precision (ns)). This means that once entanglement has been requested from the network, the nodes can use only the slots in the network schedule to produce entanglement between them, imposing constraints on the ability to schedule applications. What's more, in present day systems [33, 43] limitations in the physical devices prohibit the execution of local operations while engaging in network operations (entanglement generation), creating further dependencies between the local quantum execution and entanglement generation. As the specifics of network scheduling [6, 50] are not within scope of this thesis, we assume the existence of a *network controller* that takes application demand for entanglement and issues a network schedule to the nodes. A schedule consists of sequential time slots, each with a start time and duration, when the node will trigger entanglement generation. Nodes are not forced to attempt entanglement in corresponding time slots, and can instead choose to do local processing instead.

Performance metrics and noise. Quantum internet applications have classical outcomes that are typically probabilistic in nature: (1) applications may intentionally do measurements on quantum states that have fundamentally probabilistic outcomes (e.g. quantum cryptography), (2) in practice, quantum hardware is imperfect (or *noisy*). That is, undesired errors occur when performing operations (such as gates, measurements, or entanglement generation) or when keeping quantum states in memory for too long.

In many quantum internet applications (e.g. BQC), a single execution of the application can result in failure or success (e.g. a BQC client receives correct measurement results

from the server program [36]). Applications are often executed many times, where outcome statistics are computed in order to validate successful execution (e.g. by majority of outcomes). We consider two metrics: a *quantum metric* — the *success probability* of executing a single instance of the application (on average), and a *classical metric* — the *makespan*, i.e. the average execution time of an application instance.

5.3.2 Considerations

Considerations can be categorized into three main groups: fundamental, technological, and enabling.

Fundamental Considerations (1) *Hybrid nature of applications (FC1)*: Quantum internet applications inherently consist of both classical and quantum segments, as well as local and networking operations. The execution environment must account for this hybrid nature, and the program structure should accommodate all types of operations. (2) *Interactive nature of applications (FC2)*: Quantum internet applications require classical communication between nodes. This communication may take place in between classical and quantum segments of a single program. This implies the need for application-level interfaces between programs on different nodes, and for interfaces between classical and quantum code segments on a single node. (3) *Multitasking (FC3)*: Programs may spend a significant amount of time waiting for messages from a remote node (ms), motivating multitasking to make optimal use of the classical and quantum computing resources at each node. This requires scheduling of time and resources.

Technological Considerations (1) *Limited qubit lifetime (TC1)*: Quantum memory quality degrades over time, presenting a significant challenge for the execution environment, especially in near-term hardware (sub-millisecond to multiple seconds memory lifetimes [32, 43, 47]). As such, there are natural deadlines to application execution after which a desired performance (success probability) can no longer be reached. We thus desire that a program specification allows indication of memory quality constraints (deadlines), which the runtime environment can act upon (e.g. by appropriate scheduling or restarting). (2) *Integration of processing and networking (TC2)*: We assume that near-term nodes only have a single quantum processor, which needs to perform both local quantum gates as well as remote entanglement generation. That is, while performing local operations the processor is blocked from networking operations and vice versa, as is the case for all current implementations [33, 43] but may be mitigated partially using future proposals [53]. The node must hence allocate time for local computation while at the same time adhering to the network schedule which constrains timing of the entanglement operations.

Enabling Considerations (1) *Different compilation strategies and programming languages (EC1)*: The execution environment should support various compilation strategies and accessible programming languages. In order to enable compilation, we furthermore want a representation of the program that can be integrated with existing compiler frameworks. (2) *Different scheduling strategies (EC2)*: Since we expect that scheduling plays a vital role in optimizing application performance, the execution environment should enable scheduling, and support different scheduling algorithms and policies, allowing for their comparison and evaluation. (3) *Different (control) hardware implementations (EC3)*: The architecture should make minimal assumptions on the classical control hardware, and be independent of the choice of quantum hardware platform, allowing for integration with multiple

(future) technologies such as NV centers [42] or trapped ions [21].

5.4 Architecture

Based on these design considerations, we propose Qoala (see Figure 5.4), an execution environment for programmable nodes in a quantum internet. Provided minimal hardware assumptions are met (Section 5.4.1), each node implements its own Qoala execution environment, supporting a specific *program structure* (Section 5.4.2) and implementing a specific *runtime environment* (Section 5.4.3) that is able to *schedule tasks* (Sections 5.4.4 to 5.4.6). Details in Sections 5.9 to 5.11.

5.4.1 Minimal hardware assumptions

Qoala is based on only a few core assumptions on the processing node (consideration EC3, Figure 5.3):

CPS-QPS distinction. We assume the node distinguishes between a *classical processing system (CPS)* managing classical computing resources (e.g. CPU, classical memory and networking), and a *quantum processing system (QPS)*, responsible for executing quantum operations (gates, measurements, entanglement generation) on quantum hardware including a quantum memory as in Chapters 3 and 4. We hence use a similar distinction as with the CNPU and QNPU from Chapters 3 and 4 but with a slightly different terminology and division of abstractions: the CPS is on the same level as the CNPU from Chapters 3 and 4, but the QPS is either just the QNPU level or the combined 'QNPU + QDevice' level. Unlike in the implementation of QNodeOS, we assume a shared classical memory is accessible to both the CPS and QPS, enabling communication between the two processing systems, addressing the interactive property of quantum internet programs. The CPS can act as a fully-fledged classical computer, and performs application-level classical communication with other nodes as well as with a network controller who sets a network schedule. The QPS can execute routines consisting of low-level quantum gates, basic classical control logic (branching), and entanglement generation. This opens the door for the QPS to be based on essentially any quantum hardware platform where a specialized microcontroller is used to control the quantum hardware, and a separate microprocessor implements the CPS, where a shared memory could be realized next to the two processors on-chip. The scheduler controls both CPS and QPS execution, and may physically be realized on either one.

Time granularity. Both CPS and QPS are assumed to have knowledge of time, albeit operating with different timing precision (*ms* precision for CPS mirroring node-to-node communication latencies vs. μs and *ns* precision needed for synchronized entanglement generation [16, 42].)

Network stack. A quantum network stack including a network layer [16] is implemented on the node with which Qoala can interface. This stack can receive and fulfill requests for remote entanglement generation.

5.4.2 Program structure

Qoala defines a hybrid format for programs, mapping naturally to their hybrid nature (consideration FC1 in Section 5.3). A Qoala program is a combination of quantum and

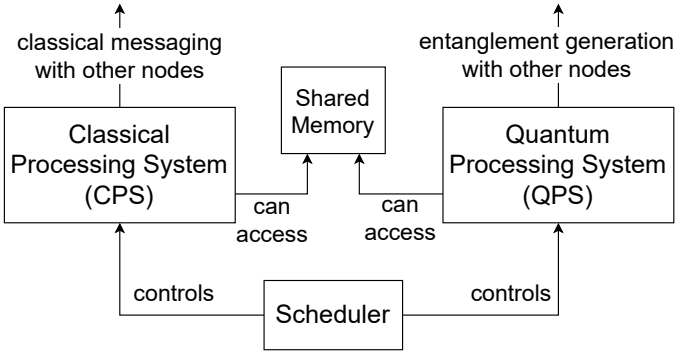


Figure 5.3: Minimal hardware assumptions for a single node. A Classical Processing System (CPS) can execute classical code and can communicate classical messages with other nodes in the network. A Quantum Processing System (QPS) can execute quantum code and can realize entanglement (quantum connections) with other nodes in the network. The CPS and QPS are controlled by a scheduler, and have access to shared memory. In the QNodeOS architecture [20], the CPS is realized as the CNPU, and the QPS as the combined QNPU-QDevice system. For Qoala, we only focus on the classical-quantum distinction, and not the internal implementation (such as a QNPU-QDevice separation), hence the different terminology.

classical instructions, organized into three main sections: *host code* (containing classical instructions), *local routines* (containing local quantum instructions), and *request routines* (for remote entanglement generation). This hybrid format allows a compiler to optimize the whole program, including critical code paths with dependencies between classical and quantum segments. Local routines and request routines can be triggered from within host code as function calls, addressing the interactivity between them (consideration FC2).

A Qoala program is an *executable* and output of a compiler. The format is separate from any high-level language in which a programmer might write code; hence Qoala in theory allows for compatibility with any such language (consideration EC1). Entry and exit points of a program are in host code. Figure 5.5 shows an example program in text format. We contrast Qoala's program format with that of NetQASM (see [17] and Chapters 3 and 4), in which there was no way to compile across classical and quantum code segments. A Qoala program has *program arguments* that are filled in during program instantiation (Section 5.4.5).

Host code. Host code, executed on the CPS, encompasses local computation, control-flow, inter-node messaging, and can initiate local and request routines. For example, in a program that is part of a QKD application, classical post-processing (including sending bases, local error correction, and privacy amplification [54]) would be represented in host code. Host code is structured as a sequence of *blocks*, each holding a list of instructions. Blocks dictate control-flow by ending with a (conditional) jump instruction (default: next block in the sequence). This block division not only facilitates task creation and scheduling (see Section 5.4.4) but also streamlines compiler integration (which may use blocks in its intermediate representation). Blocks can contain metadata about their expected *duration*, (relative) deadlines, and they may be inside *critical sections*, encompassing a sequence of blocks with a maximally allowed execution duration. This metadata is propagated to corresponding tasks and used by the scheduler in order to mitigate quantum decoherence

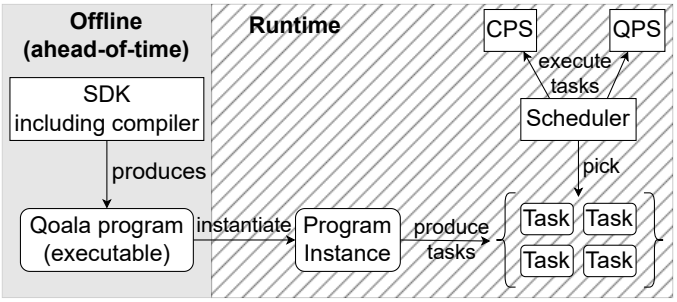


Figure 5.4: High-level overview Qoala: An SDK allows program code in a high-level language (e.g. Python). A compiler translates this code into a Qoala program (specific compiler not in scope of this work). To run, a program is instantiated with concrete values for program arguments. Tasks are created for the program instance, which are scheduled and executed by the scheduler. Multiple program instances may exist at the same time (both multiple instances of the same or different programs). All tasks from all instances are added to a single task graph (Section 5.4.4) used by the scheduler.

due to limit qubit lifetime (consideration TC1). Asynchronous execution is possible by 'submitting' multiple routines for execution, and waiting for all of them to finish. At runtime, the scheduler can decide in which order to execute the routines.

Local routine. A local routine (LR) represents a series of quantum operations (like gates and measurement), to be executed by the QPS locally (no interaction with external nodes or controllers). An LR may also contain limited classical computation and control-flow code allowing for fast feedback, which can increase quantum performance (Section 5.3.1) due to less decoherence. An updated version of NetQASM (see [17] and Chapter 3) is used to represent the instructions, which allows both hardware-specific and hardware-agnostic instructions. Therefore, the program format is compatible with different quantum hardware. In contrast to [17], Qoala's version of NetQASM does not have instructions for entanglement generation (cleanly separating local and networked quantum operations) nor 'wait' instructions. This allows routines to be treated as atomic non-preemptable blocks.

Request routine. A request routine (RR) consists of a request for entanglement generation with another node, and represent requests to the node's quantum network stack. It can have local routines as callbacks, allowing quick local (quantum) processing of entangled qubits on the QPS without returning to the CPS, decreasing waiting time and decoherence.

5.4.3 Runtime environment

The Qoala runtime environment provides various resources that programs can leverage during execution.

Exposed Hardware Interface (EHI). The Exposed Hardware Interface provides information about the hardware and software capabilities and restrictions of the node and the network, like available quantum memory and expected latencies. Each node provides their own EHI which is used in capability negotiation (see below), and allows a choice of executable code optimized by a compiler for those capabilities ahead of time.

Shared memory. To address the classical-quantum interactivity in programs, the CPS

```
HOST
  ^b0: // start basic block b0
    run_request() : req1            // call req1
  ^b1: // start basic block b1
    angle = recv_cmsg(0)            // receive message
  ^b2 { deadline: [b0 = 0.1*T2] }:
    m = run_routine(angle) : subrt1 // call subrt1
  ^b3:
    return_result(m)                // program result

ROUTINE subrt1           // Local routine definition
    params: angle        // Argument (in @input[0])
    result: m            // Result (in @output[0])
  NETQASM_START
    load C0 @input[0]   // load angle
    set Q0 0            // use qubit 0
    rot_y Q0 C0 4       // rotate qubit 0
    meas Q0 M0          // measure qubit 0
    store M0 @output[0] // return outcome
  NETQASM_END

REQUEST req1            // Request routine definition
    remote_id: alice_id // Node to create EPR pair with
    epr_sck_id: 0       // EPR socket ID
    num_pairs: 1        // Number of pairs to create
    virt_ids: all 1     // Store EPR qubit as virt ID 1
    type: create_keep   // Keep EPR qubit in memory
```

Figure 5.5: Example Qoala program containing a host section with 4 blocks, a local routine (`subrt1`), and a request routine (`req1`). Block `b2` has a relative deadline to `b0` of 0.1 times qubit noise parameter T_2.

and QPS share data with each other via *shared classical memory*. Write conflicts are avoided by explicit read/write rules for shared memory regions (Section 5.10.3). Our conceptual model of a shared memory leaves open different implementation choices, including a physical shared memory or a message-passing protocol. Calls in host code to local or request routines use the shared memory to communicate routine arguments and results.

Quantum memory. Quantum memory is organized into a *virtual quantum memory space (VQMS)* for each program instance (see Section 5.4.5 for instantiation), represented as Unit Modules (specifying the qubit topology [17]). Qoala maps each VQMS to the physical qubits available in the QPS. VQMS information like qubit connectivity and noise characteristics is provided by the EHI, which a compiler can use to optimize a program. The VQMS enables multitasking since programs have their own runtime context, while a scheduler (Section 5.4.6) sees the whole physical memory space and can schedule programs accordingly.

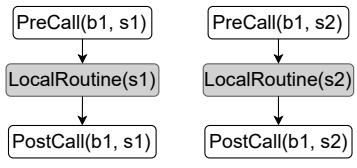
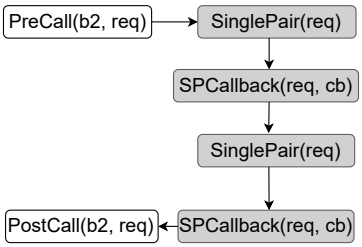
Remote interaction. For interaction with programs on remote nodes, the runtime provides *classical sockets* and *EPR sockets* based on [17]. Host code uses classical sockets for sending and receiving messages; EPR sockets are indicated in request routines (see e.g. Figure 5.5).

5.4.4 Tasks

We introduce *tasks* to enable multitasking (consideration FC3). Each task represents a code segment of a running program with a context of runtime variables. Tasks have different types (Figure 5.6) based on the code they represent. By splitting a program into distinct executable tasks, we can utilize the parallel execution on the CPS and QPS (by

Task Type	Processing Sytem	Pre-emptable	Trigger	Explanation
HostLocal	CPS	Yes	-	Local classical processing (LCP)
HostEvent	CPS	Yes	Remote message arrival	LCP starting with "receive classical message"
PreCall	CPS	Yes	-	Allocate shared memory and write args for LR or RR call
PostCall	CPS	Yes	-	Free shared memory and read results of LR or RR call
LocalRoutine	QPS	No	-	Execute a full Local Routine (LR)
SinglePair/ MultiPair	QPS	No	Start of slot in network schedule	Create one or more pairs for a Request Routine (RR)
SPCallback/ MPCallback	QPS	No	-	Execute a callback LR after creating one or more pairs of an RR

(a)

Host code snippet	Task Graph
^b1: submit_subroutine() : s1 submit_subroutine() : s2 join_routine(s1, s2)	
^b2: run_request() : req REQUEST req: callback: cb callback_type: sequential num_pairs: 2	

(b)

Figure 5.6: (a) Overview of task types. (b) Examples of host code and corresponding task graphs. Shaded tasks are executed by the QPS, the others by CPS. Top: asynchronous submission of local routines `s1` and `s2`. The graph consists of two separate chains of tasks and the scheduler can choose in which order to execute these chains (possibly interleaved). Bottom: a request routine uses the callback entry to immediately add tasks for executing local routine `cb` after each entangled pair generation.

assigning tasks to the corresponding system), and we can interleave execution of multiple programs by filling waiting times of one program by execution of tasks of another. Code segments indicated to run asynchronously (Section 5.4.2) can also be represented by tasks, the execution order of which can then be governed by a scheduler. Further, tasks enable interleaving of local operations and quantum network (entanglement generation) operations. A scheduler can choose when to execute entanglement tasks (with strict timing requirements from the network schedule) and when to execute local tasks (less strict requirements), addressing consideration TC2.

Task graph. Tasks are organized in a *task graph*, a directed acyclic graph (DAG) where each node represents a single task. Edges can be *precedence constraints* (task A must conclude before task B initiates) or *relative deadlines* (task B should start within maximum duration t after completion of task A). Using a task graph introduces a well-defined and isolated scheduling problem: given a graph of tasks, which task(s) should be executed next? Deadlines are used to assist the scheduler (see below) in mitigating the gradual quality degradation of quantum states over time (decoherence) by choosing appropriate tasks. Some tasks are only enabled after certain events happen. `HostEvent` tasks are enabled by an incoming classical message and `SinglePair` or `MultiPair` tasks are enabled by network schedule timestamps. Tasks also have information about what quantum memory they use, helping the scheduler decide which tasks it can execute at a given time.

Task creation. A task is created for a segment of a running program. If a program segment is executed multiple times (e.g. because of a loop in the code), this results in multiple tasks. A host code block is translated into a `HostLocal` task (block contains only local instructions) or a `HostEvent` task (block starts with a 'receive message' instruction). A local routine call is represented by (1) a `PreCall` task (CPS allocates shared memory and writes routine arguments), (2) a `LocalRoutine` task (QPS executes routine), and (3) a `PostCall` task (CPS reads routine results from shared memory). Request routine calls are handled similarly (with `SinglePair` or `MultiPair`). `MultiPair` tasks can be more time- and resource efficient since the network stack can handle multiple pair generations at once. Callback tasks for local routines acting as entanglement generation callbacks allow quick successive execution. For each task, its expected duration is calculated based on the metadata of the corresponding block or routine in the program, together with information from the EHI (see below). See Figure 5.6a for task types and Figure 5.6b for examples of host code and corresponding tasks (details in Section 5.11).

5.4.5 Program instantiation

A program is part of an application that uses entanglement generation orchestrated by a network controller (Figure 5.1). Therefore, before execution, the program must align with the other programs of its application as well as with the network controller. (1) *Capability negotiation and entanglement demand registration.* First, all collaborating nodes exchange their EHI and agree on concrete values for deadlines and task duration estimations (using advice pre-computed by the compiler). These values are needed to do effective scheduling at runtime. Second, the nodes together register their entanglement demands to the network controller, which then creates a *network schedule* based on these. This schedule consists of time slots, each of which is assigned to an individual *application instance* (tuple of program instances, one per node). (2) *Program instantiation.* Concrete values for

program arguments can be filled in such as deadlines, durations and program-specific input values. Typically, for a given application, the involved nodes create many *program instances* of the same program (to gather statistics, Section 5.3.1).

5.4.6 Scheduling and execution

Tasks produced for program instances are executed by the *node scheduler*. This scheduler manages a global task graph containing all tasks that have been created for instantiated programs and that are awaiting execution. Among the tasks that do not have any precedence constraints going into them (anymore), the scheduler continuously chooses the next task(s) to execute. It may choose to run a task on the CPS and a task on the QPS in parallel. If a task completed successfully, it is removed from the task graph, and precedence constraints and relative deadlines are updated accordingly. Based on the control flow of the program that this task was for, new tasks may be created representing the next segment of the program. These tasks are then added to the task graph. If a task failed (for example, entanglement generation did not succeed for a `SinglePair` task), it either (a) remains in the task graph and may be scheduled again at a later time, or (b) the whole program instance is aborted, depending on the scheduler implementation. For *predictable programs* (where control-flow and hence all corresponding tasks are known beforehand), their entire task graph may be created ahead of execution (no need to add new tasks at runtime). Tasks for entanglement generation (like `SinglePair`) additionally contain information about when they are allowed to start according to the network schedule, allowing the scheduler to make sure that the network schedule is respected. The scheduler allows pre-emption of CPS tasks. For instance, the arrival of a message from a remote node might activate a `HostEvent` task with high priority; if the CPS was executing another lower priority task, it may be pre-empted and resumed at a later time. Since quantum tasks cannot in general be rolled back or resumed (e.g. measurements are destructive and cannot be undone), Qoala does not allow the pre-emption of QPS tasks. Although we define a scheduling problem, and a framework for designing and implementing scheduling algorithms, we on purpose do not prescribe an explicit implementation and leave the question of an optimal scheduling approach open for further research (consideration EC2, see also Section 5.7).

5.5 Implementation

We implement our architecture in the form of an open-source simulator [44]. Implementation on real hardware requires developing new classical control hardware which is outside the scope of this work. The simulator is built on top of NetSquid [15] which can simulate quantum behavior as well as asynchronous classical processes. Specifically, NetSquid provides detailed configuration allowing for simulations of hardware with parameters that are validated in real experiments, not possible using other simulators such as QuNetSim [19], QNET [45], and QuISP [48]. SquidASM [46] simulates the software and hardware stack used in the NetQASM/QNodeOS system mentioned in Section 5.2, and hence misses the scheduling capabilities that we introduce in Qoala.

The simulator has on purpose been made modular and composable: components of Qoala's architecture (like CPS, QPS, scheduler, shared memory) are provided by the simulator as building blocks that can be configured and put together in different ways (details

Section 5.12). Both classical software parameters and quantum hardware noise models can be configured. In this way, the simulator allows one to investigate different architecture and parameter choices. In the simulator, a network of quantum nodes implementing Qoala can be constructed, and Qoala programs can be submitted for execution to these nodes. Static network schedules can be provided (capability negotiation and automatic network schedule creation are not simulated). The simulator then executes the programs, providing application results and statistics. Our implementation allows researchers to not only test Qoala, but also configure parameters and architectures to investigate scheduling algorithms and hardware implementation choices.

5.5.1 Scheduler implementation

In our implementation, we use a two-level hierarchical scheduler architecture, consisting of a node-wide *node scheduler* which controls two *processor schedulers*, one for the CPS and one for the QPS (Figure 5.7, details in Section 5.11). Such an approach has been used in other contexts not related to quantum networks [24, 41].

Each scheduler maintains their own task graph. The node scheduler task graph contains all tasks (CPS or QPS) that are to be executed. Each processor scheduler task graph is a partial copy of the node scheduler task graph containing only the tasks that can be executed by its own processor. Edges in the node scheduler graph between heterogeneous tasks (i.e. between CPS and QPS tasks) are represented in the partial processor graphs by an `external-dependencies` node attribute. When a processor scheduler finishes a task, it is removed from the task graph and a signal is sent to the node scheduler. The node scheduler updates its own task graph accordingly, and may then add new tasks to the task graph of the processor scheduler. Write conflicts on the processor task graphs are avoided since tasks can only be added by the node scheduler, and tasks can only be removed by the processor scheduler.

The processor schedulers support both a first-come-first-serve (FCFS) and an earliest-deadline-first (EDF) [49] scheduling mechanism. In our evaluation (Section 5.6), deadlines are used as *soft deadlines*, i.e. there is no guarantee about meeting deadlines.

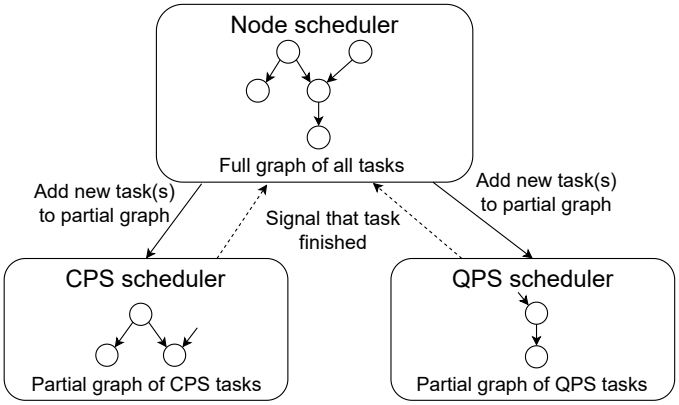


Figure 5.7: Overview of our hierarchical scheduler implementation. The node scheduler maintains a graph of all tasks. The CPS and QPS maintain partial graphs with only tasks they can execute themselves. Partial graphs are updated by the node scheduler. The CPS scheduler has access to a buffer with classical messages from other nodes, activating `HostEvent` tasks. The QPS scheduler has access to the network schedule, determining allowed start times of `Pair` tasks.

5.6 Evaluation

All simulations were run on a machine using 80 Intel Xeon Gold cores at 3.9 GHz and 192 GB of RAM. Each subsection describes an independent evaluation (details in Section 5.13):

5.6.1 Demonstrating the architecture's effectiveness

We first validate the functionality of our architecture by demonstrating that applications of different CPS-QPS interactions types can successfully be executed on two or more nodes. Using our implementation (Section 5.5), we report that we successfully simulated the following applications: (A1) quantum key distribution (2 nodes, first QPS generating 10^3 EPR pairs followed by only CPS actions (classical computation and messaging)), (A2) blind quantum computation (1 client and 1 server node, first QPS generating 2 EPR pairs, then CPS performing rounds of classical messaging followed by local quantum gates by QPS), (A3) single-qubit teleportation across two nodes (1 sender and 1 receiver node, QPS generating one EPR pair followed by QPS measurement by the sender, CPS classical messaging and QPS local quantum gates by the receiver), (A4) a ping-pong application which repeats the single-qubit teleportation application to transfer states back and forth, and (A5) a multi-node GHZ-state [25] creation application (3 nodes, QPS creating a tripartite entangled state using multiple EPR pairs, using CPS classical messaging and QPS local quantum gates).

Each program is instantiated 1000 times and all tasks are immediately added to the task graph (since the the programs are *predictable* (Section 5.4.6)). Precedence constraints are added such that instances are executed sequentially for simplicity. We use a fixed network schedule (no demand registration (Section 5.4.5) since network schedule generation is not handled by Qoala itself and hence not part of the evaluation). To demonstrate the hardware independent performance of Qoala, all simulations are performed on three different hardware models: a generic quantum platform (uniform qubit connectivity and

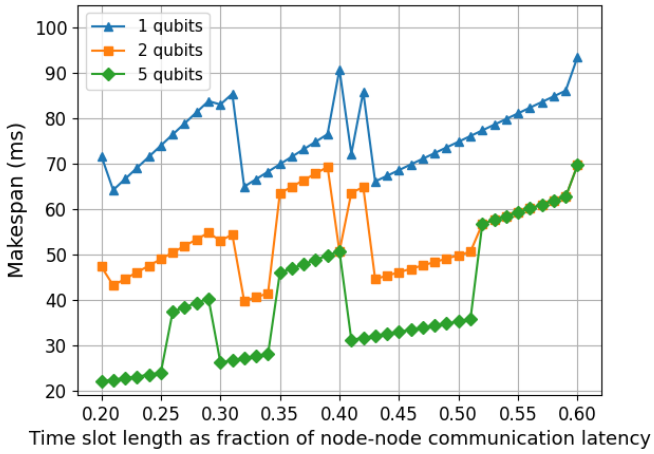


Figure 5.8: Self-preemption of a teleportation program. For certain durations of the time slot length (as fraction of node-node communication latency, x-axis), the makespan is considerably higher (spikes in the plot). Reason: a classical message arrives for some teleportation instance i, making the node scheduler choose to perform the local quantum gates for i. During this, the time slot for instance $j > i$ starts. Since the QPS is busy with i, it cannot work on entanglement generation for j. Therefore, j must wait for the next repetition of the network schedule, leading to a higher overall makespan.

vanilla (hardware-agnostic) NetQASM instruction set ([17] and Chapter 3), and two models based on data validated on real hardware (NV centers [8, 27] and trapped ions [33]). We observe successful execution (desired deterministic outcome when setting noise parameters (Section 5.3.1) to 0, and expected non-deterministic outcome distributions with realistic noise parameters) for all types of applications (details in Section 5.13).

5.6.2 Demonstrating Qoala's multitasking potential

Next, we demonstrate that Qoala can execute multiple instances of (different) programs concurrently by interleaving. We examine (1) makespan decrease (Section 5.3.1) when interleaving the instances compared to sequential execution, (2) whether makespan depends on the network schedule.

We first evaluate multitasking instances of the same application: teleportation (same as A3 in Section 5.6.1), 100 instances, with a fixed network schedule (no time slots; entanglement generation always allowed). Sequential scheduling of instances results in makespan $N \cdot CC$ while interleaved scheduling (tasks for all instances created at the same time; no precedence constraints between instances) results in $\lceil N/Q \rceil \cdot CC$ (number of instances N, classical node-node communication latency CC, number of available memory qubits at receiving node Q). We also evaluate the effect of network schedules with time slots (repeating pattern of slots assigned to A3 instances), and find that the time slots length influences the makespan (Figure 5.8) in a non-trivial manner due to instances pre-empting each other. BQC (same as A2 in Section 5.6.1, 100 instances) interleaved gives a makespan decrease over sequential of $(21\%, 56\%, 65\%)$ for (2, 5, 10) server qubits, respectively. The

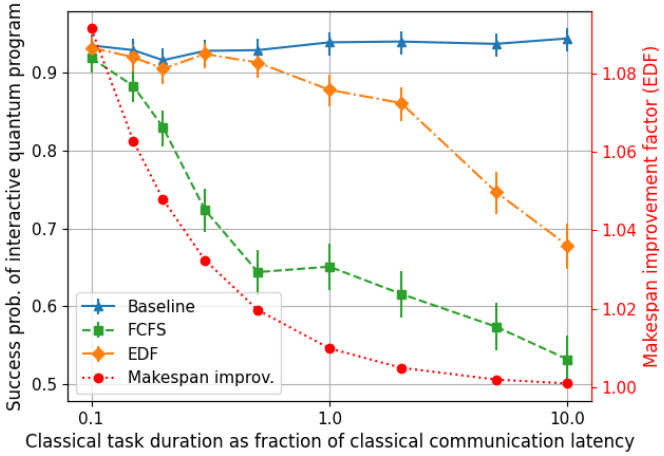


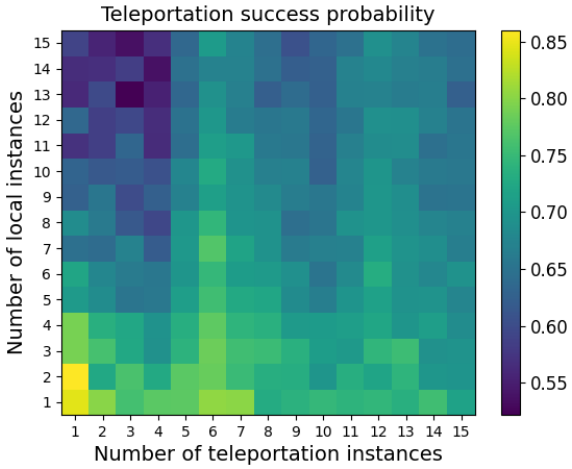
Figure 5.9: Execution of interactive quantum program in the presence of a 'busy' CPS program (tasks with duration $f \cdot CC$ for fraction f of the classical node-to-node latency CC, x-axis). **Comparison of schedulers:** *Baseline* (no scheduling nor interleaving), *FCFS*: first-come-first-serve scheduler (interleaving possible, no deadlines to prioritize quantum tasks), *EDF*: earliest-deadline-first scheduler (with deadlines to prioritize quantum tasks). The interactive program regularly waits (duration CC), with quantum states in memory, for incoming classical messages. Task interleaving allows busy CPS tasks to fill waiting times. EDF leads to higher success probability than FCFS, showcasing usefulness of deadlines. **Tradeoffs:** The baseline of sequential execution leads to the best possible success probability (quantum metric) at the expense of longest makespan. EDF allows a lowering of makespan (classical metric) at the expense of a lower succ. prob. (quantum metric).

network schedule affects the makespan decrease: doubling the time slot length results in a smaller decrease (12%, 48%, 48%).

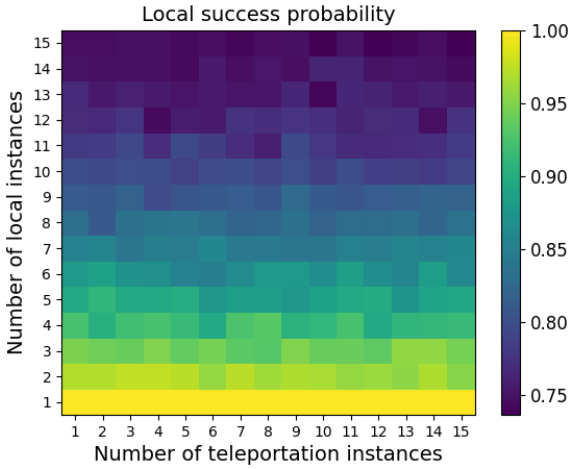
We then execute instances of different applications and again examine the effect of the network schedule on the makespan decrease: 50 QKD (A1 in Section 5.6.1) and 50 BQC (A2) instances give a makespan decrease of 9.5% (fixed schedule which first has time slots for QKD and then for BQC) and 39% (schedule with time slots alternating between QKD and BQC). We observe that multitasking can lead to improved (lower) makespan and that the network schedule can have considerable impact on the makespan.

5.6.3 Improvement over NetQASM architecture

We compare the Qoala architecture with the NetQASM runtime approach for executing programs on a node from [17] and show that Qoala provides new compilation possibilities (optimizing across classical and quantum code) and can lead to a better application execution makespan. We consider a remote measurement-based quantum computing program written in Python (the program format of the NetQASM runtime) which has suboptimal code logic on purpose. Executing this program in the NetQASM runtime performs worse (success probability 66%) than the same program but compiled manually into a Qoala program and executed in the Qoala runtime (succ. prob. 82%). We note that manual compilation allowed optimization that is not possible in the NetQASM program format, exemplifying the new compilation potential provided by Qoala.



(a)



(b)

Figure 5.10: Concurrent execution of teleportation (A3 in Section 5.6.1) and a local application (only preparing and measuring qubits). (a) Success probability of teleportation for different numbers of teleportation and local instances. More local instances lead to lower teleportation succ. prob. (effect more pronounced with few teleportation instances). (b) Success probability of local program. More local instances lead to lower local succ. prob., independent of the number of teleportation instances.

5.6.4 Tradeoffs between classical and quantum performance metrics

We compare different scheduling modes enabled by Qoala and evaluate tradeoffs between makespan and success probability, noting that the NetQASM runtime did not allow scheduling at all (Figure 5.9). We expect that interleaving of tasks reduces the makespan, but may lead to lower success probability due qubits degrading in memory while tasks wait for each other. We compare 3 scheduling modes: no scheduling (baseline), FCFS scheduling, and EDF scheduling. We consider a simple runtime scenario with (1) a local quantum program which alternates between doing local quantum gates and waiting for a remote classical message before continuing and (2) a classical 'busy program' consisting only of CPS tasks (duration defined as fraction of classical node-node latency). We find that (a) scheduling (FCFS or EDF) decreases success probability (EDF less than FCFS); impact larger for long task durations, but (b) EDF provides a better makespan than no scheduling. Note that the baseline necessarily gives the highest success probability due to no waiting, but at the expense of maximal makespan (sequential execution).

5.6.5 Success probabilities with quantum multitasking

Next, we consider a quantum multitasking scenario where we investigate trends in application success probability while varying the number of concurrent applications (Figure 5.10). In addition to a teleportation application (A3 in Section 5.6.1), the receiver node also executes multiple instances of a local quantum program (only applying quantum gates). Whenever the receiver node must wait for classical messages to come in for A3, it can work on its local quantum programs. We find that success probability of both types of programs decreases in the presence of another program.

5.6.6 Performance sensitivity

Finally, we investigate the influence of classical message-passing latencies, internal latencies, and network schedule contents on application success probability of BQC (A2, 100 instances). We find that the duration of sending classical messages between nodes has a large impact on the success probability: node-node latencies [0.01, 0.1, 1] times the qubit coherence time lead to success probabilities [0.89(2), 0.83(2), 0.54(4)], respectively. Internal latencies (between CPS and QPS, and between the scheduler and CPS or QPS) only have a significant impact when message-passing durations are low (0.01 times the qubit coherence time). We also compare different network schedules (simple linear repeating schedule where each client-server pair gets a time slot consecutively; slot length is varied). We obtain success probabilities [0.90(2), 0.69(3), 0.48(4)] for time slot lengths [0.01, 0.1, 1] times the qubit coherence time, respectively.

5.7 Conclusion

Qoala is the first architecture for executing quantum applications that addresses the need for scheduling and compiling hybrid classical-quantum programs for a quantum internet. This allows Qoala to ensure successful execution of quantum programs even in the presence of limited quantum memory lifetimes, and opens the door for a compile time optimization of the hybrid classical-quantum program. By building on an existing quantum network stack [16, 42] and the implementation of QNodeOS on quantum hardware [20,

42] (see also Chapter 4) we pave the way for the real-world implementation of Qoala in a platform-independent way on diverse hardware platforms including NV centers in diamond [42, 43] and trapped ions [32, 33]. Such an implementation may require, however, a new classical control hardware as opposed to [20, 42], e.g. by placing CPS and QPS on a single board with access to an on-chip shared memory.

Our work opens the door for further computer science research in executing quantum internet applications: *Advanced scheduling algorithms:* More sophisticated scheduling strategies may lead to higher success probabilities and lower makespan when concurrently executing multiple program instances, where inspiration may come from [1, 5, 41, 51]. In the quantum domain, missing the deadline will result in a degradation of the success probability as a function of the time by which the deadline was exceeded. This suggest the use of time-utility functions (TUF, see e.g. [29, 37]) to inform scheduling decisions, where it is an open question how such TUF could even be defined in the quantum domain. Our work also raises the question on what fundamental tradeoffs between the classical (makespan) and quantum (success probability) performance metrics are at all possible. *Compiler design:* Qoala's program format now allows for a compiler design that takes into account the hybrid and networked nature of programs. It is an open question to design compilers enabling effective code optimization and translation of different types of high-level code into executables. *Capability negotiation:* We assumed that the compiler provides advice that the nodes use in a capability negotiation and demand registration (Section 5.4.5). It is an open question how to best compute such advice, and find efficient protocols for negotiating capabilities and register demand. *Network schedule:* As expected, our evaluation shows that application performance depends on the network schedule, where we emphasize that ensuring network service is out of scope for Qoala as an environment for executing applications. This highlights a need for understanding the quality of service a quantum network should provide, as well as to design good network scheduling algorithms to satisfy them, in order to achieve good application performance.

5.8 Data availability

The implementation of Qoala as a simulator can be found online [44]. The code and data supporting the evaluation can be found at [14].

5.9 Program structure

This section provides details about the structure and contents of Qoala programs as described in Section 5.4.2.

5.9.1 Program representation and components

A Qoala program is represented in human-readable text format. This allows one to directly write Qoala programs, although our vision is that programmers write their code in a higher-level language, and that a compiler translates this into a Qoala program.

In the main text, some parts of example programs were omitted for brevity. In Figure 5.11 we show an example of a full Qoala program.

A Qoala program encompasses both classical and quantum code. These different code segments are put into different sections in the program. The host section contains Qoala-

Host code which is to be run on the CPS. The NetQASM section contains local routines (containing NetQASM instructions) which are meant to be run on the QPS. The request section contains specifications of requests for remote entanglement generation, to be handled by the QPS. Furthermore, there is a meta section which defines global information about the program. Each of these sections is explained in more detail below.

In all of the sections in a Qoala program, values may be replaced by a **template**. A template represents a value that is not defined for the program, but is filled in at program instantiation. For example, a QKD program might have a request object in its request section containing the entry `num_pairs: N`, where `N` is a template. This construction allows one to instantiate the same program with different values for `N`, and it is hence not needed to define separate programs for each different number of pairs to generate in the QKD program.

5.9.2 Program metadata

Program metadata contains:

- **Name**: The name of this program.
- **Parameters**: Global arguments to this program. These arguments may be used as templates (see above) in the program. Examples may be the name of a remote node, or the number of EPR pairs to generate.
- **Classical Sockets**: A mapping from IDs to remote node names. The IDs are local identifiers that can be used by Host code to distinguish different classical sockets.
- **EPR Sockets**: A mapping from IDs to remote node names. The IDs are local identifiers that can be used by Host code to distinguish different EPR sockets.

5.9.3 Host section

The host section contains code the be executed by the CPS. It consists of both local processing (like calculation and conditional logic), and communication (sending and receiving classical messages to and from other nodes in the network).

The language in which host code is represented is called QoalaHost. This is a low-level instruction set with well-defined semantics and types, and is meant to be executed by a virtual machine or interpreter. One can also imagine QoalaHost code to be translated (either ahead-of-time or at just-in-time) to native CPS code, such as x86 or ARM. However, for the sake of simplicity and of implementation independence, we treat here only the QoalaHost language and its semantics itself.

The QoalaHost (QH) language was designed to resemble intermediate representations as found in LLVM [34] and MLIR [35], such that integration with future compilers is accessible. Specifically, one may imagine a compiler that uses MLIR for its intermediate representation (IR). When this compiler then produces the host code of the program, the translation of its own IR to QoalaHost code should be straightforward.

Blocks. The Host section consists of a list of blocks. A block consists of a block metadata and a list of QH instructions.

The block metadata contains the following entries:

- **Name**: The name of this block. Host code can refer to this name in QH branch instructions.
- **Type**: one of CL, CC, QL or QC (see below).
- **Deadlines**: Deadlines relative to other blocks. The deadlines are specified in terms of EHI arguments. Upon program instantiation, concrete values are filled in based on the actual EHI value.
- **Time hints**: Duration estimate of executing the block. The estimates are specified in terms of EHI arguments. Upon program instantiation, concrete values are filled in based on the actual EHI value.

5.9.4 Block types

Blocks are categorized into the following four types:

- **CL**: Classical Local. The block contains only instructions that are classical, local and only involve the CPS
- **CC**: Classical Communication. CPS-only instructions, but starts with a 'receive message' instruction.
- **QL**: Quantum Local. The block contains calls to local routines.
- **QC**: Quantum Communication. The block contains calls to request routines.

QoalaHost Language. The QH Language describes a fixed set of QH instructions as well as QH Variable types. Host code is represented as blocks containing QH instructions. These instructions may be directly interpreted by a processor or OS.

All basic values are 32-bit signed integers (i32) or floating point values (f32). A variable in Host code can either be

- singleton variable, holding one basic value. Has a single name. E.g. `x`
- vector, holding an arbitrary number of basic values. Has a single name. E.g. `x<>`

The QH Language allows for expressing multiple variables in a single expression, called a *tuple*. A tuple holds a fixed number of basic values. E.g. `tuple<x, y, z>`.

Local Memory. Host code is assumed to have access to a local memory space that is logically organized as a mapping of *names* to *values*. For example, the local memory may at some point during execution contain the following items:

```
"var_x" -> 3
"my_vec" -> <1, 2, 5>
```

Shared Memory. The QH Language does not allow direct access to shared memory. Only variables from the local memory can be used. When calling and getting results from Local Routines (LRs) and Request Routines (RRs), values are automatically moved from local memory to shared memory. Shared memory is discussed in more detail in Section 5.10.3.

Block format
A block has the following format:

```
^#name {type = #type}:
    <list of QH instructions>
```

Example:

```
^b0 {type = CL}:
    x = assign() : 3
    return_result(x)
```

QH instructions
A full list of QoalaHost instructions is given in Figure 5.12.

5

5.9.5 NetQASM section

The NetQASM section consists of a list of local routines that are to be executed on the QPS. A local routine is only executed when it is called by host code using the `run_routine` instruction. A local routine may be run multiple times, again depending on the host code.

The instructions of a local routine are represented using the NetQASM 2.0 format. This is an updated format compared to NetQASM 1.0 as presented in [17] and Chapter 3.

NetQASM values. All values are 32-bit signed integers. Floating-point values are not supported. Angles for qubit rotations must be expressed as discrete values. Booleans are represented as follows: `true` is the 32-bit 0 value, `false` is the 32-bit 1 value. Any other 32-bit value is not a valid boolean. The reason for keeping the different types limited is to keep the QPS implementation simple.

NetQASM Local Memory The QPS is expected to have a local memory (only accessible by the QPS itself) consisting of 64 32-bit registers:

- 16 **R** registers: `R0` to `R15`
- 16 **C** registers: `C0` to `C15`
- 16 **M** registers: `M0` to `M15`
- 16 **Q** registers: `Q0` to `Q15`

The four groups of registers are not inherently different. A compiler producing NetQASM code may use a certain group only for certain values, but this is not mandatory.

Shared Memory See Section 5.10.3 for more information about Shared Memory and arrays. The QPS is expected to have access to Shared Memory (accessible by both the CPS and QPS). Two shared memory Arrays are available:

- an `@input` array, containing the LR input variables
- an `@output` array, with space to write the LR results to

The length of the `@input` array is equal to the number of LR parameters. The length of the `@output` array is equal to the number of LR return variables.

- The `@input` and `@output` arrays are the only arrays accessible from within the LR.
- The QPS can **only read** from the `@input` array (see `load` instruction below).
- The QPS can **only write** to the `@output` array (see `store` instruction below).

NetQASM Instruction Each instruction consists of the instruction type followed by a list of operands. The text form of an instruction is:

```
instr_name  op0 op1 ... opn
```

where the number of operands can be 0 or more (no limit).
A list of all NetQASM 2.0 instructions can be found in Figure 5.13.
These instructions can be classified as:

- shared memory access: `load` for reading LR inputs, `store` for writing LR results
- classical logic and control-flow: like `set` , `add`, or `jmp`
- quantum operations: gates from a specific flavor [17]

NetQASM instructions representing quantum operations are either *core instructions* or *flavor-specific* instructions. Core instructions are quantum hardware independent and are expected to be compatible with any QPS implementation. On top of the core instructions, flavor-specific instructions may be added and supported by a specific QPS implementation. For example, a QPS that controls an NV-centre may support NetQASM instructions of the NV flavor, which contain gate operations only available on this particular quantum hardware. Which NetQASM instructions are supported by the QPS is exposed to higher layers (including a compiler) as part of the EHI (see Section 5.10.5). Using this information, a compiler may produce optimized NetQASM code using the flavor-specific NetQASM instructions.

Note that NetQASM 2.0 **does not** contain (in contrast to NetQASM 1.0 [17]):

- Allocation instruction (`qalloc` in NetQASM 1.0): The memory manager allocates virtual qubits based on the LR header information. Note that qubit allocation is different from *qubit initialization* (`init` instruction).
- Instructions for EPR generation: This is handled by request routines.
- Waiting instructions: Waiting is handled by the scheduler choosing which tasks to execute when.

Local Routine A Local Routine (LR) represents a block of local program operations that are executed on the QPS. An LR is:

- local: there is no interaction whatsoever with external nodes or controllers
- atomic: execution of an LR cannot be pre-empted; when the QPS start executing an LR, it will not do anything else until the LR has finished (unless an abort happens)

An LR consists of a *header* and a *body*. The header contains metadata such as the resource usage of the LR, and its input/output interface. The body contains the actual instructions in the form of NetQASM code.

Arguments and Returns. An LR may have zero or more *arguments*: values that are provided to the LR only at runtime. They can be seen as inputs or parameters to the LR. These values appear in the `@input` array in shared memory, and are put there by the CPS.

An LR may also have zero or more `returns`: values that are provided by the LR only at runtime. They can be seen as outputs or results of the LR. These values must be written to the `@output` array in shared memory, and can then be used by the CPS.

Arguments and returns are always 32-bit signed integers. There is no limit to the number of arguments and returns an LR may have.

Local routine header. A Local routine (LR) header contains the following entries:

- **Name**: The name of this LR. Host code refers to this name in a `run_routine` QoalaHost instruction.
- **Uses**: A list of virtual qubits IDs. These refer to all virtual qubits that are used by this LR. At runtime, the memory manager makes sure that these virtual qubits are allocated before execution of the LR starts. (They may already have been allocated earlier; alternatively the memory manager allocates them just before the LR starts.)
- **Keeps**: A list of virtual qubit IDs. These refer to all virtual qubits that should *remain allocated* after finishing the LR. (They may e.g. be used in subsequent LRs.)
- **Args**: A list of names for the arguments of the LR. They are in the same order as how their values are accessible from the `@input` Array.
- **Returns**: A list of names for the returns of the LR. They are in the same order as how their values are put into the `@output` Array.

Quantum memory usage annotations. The LR header indicates which virtual qubits are used and freed by the LR. This makes it possible for the scheduler to decide which `LocalRoutine` task it may schedule when. For more information, see section Section 5.11 on scheduling. The following listing provides an example:

```
SUBROUTINE subrt1
    uses: 0, 1
    keeps: 0
    returns: m0
    <rest omitted>
  NETQASM_START
    set Q0 0
    set Q1 1
    init Q0
    init Q1
    cnot Q0 Q1
    meas Q1 M1
    store M1 @output[0]
  NETQASM_END
```

This local routine initializes virtual qubits 0 and 1 and then applies a CNOT gate on them. It measures qubit 1 and stores the output in the `@output` array which can then be accesses by host code using the name `m0`. Using the metadata, a scheduler knows the following information even before executing this LR: virtual qubits 0 and 1 need to be free before this LR can run, and after running the LR, qubit 1 is free (again) but qubit 0 remains occupied.

It is the responsibility of the compiler to make sure that the use and free values correspond to the actual NetQASM code.

5.9.6 Request section

The callback (which is an LR) can have zero or more arguments (just like standard LRs). The runtime values of these arguments are provided by the QPS directly. A Request Routine (RR) may have zero or more returns: outputs or results of the entire RR. The only allowed results at this moment are measurement outcomes in case of Measure Directly requests. RR callbacks can have (just like standard LRs) zero or more returns.

Request routine header A Request routine (RR) header contains the following entries:

- **Name**: The name of this RR. Host code refers to this name in a `run_request`.
- **Returns**: A list of names for the returns of the RR. Since the returns can only be measurement outcomes, these names are either (1) the name of a single QoalaHost vector variable which will hold all outcomes, or (2) a list of names for each individual outcome stored in its own QoalaHost int variable.
- **Callback type**: Either `sequential` or `wait_all`. Sequential means that the callback of this RR is executed for each generated pair, before the next pair is generated. Wait-all means that the callback is only executed once, namely when all pairs have been generated.
- **Callback**: The name of the LR that acts as the callback for this RR. Can be empty (no callback is used).

Request Parameters

- **Remote ID**: The node ID of the remote node with which to generate entanglement.
- **EPR Socket ID**: The ID of the EPR Socket to use.
- **Number of pairs**: The number of entangled pair to generate.
- **Virtual IDs**: A specification of the virtual IDs to assign to the entangled qubits. This may be in one of three formats:
  - `all <N>`: all qubits get virtual ID `<N>`. This might be used when a sequential callback is used that measures the qubit immediately after generating; thereby freeing up virtual ID `<N>` immediately for the next pair
  - `increment <N>`: the first generated qubit gets ID `<N>`, the next `<N> + 1`, etc.
  - `custom <N1, N2, ...>`: a custom list of IDs that should have the same length as the number of pairs
- **Fidelity**: The desired fidelity `F` of the generated pairs. If this request routine is for multiple pairs and the callback type is `wait_all`, this value is used to specify that all pairs, after they have all been created, should have fidelity at least `F`. (How this is realized, which may involve multiple retries, is up to the network stack implementation in the QPS.)
- **Type**: Create and Keep (`create_keep`), Measure Directly (`measure_directly`), or Remote

State Preparation (`rsp`) [16].

- **Role**: `create` or `receive`. These roles are used to break symmetry between two nodes participating in entanglement generation (they should always have different roles). The 'create' node is the initiating one.

```
META_START
    name: bob
    parameters: alice_id
    csockets: 0 -> alice
    epr_sockets: 0 -> alice
META_END

^b1 {type = QC}:
    run_request(vec<>) : req
^b2 {type = QL}:
    vec<m0> = run_routine(vec<>) : post_epr
^b3 {type = CL}:
    return_result(m0)

SUBROUTINE post_epr
    params:
    returns: m0
    uses: 0
    keeps:
    request:
  NETQASM_START
    set Q0 0
    meas Q0 M0
    store M0 @output[0]
  NETQASM_END

REQUEST req
  callback_type: wait_all
  callback:
  return_vars:
  remote_id: {alice_id}
  epr_socket_id: 0
  num_pairs: 1
  virt_ids: all 0
  timeout: 1000
  fidelity: 1.0
  typ: create_keep
  role: receive
```

Figure 5.11: Example Qoala program which creates an EPR pair with remote program Alice, measures the local qubit, and returns the classical outcome value. *Meta section.* Defines the name of this program, global arguments (in this case: the node ID of the Alice program), classical sockets used (mapping local socket ID to name of remote node, and EPR sockets used (similar mapping)). *Host section.* In this example: consists of three blocks (`b1`, `b2`, `b3`). `b1` calls request routine `req` (no result values). `b2` calls local routine `post_epr`, resulting in a classical vector with one value (`m0`). `b3` returns `m0` as the result of this program. *Local routines section.* Consists of a single local routine called `post_epr`. It requires the virtual qubit (see Section 5.10) with ID 0 to be allocated, and acts on this qubit. Upon finishing the local routine, this qubit is not in use anymore (the `keeps` entry is empty). The NetQASM code represents measuring the qubit, and then storing the result (in register `M0`) to the `@output` array (see Section 5.10), which is in shared memory and can be accessed by host code by the name `m0`. *Request routines section.* Consists of a single request routine called `req`. It represents a request to the network stack for generating a single entangled pair (`num_pairs` is 1), which is kept in memory (`typ: create_keep`; not measured immediately). This program acts as a 'receiver' for entanglement generation (`role` attribute), which breaks symmetry in the entanglement generation process (the remote Alice program must have `role: sender`). Symmetry breaking is needed for the network stack to organize the entanglement generation. No callbacks are used, and all qubits (in this case: one) are stored in virtual qubit 0.

Instruction	Syntax	Semantics
Assign	`%name = assign() : <i32>`	Put the immediate value <i32> in variable %name
Add	`%result = add(%op1, %op2)`	Add the values of variables %op1 and %op2 and store the result in variable %result
Subtract	`%result = sub(%op1, %op2)`	Subtract the value of variable %op2 from the value of variable %op1 and store the result in variable %result
Multiply constant	`%result = mult_const(%op) : <i32>`	Multiply the value of variable %op with the immediate <i32> and store the result in %result
Bit-conditional multiply constant	`%result = bcond_mult_const(%op1, %op2) : <i32>`	• If %op2 is 0, do nothing • If %op2 is 1, multiply the value of variable %op1 with the immediate <i32> and store the result in %result • If %op2 is any other value, Undefined Behavior
Jump	`jump() : #name`	Jump to the Block with name #name
Branch if equal	`beq(%op1, %op2) : #name`	Jump to the Block with name #name only if the values of variables %op1 and %op2 are equal
Branch if not equal	`bne(%op1, %op2) : #name`	Jump to the Block with name #name only if the values of variables %op1 and %op2 are not equal
Branch if greather than	`bgt(%op1, %op2) : #name`	Jump to the Block with name #name only if the value of variable %op1 is greater than the value of variable %op2
Branch if less than	`blt(%op1, %op2) : #name`	Jump to the Block with name #name only if the value of variable %op1 is less than the value of variable %op2
Send message	`send_msg(%csck, %op)`	Send the value in variable %op over the classical socket to another node
Receive message	`%msg = recv_msg(%csck)`	Block and wait until a message is received over the classical socket and store its value in the variable %msg
Run Local Routine	`%result = run_routine(%args) : #name`	Run the Local Routine named #name with arguments %args and store results in %result
Run Request Routine	`%result = run_request(%args) : #name`	Run the Request Routine named #name with arguments %args and store result in %result
Submit Routines	`submit_routines(%args) : [#name]`	Submit a batch of Local and/or Request Routines with names [#name] for execution, with arguments %args
Join on Routine results	`%results = join_routines() : [#name]`	Block until the results are available for all Local and/or Request Routines with names [#name] and store the results in %results

Figure 5.12: Overview of all host code (QoalaHost) instructions, their syntax and their semantics.

Instruction	Syntax	Semantics
Set	`set <reg0> <imm0>`	Put the immediate value <imm0> in register <reg0>
Add	`add <reg0> <reg1> <reg2>`	Add the values in <reg1> and <reg2> together and store the result in <reg0>
Subtract	`sub <reg0> <reg1> <reg2>`	Subtract the value in <reg2> from the value in <reg1> and store the result in <reg0>
Multiply	`mul <reg0> <reg1> <reg2>`	Multiply the values in <reg1> and <reg2> together and store the result in <reg0>
Quotient	`quot <reg0> <reg1> <reg2>`	Divide the value in <reg1> by the value in <reg2> and store the quotient in <reg0>
Remainder	`rem <reg0> <reg1> <reg2>`	Divide the value in <reg1> by the value in <reg2> and store the remainder in <reg0>
Jump	`jmp <imm0>`	If the current instruction index (in the LR) is n, the next instruction to execute is at index n + imm0
Branch if equal	`beq <reg0> <reg1> <imm0>`	If the value in <reg0> is equal to the value in <reg1>, the next instruction to execute is at index n + imm0, otherwise it is at n + 1
Branch if not equal	`bne <reg0> <reg1> <imm0>`	If the value in <reg0> is not equal to the value in <reg1>, the next instruction to execute is at index n + imm0, otherwise it is at n + 1
Branch if greater or equal	`bge <reg0> <reg1> <imm0>`	If the value in <reg0> is greater than or equal to the value in <reg1>, the next instruction to execute is at index n + imm0, otherwise it is at n + 1
Branch if less or equal	`ble <reg0> <reg1> <imm0>`	If the value in <reg0> is less than or equal to the value in <reg1>, the next instruction to execute is at index n + imm0, otherwise it is at n + 1
Load from shared memory	`load <reg0> @input[<reg1>]`	Load the value at index "value of <reg1>" in the input array and store the value in <reg0>
Store to shared memory	`store <reg0> @output[<reg1>]`	Store the value of <reg0> in the output array at index "value of <reg1>"

Figure 5.13: Overview of all NetQASM classical instructions, their syntax and their semantics. Quantum instructions depend on the particular flavor [17] that is being used.

5.10 Runtime environment

In this section we provide more information about the runtime environment described in Section 5.4.3. Figure 5.18 provides an overview of the runtime architecture.

5.10.1 Program instantiation

A program instance is a Qoala program with additional runtime- and context-specific information that is supplied when preparing execution of the program. A program instance represents a single execution of a Qoala program.

The additional information consists of: concrete values for the global arguments of the program, the Exposed Hardware Info (EHI), an explicit Unit Module (see below), and results from capability negotiation.

Based on the above additional information, a program instance can be created which has the following properties:

- **Program ID**: A unique ID for distinguishing multiple program instances that all need to be scheduled and run.
- **Program**: The static Qoala program (without runtime information).
- **Program Inputs**: The values for the program's global arguments.
- **Unit Module**: The virtual quantum memory space that this program instance may use at runtime.
- **Timing Information**: Deadlines for individual tasks. Computed using both the program's timing hints and information from the EHI.

Figure 5.14 provides a schematic example of program instantiation.

5.10.2 Program versus program instance

A program is typically the output of a compiler. For example, a compiler might produce a BQC-server program, including global arguments for the remote ID of the client (i.e. the client ID is *not* hardcoded into it). A program instance represents a single execution of a Qoala program with concrete values for its global arguments. For instance, the client ID now has the explicit value of 3, since the remote client happens to have node ID 3. Often many program instances may be created for a single program. For example, if 1000 runs of the BQC program are desired, 1000 program instances are created based on the single Qoala program.

Batches. A program may be submitted for execution in a batch.

A batch B consists of a program P, the number of execution N and inputs for each execution. Based on this, N program instances are created.

5.10.3 Shared memory

The CPS and QPS need to exchange information in order to execute local routines and request routines. They do so using shared memory. The CPS writes routine arguments and reads results. The QPS reads routine arguments and writes results.

Conflicts in writing and reading are avoided by the runtime itself (it is not assumed the hardware itself enforces read-only or write-only regions of memory). This is achieved

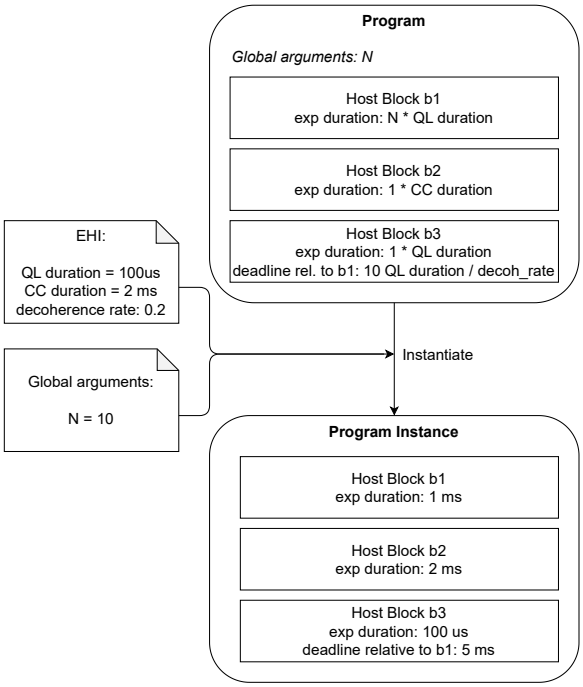


Figure 5.14: Schematic example of program instantiation. A program containing global arguments (N) is instantiated using a concrete value for the arguments ($N = 10$) and the EHI (containing values for the expect duration of a QL block, the expected duration of a CC block, and the qubit noise parameter expressed as the *decoherence rate*). This results in a program instance for which the expected durations have concrete values.

by strict read/write rules in Qoala: certain regions can only be written to by the CPS (QPS) while only be read from the QPS (CPS). No region can be written to by both CPS and QPS. Note that this design leaves open how the shared memory can be implemented: either as real physical shared memory, or as a message passing protocol.

Arrays. The shared memory is logically divided into *array elements* that can be allocated only by the CPS (Figure 5.15). Each element can hold a single 32-bit signed integer. The CPS can allocate shared memory space by specifying a *size*, resulting in an allocated array. An array is an ordered list of array elements. One can think of an array being a region in Shared Memory consisting of a consecutive list of elements.

Shared Memory is similar to the heap in classical OSes. Allocating an array is similar to `malloc` in C. Each program instance has its own view in the global shared memory, just like in classical OS, each program instance (or 'process') has its own virtual memory space.

Elements that have been allocated but never written to have an undefined value.

An array may be named; it is written as `@arrayname`. An element in an array at index `i` is written as `@arrayname[i]`. This notation is used in NetQASM(Section 5.9.5).

Arrays are used to share data between the CPS and the QPS. They are used for executing both LRs and RRs.

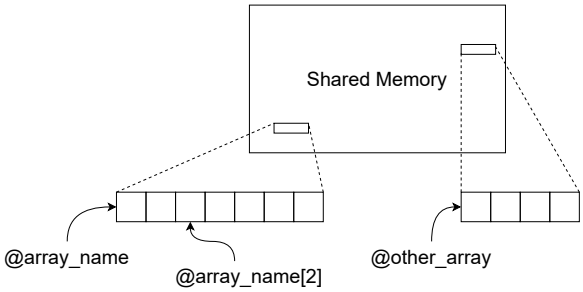


Figure 5.15: Schematic overview of shared memory, which is organized as *arrays*. Arrays are allocated by the CPS with a certain size (the number of *array elements*). Each array element holds a single classical value. Arrays are identified using the `[@<name>]` syntax. Particular array elements may be accessed using the `[index]` syntax.

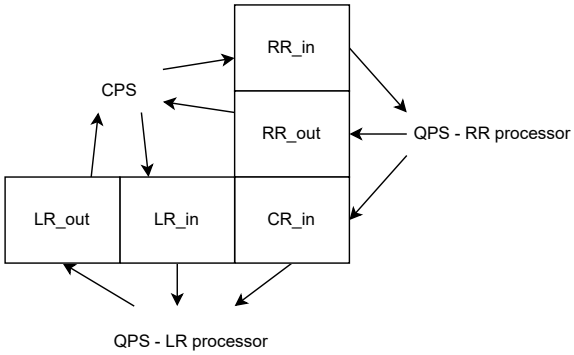


Figure 5.16: Shared memory regions. The CPS writes local routine arguments to the `LR_in` section and request routine arguments to the `RR_in` section. The CPS reads local routine results from the `LR_out` section and request routine results from the `RR_out` section. The QPS reads local routine arguments from `LR_in` and write results to `LR_out`. The QPS reads request routine arguments from `RR_in` and write results to `RR_out`. Callbacks for request routines use the separate `CR_in` section to use request routine results as arguments of the callback local routine.

The shared memory is logically divided into 5 *regions* (Figure 5.16). Each of the regions contains array elements, and in each region, arrays can be allocated. The regions are only a logical division, where each arrays in a certain region are only used to hold data for a specific use-case:

- `LR_in`: Argument values for LRs. CPS writes, QPS reads.
- `LR_out`: Result values for LRs. CPS reads, QPS writes.
- `RR_in`: Argument values for RRs. CPS writes, QPS reads.
- `RR_out`: Result values for RRs. CPS reads, QPS writes.
- `CR_in`: Argument values for callback LRs. QPS reads, QPS writes.

Arrays for local routines. Before an local routine (LR) can be executed, two arrays must be allocated by the CPS:

- An array in the `LR_in` region. Its size needs to match the number of arguments for the LR.
- An array in the `LR_out` region. Its size needs to match the number of results of the LR.

The array in the `LR_in` region can be accessed by the NetQASM code in the LR body using the name `@input`. The array in the `LR_out` region can be accessed by the NetQASM code in the LR body using the name `@output`.

Note that each program instance allocates (at runtime) its own arrays. Each individual LR in each individual program instance has access to two arrays called `@input` and `@output`, but in practice there can hence be multiple "input" and "output" arrays, each occupying a different part of the global Shared Memory.

Arrays for request routines. Before a request routine (RR) can be executed, multiple arrays must be allocated by the CPS:

- An array in the `RR_in` region.
- An array in the `RR_out` region. Its size needs to match the number of names in the "Results" entry in the RR header.
- An array in the `CR_in` region. Its size needs to match the number of arguments for the callback LR of the RR.

The results of the RR are written to the array in the `RR_out` region. Arguments to the callback LR are written to the array in the `CR_in` region.

5.10.4 Quantum memory

The QPS is assumed to have access to a quantum random access memory (QRAM) consisting of *qubits*. Each qubit is a single location in the QRAM and can hold a single 2-dimensional quantum value, like $|0\rangle$ or $|+\rangle$.

We distinguish between (1) the *physical quantum memory space (PQMS)* consisting of *physical qubits* and (2) a *virtual quantum memory space (VQMS)* for each program instance (Section 5.4.3).

The topology (qubit connectivity) and noise characteristics of the PQMS are exposed as part of the EHI. Each program instance has access to its own VQMS, which is represented as a Unit Module ([17] and Chapter 3). The VQMS for each program instance is created when instantiating the program. This can be seen as virtual memory allocation for the program. At runtime, the VQMS of each running program instance is mapped to the PQMS.

Unit Modules. A Unit Module (UM) describes the topology of a VQMS as well as its noise characteristics. That is, a UM contains:

- **Qubit Info**: a list of all qubits available in the VQMS, with for each qubit the following information: its virtual ID, whether it is a communication qubit or not, and its decoherence rate per second.
- **Gate Info**: a list of all quantum gates and quantum local operations available for the qubits in the VQMS, with for each item the following information:

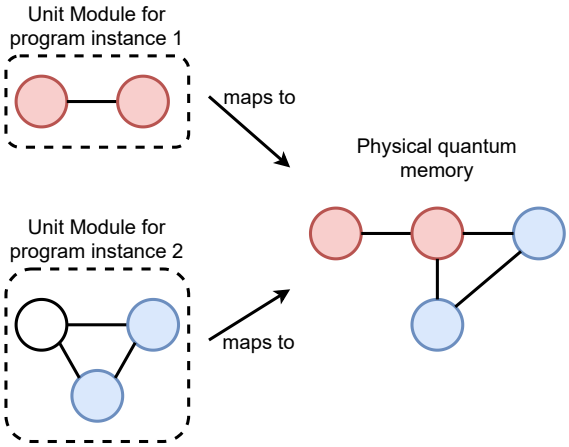


Figure 5.17: Example of a physical quantum memory available in a node (four qubits) and two allocated unit modules. The colors of the qubits represent the physical locations they map to. Note that the top-left qubit of Unit Module 2 is not currently mapped and that it also cannot be mapped. Therefore, tasks that require program instance 2 to use a third qubit cannot be executed at this time.

- Which NetQASM instruction it is represented by (may be in a particular NetQASM flavor).
- On which sets of qubits the gate or operation can be applied.
- Its duration.
- The decoherence rate per second on each of the qubits it acts on.

A UM can be seen as a subset of the full EHI of a node, specifically containing a subset of all qubits available in the node.

Qubits in the Unit Module are called *virtual qubits*. They are identified by their *virtual IDs* and are mapped to physical qubits (Figure 5.17).

Memory manager. Quantum memory allocation and freeing is handled by a memory manager, which lives in the QPS. The memory manager keeps track of the unit modules of all program instances, and maps virtual qubits to physical qubits.

Before starting a local routine or request routine, the memory manager allocates the corresponding qubits. For example, if a local routine for program instance P defines in its metadata (see Section 5.9) that it uses virtual qubits 0 and 1, the memory manager allocates virtual qubits 0 and 1 (if not already allocated). This involves finding currently unused physical qubits and mapping new virtual qubit to these free physical qubits.

5.10.5 Exposed hardware interface

The Qoala execution environment exposes certain information related to the hardware and software capabilities. This information includes noise characteristics of quantum memory and of entanglement generation, as well as estimates of classical latencies.

All information that is exposed falls under the Exposed Hardware Interface (EHI). The EHI can be divided into *node info* and *network info*.

EHI Node Info. The EHI node info consists of:

- **Qubit Info**: a list of all qubits available at the node, with for each qubit the following information: (1) its ID, (2) whether it is a communication qubit or not, and (3) its decoherence rate per second.
- **Gate Info**: a list of all quantum gates and quantum local operations available at the node, with for each item the following information: (1) which NetQASM instruction it is represented by (may be in a particular NetQASM flavor, (2) on which sets of qubits the gate or operation can be applied, (3) its duration, and (4) the decoherence rate per second on each of the qubits it acts on.
- **NetQASM flavor**: a list of all supported NetQASM instructions. All NetQASM instructions mentioned in Gate Info must be in this list
- **Classical latencies**: Covers (1) duration of executing a single QH Instruction, and (2) duration of executing a classical NetQASM instruction (Note that the duration of quantum operations is covered by the Gate Info).

EHI Network Info. The EHI network info consists of **Link Info** for each link in the network, with (1) the expected duration of generating an entangled pair on this link, and (2) the expected fidelity of generating an entangled pair on this link.

5.10.6 Sockets

Connections with remote nodes are modeled as *sockets*. Each program instance running on a node has access to classical sockets an EPR sockets. Classical sockets represent an endpoint for connections over which classical messages can be sent. A program instance can have classical sockets with any other nodes in the network.

An EPR socket represents an endpoint of a quantum connection. Through the EPR socket, a program can ask for entanglement with a remote node.

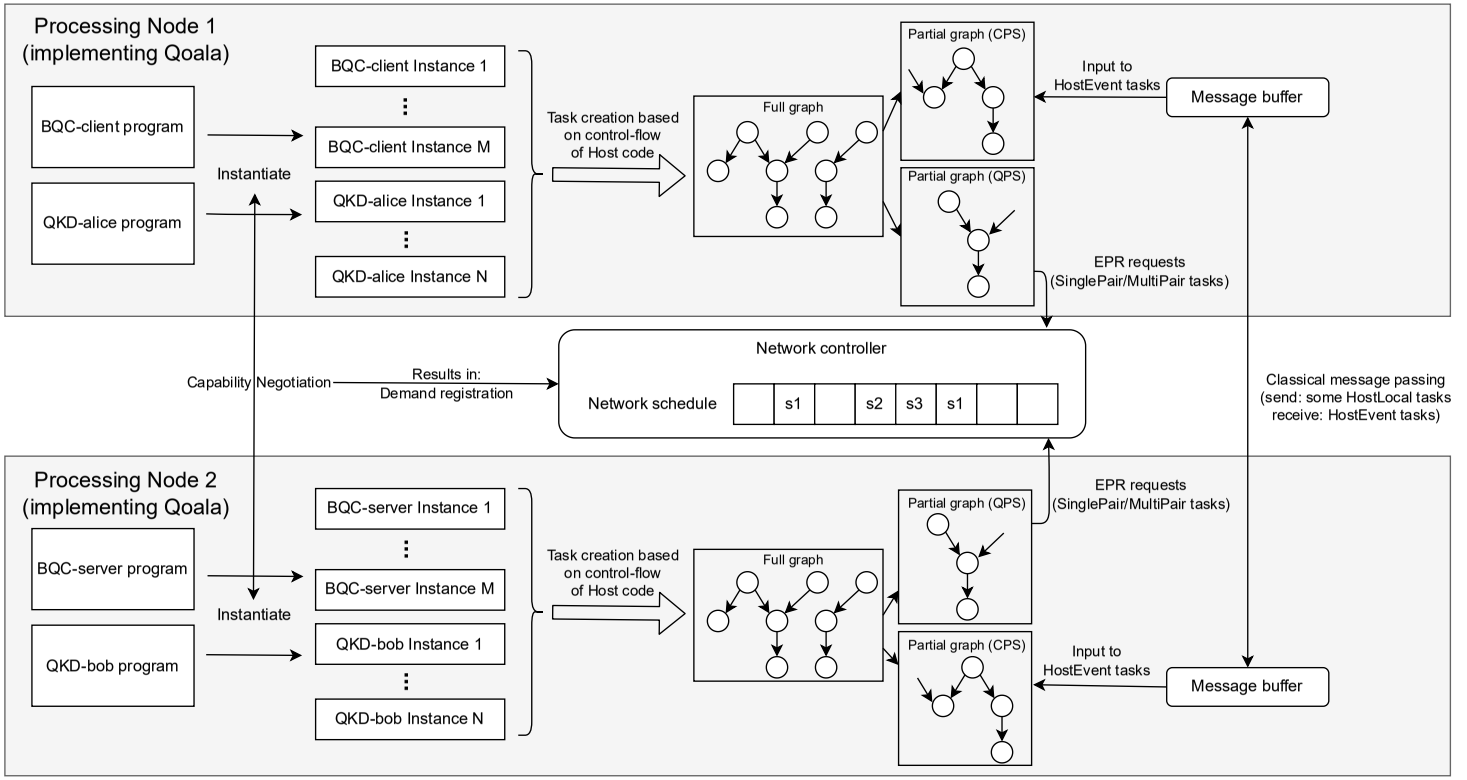


Figure 5.18: Detailed Qoala runtime overview.

5.11 Scheduling and execution

This section provides more details about tasks, task creation and scheduling (Section 5.4) as well as about our scheduler implementation Section 5.5.

5.11.1 Tasks

Task creation. Tasks are created based on the blocks in a program. Specifically, a block B in the program is mapped to a set $T(B)$ of tasks. Since a block may be executed multiple times, multiple instances of $T(B)$ can be created at runtime.

CL and CC blocks are mapped to CPS tasks only. QL and QC blocks are mapped to a sequence of CPS- and QPS tasks.

- CL block. A single `HostLocal` task is created.
- CC block. A single `HostEvent` task is created.
- QL block. If there is a single `run_routine` call, a `LocalRoutine` task is created for the QPS, as well as a `PreCall` tasks and a `PostCall` task for the CPS. Two precedence constraints are added: the `PreCall` task precedes the `LocalRoutine` task, and the `LocalRoutine` task preceded the `PostCall` task. If there is a `join_routines` on multiple local routine, multiple `PreCall`-`LocalRoutine`-`PostCall` task sets are created, without any dependencies between the task sets.
- QC block. If the request that is called from this block is for a single pair, a `SinglePair` task is created. If the request is for more than 1 pair, a `MultiPair` task is created. In both cases, an additional `PreCall` and a `PostCall` task are created with precedence constraints like for QL blocks. If there is a `join_routines` on multiple request routines, multiple `PreCall`-Pair-`PostCall` task sets are created, without any dependencies between the task sets.

Figure 5.20 shows an overview of blocks and corresponding tasks and their precedence constraints.

Predictable vs unpredictable programs. Tasks are created based on the contents of a program instance, and their precedence relations are defined by the control-flow of the blocks in the program's host code. Because of jump and branch instructions in the host code, a block may be executed zero, one or multiple times. Furthermore, the exact number of executions of a block may not be known ahead of time. For example, a program might loop through a sequence of blocks by using a conditional branch instruction at the end of the last block of the sequence. The condition could depend on a runtime value (such as the result of a quantum measurement). We say that control-flow is *predictable* if it can be completely known before runtime. *Unpredictable* control-flow, on the other hand, depends on values available only at runtime. For predictable programs, all its tasks can be created before runtime. For unpredictable programs, (some of) its tasks must be created on-the-fly during program execution. Figure 5.19 illustrates the difference between predictable and unpredictable programs.

Task execution. Tasks are executed by the CPS or the QPS, and the specific operations involved depend on the type of the task.

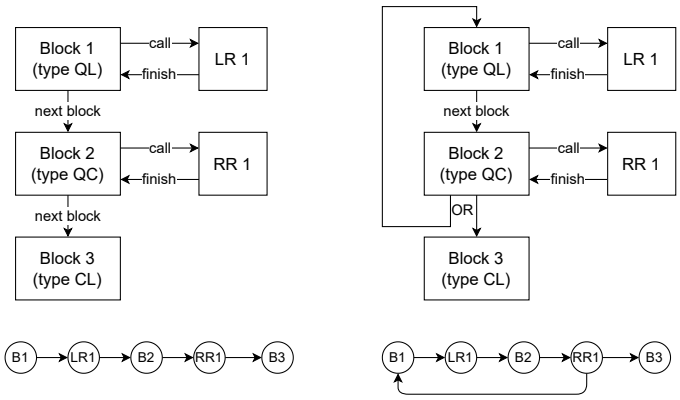


Figure 5.19: Schematic overview of the difference between predictable and non-predictable programs. The control-flow of the predictable program (left) is linear: first block 1 is executed (calling local routine (LR) 1), then block 2 (calling request routine (RR) 1), and finally block 3. Therefore, the number of tasks is fixed and known before execution. The non-predictable program is similar but after executing block 2, control-flow may go back to block 1 (again), depending on a runtime value (e.g. the result of RR 1). Hence, the number of times that blocks 1 and 2 are executed is not known beforehand, and therefore the number of tasks is also not known.

5

`HostLocal` *task execution.* A `HostLocal` task $t_{hl} = (P, B)$ for program instance P and block B is handled by executing each of the instructions in B. When the task finishes, the name of the next block to execute is recorded. If B ends with a branch instruction, this is the target block; otherwise it is the next block in the program (if this was the last block, the next block is nil).

`HostEvent` *task execution.* A `HostEvent` task represents a block B of type CC, which must start with exactly one `recv_cmsg` instruction. Handling the task involves reading a message from the message buffer and assigning it to the result variable of the receive instruction. Then, the remaining instructions in B are executed just like in a `HostLocal` task.

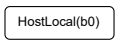
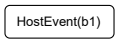
`PreCall` *task execution.* A `PreCall` task corresponds to a LR call instruction in Host code. The CPS allocates space in the shared memory for arguments and results. It then writes argument values to the shared memory.

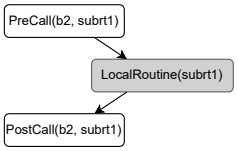
`PostCall` *task execution.* A `PostCall` task corresponds to a LR call instruction in Host code. The CPS reads the results from the shared memory and copies them to the corresponding variables in the host local memory.

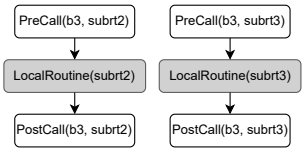
`LocalRoutine` *task execution.* A `LocalRoutine` task is executed by the QPS. It involves the following steps. First, based on information in the uses/keeps metadata, virtual quantum memory is allocated. Then all NetQASM instructions are executed, which may involve loading values from shared memory (reading arguments) and storing values to shared memory (populating results). Finally, quantum memory is freed.

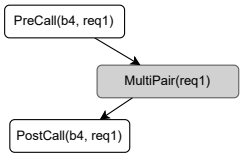
`SinglePair` *task execution.* A `SinglePair` task is executed by the QPS. First, arguments are read from shared memory. Then, an EPR request (see Section 5.11.5) is sent to the network controller.

`MultiPair` *task execution.* A `MultiPair` task is executed by the QPS. First, arguments

Host code	Task types	Example host code	Task graph based on example host code
CL block	HostLocal	`^b0:` `  send_cmsg(m)`	
CC block	HostEvent	`^b1:` `  m = recv_cmsg()`	

QL block	PreCall
	LocalRoutine
	PostCall
		`^b2:`
		`  m = run_subroutine(x) :`
		`subrt1`
			

QL block with a join	**Multiple times:**
	PreCall
	LocalRoutine
	PostCall
		`^b3:`
		`  submit_subroutine() : subrt2`
		`  submit_subroutine() : subrt3`
		`  join_routine(subrt2, subrt3)`
			

QC block without callback	PreCall
	MultiPair
	PostCall
		`^b4:`
		`  run_request() : req1`
		
		`REQUEST req1:`
		`  callback:`
			

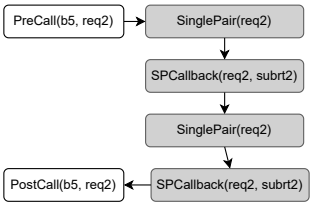
QC block with callback	PreCall
	MultiPair
	MPCallback
	PostCall
		`^b5:`
		`  run_request() : req2`
		
		`REQUEST req2:`
		`  callback: subrt2`
		`  callback_type:`
		`sequential`
		`  num_pairs: 2`
			

Figure 5.20: Overview of different host blocks with corresponding tasks. In the rightmost column, tasks with a dark background are QPS tasks, the others are CPS tasks. This example shows that tasks contain data about the program segment they correspond to, such as `LocalRoutine` tasks having the name of the routine they are executing.

are read from shared memory. Then, multiple EPR requests are sent to the network controller. Whether these requests are all sent at once or consecutively and waiting for intermediate responses is up to the implementer; the choice may depend on efficiency and resource considerations.

`SinglePairCallback` *and* `MultiPairCallback` *task execution.* First read results (from a `SinglePair` or `MultiPair` task) from shared memory. Then execute the callback routine just like a `LocalRoutine` task.

Deadlines. Deadlines can be specified for blocks relative to other blocks using the syntax:

```
^block_0:
    ...

^block_1 { deadlines = [b0: 3ms] }:  // relative deadline of 3 ms compared to
 block_0
    ...
```


A relative deadline to some block B is always with respect to the last task in $T(B)$, for the last task set instance (in case of multiple execution of this task set). The deadline value may be an explicit value (like $3ms$) or it can be in terms of EHI values, such as for example $0.1 * CC$ where CC is the expected classical node-node latency provided by the EHI.

Precedence constraints. By default, blocks are executed in the order they are given in the program. Blocks ending with a jump or branch instruction define precedence constraints at runtime for unpredictable programs.

Scheduling happens at runtime and involves choosing which task to execute next. In Qoala, there are three schedulers per node: the *CPS scheduler* controls task execution on the CPS, the *QPS scheduler* controls task execution on the QPS, and the *node scheduler* controls the CPS- and QPS schedulers. The CPS- and QPS schedulers are both processor schedulers.

5.11.2 Scheduling

In this and the following sections we describe the scheduler from our implementation (Section 5.5).

Each scheduler maintains their own task graph, which is a directed acyclic graph (DAG) in which the nodes represent tasks and edges represent precedence constraints. The node scheduler task graph contains all tasks (CPS or QPS) that are to be executed. Each processor scheduler task graph is a partial copy of the node scheduler task graph containing only the tasks that can be executed by its own processor. Edges in the node scheduler graph between heterogenous tasks (i.e. between CPS and QPS tasks) are represented in the partial processor graphs by the *external dependencies* node attribute. See Figure 5.21 for an example. When a processor scheduler finishes a task, it is removed from the task graph and a signal is sent to the node scheduler. The node scheduler updates its own task graph accordingly, and may then add new tasks to the task graph of the processor scheduler. Note that although the processor task graphs are accessible by both the owning processor scheduler and the node scheduler, there are no read/write conflicts

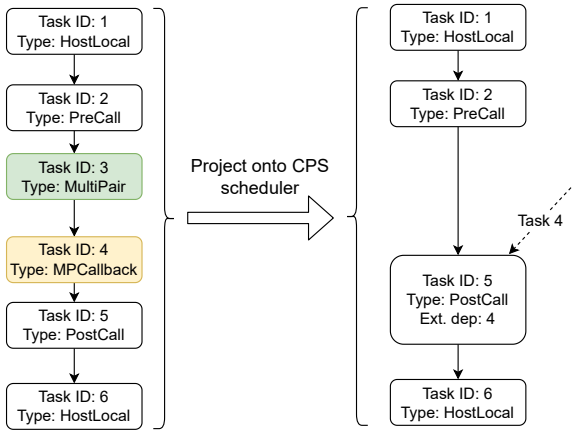


Figure 5.21: Example of a mapping from a full task graph (containing both CPS and QPS tasks) to a partial graph (containing only CPS tasks). Task 5 depends on task 4, which is external from the perspective of the CPS scheduler (indicated using the `external-dependencies` attribute). Note that Task 3 is not needed at all in the partial graph; only the dependency on task 4.

since tasks can only be added by the node scheduler, and tasks can only be removed by the processor scheduler.

Task graph. A task graph consists of

- *tasks* to be scheduled (the nodes),
- *precedence constraints* between the tasks (precedence edges),
- *external precedence constraints* for tasks in the case of processor task graphs (annotated on the nodes),
- *relative deadlines* between tasks (deadline edges),
- *trigger annotations* for some tasks (like incoming messages or network schedule timestamps)

Upon program instantiation, all created tasks are added to the node scheduler task graph, and the relevant tasks are added to the processor schedulers. The number of tasks that are created (and can hence be added to the task graphs) depends on the predictability of the program. During runtime, the node scheduler may create new tasks based on the control-flow of the program.

Task graph splitting. The node scheduler creates a heterogenous task graph consisting of both CPS and QPS tasks. This graph needs to be split into a partial CPS and a partial QPS graph. This is done using the following algorithm.

We consider creating the partial graph for the CPS, and hence the QPS is 'the other processor'. For the partial graph of the QPS the procedure is exactly the same but with reversed roles.

For a heterogenous task graph G containing tasks T (all tasks for both CPS and QPS), precedence constraints P ($(t_1, t_2) \in P$ means that t_1 must precede t_2), compute the partial

CPS graph G_{CPS} as follows:

- Split T into a set T_{CPS} consisting of all tasks that run on the CPS, and T_{QPS} consisting of all other tasks in T. G_{CPS} will consist of only tasks in T_{CPS}.
- Let $P_{CPS} \subset P$ consist of all precedence constraints (t_1, t_2) where $t_1 \in T_{CPS}$ and $t_2 \in T_{CPS}$. These constraints will remain the same in G_{CPS} since they are between tasks in T'.
- Compute the 'immediate cross-predecessors' set I of all tasks $t_{cp} \in T_{QPS}$ such that there exists a task $t \in T_{CPS}$ and $(t_{cp}, t) \in P$. In other words, I contains all tasks running on the QPS that are immediate predecessors of CPS tasks.
- For each $t_i \in I$, compute the 'closest CPS ancestor' task $t_{anc} \in T_{CPS}$, which is a CPS task that has a direct precedence constrain with the closest ancestor of t_i. Add (t_{anc}, t_i) to the precedence constraints of G_{CPS}.

Scheduler communication. Here we describe how the schedulers communicate in our implementation (Section 5.5).

The three schedulers need to exchange information in order to work together. All schedulers can broadcast a *signal* with short information such as 'task N completed' or 'memory freed'. Each scheduler receives these signals. Furthermore, the following read and write access is given:

- The CPS scheduler can read from the completed task ID list of the QPS and vice versa. This makes it possible for the CPS (QPS) scheduler to directly update their remote dependencies without having to wait for a signal from the node scheduler, leading to overall improvement in efficiency
- The node scheduler can add new tasks to the partial graphs of the CPS and QPS. Note that the node scheduler will only add tasks to the partial graph of a processor scheduler when this scheduler is in a waiting state; that is, after the processor scheduler has sent a 'waiting' signal and before the node scheduler has sent a 'task added' signal (only after this signal will the processor scheduler continue). In this way, there are no read/write conflicts in the partial graphs of processor schedulers.
- The CPS (QPS) scheduler can only remove tasks from its own partial graph, not add any.

5.11.3 Scheduler algorithms

Node scheduler algorithm. Below we describe the high-level steps involved in the node scheduler algorithm implementation of Section 5.5.

1. Split the current task graph into a partial CPS graph and a partial QPS graph. For the algorithm, see 'Task graph splitting' above.
2. Add the CPS (QPS) tasks to the partial graph of the CPS (QPS) scheduler
3. Wait for a 'task finished' signal from either CPS or QPS scheduler
4. Remove the corresponding task from the task graph.
5. If the finished task was a `HostLocal` task for some program instance P, and if the CPS partial graph is empty, check which block the program instance should jump to. This information is given by the task itself (and stored in the completed task list of the CPS

scheduler), after evaluating the last instruction (a jump or branch instruction) in the BB that the task represented. For this new BB, create corresponding tasks for both the CPS and QPS. Task creation is discussed in Section 5.11.1.

6. If the task graph is empty, idle until new programs are instantiated.
7. Go back to step 1.

Note that the role of the node scheduler is much smaller when only predictable programs are run. When predictable programs are instantiated, all of their tasks are created at once, resulting in a large task graph in the node scheduler, which never gets new tasks created at runtime. In this scenario, after steps 1 and 2 the CPS and QPS schedulers possess a partial graph which will never get any new tasks. Both processor schedulers will work on their tasks until they are both empty, after which all program instances have finished. Meanwhile, the node scheduler just loops through steps 1, 2, 3 and 6, not doing anything.

CPS scheduler algorithm. Below we describe the high-level steps involved in the CPS scheduler algorithm implementation of Section 5.5.

1. Check which new tasks were completed by the QPS by reading from the shared task memory. Remove external dependency edges that correspond to QPS tasks that have completed.
2. Find all tasks in the partial graph that are ready to execute. These are tasks that fulfill all following requirements:
   - The task has no incoming precedence constraints (there are no unfinished tasks in the task graph that must precede this task)
   - The task has no external precedence constraints (there are no unfinished QPS tasks that must precede this task)
   - If the task is a `HostEvent` task, there must be at least one message in the CPS' message buffer
   - If the task has a specific start time, the current time should be at least the start time
3. If there is no task ready to execute, send a 'waiting' signal and wait until a signal is received that indicates one of the following events:
   - The node scheduler has added one or more tasks to the partial graph
   - The QPS scheduler has completed a task
   - The start time has arrived of one of the tasks that were previously not ready only because their start time had not yet passed
   - One or more new messages have been put into the message buffer

   After one of these signals is received, go back to step 1.
4. If there is at least one task ready to execute, choose which one to execute now. This depends on the scheduling policy that is being used. The policy may or may not use information about the deadlines of the available tasks. Scheduling policies that were implemented for our evaluation are described in Section 5.13.
5. If the task failed, go back to step 1

6. If the task completed, remove it from the partial graph, add its ID to the completed task ID list, and broadcast a signal that the task was finished. If the task was a `HostLocal` task, then also store (in the completed task list) an entry containing the name of the next block to execute. (In this way, the node scheduler knows which task(s) to create and add to the full task graph. See Section 5.11.1 for more details.) Update the deadlines of all other tasks in the task graph. Then go back to step 1.

QPS scheduler algorithm. Below we describe the high-level steps involved in the QPS scheduler algorithm implementation of Section 5.5.

1. Check which new tasks were completed by the CPS by reading from the shared task memory. Remove external dependency edges that correspond to CPS tasks that have completed.
2. Find all tasks in the partial graph that are ready to execute. These are tasks that fulfill all following requirements:
   - The task has no incoming precedence constraints (there are no unfinished tasks in the task graph that must precede this task)
   - The task has no external precedence constraints (there are no unfinished CPS tasks that must precede this task)
   - If the task is a `SinglePair` or `MultiPair` task, the current time should be the beginning of a network time slot that corresponds to this task. (For example, if the task is for creating EPR pairs for program instance 1 on this node (called 'Alice') and program instance 2 on node 'Bob', then the current time should be the start of a $(Alice, 1, Bob, 2)$ time slot).
   - If the task has a specific start time, the current time should be at least the start time
3. If there is no task ready to execute, wait for a signal that indicates one of the following events:
   - The node scheduler has added one or more tasks to the partial graph
   - The CPS scheduler has completed a task
   - The start time has arrived of one of the tasks that were previously not ready only because their start time had not yet passed
   - The start of a time slot has arrived which corresponds to one of the tasks that were previously only blocked on the arrival of this time slot

   After one of these signals is received, go back to step 1.
4. If there is at least one task ready to execute, choose which one to execute now. This depends on the scheduling policy that is being used. The policy may or may not use information about the deadlines of the available tasks.
5. If the task failed, go back to step 1
6. If the task completed, remove it from the partial graph, add its ID to the completed task ID list, and broadcast a signal that the task was finished. Update the deadlines of all other tasks in the task graph. Then go back to step 1.

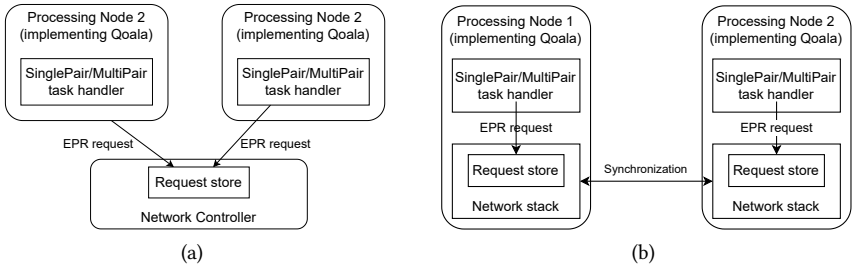


Figure 5.22: Different implementations of network controller and network stack. (a) The network controller is centralized and the nodes send requests to this controller whenever they are executing `SinglePair` or `MultiPair` tasks. (b) The network controller is distributed over the nodes. Inside each node there is a network stack which autonomously talks with the network stack of other nodes and synchronizes entanglement generation. Execution of `SinglePair` and `MultiPair` tasks involves sending a request to the network stack within the node, which then handles pair generation by synchronizing with the network stack in other nodes.

Task graph updates. The node scheduler may add tasks to the current task graph of the CPS or QPS. When a processor scheduler has finished a task, it is removed from the task graph. This has the following effects:

- Precedence edges from this task are removed, potentially making other tasks available for execution
- The time of finishing is recorded; and the deadlines and relative deadlines of all other tasks are updated accordingly

5.11.4 Other algorithms

Linear graphs. When instantiating a program multiple times (for example instantiating a BQC program 1000 times), one has the option to linearize the graphs. Each instantiation has its own graph, and the full graph of all instances result in many independent tasks. One can force all instances to be run in sequence, rather than interleaved, resulting in a linear chain of single-instance graphs. This is done using the following algorithm:

- For each pair (i_1, i_2) of consecutive instances, add a precedence constraint between the last tasks(s) of i_1 and the first task(s) i_2.

Estimating task durations. The scheduler uses the EHI to estimate the duration of a task. This duration may then be used by the scheduler to decide which task to execute when. In our implementation, the scheduler does not make use of these estimates, but we did implement a simple estimator algorithm:

The estimated duration E of a task is computed as follows:

- For a `HostLocal` or `HostEvent` task representing a program block B, E is $N \cdot$ `host_latency` where N is the number of HostLanguage operations in B and `host_latency` is given in the EHI.
- For a `LocalRoutine` tasks representing a block that call a NetQASM routine S, E is the sum of estimated durations of each NetQAM instruction in S. The duration of each quan-

tum instruction is obtained from the EHI, and the duration of each classical instruction is given by the `qnos_latency` entry in the EHI.

- For a `SinglePair` or `MultiPair` task based on a block that calls a request R for N EPR pairs, E is N times the duration of a single EPR generation as listed in the EHI.
- For `PreCall` and `PostCall` tasks, the duration is set to the `host_latency` entry in the EHI.

5.11.5 Entanglement distribution

Qoala only defines how program are executed on a node in a quantum network, and not how and when entanglement is created between nodes. However, Qoala does assume certain things about how nodes can interact with the entanglement distribution system, however this is implemented. The assumption about entanglement generation are as follows.

Network controller with time slots. Conceptually, there is a network controller that oversees entanglement generation and distribution across the whole network. Qoala does not care whether this controller is implemented as a single entity, or is distributed in some way across multiple (processing) nodes (Figure 5.22). The network controller maintains a global timeline divided into *time slots*, which can have arbitrary length. Each time slot may be assigned to a *session*, which is a 4-tuple $(N1, P1, N2, P2)$ where $N1$ ($N2$) is the name of a node in the network and $P1$ ($P2$) is an ID of a program instance running within $N1$ ($N2$). A session hence represents a pair of running program instances across two nodes, and it is such pairs of program instances that want to create entanglement with each other. If a time slot is assigned to some session $(N1, P1, N2, P2)$, only program instances $P1$ and $P2$ (on nodes $N1$ and $N2$) may create entanglement with each other during this time slot.

Populating the network controller's time slot with sessions is the result of (1) demand registration by nodes in the network, followed by (2) network schedule generation by the network controller itself, which we do not consider here (Figure 5.23). In the following, we simply assume that the network controller has a list of time slots assigned to sessions relating to program instances that are being run, and that these time slots are also known by the individual processing nodes.

On-demand entanglement requests. At runtime, nodes implementing Qoala may send requests to the *network stack*. This network stack then issues *EPR requests* to the network controller. Upon receiving an EPR request, the network controller stores it and potentially acts on it:

- If there is a matching EPR request from the other node, and if the current time slot is assigned to the corresponding session, perform the actual entanglement generation process.
- If at least one of the two above conditions does not hold, keep the request until both conditions are satisfied (a matching request from the other node arrives, or the corresponding time slot arrives, or both).

An EPR request is a request for a single EPR pair. A `SinglePair` task is handled by the network stack sending a single EPR request to the network controller. A `MultiPair` task

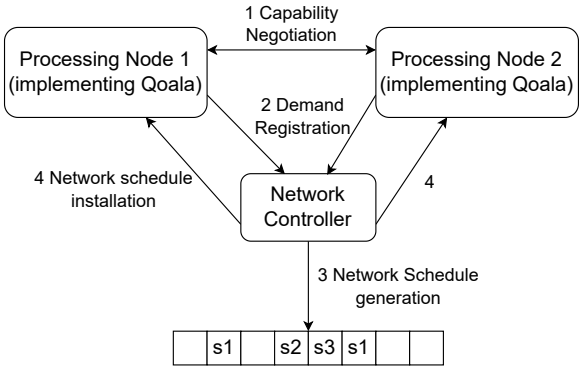


Figure 5.23: High-level steps of using the network controller. 1. Nodes discuss among each other constraints about application execution (Capability Negotiation). 2. The outcome of Capability Negotiation, which contains demands about entanglement generation, is sent to the network controller (Demand Registration). 3. Based on the demands from the nodes, the network controller constructs a network schedule consisting of time slots. Each time slot is assigned to zero or more *sessions*, which correspond to program instance pairs.

is handled by sending multiple EPR requests, possibly interleaved by local QPS processing such as callback routines.

The network stack may fail handling a request. For example, it might timeout trying to produce an EPR pair. In this case, the corresponding task (`SinglePair` or `MultiPair`) also fails. Depending on the scheduler implementation, this task may be executed again at a later time, or the whole program instance may be aborted.

Entanglement generation as a black box. We assume that all nodes can create entanglement with all nodes, orchestrated by the network controller. Qoala does not assume anything about the existence of repeater nodes or entanglement routing algorithms. Rather, a node sending a request for entanglement (in a suitable time slot) will either get this entanglement (created in some way, irrelevant to Qoala) or not (creation failed for some reason, again irrelevant to Qoala). The network stack and controller may be implemented in various ways, such as illustrated in Figure 5.22.

5.12 Simulator implementation

Package overview. The simulator has been implemented as a Python package and is available at [44]. It is divided into three subpackages: (1) `lang`, defining the format of Qoala programs and of the EHI, (2) `runtime` defining common types for the runtime system, and (3) `sim` containing Netsquid objects that implement the Qoala runtime.

The division into subpackages is made in such a way that only `sim` depends on (imports from) Netsquid; the other two subpackages are implementation independent. The `lang` can be used standalone in a compiler, without having the compiler to depend on the runtime implementation, whether that is in simulation or on real hardware.

Netsquid: Protocols, Components, and Listeners. The Qoala simulator make heavy use of Netsquid's *Protocol* class, which can be used to model concurrent software systems. Each protocol defines its own run function, and the Netsquid simulator executes the run

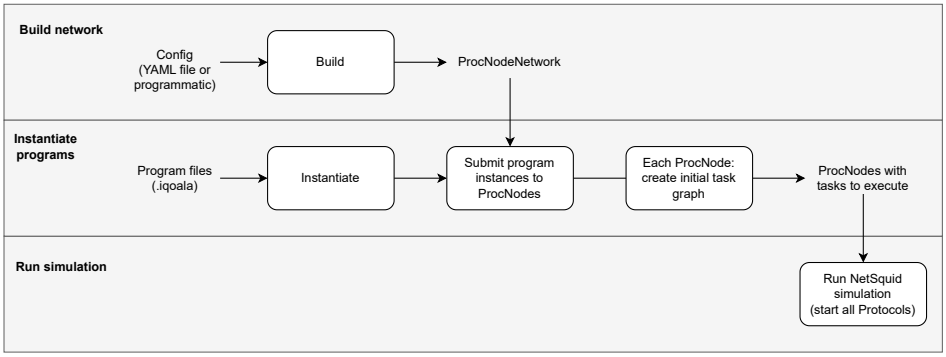


Figure 5.24: High-level overview of sequence of steps performed in order to simulate a network running programs on nodes implemented Qoala.

functions of each protocol concurrently. (Netsquid uses only a single thread, but protocols are run interleaved, i.e. NetSquid provides provides an asynchronous runtime).

The simulator implements a hierarchy of protocols. Each node in the network is a protocol, containing subprotocols for a *Host, Qnos, and Netstack*. The Host represents the CPS, and the Qnos and Netstack together represent the QPS, where Qnos handle local quantum processing, and the Netstack handles requests to the central network controller.

The protocol objects implement the runtime logic of the subsystems. The Netsquid *Component* holds static information about the subsystem, and contains *Ports* for communicating with other components. Protocols use these ports to send messages to other protocols.

Listener objects are a feature of the Qoala simulator that are protocols with the sole purpose to wait for any incoming messages on a port and then notifying the corresponding protocol of them.

Interfaces and configuration. The Qoala simulator allows for a lot of configuration. The Low-level Hardware Interface (LHI) defines a format for defining physical quantum instructions, durations, and noise models. Default values are provided for NV-centers and trapped ions, but custom hardware models can easily be added. The LHI allows for representing real-life validated hardware, but also for simulating hardware that does not (yet) exist.

The LHI allows for the configurations of

- Allowed gate types, gate durations, and gate noise models
- Qubit decoherence model and qubit topology in a node
- Topology of the network
- Entanglement fidelity and generation duration between pairs of nodes
- Classical communication latency between nodes
- Internal communication latency between scheduler components
- Duration of CPS instructions and of classical QPS instructions

The Native-To-Flavor (NTF) interface is used to define the translation from LHI physical quantum instructions to a NetQASM flavor. The QPS is expected to provide an implementation of the NTF, such that it can translate instruction in Local Routines (which are from a particular NetQASM flavor) into the corresponding hardware instructions.

The Exposed Hardware Interface (EHI) is described in Section 5.4. The simulator provides automated tools for translating a combination of an LHI instance and a NTF into an EHI.

Simulator Architecture. The simulator defines various component and protocol classes representing the concepts defined in Section 5.4. These classes can be instantiated in a custom way, and can hence be seen as building blocks. The simulator provides a default way of using these blocks (namely, in the way explained in the Qoala architecture), but it is possible to use these blocks in another way to investigate different architectures. In Figure 5.25 a schematic overview of the most important classes and their roles is given, and in Figure 5.24 an overview is provided of the general sequence of actions involved when simulating the execution of one or more applications on a quantum network.

In our simulator, the network controller (Section 5.11.5) is implemented as a single centralized entity called EntDist (Figure 5.25).

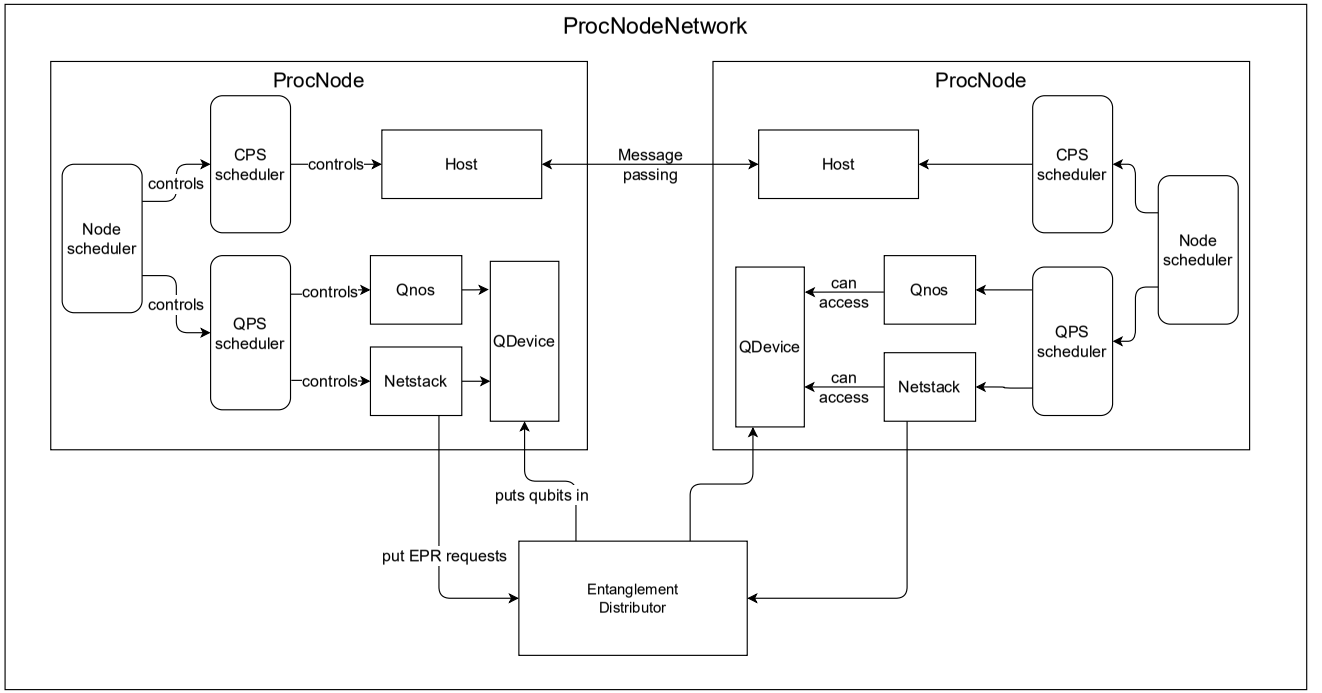


Figure 5.25: High-level overview of the simulator architecture. Each box represents a component. A single network component can contain multiple nodes (two are shown, but there can be more). Each node contains a node scheduler, CPS scheduler and QPS scheduler, implementing the algorithms as described in Section 5.11. A single entanglement distributor object creates entangled pairs and puts the qubits directly in the quantum memory of the nodes, in the QDevice.

5.13 Evaluation details

5.13.1 Simulator setup

All simulations have been done with the simulation package found at [44] and were run on a machine using 80 Intel Xeon Gold cores at 3.9 GHz and 192 GB of RAM. All code used for the evaluations is available in the `evaluation/` folder [44]. For each evaluation done (each subsection in Section 5.6, it includes the Qoala program source code, the scripts for running the simulations, the scripts for producing the plots, and a `README` that explains how to use the code. In the source code, the term *time bin* is used for what we here call *time slot*.

5.13.2 Hardware parameters

In this section we describe the characteristics of the hardware types that have been used in our evaluations. For the evaluation in Section 5.6.1 we simulated all three types below; for the other evaluations we only considered the generic hardware type.

Generic hardware. The allowed gate set is expressed as a particular NetQASM flavor ([17] and Chapter 3).

- Allowed single-qubit gates (vanilla NetQASM flavor [17]): `init`, `rot_x`, `rot_y`, `rot_z`, `x`, `y`, `z`, `h`, `meas`.
- Allowed two-qubit gates (vanilla NetQASM flavor): `cnot`, `cphase`.

Qubit decoherence times are expressed as T1 (amplitude damping) and T2 (dephasing time), which is commonly done in quantum computing. Unless stated otherwise in the evaluation details below, the default noise and duration parameters used for the generic hardware are:

- Single-qubit duration: $5 \cdot 10^3$ ns.
- Two-qubit duration: $200 \cdot 10^3$ ns.
- Qubit T1 time: 10^9 ns.
- Qubit T2 time: 10^8 ns.

NV hardware. Values from [3] and private communication.

- Allowed single-qubit gates on communication qubit (NV NetQASM flavor): `init`, `rot_x`, `rot_y`, `meas`.
- Allowed single-qubit gates on memory qubit (NV NetQASM flavor): `init`, `rot_x`, `rot_y`, `rot_z`, `meas`.
- Allowed two-qubit gates between communication qubit and memory qubit (NV NetQASM flavor): `crot_x`, `crot_y`.

Unless stated otherwise in the evaluation details below, the default noise and duration parameters used for the NV hardware are:

- Single-qubit duration on communication qubit: 300 ns.
- Single-qubit duration on memory qubit: 1.2 ms.
- Two-qubit duration: 1 ms.

- Communication qubit T1 time: 3600 ms
- Communication qubit T2 time: 500 ms
- Memory qubit T1 time: 35000 ms
- Memory qubit T2 time: 1 ms

Trapped Ion hardware. Values from [3] and private communication.

- Allowed single-qubit gates (trapped ion NetQASM flavor): `init`, `rot_z`, `meas`.
- Allowed all-qubit gates (trapped ion NetQASM flavor): `init_all`, `meas_all`, `rot_x_all`, `rot_y_all`, `rot_z_all`, `bichromatic`.

  The effect of applying a bichromatic gate is expressed as

$$U_{XX}(\theta) = \exp(-i\frac{\theta}{2}\sum_{i<j}\sigma_X^{(i)}\sigma_X^{(j)})$$

  for some angle θ.

  Unless stated otherwise in the evaluation details below, the default noise and duration parameters used for the Trapped Ion hardware are:

- Single-qubit duration on communication qubit: 26.6 μs.
- All-qubit duration: 85 ms.
- Qubit T1 time: ∞.
- Qubit T2 time: 85 ms.

NetQASM gate sequence for CNOT on Trapped Ion hardware. We list the sequence of netqasm instructions to effectively apply a CNOT gates on two qubits, which is non-trivial.

Assuming 2 qubits are in use, CNOT gate between qubit 0 and qubit 1 on trapped ion:

```
NETQASM:
  // cnot between q0 and q1
  rot_x_all 8 4
  rot_z Q0 8 4
  rot_x_all 24 4
  bichromatic 8 4
  rot_x_all 24 4
  rot_x_all 8 4
  rot_z Q0 24 4
  rot_x_all 24 4
```

5.13.3 Details for: Demonstrating the architecture's effectiveness

Details for the evaluation described in Section 5.6.1.

A free network schedule was used, meaning that there were no specific time slots, and entanglement generation was allowed at any time. Such a free network schedule was justified since we considered only whether the application ran successfully All applications

Application	# nodes	# EPR pairs per instance	Max # qubits per node	# Instances	Simulation duration (s)
A1. QKD	2	1000	1	1000	2166
A2. BQC	2	2	2	1000	227
A3. Teleportation	2	1	2	1000	24
A4. Ping-pong	2	2	2	1000	35
A5. GHZ	3	4	2	1000	41

Table 5.1: Overview of application used in the evaluation described in Section 5.6.1. Each application was simulated three times, once for each hardware type (generic, NV, Trapped Ion). Each simulation was for 1000 instances of the application. The simulation duration is an average over the three simulations per application.

were run with both (a) hardware-validated parameters (see above); all executions were successful and (b) no-noise versions of these parameters (same durations of all operations but no decoherence nor gate noise); these were used to check if the expected outcomes were obtained. Table 5.1 provides an overview of the applications.

Quantum Key Distribution (QKD). . Two programs (on two nodes): Alice and Bob. N EPR are pairs are generated. Each generated pair is immediately measured by both programs. This results in both programs having N classical outcome bits. See Figure 5.26a for the circuit. Success per instance is determined by checking that Alice and Bob got the same N outcomes bits. For the evaluation, we ran 1000 instances, each creating 1000 EPR pairs.

Teleportation. . Two programs (on two nodes): Sender and Receiver. The Sender teleports a *state* (which is state is an argument to the Sender program) to the Receiver. The Receiver measures in a *basis* (which basis is an argument to the Receiver program) and obtains a single classical outcome bit which is the result of the application. See Figure 5.26b for the circuit. Success per instance is determined by checking that the Receiver got the expected outcome bit (which depends on the combination of state and basis). For the evaluation, we ran 1000 instances.

For each of the Sender program instances, the *state* argument was chosen evenly from the following: $|0\rangle$, $|1\rangle$, $|+\rangle = 1/\sqrt{2}(|0\rangle + |1\rangle)$, $|-\rangle = 1/\sqrt{2}(|0\rangle - |1\rangle)$, $|+i\rangle = 1/\sqrt{2}(|0\rangle + i|1\rangle)$, $|-i\rangle = 1/\sqrt{2}(|0\rangle - i|1\rangle)$.

For each corresponding Receiver program instance, the *basis* argument was chosen such that the expected outcome bit is always 1. Hence, a single application instance succeeded if the Receiver outcome was 1.

Ping-pong. . Teleportation from Sender to Receiver and immediately back to Sender. Same as teleportation application, but the Receiver does not measure; the Sender receives the state back by teleportation and measures. See Figure 5.26b for the circuit. State and basis per instance were chosen similarly as for the teleportation application. Success now depends on the Sender measurement outcome being 1. For the evaluation, we ran 1000 instances.

Blind Quantum Computation (BQC). . Two programs (on two nodes): Client and Server. Two EPR pairs are generated, after which 2 rounds happen. In each round, the

client sends a classical message to the server, after which the server performs a measurement on one of its qubits, sending the measurement outcome back. The same BQC application was used as in [17], using values $\alpha = \pi/2$ and $\beta = -\pi/2$. See Figure 5.27a for the circuit. Success per instance is determined by checking that the Client received the expected classical bit m_2. For the evaluation, we ran 1000 instances.

GHZ. . Three programs (on three nodes): Alice, Bob, and Charlie. Alice creates an EPR pair with Bob, and Bob creates an EPR pair with Charlie. Then, local gates and classical messages are sent between the nodes, resulting in a 3-qubit state (one qubit per node) that is an entangled *GHZ state* [25]. At the end, each program measures its own qubit. See Figure 5.27b for the circuit. Success per instance is determined by checking that the all three programs got the same measurement outcome. For the evaluation, we ran 1000 instances.

5.13.4 Details for: Demonstrating Qoala's multitasking potential

Details for the evaluation described in Section 5.6.2.

Sequential vs Interleaved execution. Sequential: All tasks for all instances were created and added to the task graph at the beginning, but additional precedence constraints were added between the last task for each instance and the first task of the next instance. This resulting in the sequential execution of the 10 instances. Interleaved: All tasks for all instances were created and added to the task graph at the beginning, and no additional precedence constraints were added. We used an FCFS scheduler to pick tasks; since there were no precedence constraints between tasks of different instances, the execution of instances was interleaved.

Teleportation multitasking scenario. One sender node and one receiver node. The teleportation application (A3 in Section 5.6, see also Figure 5.26b) was instantiated 100 times. Classical node-node communication latency: 10^7 ns. Sender node: 2 qubits. Receiver node: sweep over range $[1,\dots,6]$. For each number of qubits $Q \in [1,\dots,6]$, we ran a simulation using both a sequential and an interleaved scheduling approach. For the self-preemption case, the teleportation application was only instantiated 5 times.

BQC multitasking scenario. 10 client nodes and one server node. The BQC application (A2 in Section 5.6, see also Figure 5.27a) was instantiated 10 times for each client, for a total for 100 program instances. Classical node-node communication latency: 10^5 ns. Client node: 2 qubits. Server node: sweep over $\{2,5,10\}$. For each number of qubits $Q \in \{2,5,10\}$, we ran a simulation using both a sequential and an interleaved scheduling approach.

QKD-BQC multitasking scenario. One client node and one server node. Client and server execute both (a) 50 instances of QKD (A1 in Section 5.6, see also Figure 5.26a) and (b) 50 instances of BQC (A2 in Section 5.6, see also Figure 5.27a). We compared two network schedules. Sequential network schedule with repeating pattern P_{seq}. P_{seq} consists of time slots QKD_1, QKD_2, ..., QKD_{50}, BQC_1, BQC_2, ..., BQC_{50}. Alternating network schedule with repeating pattern P_{alt}. P_{alt} consists of time slots QKD_1, BQC_1, QKD_2, BQC_2, ..., QKD_{50}, BQC_{50}.

5.13.5 Details for: Improvement over NetQASM architecture

Details for the evaluation described in Section 5.6.3.

Scenario. Two nodes: client and server. The client and server execute a remote measurement-based quantum computing (MBQC) application. The server initializes local qubits and applies two-qubit gates on them, resulting in a cluster state of three qubits. Then, three rounds of communication happen. In each round, the client sends a classical message containing a measurement basis to the server, the server measures one of its qubits, and finally sends the measurement outcome to the client. After three rounds, the application ends; the last message from the server is the result of the application. This result has an expected value, which is used to determine if a single application instance succeeded or not. The success probability is calculated as the fraction of instances that resulted in the expected value.

We consider a program implementation P for the server. The steps of P are as follows.

1. (Quantum) Initialize all three qubits and apply gates until the desired cluster state is realized.
2. (Classical) Wait for a message θ_0 from the client, representing the first measurement basis.
3. (Quantum) Measure the first qubit in basis $B(\theta_0)$, resulting in classical bit m_0.
4. (Classical) Send m_0 to the client.
5. (Classical) Wait for a message θ_1 from the client, representing the second measurement basis.
6. (Quantum) Measure the second qubit in basis $B(\theta_1)$, resulting in classical bit m_1.
7. (Classical) Send m_1 to the client.
8. (Classical) Wait for a message θ_2 from the client, representing the third measurement basis.
9. (Quantum) Measure the third qubit in basis $B(\theta_2)$, resulting in classical bit m_2.
10. (Classical) Send m_2 to the client.

In our evaluation, we considered a program $P_{netqasm}$ written in the NetQASM runtime format [17], which would be written in Python. Specifically, $P_{netqasm}$ contains the above steps in Python code, in the same order. The quantum steps are converted on-the-fly into NetQASM subroutines. This means that, in the NetQASM runtime, we have the following execution:

$P_{netqasm}$ execution:

1. NetQASM subroutine for initializing the three qubits.
2. Classical Python code for waiting for θ_0.
3. NetQASM subroutine for measuring the first qubit.
4. Classical Python code for sending for m_0.
5. Classical Python code for waiting for θ_0.
6. NetQASM subroutine for measuring the second qubit.

7. Classical Python code for sending for m_1.
8. Classical Python code for waiting for θ_2.
9. NetQASM subroutine for measuring the third qubit.
10. Classical Python code for sending for m_2.

We note that since the NetQASM runtime does not allow for compilation across classical and quantum segments of the code, there is no way to change the order of the steps. In our evaluation, we represented $P_{netqasm}$ as a Qoala program $Q_{netqasm}$ with the exact same contents, but with classical code represented as host code, and the NetQASM subroutines as Qoala local routines.

P can be optimized by noting that some qubit operations can be delayed until a later time, decreasing the duration the some qubits have to stay in memory. This mitigates decoherence and it is expected that overall such an optimized program P_{opt} leads to a higher success probability.

The steps of P_{opt} are:

1. Wait for a message θ_0 from the client, representing the first measurement basis.
2. Initialize the first 2 qubits and apply gates until a partial cluster state is realized.
3. Measure the first qubit in basis $B(\theta_0)$, resulting in classical bit m_0.
4. Send m_0 to the client.
5. Wait for a message θ_1 from the client, representing the second measurement basis.
6. Initialize the third qubit and apply gates until the remaining partial cluster state is realized.
7. Measure the second qubit in basis $B(\theta_1)$, resulting in classical bit m_1.
8. Send m_1 to the client.
9. Wait for a message θ_2 from the client, representing the third measurement basis.
10. Measure the third qubit in basis $B(\theta_2)$, resulting in classical bit m_2.
11. Send m_2 to the client.

where we note that the end-to-end behavior of $P_{netqasm}$ and P_{opt} are the same, and hence P_{opt} is a valid optimized version of P. In our evaluation, we represented P_{opt} as a Qoala program Q_{opt} with the exact same steps.

For the client, we used a single program implementation Q_{client}, optimized for Qoala.

We compared running (NETQASM): Q_{client} on the client node and $Q_{netqasm}$ on the server node with (QOALA): Q_{client} on the client node and Q_{opt} on the server node. In both cases we instantiated the application 1000 times. We obtained success probabilities 66% for NETQASM and 82% for QOALA.

5.13.6 Details for: Tradeoffs between classical and quantum performance metrics

Details for the evaluation described in Section 5.6.4.

Scenario. Two nodes: Alice and Bob. Bob executes an interactive quantum program where classical input is given by Alice. Bob also executes a 'busy' program consisting only of CPS tasks.

Interactive program. The interactive program does the following steps: (1) prepare a local qubit in a state (state given as program instance argument) by initialization and qubit rotation, (2) send an 'acknowledge' message to Alice, (3) wait for a message from Alice, (4) measure the local qubit in a basis (basis given as program instance argument) (5) return the measurement result (classical bit). For each combination of state and basis, an expected measurement result value is computed. The success probability of the interactive program is given by the fraction of program instances that produces the expected value. The interactive program was instantiated 1000 times.

Busy program. The busy program consists only of a block which waits for some duration (input argument). This waiting time mimics the CPS being busy with some local classical computation.

Fixed parameters. Qubit coherence times: $T_1 = 10^{10}$ ns, $T_2 = 10^8$ ns. Classical node-to-node communication latency: 10^7 ns. Rate of arrival of busy programs instances: once every 10^6 ns.

5.13.7 Details for: Success probabilities with quantum multitasking

Details for the evaluation described in Section 5.6.5.

Local program. A program which prepares a single qubit to the $|-\rangle = 1/\sqrt{2}(|0\rangle + |1\rangle)$ state, then waits for duration d, and then measures the qubit in the X-basis. The expected outcome bit is hence 1.

Scenario. Two nodes: Alice and Bob. Alice and Bob execute the teleportation application (A3 in Section 5.6, see also Figure 5.26b) T times. Bob concurrently executes the local program described above L times. For each combination of $T \in [1, 15]$ and $L \in [1, 15]$, we ran a simulation SIM 200 times, where in each simulation, all instances and all their tasks are created at the same time and added to the task graphs of the node. The success per teleportation instance is calculated as in Section 5.13.3. Success probability is calculated as the fraction of successful instances. The success probability of the local program is calculated as the fraction of all local program instances (across all 200 runs) gave the expected classical result 1.

Fixed parameters. Qubit coherence times: $T_1 = 10^{10}$ ns, $T_2 = 10^7$ ns. Classical node-node communication latency: 0.1 ms. Network schedule: repeating pattern P of time slots, where $P = \langle 0, \ldots, n \rangle$ where n is the number of teleportation instances and where each i is associated with a single teleportation instance. Number of qubits at Bob: 10. Number of qubits at Alice: 20.

5.13.8 Details for: Performance sensitivity

Details for the evaluation described in Section 5.6.6.

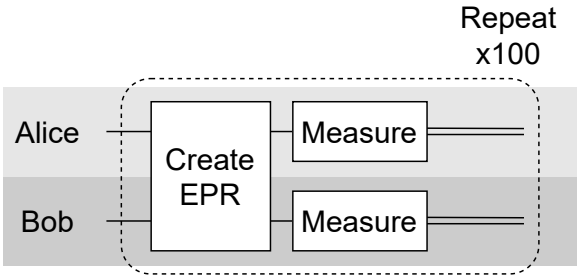
Scenario. One server node runs 10 BQC applications (A2 in Section 5.6, see also Figure 5.27a) concurrently with 10 client nodes (one BQC application per client node). One BQC instance is run for each client. This scenario was repeated 100 times for each of the three evaluations (impact of node-node-latencies, impact of internal latencies, impact of time slot length) in order to obtain statistics. The success per BQC instance is calculated as in Section 5.13.3. Success probability is calculated as the fraction of successful instances.

Fixed parameters. Number of client nodes: 10. Number of qubits per client node: 1. Number of qubits for server node: 20. Qubit coherence time: $T_2 = 1 \cdot 10^7$ ns. Network schedule: repeating pattern of 10 slots, each assigned to one client-server pair.

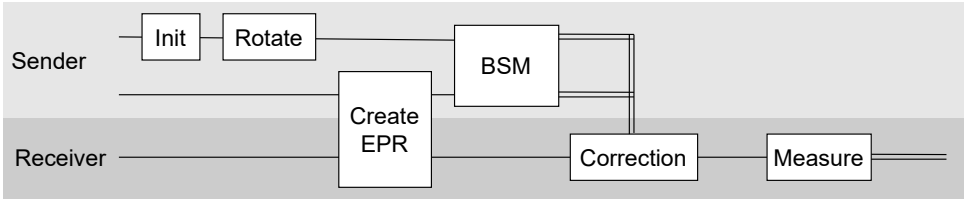
Impact of node-to-node classical communication latencies. Network time slot length: $1 \cdot 10^5$ ns. Internal scheduler communication latency: 0 ns. Values used for node-to-node classical communication latencies: 10^5, 10^6, 10^7 ns (i.e. 0.01, 0.1, 1 times the T_2 coherence time).

Impact of internal scheduler latencies. Network time slot length: $1 \cdot 10^5$ ns. Classical communication latency: 10^5 ns. Values used for internal scheduler communication latencies: 10^3, 10^5, 10^7 ns, where we obtained success probabilities 0.89(2), 0.89(2), 0.83(2), respectively.

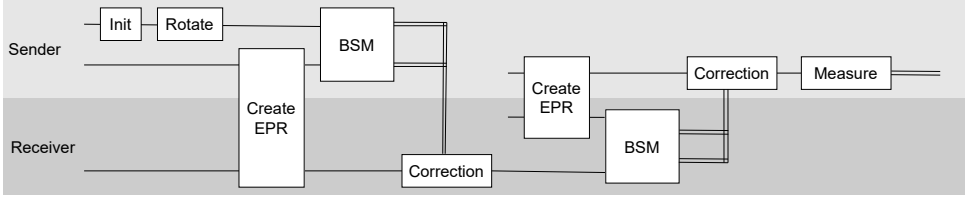
Impact of network schedule time slot length. Classical communication latency: 10^5 ns. Internal scheduler communication latency: 0 ns. Values used for time slot length: 10^5, 10^6, 10^7 ns, (i.e. 0.01, 0.1, 1 times the T_2 coherence time).



(a)

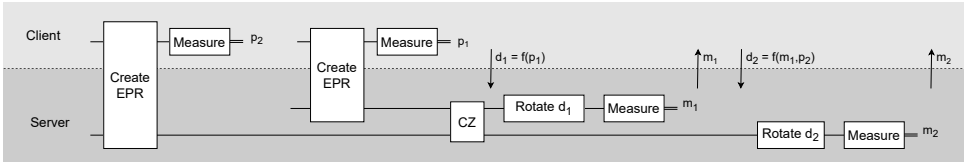


(b)

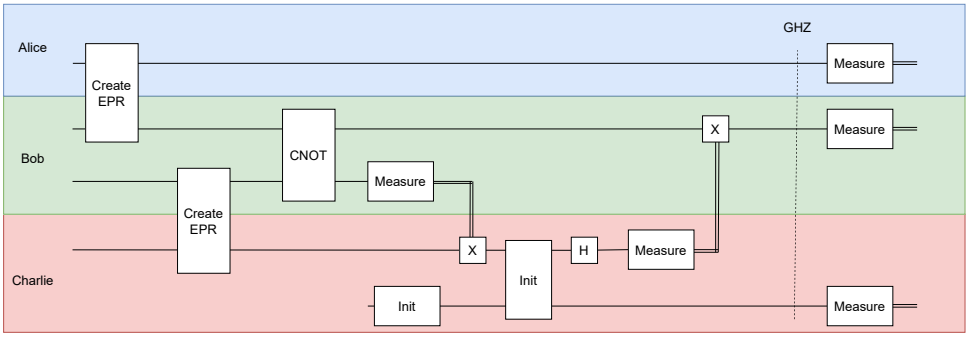


(c)

Figure 5.26: Circuit for applications A1 (QKD), A3 (Teleport) and A4 (ping-pong) from Section 5.6. Single lines represent qubits. Double lines represent classical values. (a) QKD (A1). Two nodes repeatedly generate entangled pairs which are immediately measured. (b) Teleport (A3). A sender node (having 2 qubits) teleport a state to a receiver node. The sender applies local quantum operations (initialization, qubit rotation gates). The sender and receiver create an entangled pair. The sender performs local quantum gates and measurements resulting in classical outcomes. The sender sends the classical outcomes to the receiver. Based on the outcomes the receiver applies local quantum gates and measurement. (c) Ping-pong (A4). The sender teleports a state to the receiver and the receiver immediately teleports it back to the sender. In total, 2 entangled pairs are created.



(a)



(b)

Figure 5.27: Circuit for applications A2 (BQC) and A5 (GHZ) from Section 5.6. Single lines represent qubits. Double lines represent classical values. (a) BQC (A2). A client node remotely prepares two qubits on a server node by creating an entangled pair and locally measuring its qubits, resulting in classical outcomes p_1 and p_2. The server applies a local two-qubit gate (CZ or cphase) on its qubits. The client sends a classical value d_1 which it calculates based on p_1 and other application input values. The server applies local gates based on d_1 and measures, resulting in classical value m_1 which it sends to the client. The client sends a classical value d_2 which it calculates based on p_2, m_1 and other application input values. The server applies local gates based on d_2 and measures, resulting in classical value m_2 which it sends to the client. The client uses the values m_2 to calculate the final result (not in the Figure). (b) Three nodes (Alice, Bob, and Charlie) create pair-wise entangled pairs. Bob applies local gates and measures one of his qubits, sending the outcomes to Charlie. Based on this outcome, Charlie performs local operations and sends a measurement outcome back to Bob. At the time of the vertical dashed line, the three nodes share a 3-qubit GHZ state. They all measure and check their correlations.

References

[1] B. Andersson and E. Tovar. "Multiprocessor scheduling with few preemptions". In: *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. IEEE. 2006, pp. 322–334. DOI: 10.1109/RTCSA.2006.45.

[2] P. Arrighi and L. Salvail. "Blind quantum computation". In: *International Journal of Quantum Information* 4.05 (2006), pp. 883–898. DOI: 10.1142/S0219749906002171.

[3] 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". In: *npj Quantum Information* 9.1 (2023), p. 100. DOI: 10.1038/s41534-023-00765-x.

[4] K. Azuma, S. Bäuml, T. Coopmans, D. Elkouss, and B. Li. "Tools for quantum network design". In: *AVS Quantum Science* 3.1 (2021). DOI: 10.1116/5.0024062.

[5] S. Baruah, V. Bonifaci, G. d'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. "Scheduling real-time mixed-criticality jobs". In: *IEEE Transactions on Computers* 61.8 (2011), pp. 1140–1152. DOI: 10.1109/TC.2011.142.

[6] T. Beauchamp, H. Jirovská, and S. Wehner. "An Architecture for scheduling network and local operations in a small-scale Quantum Network". In preparation. Private communication. 2023.

[7] C. Bennett and G. Brassard. "Quantum cryptography: Public key distribution and coin tossing". In: *Theoretical computer science* 560 (2014), pp. 7–11. DOI: 10.1016/j.tcs.2014.05.025.

[8] C. Bradley, J. Randall, M. Abobeih, R. Berrevoets, M. Degen, M. Bakker, M. Markham, D. Twitchen, and T. Taminiau. "A ten-qubit solid-state spin register with quantum memory up to one minute". In: *Physical Review X* 9.3 (2019), p. 031045. DOI: 10.1103/PhysRevX.9.031045.

[9] A. Broadbent, J. Fitzsimons, and E. Kashefi. "Universal Blind Quantum Computation". In: *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '09. Washington, DC, USA: IEEE, Oct. 2009, pp. 517–526. ISBN: 978-1-4244-5116-6. DOI: 10.1109/FOCS.2009.36. arXiv: 0807.4154. URL: http://ieeexplore.ieee.org/document/5438603/.

[10] A. Burns and R. Davis. "A survey of research into mixed criticality systems". In: *ACM Computing Surveys* 50.6 (2017), pp. 1–37. DOI: 10.1145/3131347.

[11] G. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[12] A. S. Cacciapuoti, M. Caleffi, F. Tafuri, F. S. Cataliotti, S. Gherardini, and G. Bianchi. "Quantum internet: networking challenges in distributed quantum computing". In: *IEEE Network* 34.1 (2019), pp. 137–143. DOI: 10.1109/MNET.001.1900092.

[13] M. Caleffi, M. Amoretti, D. Ferrari, D. Cuomo, J. Illiano, A. Manzalini, and A. S. Cacciapuoti. "Distributed quantum computing: a survey". In: *arXiv preprint arXiv:2212.10609* (2022). DOI: 10.48550/arXiv.2212.10609.

[14] *Code and data for running the evalutions*. 4TU.ResearchData. 2025. URL: https://doi.org/10.4121/f1e3a0ba-17d5-48f9-a66b-2c45520f229c.

[15] T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J. de Oliveira Filho, M. Papendrecht, J. Rabbie, F. Rozpędek, M. Skrzypczyk, et al. "Netsquid, a network simulator for quantum information using discrete events". In: *Communications Physics* 4.1 (2021), pp. 1–15.

[16] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveira Filho, R. Hanson, and S. Wehner. "A link layer protocol for quantum networks". In: *Proceedings of the ACM special interest group on data communication*. 2019, pp. 159–173. DOI: 10.1145/3341302.3342070.

[17] 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". In: *Quantum Science and Technology* 7.3 (2022), p. 035023. DOI: 10.1088/2058-9565/ac753f.

[18] S. DiAdamo, M. Ghibaudi, and J. Cruise. "Distributed quantum computing and network control for accelerated vqe". In: *IEEE Transactions on Quantum Engineering* 2 (2021), pp. 1–21. DOI: 10.1109/TQE.2021.3057908.

[19] S. DiAdamo, J. Nötzel, B. Zanger, and M. M. Beşe. "QuNetSim: A software framework for quantum networks". In: *IEEE Transactions on Quantum Engineering* 2 (2021), pp. 1–12. DOI: 10.1109/TQE.2021.3092395.

[20] C. D. Donne, M. Iuliano, B. van der Vecht, G. M. Ferreira, H. Jirovská, T. van der Steenhoven, A. Dahlberg, M. Skrzypczyk, D. Fioretto, M. Teller, P. Filippov, A. R.-P. Montblanch, J. Fischer, B. van Ommen, N. Demetriou, D. Leichtle, L. Music, H. Ollivier, I. t. Raa, W. Kozlowski, T. Taminiau, P. Pawełczak, T. Northup, R. Hanson, and S. Wehner. "An operating system for executing applications on quantum network nodes". In: *Nature* 639 (Mar. 12, 2025), pp. 321–328. DOI: 10.1038/s41586-025-08704-w.

[21] P. Drmota, D. Main, D. Nadlinger, B. Nichol, M. A. Weber, E. Ainley, A. Agrawal, R. Srinivas, G. Araneda, C. Ballance, and D. Lucas. "Robust quantum memory in a trapped-ion quantum network node". In: *Physical Review Letters* 130.9 (2023), p. 090803. DOI: 10.1103/PhysRevLett.130.090803.

[22] A. K. Ekert. "Quantum cryptography based on Bell's theorem". In: *Physical Review Letters* 67.6 (1991), p. 661.

[23] E. Farhi, J. Goldstone, and S. Gutmann. "A quantum approximate optimization algorithm". In: *arXiv preprint arXiv:1411.4028* (2014). DOI: 10.48550/arXiv.1411.4028.

[24] M. Girkar and C. Polychronopoulos. "The hierarchical task graph as a universal intermediate representation". In: *International Journal of Parallel Programming* 22.5 (1994), pp. 519–551. DOI: 10.1007/BF02577777.

[25] D. Greenberger, M. Horne, and A. Zeilinger. "Going beyond Bell's theorem". In: *Bell's theorem, quantum theory and conceptions of the universe*. Springer, 1989, pp. 69–72. DOI: 10.1007/978-94-017-0849-4.

[26] P. Hambarde, R. Varma, and S. Jha. "The survey of real time operating system: RTOS". In: *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. IEEE. 2014, pp. 34–39. DOI: 10.1109/ICESC.2014.15.

[27] S. Hermans, M. Pompili, H. Beukers, S. Baier, J. Borregaard, and R. Hanson. "Qubit teleportation between non-neighbouring nodes in a quantum network". In: *Nature* 605.7911 (2022), pp. 663–668. DOI: 10.1038/s41586-022-04697-y.

[28] M. Hillery, V. Bužek, and A. Berthiaume. "Quantum secret sharing". In: *Physical Review A* 59.3 (1999), p. 1829. DOI: 10.1103/PhysRevA.59.1829.

[29] D. Jensen. "A timeliness model for asychronous decentralized computer systems". In: *Proceedings ISAD 93: International Symposium on Autonomous Decentralized Systems*. IEEE. 1993, pp. 173–182. DOI: 10.1109/ISADS.1993.262706.

[30] H. Jnane, B. Undseth, Z. Cai, S. Benjamin, and B. Koczor. "Multicore quantum computing". In: *Physical Review Applied* 18.4 (2022), p. 044064. DOI: 10.1103/PhysRevApplied.18.044064.

[31] H. J. Kimble. "The quantum internet". In: *Nature* 453.7198 (2008), pp. 1023–1030. DOI: 10.1038/nature07127.

[32] V. Krutyanskiy, M. Canteri, M. Meraner, J. Bate, V. Krcmarsky, J. Schupp, N. Sangouard, and B. Lanyon. "Telecom-wavelength quantum repeater node based on a trapped-ion processor". In: *Physical Review Letters* 130.21 (2023), p. 213601. DOI: 10.1103/PhysRevLett.130.213601.

[33] V. Krutyanskiy, M. Galli, V. Krcmarsky, S. Baier, D. Fioretto, Y. Pu, A. Mazloom, P. Sekatski, M. Canteri, M. Teller, J. Schupp, J. Bate, M. Meraner, N. Sangouard, B. Lanyon, and T. Northup. "Entanglement of trapped-ion qubits separated by 230 meters". In: *Physical Review Letters* 130.5 (2023), p. 050803. DOI: 10.1103/PhysRevLett.130.050803.

[34] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[35] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. "MLIR: Scaling compiler infrastructure for domain specific computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[36] D. Leichtle, L. Music, E. Kashefi, and H. Ollivier. "Verifying BQP computations on noisy devices with minimal overhead". In: *PRX Quantum* 2.4 (2021), p. 040302. DOI: 10.1103/PRXQuantum.2.040302.

[37] P. Li. "Utility accrual real-time scheduling: Models and algorithms". PhD thesis. Virginia Polytechnic Institute and State University, 2004.

[38] C. L. Liu and J. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61. DOI: 10.1145/321738.321743.

[39] X. Liu, A. Angone, R. Shaydulin, I. Safro, Y. Alexeev, and L. Cincio. "Layer VQE: A variational approach for combinatorial optimization on noisy quantum computers". In: *IEEE Transactions on Quantum Engineering* 3 (2022), pp. 1–20. DOI: 10.1109/TQE.2021.3140190.

[40] A. Ovide, S. Rodrigo, M. Bandic, H. Van Someren, S. Feld, S. Abadal, E. Alarcon, and C. Almudever. "Mapping quantum algorithms to multi-core quantum computing architectures". In: *arXiv preprint arXiv:2303.16125* (2023). DOI: 10.48550/arXiv.2303.16125.

[41] C. Polychronopoulos. "The hierarchical task graph and its use in auto-scheduling". In: *Proceedings of the 5th International Conference on Supercomputing*. 1991, pp. 252–263. DOI: 10.1145/109025.109089.

[42] M. Pompili, C. Delle Donne, I. te Raa, B. van der Vecht, M. Skrzypczyk, G. Ferreira, L. de Kluijver, A. Stolk, S. Hermans, P. Pawełczak, W. Kozlowski, R. Hanson, and S. Wehner. "Experimental demonstration of entanglement delivery using a quantum network stack". In: *npj Quantum Information* 8.1 (2022), p. 121. DOI: 10.1038/s41534-022-00631-2.

[43] M. Pompili, S. Hermans, S. Baier, H. Beukers, P. Humphreys, R. Schouten, R. Vermeulen, M. Tiggelman, L. dos Santos Martins, B. Dirkse, S. Wehner, and R. Hanson. "Realization of a multinode quantum network of remote solid-state qubits". In: *Science* 372.6539 (2021), pp. 259–264. DOI: 10.1126/science.abg1919.

[44] *Qoala simulator implementation*. https://github.com/QuTech-Delft/qoala-sim.

[45] *Quantum NETwork in Baidu Quantum Platform*. 2022. URL: https://quantum-hub.baidu.com/qnet/.

[46] QuTech. *SquidASM*. https://github.com/QuTech-Delft/squidasm. 2022.

[47] M. Ruf, N. Wan, H. Choi, D. Englund, and R. Hanson. "Quantum networks based on color centers in diamond". In: *Journal of Applied Physics* 130.7 (2021). DOI: 10.1063/5.0056534.

[48] R. Satoh, M. Hajdušek, N. Benchasattabuse, S. Nagayama, K. Teramoto, T. Matsuo, S. A. Metwalli, P. Pathumsoot, T. Satoh, S. Suzuki, and R. Van Meter. "QuISP: a quantum internet simulation package". In: *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE. 2022, pp. 353–364. DOI: 10.1109/QCE53715.2022.00056.

[49] A. Silberschatz, P. B. Galvin, and G. Gagne. "Operating System Concepts, Windows XP update". In: John Wiley & Sons, 2006. Chap. 6.

[50] M. Skrzypczyk and S. Wehner. "An architecture for meeting quality-of-service requirements in multi-user quantum networks". In: *arXiv preprint arXiv:2111.13124* (2021). DOI: 10.48550/arXiv.2111.13124.

[51] H. Topcuoglu, S. Hariri, and M.-Y. Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274. DOI: 10.1109/71.993206.

[53] G. Vardoyan, M. Skrzypczyk, and S. Wehner. "On the quantum performance evaluation of two distributed quantum architectures". In: *ACM SIGMETRICS Performance Evaluation Review* 49.3 (2022), pp. 30–31. DOI: 10.1145/3529113.3529123.

[54] T. Vidick and S. Wehner. *Introduction to Quantum Cryptography*. Cambridge University Press, 2023. DOI: 10.1017/9781009026208.

[55] S. Wehner, D. Elkouss, and R. Hanson. "Quantum internet: A vision for the road ahead". In: *Science* 362.6412 (2018), eaam9288. DOI: 10.1126/science.aam9288.

[56] S.-H. Wei, B. Jing, X.-Y. Zhang, J.-Y. Liao, C.-Z. Yuan, B.-Y. Fan, C. Lyu, D.-L. Zhou, Y. Wang, G.-W. Deng, H.-Z. Song, D. Oblak, G.-C. Guo, and Q. Zhou. "Towards real-world quantum networks: a review". In: *Laser & Photonics Reviews* 16.3 (2022), p. 2100219. DOI: 10.1002/lpor.202100219.

6

Compiling Qoala programs

In this chapter, we discuss the problem of compiling quantum network programs. We review what we want compilers to do for quantum network applications and discuss related work. We then discuss design consideration for a compiler for Qoala programs. We give recommendations on the design of such a compiler and conclude with pointers for further research and evaluation.

6.1 Introduction

Recall that our overarching objectives for programming and executing quantum network applications are, as also discussed in Chapter 1:

- Improve accessibility by enabling programming using high-level concepts and tools.
- Execute quantum internet programs efficiently in terms of application success probability and success rate.

In the previous chapters we presented building blocks towards these objectives. In Chapter 3 the low-level instruction set NetQASM was introduced for representing quantum operations. We also defined a software development kit (SDK) that allows one to write programs in Python and which compiles quantum code into NetQASM subroutines. In Chapter 4 we introduced an operating system, QNodeOS, that executes these programs. In Chapter 5 we improved on the design of the operating system and presented the Qoala program format and execution model. We also noted that Qoala enables a compiler design that takes into account the hybrid and networked nature of programs. In this chapter we discuss what would be needed for such a design and provide recommendations for working out such as design.

Why a compiler? Although programmers could express their program logic directly in the Qoala format, this is not practical for several reasons. First, since the format is low-level, it is quite verbose. This is a similar issue as with classical machine code (or lower-level languages): although possible, it is often very impractical to write complete programs

This chapter is based on the article in preparation: **B. van der Vecht**, S. Oslovich, D. Rivera, and S. Wehner. "Qoala Compiler".

```
q = create_epr()
q.H()  # hadamard
t = recv_from_remote_node()
q.H()  # hadamard
q.rot_Z(angle=t)
q.measure()
```

Figure 6.1: Simple program source code which cannot be optimized with the NetQASM SDK. The NetQASM SDK splits the quantum code into two separate subroutines — S1 containing the entanglement creation and the hadamard gate, and S2 containing the second hadamard gate until the measurement — because of the classical communication operation (receiving t). A compiler that can compile across such classical operations would cancel out the two hadamard gates since their consecutive execution is equal to the identity operation.

in machine code itself. Second, the low-level format can be quite far from the high-level program logic. Someone who wants to implement a certain quantum network program typically only cares about the overall (high-level) behavior of the program. For example, they might simply want to express that the program should create entangled states repeatedly, and then measure all created qubits. Having to deal with low-level details about how to express looping behavior, which and how much (quantum) memory to allocate, and how to integrate the classical and quantum code is in most cases not something a programmer wants to deal with. Third, the Qoala program format allows hardware-specific details, such as hardware-specific quantum instructions (using NetQASM flavors, see Chapter 3). Although not required to add these details by the programmer, if one wants to make use of these details to improve execution (e.g. increasing success probability), the programmer should have knowledge about the hardware. Ideally, the programmer should not have to write different versions of a program, one for each hardware platform that the program is intended to run upon.

In Chapters 3 and 4 we described how NetQASM and QNodeOS already provide a way for programmers to express program logic in a high-level language, Python. However, in QNodeOS, compilation would happen on-the-fly (also called *interpretation*): while executing a program (i.e. at *runtime*), program code is evaluated and quantum instructions are translated into NetQASM subroutines. This lead to certain problems, also discussed in the evaluation of Qoala (Section 5.6):

- Compilation slows down execution: Since compilation needs to happen at runtime, overall execution time increases. This can be especially bad if quantum states must stay alive during this compilation time. This can be seen clearly in Figure 4.3.
- No cross-compilation possible: Since code is interpreted on the fly, optimizations taking into account future code cannot be made. See Figure 6.1 for a simple example piece of code that cannot be optimized further in NetQASM/QNodeOS.

Qoala, with its new model of programs (Section 5.4), enables addressing of the above problems, by requiring hybrid classical-quantum ahead-of-time compilation. Furthermore, Qoala's feature of deadlines may be used to aid the scheduler at runtime, improving runtime performance. Before we discuss how a compiler for Qoala should look like, we first

review what compilers are and related work.

6.2 Compilers

Compilation is the translation from a high-level source code representation of a program into a lower-level representation of that program, typically machine code, that a computer's processor can execute. Compilation is itself done by a computer program, namely the compiler. The type of machine code that a compiler produces depends on the *target* of the compiler. The target is typically a combination of processor architecture (such as x86 or ARM for classical computers) and operating system (such as Linux or Windows for classical computers).

Program compilation is a well-studied topic in both classical [1] and quantum computing [8] and is a vital tool in everyday practical software development. Many existing ideas and strategies can be re-used for compiling quantum network applications. However, quantum network applications present unique challenges that need to be solved by a compiler.

Generally, a compiler has three purposes:

- **Translation**. Allowing a programmer to write application logic in a high-level language improves the accessibility: it relieves the programmer of the burden of thinking about low-level details, and makes it easier to express complex logic. However, in the end, hardware must execute low-level code and a translation step is required. Such translation must maintain the intended behavior of the program (correctness).
- **Error detection**. A compiler checks whether the program source code is valid and throws an error if it is not.
- **Optimization**. A compiler tries to perform optimization with respect to various metrics. It may try to minimize the size of the final executable. It may also try to produce an executable such that the expected runtime performance is optimized. Increasing expected runtime performance can be done in many ways, which are discussed in more detail below.

6.3 Related work

6.3.1 Compilers for classical computing.

Compilers for classical computers have been developed and used since the creation of the first digital computers and are therefore a well-studied topic [1, 20, 47, 54]. A common architecture for compilers is to use one or more Intermediate Representation (IR)s. In this model, the high-level program source code is first translated by a *front-end* compiler to the (highest-level) IR, then possibly translated to lower-level IRs, before it is translated from the (lowest-level) IR by a *back-end* compiler to machine code (Figure 6.2). Separating the front-end from the back-end enables extensibility of the compiler: for each high-level language for which one wants to implement a compiler, only a front-end must be developed, since further compilation can happen using existing back-ends. Similarly, for each machine target, only one back-end needs to be created. In this way, many different programming languages and machine targets may use the same infrastructure, by 'going

Figure 6.2: Schematic of high-level steps in compilers that use Intermediate Representation (IR)s. A source language-specific front-end compiler translates the source code into the IR. There may be different front-end compilers for different source code languages, all translating to the same IR. A machine code-specific back-end compiler translates the IR to machine code. There may be different back-end compilers for different machine code types (i.e. targets). Optimizations may be applied on the IR, or during front-end and/or back-end translation. This model allows flexibility: making a new source code language compatible only requires developing a new front-end compiler; making a new machine target compatible only requires a new back-end compiler. Furthermore, optimizations on the IR code do not need to know about source code language nor target machine code.

through' the same IR(s). A well-known compiler that uses this model is LLVM [30] which uses the *LLVM IR*.

Compilers for classical computers perform code optimizations. Code optimization techniques include

- *Target-independent optimization* [17], such as including removal of dead (unused) code, optimizations of looping constructs and combining common expressions to reduce code, and
- *Target-dependent optimization* [6, 48], such as register allocation, instruction re-ordering, branch prediction and elimination, vectorization and cache optimization.

These optimizations typically aim to improve execution speed and memory efficiency [1, 47].

6.3.2 Compilers for classical networking.

Classical applications that use networking are also compiled by classical compilers. Typically, compilers do not apply network-specific optimizations. This is since network operations are often handled by system calls, which compilers do not have control over. Moreover, compilers also don't have control over the execution and timing of other applications in the network, nor over the network itself. Network-specific optimization is therefore indirect, namely by optimizing the number of network system calls, reducing memory that would need to be context-switched upon network events, and optimizing local code that affects networking code [11]. We note that there exist compilers for network programming languages [34, 39]. However, these relate to programming the control plane of networks (a paradigm called software-defined networking (SDN)), including how data is routed through the network, and are hence different from user applications that 'merely use' the network.

6.3.3 Compilers for quantum computing.

As already described in Chapter 2, quantum computing programs are typically expressed as quantum circuits. Circuits consist of qubits, which are memory locations on which operations are applied. Limit classical control may be added.

There exist many languages for quantum computing [8, 22]. Many frameworks make use of assembly-like languages like OpenQASM [9] or cQASM [28]. Also more high-level

languages exist such as Q# and Scaffold [22, 35, 50].

Circuit optimizers. Compilers for quantum computing programs often try to optimize circuit depth (number of gates) and width (number of qubits). Most compilers focus on NISQ computers, dealing with resource constraints (like limited qubit availability, and restricted topology) [3, 8, 41] and qubit mapping [4, 33, 44, 56]. Benchmarks exist [32, 51] with which to compare compilers, by looking at the output, like circuit size.

Hybrid classical-quantum compilers. In the last years, more focus has been brought on hybrid classical-quantum compilers. This can be seen in the proposal of full-stack architectures, including classical and quantum processing [27, 38, 41], and the focus on hybrid quantum computing programs, like Variational Quantum Eigensolvers (VQE) [13, 36] or Quantum Approximate Optimization Algorithms (QAOA) [14]

One can distinguish two approaches:

- integration of classical and quantum code but optimization is separate or limited [18, 26, 35, 55]
- joint classical-quantum optimization (MLIR-based [31], LLVM based) [25, 37, 42, 46].

In order to deal with the hybrid code of hybrid programs, often intermediate representations (IRs) are used [43, 45]. The existing LLVM framework has been re-used and integrated in quantum compilers [35, 40, 41]. A quantum-specific IR has been created: QIR [19, 21].

6.3.4 Compilers for quantum networking.

For quantum networking, compilation has not much been studied. For distributed quantum computing, compilers do exist [7, 10, 15, 16]. One approach is to take the input circuit and first split it into separate circuits: circuit cutting [7]. However, as also explained in Chapter 1, distributed quantum computing — although also dealing with multiple nodes or cores — is different from quantum networking: in distributed quantum computing, a single compiler produces circuits for each node, whereas in quantum networking each node is independent and has its own compiler.

Intermediate representations have also been proposed for network-related quantum programs: NetQIR [53] (an extension to QIR for network-related programs), and InQuIR [45]. However, these are also for distributed quantum computing, and hence not for quantum networking which we are dealing with in this work.

6.3.5 Session types

As mentioned above, compilers may also perform correctness checking. In the context of networked programs, one could also say that correctness should include adherence to a (communication) protocol. *Session types* is a formalism used for defining communication protocols between different parties [23]. They can be used to for example specify the order in which messages need to be sent and received. By having the protocol in a well-defined format, the parties can check whether they adhere to the agreed upon protocol. Tools exist that can check at compile-time if a program adheres to a communication protocol [2, 12]. Session types may be used for two parties, but multi-party session types

also exist [24]. Multiparty session types have been considered for quantum network programs [29]. Timed session types are an extension to session types, adding explicit time constraints to operations [5]. Timed session types have not yet been applied to quantum.

6.4 Design considerations

In this section we go through considerations to take into account when designing a compiler for Qoala programs. In general, we want to re-use existing compilation strategies, but also need to address the unique challenges for compiling quantum network programs, as well as using features from Qoala like the hybrid representation and deadlines. For optimization, the metrics we are interested in are makespan and application success probability.

Program elements. Quantum network programs consist of four types of code: classical local (CL), classical communication (CC), quantum local (QL), and quantum communication (QC). For purely CL and QL segments in the source code, existing compilation techniques can be used as described in Section 6.3.

Hybrid classical-quantum code. A compiler should take advantage of the fact that the Qoala executable contains both classical and quantum code. This enables the compiler to perform optimizations on the mixed classical-quantum segments of the source code.

Networking operations. Network operations (i.e. classical communication and entanglement generation) are slow compared to local operations. For example, waiting to receive a classical message can take in the order of milliseconds (see Figure 4.2). Also, entanglement generation may take milliseconds (Figure 4.17 in Section 4.11.6). By contrast, individual local quantum operations like gates take only microseconds (Table 4.6 in Section 4.11.6) and local classical operations can be as fast as nanoseconds. As can be seen in Table 4.7, most of the time is spent on network processing.

Since quantum memory decoheres over time, a general optimization for the compiler is to minimize the time that quantum memory must remain idle during some other operation. Because of the comparably long duration of network operations, special attention must be made by the compiler to minimize the time that there is quantum memory alive while doing network operations. This may mean that the compiler should re-order operations (without changing the behavior of the program) around network operations, such as exemplified in Figure 6.4.

Non-deterministic durations. Not only are network-related operations slow, their duration is also non-deterministic (Figure 4.15 and Figure 4.17). Even with optimizations (like re-ordering local operations) to minimize the time that quantum memory must remain alive, a network operation (such as entanglement generation) may take too long — resulting in a too low-quality quantum memory to produce meaningful results. This raises two points: (1) the compiler can use Qoala's feature of *deadlines*: by adding (relative) deadlines to (parts of) the code, the compiler can hint to the scheduler to, at runtime, either abort the program or to retry a certain piece of code, and (2) the question is what deadline the compiler should insert. For this, it may be possible for the developer to insert fidelity constraints in the high-level language (or as a compiler flag), such that the compiler can in-

sert the corresponding deadlines (possibly parametrized by hardware characteristics from the Exposed Hardware Interface (EHI), see Section 5.4.3).

Correctness guarantee of compiler. A compiler must at all times guarantee that the behavior of the program does not change due to optimizations or rewrites. For quantum network programs, this also means that any communication operations (classical communication of entanglement generation) must not be re-ordered among themselves. This is because programs on other nodes in the network may expect these operations to happen in a certain order, and they cannot know of any re-orderings on this side.

Runtime performance depends on scheduler. The compiler only affects the contents of the Qoala executable. How well the program does at runtime also depends on (as shown in Section 5.6) (1) the network schedule, and (2) the node scheduler, especially in the case of multitasking (i.e. when other programs are executed concurrently with this program). Specifically, the network schedule and the node scheduler affect both the makespan (time it takes to execute the whole program) and the average application success probability (see Chapter 2 for metrics).

Private program in networked context. Recall that Qoala programs are part of multi-node applications, but that the compiler individually compiles the programs for a single node. This means that program code must be compatible with code in the programs that are to be run on other nodes. These other programs may not be in control by the developer nor the compiler. For example, in a client-server application like BQC, a server node may already have implemented the server-side program, and advertises a service for clients to connect with, by describing a protocol that a client program should adhere to. On the client side, a developer may write the client-side program. This client program must be compatible with the server-side program, including the correct order of communication operations (both classical communication and entanglement generation operations), which may also include retries. We may consider two alternatives:

- It is the developer's responsibility to make sure the client program contains the correct order of communication operations, and hence adheres to the protocol. In this case, the compiler does not specifically know about the protocol; it will just optimize the program, and makes sure it does not re-order the internal ordering of communication operations (see consideration above). However, if the developer wrote the program in such a way that the program is not fully adherent to the protocol, the compiler cannot correct for this.
- There is a formal specification of the protocol, for example in the form of session types (Section 6.3.5), which the compiler has access to. The developer must still make sure the program code adheres to the specification, but the compiler can perform an additional correctness check by evaluating protocol adherence, and warn the developer if this is not the case. If the protocol specification contains requirements about fidelity or timing, the compiler can, on top of checking for correctness, also try to meet these requirements by performing appropriate optimizations and inserting deadlines to guide the scheduler.

Target-specific optimization. The compiler may perform optimizations specific to the hardware that a program will run on. The Exposed Hardware Interface (EHI) (Section 5.4.3)

can serve as the target for a compiler, such that the compiler can insert advice (such as deadlines to code blocks) for the scheduler.

6.5 Architecture recommendations

We list recommendations for an architecture of a Qoala program compiler, based on the considerations above.

6.5.1 Translation from high-level to Qoala

Decouple front-end language from compilation framework by using Intermediate Representations (IRs). As explained in Section 6.3, classical compilers typically use a hierarchy of steps: a front-end for any high-level language to an Intermediate Representation (IR), and a back-end for IR to machine code. Using an IR decouples the main compiler design from specific input and output formats, making it more flexible and scalable. Furthermore, if an IR would be used that is already part of existing compilation frameworks, it can re-use existing compiler techniques. The Quantum Intermediate Representation (QIR) [19, 21] may be used. However, QIR uses opaque pointers for quantum types, which prevents certain optimizations [25, 46]. Multi-Level Intermediate Representation (MLIR) avoids these issues, since it allows for creating domain-specific *dialects*. Dialects have been created for quantum such that quantum types can be represented natively, allowing optimizations to be applied on operations including quantum ones [25, 46]. [37, 42] also use MLIR before then compiling to QIR.

The front-end language is not a primary concern; we want to leave design and implementations open. We also want to allow multiple (existing) high-level formats. We recommend using the existing technique of IRs, gradually lowering code (translating to a lower-level IR) and performing optimizations at each level. Specifically, we recommend MLIR [31] since it enables defining intermediate representations specifically for quantum networking, while still having access to standard (classical) LLVM optimizations.

One may consider at least three IRs (Figure 6.3):

- A high-level IR (QoalaHIR), consisting of a uniform hybrid classical-quantum representation of operations, where quantum values are native types. Native quantum values allow *value-semantics*, meaning that quantum operations consume and produce quantum state values. This is in contrast with memory semantics where qubits are seen as pointers (such as in QIR [19, 21]) and where quantum operations have side-effects. By using value-semantics, the compiler can perform (quantum) data-flow analysis in order to apply classical optimization techniques such as loop unrolling or dead code elimination [25, 46].
- A mid-level IR (QoalaMIR), where classical code is explicitly separated from quantum code. The classical code segments will eventually compile down to classical Qoala blocks (classical local and classical networking blocks, Section 5.4.2), while quantum code segments will eventually compile down to quantum Qoala blocks (local routines and request routines, Section 5.4.2). The explicit separation (1) makes it easier to perform the back-end compilation step (going from low-level IR to Qoala executable) since the formats are already similar and (2) forces the compiler to decide on *where* to make the separations, allowing it to optimize it (see below). QoalaMIR explicitly maps qubit values to

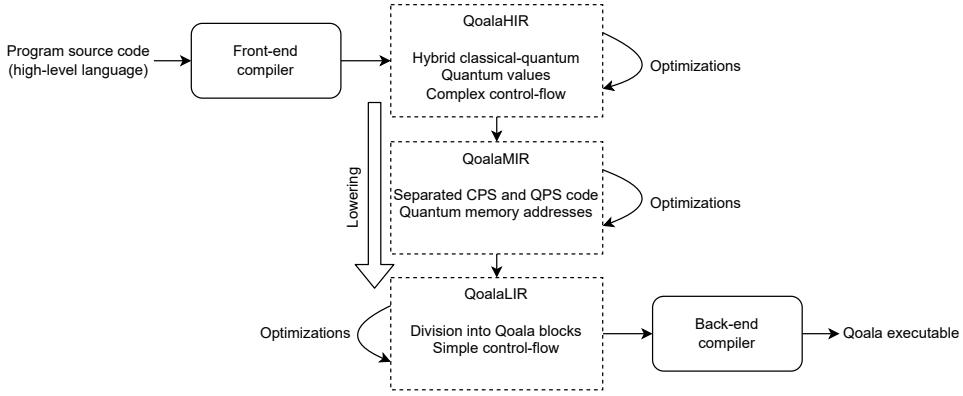


Figure 6.3: Schematic of a high-level architecture for a Qoala program compiler. Overall, the same structure is used as in Figure 6.2, but now with three IRs. Source code (in any high-level language) is first translated into a high-level IR called QoalaHIR. In this IR, classical and quantum code can be mixed arbitrarily, allowing optimizations to be applied on the whole hybrid code. QoalaHIR has quantum values as native types, which enables the compiler to do (quantum) data-flow analysis in order to apply existing optimizations that make use of these def-use chains of quantum values [46]. Control-flow may be complex. After applying optimizations on QoalaHIR, the program is translated to a mid-level IR (QoalaMIR) in which an explicit distinction is made between code that will run on the CPS and code that will be run on the QPS. Quantum values are now represented as pointers to memory locations, which means that quantum operations are in the circuit model, enabling existing quantum circuit optimization techniques. Then, a translation is made to the low-level IR QoalaLIR, in which the program is explicitly represented in a format close to the final Qoala executable. Optimizations on QoalaLIR may be hardware-specific and noise-aware. Finally, a back-end compiler translates the QoalaLIR code into an actual Qoala executable (`.iqoala`).

quantum memory locations, hence not using value-semantics. By including quantum memory, the compiler can do hardware-specific optimizations such as qubit mapping.

- A low-level IR (QoalaLIR), where operations are as close as possible to the native Qoala instructions (including QoalaHost instructions and NetQASM instructions). From this QoalaLIR representation, it is then relatively straightforward to convert the program to the Qoala executable (`.iqoala`) format.

Support fidelity and timing constraints in high-level language. As mentioned above, we do not want to specify one specific high-level language for developers to write their quantum network programs in. Instead we want to allow multiple (existing) high-level formats in order to provide flexibility in adoption. The NetQASM SDK (Chapter 3 and Chapter 4) may be used, which is implemented in Python. However, we recommend adding constructs to the high-level language for supplying fidelity and timing constraints. In this way, the developer can indicate that certain parts of the code are critical, and that the compiler should handle this by adding deadline constraints. For example, a developer may indicate that a particular qubit variable must have at least F fidelity compared to some ideal state at the moment it gets measured; the compiler, seeing that certain network-related operations happen while this qubit is alive, may estimate the fidelity of the qubit as a function of how long these operations take. In order to meet the required fidelity, then, the compiler can insert a deadline for the network-related operations corresponding to the

maximum time that still realizes fidelity F.

Use Exposed Hardware Interface (EHI) as compilation target. A compiler must have an explicit *target* that describes the software and hardware characteristics that the compiled executable must be compatible with. This target includes at least the qubit topology of the quantum memory and the NetQASM flavor (i.e. the allowed quantum instructions, see also Section 3.5.2). This information can come from the EHI. We note that the EHI also contains information about noise and durations (of gates and network operations). This information may or may not be used by the compiler for optional optimization, but is not part of the target.

Optionally use session types for protocol adherence checking. As mentioned in Section 6.4, a program typically implements some multi-node protocol that has certain requirements on communication order and possibly fidelity or timing. If such a protocol would have a formal description in the form of session types, a compiler for Qoala programs should check the program's contents and see if it adheres to the protocol.

6.5.2 Optimization

The compiler should optimize programs with respect to various metrics. Some of these metrics are those from standard compilers: memory usage and execution time (or makespan). This holds for both classical and quantum code. For classical code, existing optimization can be used from MLIR (part of LLVM), applied on the program when it is in one of the IRs (Figure 6.3). For quantum code, also existing techniques can be used, see Section 6.3.3. These include circuit mapping and gate optimization techniques.

Moreover, by using the hybrid classical-quantum format of QoalaHIR we can re-use existing optimizations such as loop unrolling and others, see [25, 37, 42, 46]. This also enables cross-subroutine optimizations, something not possible in NetQASM.

Other metrics the compiler should optimize for are (1) amount of CPS-QPS communication, since this negatively impacts execution time and possibly the time that quantum states must remain in memory, and (2) success probability of the application. In general this is application specific, but a good heuristic is to reduce the time that quantum states remain in memory, which may be done by adding deadlines that help the scheduler.

As with classical network or internet applications, the performance of quantum network applications also depends on the network itself, on which the compiler does not have influence. A large factor in performance is the fidelity (quality) of entangled states produced by the network. Although the compiler does not control this fidelity directly, it can indirectly help by doing clever capability negotiation (see Chapter 5) and adding deadlines.

Furthermore, the compiler should do noise-aware optimization including existing techniques for local quantum operations [40, 49].

Re-use existing (classical and quantum) compilation techniques for local code These include classical techniques like loop unrolling, and quantum techniques (qubit mapping, gate optimization, see Section 6.3). Existing quantum circuit optimizations can be applied to local quantum code. For network-related operations, see below.

```
q = init_local_qubit()
q.H()  # hadamard
e = create_epr()
q.H()  # hadamard
q.measure()
```

Figure 6.4: Simple program that (1) initializes a local qubit q, (2) applies a Hadamard gate on it, (3) creates an entangled qubit e, (4) applies another Hadamard gate on q and (5) measures q. q and e are completely independent in terms of application logic. Since the creation of the entangled qubit e may take a long time, executing the operations in the order they are given means that q —after the first Hadamard — must remain in memory while waiting for the entanglement generation to finish. During this time q might decohere considerably. An optimization would be to move the entanglement creation to the very end (after measuring of q) such that the initialization, gates, and measurement of q happen in one go, providing best performance.

Perform optimizations on joint classical-quantum representation Make use of the fact that classical and quantum code can be jointly analyzed and optimized. Use existing techniques, see [25, 37, 42, 46]. Use existing MLIR infrastructure.

Optimize using noise characteristics As mentioned above, the compiler's target comes from the EHI, which contains information about noise and durations (of gates and network operations). The compiler can use this information to do specific optimizations, including

- compiling to specific NetQASM flavor,
- mapping to specific qubits with better topology, and
- noise-aware optimizations such as [40, 49].

Give hints to scheduler (deadlines) In Section 6.4 we mentions that the compiler could add deadlines for parts of the code in order to meet fidelity requirements that are put in the code by a developer or that may come from a protocol description. The compiler may do so by analyzing the code and estimating the fidelity of certain quantum memory at a given moment in time, as a function of the time it took to do the operations before. It can use the EHI for this, which contains information about gate duration, estimated network operation duration, and coherence times of qubits. For this, algorithms would need to be developed to perform the estimation. The deadline can then be used by a scheduler, and depending on the scheduling algorithm, these may increase the application success probability.

Do network-specific optimizations On top of the existing optimizations for classical and quantum local code (mentioned above), the compiler should try to perform optimizations relating to networking operations, as mentioned in Section 6.4:

- Re-order operations to minimize qubit memory time (see Figure 6.4 for a simple example).
- Even without protocol description: do not re-order external operations.
- Add deadlines to code blocks to guide scheduling.

6.6 Implementation

We report on a preliminary implementation of our compiler, using the LLVM framework (C++), specifically making use of the MLIR subproject of LLVM [31]. This implementation focuses mainly on translating high-level code to a Qoala executable, using Intermediate Representation (IR)s as proposed in the previous section. We have not yet implemented any optimizations, but we emphasize that our implementation can easily incorporate optimizations by writing these as *passes* (a construct in LLVM [30]) that are applied on the IRs.

6.6.1 Overview

The Qoala compiler uses three intermediate representations (IRs): a high-level IR (QoalaHIR), a mid-level IR (QoalaMIR), and a low-level IR (QoalaLIR). Each IR is associated with a particular set of MLIR *dialects* [31]. The Qoala compiler uses four custom MLIR dialects, called `qnet`, `qmem`, `qoalahost`, and `netqasm`, explained in more detail below.

- QoalaHIR (High-level): A higher-level IR, where operations are closely related to the Python source code. Quantum operations are represented using the `qnet` dialect, which consume and produce quantum *values*. Programs in the QoalaHIR format use the following dialects: `qnet` (Qoala-specific dialect), and `arith`, `scf`, `affine`, `async`, `tensor` (default dialects in MLIR, which support complex control-flow and data structures).
- QoalaMIR (Mid-level): A mid level IR, similar to QoalaHIR but with explicit memory locations, using the Qoala-specific `qmem` dialect instead of `qnet` (`qmem` uses quantum pointers instead of quantum values).
- QoalaLIR (Low-level): A lower level IR, where operations are closer to the classical and quantum assembly instructions. Quantum operations are represented using the `netqasm` dialect, which take quantum quantum memory pointers as operands and have side-effects on the quantum value stored in the registers. Programs in the QoalaLIR format use the following dialects: `qoalahost` (Qoala-specific), `netqasm` (Qoala-specific), and `arith`, `cf`, `memref`, `async` (standard MLIR dialects). We note that only simple control-flow and data structures are supported (`cf` and `memref`), since they map more directly onto the allowed control-flow and data structures in Qoala executables.

6.6.2 Lowering passes

We have implemented the following compiler passes for lowering QoalaHIR code into QoalaLIR code (Figure 6.3).

QoalaHIR to QoalaMIR. Go from a value-semantics representation (qubits are values that are consumed and produced by operations) in the `qnet` dialect to a memory-semantics representation (qubits are pointers and quantum operations acting on qubit pointers have side effects) in the `qmem` dialect.

QoalaMIR functionizing. Split QoalaMIR code into functions. This 'functionizing' is done in such a way that the program structure already resembles the final Qoala executable structure.

Functionizing algorithm:

1. Loop over all operations in the order they are given until either a classical receive operation or an entanglement operation is found. (When the end of the program is reached, exit.)
2. Create one or more new functions:
   - If a classical receive operation was encountered: gather all quantum operations found so far, plus all (possibly non-quantum) operations that produce values used by these quantum operations. Put all these operations inside a new function and replace the original QoalaMIR code with a call to this new function.
   - If an entanglement operation was encountered: gather all *non-entanglement* operations found so far, and put them in a new function (F1) (like in the previous point). Additionally, create a second new function (F2) containing the entanglement operation itself, including the corresponding allocation operation. Replace the original QoalaMIR code with calls to F1 and F2.
3. Go back to (1) starting at the operation after the previously found receive or entanglement operation. Note that classical receive operations always remain *outside* newly created functions, whereas entanglement operations are always moved *into* their own new functions.

QoalaMIR to QoalaLIR. Convert operations inside functions to the `netqasm` dialect. Convert other operations to the `qoalahost` dialect. The format is now already close to a Qoala executable (`.iqoala`).

6.6.3 Python SDK

We have implemented a new, bare-bones, SDK for the purpose of trying our compiler; in the future we aim to merge this with the NetQASM SDK (Chapter 3) such that programs written in this existing SDK can immediately be compiled using our compiler. The SDK enables the following program development flow:

- Programmer writes source code in Python using the Qoala Python SDK.
- The Python SDK itself provides a `compile` function that executes the source code which produces a `.mlir` file containing the program code in QoalaHIR format.
- This Python SDK implements *some* functionality present in the "frontend" of a classical compiler, for example, a semantical analysis to check the validity of the classical and quantum operations specified in the program.
- A separate `opt`-like tool (built using LLVM/MLIR) takes the `.mlir` as input and produces (after several applying passes) a `.iqoala` file.
- The `.iqoala` file can then be executed by a Qoala runtime.

6.6.4 Example lowering

Take the below example program written in the high-level Qoala Python SDK. The program first creates an entangled qubit (`q`) with remote node Bob, after which a Hadamard gate is applied on `q`. Then, the program waits to receive a classical real number (a float) as a classical message from Bob, calling it `t1`. An x-rotation gate is applied on `q` with angle

`t1`. Finally, `q` is measured, producing bit `m` which is returned as result of this program.

```
class MyProgram(QoalaProgram):
    def main(self, ctx: QoalaContext):
        q = ctx.entangle_keep("Bob")
        ctx.hadamard(q)
        t1 = ctx.recv_float("Bob")
        ctx.rot_x(q, t1)
        m = ctx.measure(q)
        ctx.add_return(m)

program = MyProgram()
mlir = program.compile()  # 'compile' is defined in 'QoalaProgram'
with open("program.mlir", "w") as f:
    f.write(mlir.text())
```

The `program.compile()` method produces an in-memory representation of this program but in QoalaHIR format. Finally, it is written to the file `program.mlir`, which can then be fed to `qoala-opt`, the main Qoala compiler. The QoalaHIR format looks as follows:

```
qnet.func @main() -> i1 {
    %q = qnet.eprs {N = 1, remote = @Bob} : !qnet.qubit
    %q2 = qnet.hadamard %q : !qnet.qubit
    %floats1 = qnet.recv_floats {remote = @Bob, length = 1 : i32} : tensor<1xf32
  >
    %t1 = tensor.extract %floats1[%zero] : tensor<1xf32>
    %q3 = qnet.rot_x %q2, %t1 : !qnet.qubit
    %m = qnet.measure %q3 : i1
    qnet.return %m : i1
}
```

As can be seen, the variable `q` from the original Python program is now called `%q` (using `%` is MLIR syntax), and is the result of the `eprs` operations defined in the `qnet` dialect. This operation produces a variable of type `!qnet.qubit`, a native `qnet` type.

The hadamard operation in the `qnet` dialect *consumes* `!qnet.qubit` values (in this case, `%q`) and *produces a new* `!qnet.qubit` value (in this case, `%q2`). Using value-semantics (consuming and producing values) allows for already-existing MLIR optimization passes (such as dead code elimination, but not shown in this example) to be applied on our custom `qnet` dialect as well.

Lowering QoalaHIR to QoalaMIR. When lowering the above QoalaHIR code to QoalaMIR, the change is that instead of using value-semantics for qubits, now explicit allocations are done for quantum memory. Quantum operations are applied on quantum pointers, and have side effects. (In contrast to consuming a quantum *value* and producing a new one.) As can be seen below, the `qnet.eprs` operation producing qubit variable `%q` is replaced by a `qalloc` operation (in the `qmem` dialect), producing a qubit *pointer* `%qptr` (not a quantum value), followed by a `eprs` operation (in the `qmem` dialect instead of the `qnet` dialect) which acts on `%qptr`.

```
qmem.func @main() -> i1 {
    %qptr = qmem.qalloc : i32
    qmem.eprs %qptr
    qmem.hadamard %qptr
    %floats1 = qmem.recv_floats {remote = @Bob, length = 1 : i32} : tensor<1xf32
  >
    %t1 = tensor.extract %floats1[%zero] : tensor<1xf32>
    qmem.rot_x %qptr, %t1
    %m = qmem.measure %qptr : i1
    qmem.return %m : i1
}
```

Functionizing QoalaMIR. On QoalaMIR code, a *functionizing* pass is applied, which splits quantum code into separate functions that are called from the main function. The main function itself only keeps classical instructions. Here we see the first split that eventually becomes the split between CPS code (in the main function) and QPS code (in the separate functions). The result of applying the `functionize` pass on the above QoalaMIR code looks like this:

```
qmem.func @routine1() -> i32 {
    %qptr = qmem.qalloc : i32
    qmem.eprs %qptr
    qmem.return %qptr : i32
}

qmem.func @routine2(%qptr: i32) -> i32 {
    qmem.hadamard %qptr
    qmem.return %qptr : i32
}

qmem.func @routine3(%qptr: i32, %t1: f32) -> i1 {
    qmem.rot_x %qptr, %t1
    %m = qmem.measure %qptr : i1
}

qmem.func @main() -> i1 {
    %qptr = qmem.call @routine1() : () -> i32
    qmem.call @routine2(%qptr) : () -> i32
    %floats1 = qmem.recv_floats {remote = @Bob, length = 1 : i32} : tensor<1xf32
  >
    %t1 = tensor.extract %floats1[%zero] : tensor<1xf32>
    %m = call @routine3(%qptr1, %t1) : (i32, f32) -> i1
    qmem.return %m : i1
}
```

Lowering functionized QoalaMIR to QoalaLIR. When lowering to QoalaLIR, the quantum functions are replaced by either `request_routine` operations (for functions containing EPR creation operations) or `local_routine` operations (for function with only local quantum operations). Moreover, these functions are now all in the `netqasm` dialect. The

code in the main function is replaced by appropriate operations in the `qoalahost` dialect. As can be seen below, the structure of the resulting QoalaLIR code already looks close to the final Qoala executable (`.iqoala`) format: `netqasm.request_routine` functions can be mapped to Qoala Request Routines, `netqasm.local_routine` functions can be mapped to Qoala Local Routines, and the `main_func` code can be mapped to Qoala Host code (Section 5.4.2).

```
netqasm.request_routine @req1() -> i32 {
    %vqubit = netqasm.qalloc : i32
    netqasm.eprs %vqubit {remote = @Bob}
    return %vqubit : i32
}

netqasm.local_routine @subrt2(%vqubit: i32) {
    netqasm.hadamard %vqubit
    return
}

netqasm.local_routine @subrt3(%vqubit: i32, %num: i32, %denom: i32) {
    netqasm.rot_x %vqubit, %num, %denom
    %m = netqasm.measure %vqubit : i1
    return %m : i1
}

qoalahost.main_func @main() -> i1{
    %zero = arith.constant 0 : index
    %vqubit = qoalahost.call @req1() : () -> i32
    netqasm.call @subrt2(%vqubit, %num1, %denom1) : (i32, i32, i32) -> ()

    %floats1 = qoalahost.recv_floats {remote = @Bob, length = 1 : i32} : tensor
  <1xf32>
    %t1 = tensor.extract %floats1[%zero] : tensor<1xf32>
    %num1, %denom1 = func.call @conver_float_to_num_and_denom(%t1) : (f32) -> (
  i32, i32)

    %m = netqasm.call @subrt3(%vqubit, %num1, %denom1) : (i32, i32, i32) -> i1
    qoalahost.return %m : i1
}
```

6.7 Conclusion

We have discussed design considerations for a Qoala program compiler, and proposed a high-level architecture for such a compiler. Our overall recommendation is to re-use existing compilation techniques wherever possible, such as for purely classical code segments and purely local quantum code segments. We also recommend using existing infrastructure such as MLIR in order to (1) re-use existing techniques and (2) represent the hybrid nature of quantum network programs. Moreover, we have provided pointers for (quantum) network-specific optimizations, including instruction re-ordering and inserting deadlines to code blocks.

More research is needed for the ideas presented in this chapter before a detailed compiler design can be completed. For example, how should developers describe their fidelity

constraints in high-level source code? How can a compiler translate such constraints into deadlines? Also, the idea of using session types for protocol descriptions must be investigated more. Finally, evaluation of the compiler design is crucial to test the merit of these ideas and to guide further research. Evaluation may be done by inspecting the compilation output (for instance, the number of (blocks of) instructions in the Qoala executable) or the runtime performance. However, we note that the runtime performance (including makespan and application success probability) also depends on the node scheduler and network schedule. Therefore, future research may want to focus on the joint compilation-scheduling problem.

References

[1] A. Aho, J. Ullman, R. Sethi, and M. Lam. *Compilers: Principles, Techniques, and Tools*. 2nd edition. Boston Munich: Addison Wesley, Aug. 31, 2006. 1040 pp. ISBN: 978-0-321-48681-3.

[2] E. Ardeshir-Larijani, S. J. Gay, and R. Nagarajan. "Automated Equivalence Checking of Concurrent Quantum Systems". In: *ACM Trans. Comput. Logic* 19.4 (Nov. 20, 2018), 28:1–28:32. ISSN: 1529-3785. DOI: 10.1145/3231597. URL: https://dl.acm.org/doi/10.1145/3231597 (visited on Aug. 8, 2024).

[3] M. Bandic, S. Feld, and C. G. Almudever. "Full-stack quantum computing systems in the NISQ era: algorithm-driven and hardware-aware compilation techniques". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE). ISSN: 1558-1101. Mar. 2022, pp. 1–6. DOI: 10.23919/DATE54114.2022.9774643. URL: https://ieeexplore.ieee.org/abstract/document/9774643 (visited on Aug. 8, 2024).

[4] M. Bandic, L. Prielinger, J. Nüßlein, A. Ovide, S. Rodrigo, S. Abadal, H. van Someren, G. Vardoyan, E. Alarcon, C. G. Almudever, and S. Feld. "Mapping Quantum Circuits to Modular Architectures with QUBO". In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2023 IEEE International Conference on Quantum Computing and Engineering (QCE). Vol. 01. Sept. 2023, pp. 790–801. DOI: 10.1109/QCE57702.2023.00094. URL: https://ieeexplore.ieee.org/abstract/document/10313719 (visited on Aug. 13, 2024).

[5] M. Bartoletti, T. Cimoli, and M. Murgia. "Timed Session Types". In: *Logical Methods in Computer Science* Volume 13, Issue 4 (Dec. 8, 2017). Publisher: Episciences.org. ISSN: 1860-5974. DOI: 10.23638/LMCS-13(4:25)2017. URL: https://lmcs.episciences.org/4133 (visited on Aug. 8, 2024).

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. "Register allocation via coloring". In: *Computer Languages* 6.1 (Jan. 1, 1981), pp. 47–57. ISSN: 0096-0551. DOI: 10.1016/0096-0551(81)90048-5. URL: https://www.sciencedirect.com/science/article/pii/0096055181900485 (visited on Oct. 24, 2024).

[7] T. Chatterjee, A. Das, S. I. Mohtashim, A. Saha, and A. Chakrabarti. "Qurzon: A Prototype for a Divide and Conquer-Based Quantum Compiler for Distributed Quantum Systems". In: *SN Computer Science* 3.4 (June 10, 2022), p. 323. ISSN: 2661-8907. DOI: 10.1007/s42979-022-01207-9. URL: https://doi.org/10.1007/s42979-022-01207-9 (visited on Aug. 13, 2024).

[8] F. T. Chong, D. Franklin, and M. Martonosi. "Programming languages and compiler design for realistic quantum hardware". In: *Nature* 549.7671 (Sept. 2017). Publisher: Nature Publishing Group, pp. 180–187. ISSN: 1476-4687. DOI: 10.1038/nature23459. URL: https://www.nature.com/articles/nature23459 (visited on Sept. 13, 2024).

[9] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. *Open Quantum Assembly Language*. July 13, 2017. DOI: 10.48550/arXiv.1707.03429. arXiv: 1707.03429[quant-ph]. URL: http://arxiv.org/abs/1707.03429 (visited on Aug. 13, 2024).

[10] D. Cuomo, M. Caleffi, K. Krsulich, F. Tramonto, G. Agliardi, E. Prati, and A. S. Cacciapuoti. "Optimized Compiler for Distributed Quantum Computing". In: *ACM Transactions on Quantum Computing* 4.2 (Feb. 24, 2023), 15:1–15:29. DOI: 10.1145/3579367. URL: https://dl.acm.org/doi/10.1145/3579367 (visited on Aug. 13, 2024).

[11] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. "MPI-aware compiler optimizations for improving communication-computation overlap". In: *Proceedings of the 23rd international conference on Supercomputing*. ICS '09. New York, NY, USA: Association for Computing Machinery, June 8, 2009, pp. 316–325. ISBN: 978-1-60558-498-0. DOI: 10.1145/1542275.1542321. URL: https://doi.org/10.1145/1542275.1542321 (visited on Oct. 11, 2024).

[12] T. Davidson, S. J. Gay, H. M. Ik, R. Nagarajan, and N. Papanikolaou. "Model Checking for Communicating Quantum Processes". In: ().

[13] S. DiAdamo, M. Ghibaudi, and J. Cruise. "Distributed quantum computing and network control for accelerated vqe". In: *IEEE Transactions on Quantum Engineering* 2 (2021), pp. 1–21. DOI: 10.1109/TQE.2021.3057908.

[14] E. Farhi, J. Goldstone, and S. Gutmann. "A quantum approximate optimization algorithm". In: *arXiv preprint arXiv:1411.4028* (2014). DOI: 10.48550/arXiv.1411.4028.

[15] D. Ferrari, A. S. Cacciapuoti, M. Amoretti, and M. Caleffi. "Compiler Design for Distributed Quantum Computing". In: *IEEE Transactions on Quantum Engineering* 2 (2021). Conference Name: IEEE Transactions on Quantum Engineering, pp. 1–20. ISSN: 2689-1808. DOI: 10.1109/TQE.2021.3053921. URL: https://ieeexplore.ieee.org/abstract/document/9334411 (visited on Aug. 8, 2024).

[16] D. Ferrari, S. Carretta, and M. Amoretti. "A Modular Quantum Compilation Framework for Distributed Quantum Computing". In: *IEEE Transactions on Quantum Engineering* 4 (2023), pp. 1–13. ISSN: 2689-1808. DOI: 10.1109/TQE.2023.3303935. arXiv: 2305.02969[quant-ph]. URL: http://arxiv.org/abs/2305.02969 (visited on Sept. 13, 2024).

[17] D. J. Frailey. "An intermediate language for source and target independent code optimization". In: *Proceedings of the 1979 SIGPLAN symposium on Compiler construction*. SIGPLAN '79. New York, NY, USA: Association for Computing Machinery, Aug. 1, 1979, pp. 188–200. ISBN: 978-0-89791-002-6. DOI: 10.1145/800229.806969. URL: https://dl.acm.org/doi/10.1145/800229.806969 (visited on Oct. 24, 2024).

[18] FuX, YuJintao, SuXing, JiangHanru, WuHua, ChengFucheng, DengXi, ZhangJinrong, JinLei, YangYihang, XuLe, HuChunchao, HuangAnqi, HuangGuangyao, QiangXiaogang, DengMingtang, XuPing, XuWeixia, LiuWanwei, ZhangYu, DengYuxin, WuJunjie, and FengYuan. "Quingo: A Programming Framework for Heterogeneous Quantum-Classical Computing with NISQ Features". In: *ACM Transactions on Quantum Computing* (Dec. 21, 2021). Publisher: ACMPUB27New York, NY. DOI: 10.1145/3483528. URL: https://dl.acm.org/doi/10.1145/3483528 (visited on Aug. 13, 2024).

[19] A. Geller. *Introducing Quantum Intermediate Representation (QIR)*. Q# Blog. Sept. 23, 2020. URL: https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/ (visited on Sept. 23, 2024).

[20] R. L. Glass. "An elementary discussion of compiler/interpreter writing". In: *ACM Computing Surveys (CSUR)* 1.1 (1969), pp. 55–77.

[21] T. Häner, D. S. Steiger, K. Svore, and M. Troyer. "A software methodology for compiling quantum programs". In: *Quantum Science and Technology* 3.2 (Feb. 2018). Publisher: IOP Publishing, p. 020501. ISSN: 2058-9565. DOI: 10.1088/2058-9565/aaa5cc. URL: https://dx.doi.org/10.1088/2058-9565/aaa5cc (visited on Aug. 19, 2024).

[22] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore. "Quantum programming languages". In: *Nature Reviews Physics* 2.12 (Dec. 2020). Publisher: Nature Publishing Group, pp. 709–722. ISSN: 2522-5820. DOI: 10.1038/s42254-020-00245-7. URL: https://www.nature.com/articles/s42254-020-00245-7 (visited on Aug. 13, 2024).

[23] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language primitives and type discipline for structured communication-based programming". In: *Programming Languages and Systems*. Ed. by C. Hankin. Berlin, Heidelberg: Springer, 1998, pp. 122–138. ISBN: 978-3-540-69722-0. DOI: 10.1007/BFb0053567.

[24] K. Honda, N. Yoshida, and M. Carbone. "Multiparty Asynchronous Session Types". In: *J. ACM* 63.1 (Mar. 3, 2016), 9:1–9:67. ISSN: 0004-5411. DOI: 10.1145/2827695. URL: https://dl.acm.org/doi/10.1145/2827695 (visited on Aug. 8, 2024).

[25] D. Ittah, T. Häner, V. Kliuchnikov, and T. Hoefler. "Enabling Dataflow Optimization for Quantum Programs". In: *ACM Transactions on Quantum Computing* 3.3 (Sept. 30, 2022), pp. 1–32. ISSN: 2643-6809, 2643-6817. DOI: 10.1145/3491247. arXiv: 2101.11030[quant-ph]. URL: http://arxiv.org/abs/2101.11030 (visited on Sept. 23, 2024).

[26] P. Khalate, X.-C. Wu, S. Premaratne, J. Hogaboam, A. Holmes, A. Schmitz, G. G. Guerreschi, X. Zou, and A. Y. Matsuura. *An LLVM-based C++ Compiler Toolchain for Variational Hybrid Quantum-Classical Algorithms and Quantum Accelerators*. Feb. 22, 2022. DOI: 10.48550/arXiv.2202.11142. arXiv: 2202.11142[quant-ph]. URL: http://arxiv.org/abs/2202.11142 (visited on Aug. 8, 2024).

[27] N. Khammassi, I. Ashraf, J. V. Someren, R. Nane, A. M. Krol, M. A. Rol, L. Lao, K. Bertels, and C. G. Almudever. "OpenQL: A Portable Quantum Programming Framework for Quantum Accelerators". In: *J. Emerg. Technol. Comput. Syst.* 18.1 (Dec. 20, 2021), 13:1–13:24. ISSN: 1550-4832. DOI: 10.1145/3474222. URL: https://dl.acm.org/doi/10.1145/3474222 (visited on Sept. 13, 2024).

[28] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. *cQASM v1.0: Towards a Common Quantum Assembly Language*. May 24, 2018. DOI: 10.48550/arXiv.1805.09607. arXiv: 1805.09607[quant-ph]. URL: http://arxiv.org/abs/1805.09607 (visited on Sept. 23, 2024).

[29] I. Lanese, U. D. Lago, and V. Choudhury. *Towards Quantum Multiparty Session Types*. Sept. 17, 2024. DOI: 10.48550/arXiv.2409.11133. arXiv: 2409.11133[cs]. URL: http://arxiv.org/abs/2409.11133 (visited on Sept. 24, 2024).

[30] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* International Symposium on Code Generation and Optimization, 2004. CGO 2004. Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. URL: https://ieeexplore.ieee.org/abstract/document/1281665 (visited on Oct. 9, 2024).

[31] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. Mar. 1, 2020. DOI: 10.48550/arXiv.2002.11054. arXiv: 2002.11054. URL: http://arxiv.org/abs/2002.11054 (visited on Oct. 25, 2024).

[32] A. Li, S. Stein, S. Krishnamoorthy, and J. Ang. "QASMBench: A Low-Level Quantum Benchmark Suite for NISQ Evaluation and Simulation". In: *ACM Transactions on Quantum Computing* 4.2 (Feb. 24, 2023), 10:1–10:26. DOI: 10.1145/3550488. URL: https://dl.acm.org/doi/10.1145/3550488 (visited on Sept. 13, 2024).

[33] G. Li, Y. Ding, and Y. Xie. "Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 4, 2019, pp. 1001–1014. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304023. URL: https://dl.acm.org/doi/10.1145/3297858.3304023 (visited on Aug. 13, 2024).

[34] H. Li, P. Zhang, G. Sun, C. Hu, D. Shan, T. Pan, and Q. Fu. "A modular compiler for network programming languages". In: *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. CoNEXT '20. New York, NY, USA: Association for Computing Machinery, Nov. 24, 2020, pp. 198–210.

ISBN: 978-1-4503-7948-9. DOI: 10.1145/3386367.3432063. URL: https://dl.acm.org/doi/10.1145/3386367.3432063 (visited on Oct. 9, 2024).

[35] A. Litteken, Y.-C. Fan, D. Singh, M. Martonosi, and F. T. Chong. "An updated LLVM-based quantum research compiler with further OpenQASM support". In: *Quantum Science and Technology* 5.3 (May 2020). Publisher: IOP Publishing, p. 034013. ISSN: 2058-9565. DOI: 10.1088/2058-9565/ab8c2c. URL: https://dx.doi.org/10.1088/2058-9565/ab8c2c (visited on Aug. 8, 2024).

[36] X. Liu, A. Angone, R. Shaydulin, I. Safro, Y. Alexeev, and L. Cincio. "Layer VQE: A variational approach for combinatorial optimization on noisy quantum computers". In: *IEEE Transactions on Quantum Engineering* 3 (2022), pp. 1–20. DOI: 10.1109/TQE.2021.3140190.

[37] A. McCaskey and T. Nguyen. "A MLIR Dialect for Quantum Assembly Languages". In: *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2021 IEEE International Conference on Quantum Computing and Engineering (QCE). Oct. 2021, pp. 255–264. DOI: 10.1109/QCE52317.2021.00043. URL: https://ieeexplore.ieee.org/abstract/document/9605269 (visited on Aug. 8, 2024).

[38] A. J. McCaskey, D. I. Lyakh, E. F. Dumitrescu, S. S. Powers, and T. S. Humble. "XACC: a system-level software infrastructure for heterogeneous quantum–classical computing*". In: *Quantum Science and Technology* 5.2 (Feb. 2020). Publisher: IOP Publishing, p. 024002. ISSN: 2058-9565. DOI: 10.1088/2058-9565/ab6bf6. URL: https://dx.doi.org/10.1088/2058-9565/ab6bf6 (visited on Aug. 13, 2024).

[39] C. Monsanto, N. Foster, R. Harrison, and D. Walker. "A compiler and run-time system for network programming languages". In: *SIGPLAN Not.* 47.1 (Jan. 25, 2012), pp. 217–230. ISSN: 0362-1340. DOI: 10.1145/2103621.2103685. URL: https://dl.acm.org/doi/10.1145/2103621.2103685 (visited on Oct. 9, 2024).

[40] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi. "Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 4, 2019, pp. 1015–1029. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304075. URL: https://dl.acm.org/doi/10.1145/3297858.3304075 (visited on Sept. 13, 2024).

[41] P. Murali, N. M. Linke, M. Martonosi, A. J. Abhari, N. H. Nguyen, and C. H. Alderete. "Full-stack, real-system quantum computer studies: architectural comparisons and design insights". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. New York, NY, USA: Association for Computing Machinery, June 22, 2019, pp. 527–540. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322273. URL: https://dl.acm.org/doi/10.1145/3307650.3322273 (visited on Aug. 8, 2024).

[42] T. Nguyen and A. McCaskey. "Retargetable Optimizing Compilers for Quantum Accelerators via a Multilevel Intermediate Representation". In: *IEEE Micro* 42.5 (Sept. 2022). Conference Name: IEEE Micro, pp. 17–33. ISSN: 1937-4143. DOI: 10.1109/MM.

2022.3179654. URL: https://ieeexplore.ieee.org/abstract/document/9795194 (visited on Sept. 23, 2024).

[43] R. Nigam, S. Thomas, Z. Li, and A. Sampson. "A compiler infrastructure for accelerator generators". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, Apr. 17, 2021, pp. 804–817. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446712. URL: https://dl.acm.org/doi/10.1145/3445814.3446712 (visited on Aug. 8, 2024).

[44] S. Nishio, Y. Pan, T. Satoh, H. Amano, and R. V. Meter. "Extracting Success from IBM's 20-Qubit Machines Using Error-Aware Compilation". In: *J. Emerg. Technol. Comput. Syst.* 16.3 (May 28, 2020), 32:1–32:25. ISSN: 1550-4832. DOI: 10.1145/3386162. URL: https://dl.acm.org/doi/10.1145/3386162 (visited on Sept. 13, 2024).

[45] S. Nishio and R. Wakizaka. *InQuIR: Intermediate Representation for Interconnected Quantum Computers*. Feb. 1, 2023. DOI: 10.48550/arXiv.2302.00267. arXiv: 2302.00267[quant-ph]. URL: http://arxiv.org/abs/2302.00267 (visited on Aug. 8, 2024).

[46] A. Peduri, S. Bhat, and T. Grosser. "QSSA: an SSA-based IR for Quantum computing". In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. CC 2022. New York, NY, USA: Association for Computing Machinery, Mar. 18, 2022, pp. 2–14. ISBN: 978-1-4503-9183-2. DOI: 10.1145/3497776.3517772. URL: https://dl.acm.org/doi/10.1145/3497776.3517772 (visited on Sept. 23, 2024).

[47] P. B. Schneck. "A survey of compiler optimization techniques". In: *Proceedings of the ACM annual conference*. ACM '73. New York, NY, USA: Association for Computing Machinery, Aug. 27, 1973, pp. 106–113. ISBN: 978-1-4503-7490-3. DOI: 10.1145/800192.805690. URL: https://dl.acm.org/doi/10.1145/800192.805690 (visited on Oct. 24, 2024).

[48] J. E. Smith. "A study of branch prediction strategies". In: *Proceedings of the 8th annual symposium on Computer Architecture*. ISCA '81. Washington, DC, USA: IEEE Computer Society Press, May 12, 1981, pp. 135–148. (Visited on Oct. 24, 2024).

[49] K. N. Smith, G. S. Ravi, P. Murali, J. M. Baker, N. Earnest, A. Javadi-Abhari, and F. T. Chong. *Error Mitigation in Quantum Computers through Instruction Scheduling*. Nov. 10, 2021. DOI: 10.48550/arXiv.2105.01760. arXiv: 2105.01760[quant-ph]. URL: http://arxiv.org/abs/2105.01760 (visited on Aug. 23, 2024).

[50] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler. "Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL". In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*. RWDSL2018. New York, NY, USA: Association for Computing Machinery, Feb. 24, 2018, pp. 1–10. ISBN: 978-1-4503-6355-6. DOI: 10.1145/3183895.3183901. URL: https://dl.acm.org/doi/10.1145/3183895.3183901 (visited on Oct. 9, 2024).

[51] T. Tomesh, P. Gokhale, V. Omole, G. S. Ravi, K. N. Smith, J. Viszlai, X.-C. Wu, N. Hardavellas, M. R. Martonosi, and F. T. Chong. "SupermarQ: A Scalable Quantum Benchmark Suite". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). ISSN: 2378-203X. Apr. 2022, pp. 587–603. DOI: 10.1109/HPCA53966.2022.00050. URL: https://ieeexplore.ieee.org/abstract/document/9773202 (visited on Oct. 24, 2024).

[53] J. Vázquez-Pérez, F. J. Cardama, C. Piñeiro, T. F. Pena, J. C. Pichel, and A. Gómez. *NetQIR: An Extension of QIR for Distributed Quantum Computing*. Aug. 7, 2024. DOI: 10.48550/arXiv.2408.03712. arXiv: 2408.03712[quant-ph]. URL: http://arxiv.org/abs/2408.03712 (visited on Sept. 23, 2024).

[54] R. L. Wexelblat, ed. *History of programming languages*. New York, NY, USA: Association for Computing Machinery, May 1978. 782 pp. ISBN: 978-0-12-745040-7.

[55] E. Younis and C. Iancu. "Quantum Circuit Optimization and Transpilation via Parameterized Circuit Instantiation". In: *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2022 IEEE International Conference on Quantum Computing and Engineering (QCE). Sept. 2022, pp. 465–475. DOI: 10.1109/QCE53715.2022.00068. URL: https://ieeexplore.ieee.org/abstract/document/9951320 (visited on Aug. 8, 2024).

[56] Y. Zhang, H. Deng, and Q. Li. *Context-Sensitive and Duration-Aware Qubit Mapping for Various NISQ Devices*. Jan. 19, 2020. DOI: 10.48550/arXiv.2001.06887. arXiv: 2001.06887[quant-ph]. URL: http://arxiv.org/abs/2001.06887 (visited on Aug. 23, 2024).

7

Conclusion

In this final chapter, we provide a brief summary of the work presented in this thesis and list possible future lines of research and development.

7.1 Summary of results

In Chapter 1 we have explained that for quantum networks, there was previously no programming and execution framework for quantum network (or internet) applications. Our main goal has hence been to *enable programming and execution of arbitrary quantum network applications in a hardware-agnostic way while optimizing runtime performance.* In Chapters 3 to 6 we presented novel (system) architectures towards achieving this goal.

NetQASM. In Chapter 3 we introduced NetQASM — the first ever instruction set that can express quantum network instructions including remote entanglement generation. NetQASM enables the development of quantum network applications in a hardware-agnostic manner by providing a high-level SDK, and by making the instruction set hardware-independent (although we also allow hardware-specific optimizations). We also introduced a model of execution for quantum network nodes, which consists of a CNPU performing classical processing and communication with other nodes, and a QNPU responsible for executing quantum instructions as well as delivering entanglement to user applications. By using virtualized quantum memory spaces, NetQASM allows multitasking on a node: the concurrent execution of multiple applications, in order to increase device utilization. We validated and evaluated NetQASM in simulation.

QNodeOS. In Chapter 4 we extended our execution model from Chapter 3, and presented the first ever full-stack system architecture for quantum network nodes — QNodeOS — for executing arbitrary applications on quantum network nodes. QNodeOS is an architecture that spans both CNPU and QNPU, and describes how user applications written with the NetQASM SDK can be executed, by also incorporating the network stack [1, 4]. We have validated our architecture by implementing it on real hardware: we showed the successful execution of simple quantum network applications on a setup of two nodes based on NV centers. Moreover we have shown that QNodeOS is platform-agnostic by also connecting

it to a quantum device with trapped ions. Finally we validated that QNodeOS can do multitasking of multiple applications and can increase quantum device utility this way.

Qoala. We learned from QNodeOS that there are opportunities to improve compilation and scheduling, which will then lead to better runtime performance. In Chapter 5 we explained these opportunities and proposed an improved architecture — Qoala — for executing applications on quantum network nodes. In the Qoala architecture, the split between CNPU and QNPU (in Chapter 5 called CPS and QPS) is lessened: Qoala describes a hybrid classical-quantum format of programs, allowing more advanced compilation, and a node-wide scheduler that allows more advanced scheduling. We evaluated how Qoala enables improvement of both compilation and scheduling in simulation.

Compiler. While the Qoala architecture enables better compilation in the first place, in Chapter 6 we discussed how such a compiler could look like. We discussed design considerations and proposed a high-level architecture for such a compiler. Our overall recommendation is to re-use existing compilation techniques wherever possible, such as for purely classical code segments and purely local quantum code segments. We also recommend using existing infrastructure such as MLIR in order to (1) re-use existing techniques and (2) represent the hybrid nature of quantum network programs. Moreover, we have provided pointers for (quantum) network-specific optimizations, including instruction reordering and inserting deadlines to code blocks.

7.2 Future work

We believe that our work has contributed to addressing the objective of enabling programming and execution of arbitrary quantum network nodes, by designing and implementing the frameworks mentioned above. Still, there remain areas for improvement and more research.

Implement Qoala on real hardware. Although we have validated Qoala's improvements over QNodeOS in simulation, it is important to validate this on a real implementation as well. A real implementation of Qoala, by upgrading the existing QNodeOS implementation would enable running the same experiments on real hardware with Qoala as we did with QNodeOS in Chapter 4.

Scaling up. All the work presented here has been evaluated on small networks, whether it be on real hardware (Chapter 4) or in simulation (Chapters 3 and 5). Evaluating QNodeOS and Qoala on larger systems (meaning: larger networks, more applications, more qubits per node, more instructions per program) can give insights into how much our architecture scales, and whether changes in the design have to be made before larger scale adoption can be realized.

Qoala compiler. In Chapter 6 we discussed possibilities for the design of a compiler for Qoala programs. Finalizing the design and then implementing it would enable evaluation (in simulation or on hardware), which would in turn point to future improvements of the design. It would be especially interesting to see how compilation strategies may be combined with scheduling strategies, because of their interplay as shown in Chapter 5.

Advanced scheduling algorithms. In Chapter 5 we evaluated only simple node scheduling algorithms (first-come-first-serve and earliest-deadline-first) since our main objective was to showcase the ability to do node scheduling in the first place, rather than coming up with an optimized scheduler. More sophisticated scheduling strategies may lead to higher success probabilities and lower makespan when concurrently executing multiple program instances, where inspiration may come from existing strategies such as time-utility functions (TUF, see e.g. [3, 5]) to inform scheduling decisions, and where it is an open question how such TUFs could even be defined in the quantum domain. Further research on the fundamental tradeoffs between the classical (makespan) and quantum (success probability) performance metrics may also guide further design choices on scheduling algorithms.

Investigate using Qoala for distributed quantum computing. Although Qoala has been designed with quantum network (or internet) applications in mind, it may also be used for distributed quantum computing (DQC, although not to be confused with Delegated Quantum Computation from Chapter 4). Since DQC only needs to do limited classical communication (sending teleportation corrections), Qoala's architecture of having a separate CPS doing arbitrary classical computation and communication may not be the most efficient solution. However, it may be that overhead is minimal and is made up for by being able to use the same architecture (namely Qoala) as for more general quantum network applications. Indeed, using Qoala may enable DQC in larger-scale networks comprised of autonomous nodes (like a quantum internet), something not immediately possible with existing frameworks such as [2] which assume central control over all nodes. Because of the more predictable nature of DQC applications (there is typically no complex control-flow depending on runtime information) simple node scheduling algorithms may suffice, although this needs to be investigated.

Capability negotiation. In Section 5.4.5 we mentioned that the compiler provides advice that the nodes use in capability negotiation and demand registration (Section 5.4.5). It is an open question how to best compute such advice, and find efficient protocols for negotiating capabilities and register demand. This must be fleshed out in coordination with network scheduling research.

Network schedule. As expected, our evaluation shows that application performance depends on the network schedule, where we emphasize that ensuring network service is out of scope for Qoala as en environment for executing applications. This highlights a need for understanding the quality of service a quantum network should provide, as well as to design good network scheduling algorithms to satisfy them, in order to achieve good application performance.

References

[1] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveria Filho, R. Hanson, and S. Wehner. "A Link Layer Protocol for Quantum Networks". In: *ACM SIGCOMM 2019 Conference*. SIGCOMM '19. Beijing, China: ACM, 2019, p. 15. ISBN: 978-1-4503-5956-6/19/09. DOI: 10.1145/3341302.3342070. URL: http://doi.acm.org/10.1145/3341302.3342070.

[2] S. DiAdamo, M. Ghibaudi, and J. Cruise. "Distributed Quantum Computing and Network Control for Accelerated VQE". In: *IEEE Transactions on Quantum Engineering* 2 (2021). Conference Name: IEEE Transactions on Quantum Engineering, pp. 1–21. ISSN: 2689-1808. DOI: 10.1109/TQE.2021.3057908. URL: https://ieeexplore.ieee.org/abstract/document/9351762 (visited on Aug. 23, 2024).

[3] D. Jensen. "A timeliness model for asychronous decentralized computer systems". In: *Proceedings ISAD 93: International Symposium on Autonomous Decentralized Systems*. IEEE. 1993, pp. 173–182. DOI: 10.1109/ISADS.1993.262706.

[4] W. Kozlowski, A. Dahlberg, and S. Wehner. "Designing a Quantum Network Protocol". In: *CoNEXT*. ACM, 2020, pp. 1–16. DOI: 10.1145/3386367.3431293.

[5] P. Li. "Utility accrual real-time scheduling: Models and algorithms". PhD thesis. Virginia Polytechnic Institute and State University, 2004.

A

NetQASM details

A.1 Flow of messages

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

- `RegisterApp`: Sent once from the CNPU 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 [3], 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 CNPU 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 CNPU, 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 CNPU will have access to a copy of the memory allocated by the QNPU for certain registers and arrays, see Section 3.6.2. This memory is read-only by the CNPU. Updates to the copy of the memory are performed by the end of a subroutine

A

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 CNPU will be notified by an explicit message whenever the memory is updated.

- `StopApp`: Sent from the CNPU to the QNPU indicating that an application is finished.

A.2 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 A.6. 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 A.3.

The operand types of NetQASM are:

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

```
0 beq 0 0 12
```

In the binary encoding used at [2], **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`.

```
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 [2], **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 [2], `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 A.6.6.

```
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 [2], `s` and `e` are always a **REGISTER**s. This is handled by the compiler by using a `set`-commands before.

A.3 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
```

A

```
// 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
```

A.4 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 CNPU but also other data such as information about the generated remote entanglement [1, 3]. 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]
```

A.5 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.

A.6 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 [2]. Commands are specified as follows:

- `name <operand-type 1> <operand-type 2> ...`: Description of instruction.

  Description of operands.

Operand types can be `reg` for a **REGISTER**, `imm` for an **IMMEDIATE**, `addr` for an **ADDRESS**, `arr-ent` for an **ARRAY_ENTRY**, `arr-slice` for an **ARRAY_SLICE**.

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.

A

Allocation.

- `qalloc <reg>`: Start using a qubit in the unit module.

  `<reg>`: The virtual address of the qubit.

- `array <imm> <addr>`: Creates an array of a certain length (width is fixed).

  `<imm>`: Number of entries in the array. `<addr>`: Address of array.

A.6.1 Initialization

- `init <reg>`: Initializes a qubit to $|0\rangle$.

  `<reg>`: The virtual address of the qubit.

- `set <reg> <imm>`: Set a register to a certain value.

  `<reg>`: The register to assign a value to. `<imm>`: The value to assign.

A.6.2 Memory operations

- `store <reg> <arr-ent>`: Stores the value in a register to an index of an array.

  `<reg>`: The register holding the value to store. `<arr-ent>`: The array-entry to store the value to.

- `load <reg> <arr-ent>`: Loads the value from an index of an array to a register.

  `<reg>`: The register to store the value to. `<arr-ent>`: The array-entry holding the value.

- `undef <arr-ent>`: Sets an entry of an array to `null`, see Appendix A.6.6.

  `<arr-ent>`: Array-entry to make `null`.

- `lea <reg> <addr>`: Loads a given address of an array to a register.

  `<reg>`: The register to store the address to. `<addr>`: The address to the array.

A.6.3 Classical logic

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

Nullary branching

- `jmp <imm>`: Jump to a given line (unconditionally).

  `<imm>`: Line to branch to.

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

- `b{ez,nz} <reg> <imm>`: Branch to a given line if condition fulfilled, see below.

  `<reg>`: Value v in condition expression. `<imm>`: 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} <reg0> <reg1> <imm>`: Branch if condition fulfilled, see below.

  `<reg0>`: Value v_1 in conditional expression. `<reg1>`: Value v_2 in conditional expression. `<imm>`: 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`: $v_0 \geq v_1$ (branch-if-greater-or-equal)

A.6.4 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} <reg0> <reg1> <reg2>`: Perform a binary operation and store the result.

  `<reg0>`: Register to write result (r) to. `<reg1>`: First operand in binary operation (v_0). `<reg2>`: 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 <reg0> <reg1> <reg2> <reg3>`: Perform a binary operation modulo m and store the result.

  `<reg0>`: Register to write result (r) to. `<reg1>`: First operand in binary operation (v_0). `<reg2>`: Second operand in binary operation (v_1). `<reg3>`: 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`, $r = (v_0 - v_1) \pmod{m}$

A

A.6.5 Quantum gates

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

- `instr <reg>`: Perform a single-qubit gate.

  `<reg>`: The virtual address of the qubit.

  Single-qubit gates without additional arguments are the following.

- `x`: X-gate.

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

- `y`: Y-gate.

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{A.2}$$

- `z`: Z-gate.

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{A.3}$$

- `h`: Hadamard gate.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{A.4}$$

- `s`: S-gate (phase)

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \tag{A.5}$$

- `k`: K-gate.

$$K = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ i & -1 \end{pmatrix} \tag{A.6}$$

- `t`: T-gate.

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} \tag{A.7}$$

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:

$$a = \frac{n\pi}{2^d} \tag{A.8}$$

These instructions have the following structure

- `rot_{x,y,z} <reg> <imm0> <imm1>`: Perform a single-qubit rotation.

  `<reg>`: Register containing the virtual address of the qubit. `<imm0>`: `n`, for angle, see above. `<imm1>`: `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} <reg0> <reg1>`: Perform a two-qubit operation.

  `<reg0>`: Register containing the virtual address of the control qubit. `<reg1>`: Register containing the virtual address of the target qubit.

Two-qubit gates are the following.

- `cnot`: Controlled X gate.
- `cphase`: Controlled Z gate.

Measurement

- `meas <reg0> <reg1>`: Measure a qubit in the standard basis.

  `<reg0>`: Register containing the virtual address of the qubit. `<reg1>`: 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} <imm0> <imm1> <imm2> <imm3> <imm4> <imm5>`: Specify a pre-measurement rotation.

  `<imm0>`: `n0`, for angle of first rotation, see below. `<imm1>`: `d0`, for angle of first rotation, see below. `<imm2>`: `n1`, for angle of second rotation, see below. `<imm3>`: `d1`, for angle of second rotation, see below. `<imm4>`: `n2`, for angle of third rotation, see below. `<imm5>`: `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 <reg0> <reg1> <reg2> <reg3> <reg4>`: Create an EPR pair with a remote node.

  `<reg0>`: Remote node ID. `<reg1>`: EPR socket ID. `<reg2>`: 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. `<reg3>`: Provides the address to the array which holds the rest of the arguments of the entanglement generation to the network stack [1, 3]. 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 [1, 3] (except remote node ID and EPR socket ID). `<reg4>`: 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 [1, 3].

- `recv_epr <reg0> <reg1> <reg2> <reg3>`: Receive an EPR pair from a remote node.

  `<reg0>`: Remote node ID. `<reg1>`: EPR socket ID. `<reg2>`: 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. `<reg3>`: 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 [1, 3].

A.6.6 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 <arr-slice>`: Wait for all entries in a given array slice to become not `null`.

  `<arr-slice>`: Array slice to wait for.

- `wait_any <arr-slice>`: Wait for any entry in a given array slice to become not `null`.

  `<arr-slice>`: Array slice to wait for.

- `wait_single <arr-ent>`: Wait for a single entry in an array to become not `null`.

  `<arr-ent>`: Array entry to wait for.

A.6.7 Deallocation

- `qfree <reg>`: Stop using a qubit in the unit module.

  `<reg>`: Register containing the virtual address of the qubit.

A.6.8 Return

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

- `ret_reg <reg>`: Return a register.

  `<reg>`: The register to return.

- `ret_arr <addr>`: Return an array,

  `<addr>`: The address of the array to return.

A.7 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:

  – NetQASM (required): Sets the NetQASM version in the metadata.

```
0 # NETQASM 1.0
```

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

```
0 # 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 `{}`.

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

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

A.8 Examples

Here we list some examples of programs written in NetQASM. In Appendix A.8.1, we show some examples written directly in the NetQASM-language. In Appendix A.8.2, we show the corresponding examples, instead written in the Python SDK.

A

A.8.1 NetQASM

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:
```

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.

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
// 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 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.
```

A

```
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
```

A.8.2 SDK

Each of the examples in this section are functionally the same as the examples in Appendix A.8.1. 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.

Classical logic (if-statement)
Functionally the same as the NetQASM-subroutine (Appendix A.8.1).

```
# 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()
```

Classical logic (for-loop)
Functionally the same as the NetQASM-subroutine (Appendix A.8.1).

```
# 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
```

Create and recv EPR
Functionally the same as the NetQASM-subroutine (Appendix A.8.1).
This code is for the side initializing the entanglement request.

```
# 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.

```
# 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()
```

References

[1] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpędek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveria Filho, R. Hanson, and S. Wehner. "A Link Layer Protocol for Quantum Networks". In: *ACM SIGCOMM 2019 Conference*. SIGCOMM '19. Beijing, China: ACM, 2019, p. 15. ISBN: 978-1-4503-5956-6/19/09. DOI: 10.1145/3341302.3342070. URL: http://doi.acm.org/10.1145/3341302.3342070.

[2] *Git repository with code for NetQASM*. https://github.com/QuTech-Delft/netqasm. 2021.

[3] W. Kozlowski, A. Dahlberg, and S. Wehner. "Designing a Quantum Network Protocol". In: *arXiv preprint arXiv:2010.02575* (2020).

B

Application source code for QNodeOS experiments

```python
# Bases to measure the final server state in.
# Note: for efficiency reasons, only bases +Y and -Y were
# used for alpha=pi/2, and +Z and -Z for alpha=pi.
MEAS_BASES = ["+X", "+Y", "+Z", "-X", "-Y", "-Z"]

# Server code (no parameters).
class DelegatedComputationServer(Application):
    def _rotate_basis(self, qubit: Qubit, basis: str) -> None:
        right_angle = math.pi / 2
        if basis == "+X":
            qubit.rot_Y(angle=-right_angle)
        elif basis == "+Y":
            qubit.rot_X(angle=right_angle)
        elif basis == "-X":
            qubit.rot_Y(angle=right_angle)
        elif basis == "-Y":
            qubit.rot_X(angle=-right_angle)
        elif basis == "-Z":
            qubit.X()

    def run(self, context: ApplicationContext) -> Dict[str, Any]:
        outcomes = {}
        for basis in MEAS_BASES:
            # Create EPR pair with client
            epr = context.epr_sockets[0].recv_keep()[0]
            # Compile and send subroutine S1.
            context.connection.flush()
            # Wait and receive delta from client.
            delta = context.app_socket.recv_float()
            # Local gates using delta.
            epr.rot_Y(angle=math.pi / 2)
            epr.rot_X(angle=delta)
            epr.rot_X(angle=math.pi)
            # At this point, the server has qubit state |psi>.
            # Measure in particular basis (part of tomography).
            self._rotate_basis(qubit=epr, basis=basis)
            m_s = epr.measure(store_array=False)
            # Compile and send subroutine S2.
            context.connection.flush()
            # Receive and store result (m_s).
            m_s = int(m_s)
            outcomes[basis] = m_s
        return outcomes
```

Listing B.1: Delegated Quantum Computation (DQC) source code for the server.

```python
# Bases to measure the final server state in.
# Note: for efficiency reasons, only bases +Y and -Y were
# used for alpha=pi/2, and +Z and -Z for alpha=pi.
MEAS_BASES = ["+X", "+Y", "+Z", "-X", "-Y", "-Z"]

# Client code, with parameters alpha and theta.
class DelegatedComputationClient(Application):
    def __init__(self, alpha: float, theta: float):
        self._theta = theta
        self._alpha = alpha

    def run(self, context: ApplicationContext) -> Dict[str, Any]:
        outcomes = {}
        for basis in MEAS_BASES:
            # Create EPR pair with server
            epr = context.epr_sockets[0].create_keep()[0]
            # Local gates
            epr.rot_Y(angle=math.pi / 2)
            epr.rot_X(angle=self._theta)
            epr.rot_X(angle=math.pi)
            # Measurement
            m_c = epr.measure(store_array=False)
            # Compile and send subroutine C1
            context.connection.flush()
            # Receive and store result (m_c).
            outcomes[basis] = m_c
            # Compute and send delta.
            delta = self._alpha - self._theta + m_c * math.pi
            context.app_socket.send_float(delta)
        return outcomes
```

Listing B.2: Delegated Quantum Computation (DQC) source code for the client.

```python
# Bases to measure the final server state in.
MEAS_BASES = ["+X", "+Y", "+Z", "-X", "-Y", "-Z"]

# Local tomography code with axis and angle parameters.
class ClientLocalApp(Application):
    def __init__(self, axis: str, angle: float) -> None:
        self._axis = axis
        self._angle = angle

    def run(self, context: ApplicationContext) -> Dict[str, Any]:
        outcomes = {}
        # Loop over all 6 cardinal measurement bases.
        for basis in self.MEAS_BASES:
            # Create and initialize a qubit in the |0> state.
            q = Qubit(context.connection)
            # Rotate it to one of the 6 cardinal states.
            if self._axis == "X":
                q.rot_X(angle=self._angle)
            else:
                q.rot_Y(angle=self._angle)
            # Measure it in the current measurement basis.
            self._rotate_basis(qubit=q, basis=basis)
            outcomes[basis] = q.measure(store_array=False)
        # Compile send the subroutine containing the above instructions
          .
        context.connection.flush()
        return outcomes
```

Listing B.3: Local Gate Tomography (LGT) source code.

```
# Instantiate the programs to run.
programs = []
er_socket_ids = {}
for i in range(N):
    dqc_program = create_dqc_client_program()
    lgt_program = create_lgt_program()
    programs.append(dqc_program)
    programs.append(lgt_program)
    er_socket_ids[dqc_program] = i  # assign unique ID for ER socket

# Create a thread pool that can be executed by the OS hosting the CNPU.
tpe = ThreadPoolExecutor()

# For each program, submit a piece of code that executes the whole
  program.
for program in programs:
    runner = program_runner(program, er_socket_ids[progam])
    tpe.submit(runner)  # submit it to the thread pool

# Block until the OS hosting the CNPU has finished all programs.
tpe.wait()


# Code for running a single program.
def program_runner(program, er_socket_id):
    # Create connection with QNPU
    qnpu_connection = connect_qnpu()

    # Use connection to setup processes.
    qnpu_connection.register_program()
    for remote_node in program.remote_nodes:
        er_socket = ERSocket(
            remote_node=remote_node.name,
            er_socket_id=er_socket_id,
            remote_er_socket_id=er_socket_id,
        )
        qnpu_connection.open_er_socket(er_socket)

    # classical sockets with other programs do not go through the QNPU
    create_classical_sockets()

    # Execute the program code; it can use the connection to send
    # subroutines to the QNPU and receive results.
    run(program, qnpu_connection)
```

Listing B.4: Pseudocode illustrating the CNPU runner. It can instantiate multiple programs, like the ones defined in Examples B.1 to B.3. Each program is submitted for concurrent execution to a thread pool executor which is managed by the host OS. Each program independently sets up a connection with the QNPU, and executes the program code itself.

```
    array 10 @0
    array 1 @1
    store 0 @1[0]
    recv_epr(2,0) 1 0    // submit request for entanglement
    set R0 0
LOOP3:
    beq R0 1 LOOP_EXIT3
    set R3 0
    set R4 0
    set R5 0
    set R6 0
LOOP:
    beq R6 10 LOOP_EXIT
    add R3 R3 R0
    add R6 R6 1
    jmp LOOP
LOOP_EXIT:
    add R4 R0 1
    set R6 0
LOOP1:
    beq R6 10 LOOP_EXIT1
    add R5 R5 R4
    add R6 R6 1
    jmp LOOP1
LOOP_EXIT1:
    wait_all @0[R3:R5]  // wait until entangled qubit is ready
    set R3 9            // check which Bell state
    set R4 0
LOOP2:
    beq R4 R0 LOOP_EXIT2
    add R3 R3 10
    add R4 R4 1
    jmp LOOP2
LOOP_EXIT2:
    load R2 @0[R3]      // load Bell state type into R2
    set R1 0
    bne R2 3 IF_EXIT
    rot_z R1 16 4       // correction for Phi-
IF_EXIT:
    bne R2 1 IF_EXIT1
    rot_x R1 16 4       // correction for Psi+
IF_EXIT1:
    bne R2 2 IF_EXIT2
    rot_x R1 16 4
    rot_y R1 8 4
    rot_x R1 16 4
    rot_y R1 24 4       // corrections for Psi-
IF_EXIT2:
    beq R0 0 IF_EXIT3   // no correction for Phi+
```

```
IF_EXIT3:
   add R0 R0 1
   jmp LOOP3
LOOP_EXIT3:
   ret_arr @0
   ret_arr @1
```

Listing B.5: NetQASM subroutine S1 of the DQC application. Compiled by the DQC server program code listed in Example B.1.

```
set Q0 0
rot_y Q0 1 1
set Q0 0
rot_x Q0 1 0
set Q0 0
rot_x Q0 1 0
set Q0 0
meas Q0 M0
qfree Q0
ret_reg M0
```

Listing B.6: NetQASM subroutine S2 of the DQC application. Compiled by the DQC server program code listed in Example B.1. The exact gates may differ depending on the iteration of the program loop and the δ value sent by the client.

```
    array 10 @0
    array 1 @1
    store 0 @1[0]
    array 20 @2
    store 0 @2[0]
    store 1 @2[1]
    create_epr(1,0) 1 2 0    // submit request for entanglement
    set R0 0
LOOP2:
    beq R0 1 LOOP_EXIT2
    set R3 0
    set R4 0
    set R5 0
    set R6 0
LOOP:
    beq R6 10 LOOP_EXIT
    add R3 R3 R0
    add R6 R6 1
    jmp LOOP
LOOP_EXIT:
    add R4 R0 1
    set R6 0
LOOP1:
    beq R6 10 LOOP_EXIT1
    add R5 R5 R4
    add R6 R6 1
    jmp LOOP1
LOOP_EXIT1:
    wait_all @0[R3:R5]    // wait until entangled qubit is ready
    add R0 R0 1
    jmp LOOP2
LOOP_EXIT2:               // end of entanglement creation code
    set Q0 0
    rot_y Q0 1 1          // local gate in C1
    set Q0 0
    rot_x Q0 1 0          // local gate in C1
    set Q0 0
    meas Q0 M0            // measurement
    qfree Q0
    ret_arr @0
    ret_arr @1
    ret_arr @2
    ret_reg M0
```

Listing B.7: NetQASM subroutine C1 of the DQC application. Compiled by the DQC client program code listed in Example B.2. The exact gates may differ depending on the DQC parameters α and θ.

```
set Q0 0
qalloc Q0
init Q0
rot_y Q0 3 1
meas Q0 M0
qfree Q0
qalloc Q0
init Q0
rot_x Q0 1 1
meas Q0 M1
qfree Q0
qalloc Q0
init Q0
meas Q0 M2
qfree Q0
qalloc Q0
init Q0
rot_y Q0 1 1
meas Q0 M3
qfree Q0
qalloc Q0
init Q0
rot_x Q0 3 1
meas Q0 M4
qfree Q0
set Q0 0
qalloc Q0
init Q0
rot_x Q0 1 0
meas Q0 M5
qfree Q0
ret_reg M0
ret_reg M1
ret_reg M2
ret_reg M3
ret_reg M4
ret_reg M5
```

Listing B.8: NetQASM subroutine L1 of the LGT application. Compiled by the LGT program code listed in Example B.3. The exact gates may differ depending on the iteration of the program loop.

Glossary

API Application Programming Interface
ASIC Application-Specific Integrated Circuit

CNPU Classical Network Processing Unit
CPLD Complex Programmable Logic Device
CPU Central Processing Unit
CQC Classical Quantum Combiner
CR Charge-Resonance

DD Dynamical Decoupling
DQC Delegated Quantum Computation
DQP Distributed Queue Protocol

EGP Entanglement Generation Protocol
EHI Exposed Hardware Interface
EMU Entanglement Management Unit
EPR Entanglement Pair Request
ER Entanglement Request

FPGA Field Programmable Gate Array

HAL Hardware Abstraction Layer

IR Intermediate Representation

LGT Local Gate Tomography

MLIR Multi-Level Intermediate Representation
MW Microwave

NetQASM Quantum Network Assembly Language
NISQ Noisy Intermediate-Scale Quantum
NV Nitrogen Vacancy

OS Operating System
OSI Open Systems Interconnect

PID Proportional-Integral-Derivative
PMT Photomultiplier Tube
PSB Phonon-Side Band

QAOA Quantum Approximate Optimization Algorithms
QASM Quantum Assembly Language
QDevice Quantum Device
QDriver QDevice Driver
QEGP Quantum Entanglement Generation Protocol
QMMU Quantum Memory Management Unit
QNetStack Quantum Network Stack
QNodeOS Quantum Network Operating System
QNP Quantum Network Protocol
QNPU Quantum Network Processing Unit
QPU Quantum Processing Unit

RO Readout

SDK Software Development Kit
SoC System on a Chip
SP Spinpump
SPI Serial Peripheral Interface
SSRO Single-Shot Readout

TDMA Time-Division Multiple Access
TTL Transistor-Transistor Logic

VQE Variational Quantum Eigensolvers

ZPL Zero-Phonon Line