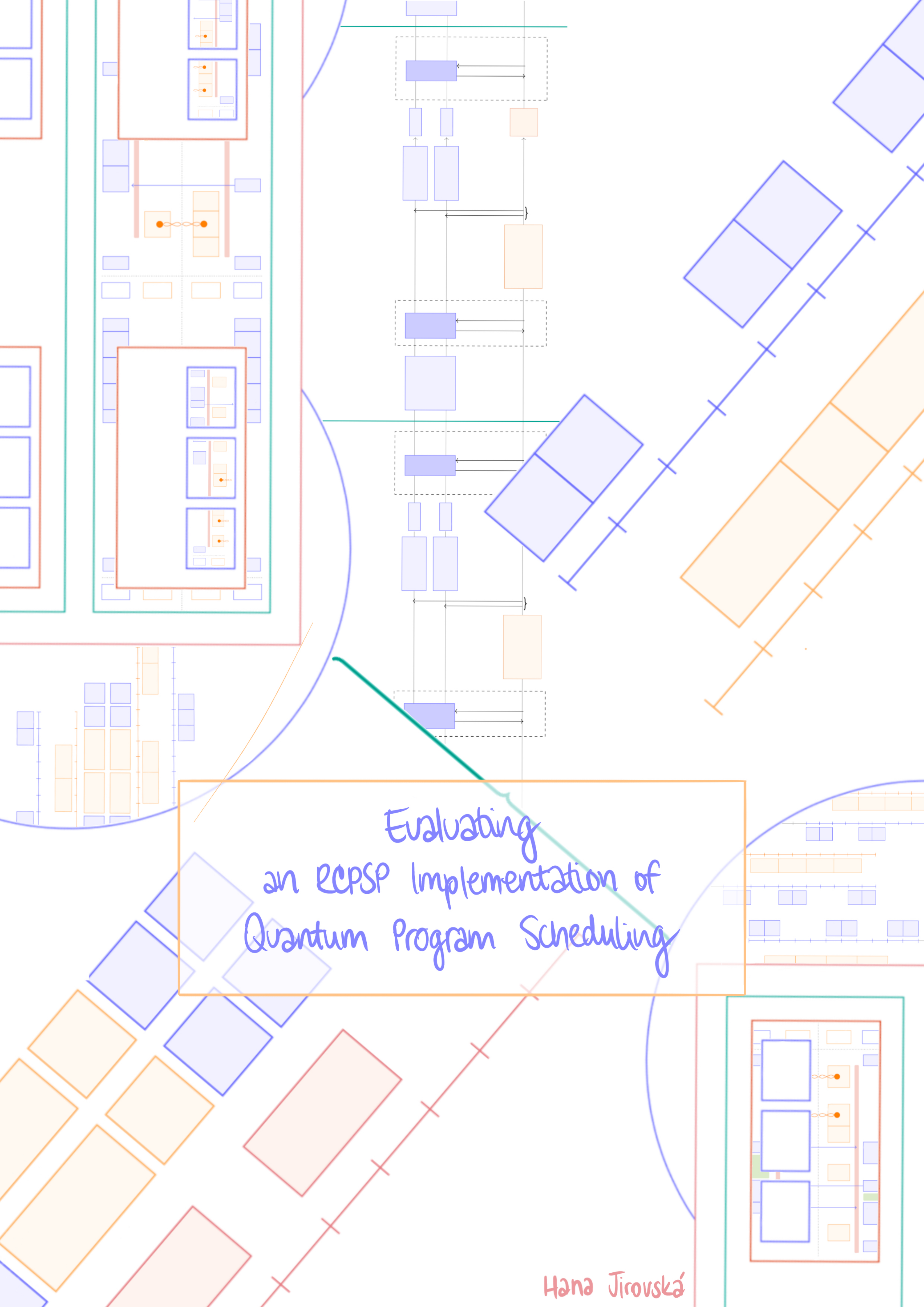


Evaluating an RCPSP Implementation of Quantum Program Scheduling

Hana Jirovská

Evaluating an RCPSP Implementation of Quantum Program Scheduling

by

Hana Jirovská

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended on the 22nd of June 2023.

Student number:	4863135	
Project duration:	November 1, 2022 – June 22, 2023	
Thesis committee:	Prof. dr. S. D. C. Wehner,	TU Delft, supervisor
	Dr. ir. M. Veldhorst,	TU Delft
	Dr. E. Demirović,	TU Delft
Daily supervisor:	PhD candidate Bart van der Vecht,	TU Delft

Abstract

The goal of quantum application scheduling is to enable the execution of applications on a quantum network. As the final step in the application scheduling process, program scheduling locally schedules execution of blocks of instructions on each node by defining so-called node schedules. In this thesis, we present a formal definition of program scheduling and propose success metrics to evaluate the quality of node schedules. We implement program scheduling using the framework of Resource-Constrained Project Scheduling Problem (RCPSP) and simulate the execution of node schedules. By evaluating the performance of program scheduling on datasets with diverse quantum applications including quantum key distribution and blind quantum computing, we observe that heuristic-driven approaches achieve comparable results to optimal program scheduling while requiring fewer computational resources. Our work offers valuable insights into the translation of classical scheduling problems into the quantum domain, contributing to the advancement of quantum application scheduling.

Acknowledgments

This thesis would not look the same if it were not for the support of many people around me. Here I would like to express my sincerest gratitude to them.

First and foremost, I would like to thank my supervisor, Prof. Stephanie Wehner, for the opportunity to do my thesis in her group. You have taught me a great deal and always pushed me to become a better researcher. I would like to give special thanks to Dr. Veldhorst and Dr. Demirović for kindly accepting the invitation to join my thesis committee and for taking the time to read this thesis.

To my daily supervisor, Bart, thank you for your valuable feedback and calm support. The often-too-long meetings in the "secret" meeting room we found always motivated me to keep trying. Thomas, thank you for embarking on the scheduling journey with me, I so appreciate how entertaining our brainstorming sessions were. I feel deeply grateful for the amazing environment in the Wehner group — you all played an instrumental part in celebrating the ups and managing the downs of my project, thank you so much. Many other people from QuTech played a role in making the past months exciting and I am very thankful for that.

To my family, I could not imagine doing this without you. Mami a tati, děkuju za všechnu vaši lásku a podporu. Jake, you have always had my back and I could not wish for a more amazing big brother.

Somewhere in the past few months, Delft truly started feeling like home. I would very much like to thank everyone who contributed to that. There are no words for how grateful I feel for all the coffee breaks, ice cream outings, Ted Lasso (& beach) mid-distance runs, and escapes to French campsites. The time spent together with the wonderful people around me always filled me with joy and made me feel at peace. Thank you from the bottom of my heart.

This thesis has been financially supported by the QuTech Academy Scholarship 2021-2023. I would like to conclude with my gratitude towards QuTech.

Contents

1 Introduction

Quantum networks allow for execution of quantum applications that are impossible to achieve classically [1, 2]. Quantum applications rely on a unique property of quantum systems called entanglement. Generation of entanglement generation between two nodes in a network must be synchronised, which means there are real-time requirements on the scheduling of entanglement generation. This is why we consider the problem of executing quantum applications on quantum networks as a scheduling problem.

The goal of *quantum application scheduling* is to support the execution of quantum applications on a quantum network [3]. This scheduling workflow describes the steps leading from two users wishing to execute an application to the individual nodes in the quantum network knowing which instructions they should execute at a given time. An important part of this workflow is called *network scheduling*: this process constructs a *network schedule* which determines when can specific pairs of nodes in a quantum network attempt to generate entanglement.

We specify the structure of a quantum application such that an application consists of one or more *sessions*. Sessions are independent runs of the application. A session describes what needs to happen on all involved nodes; the part of a session that happens locally on one node is called a *program*. Each program then consists of *blocks* of classical or quantum instructions. We assume that a quantum application is executed between two nodes in a quantum network.

In this project, we focus on so-called *program scheduling*. This is the last part of the quantum application scheduling workflow. Program scheduling is undertaken locally on each node in the quantum network. It focuses on scheduling blocks of instructions which are executed either on the classical processing unit (CPU) or the quantum processing unit (QPU). We provide a high-level summary of program scheduling here and refer to Section 3 for the full formal definition.

Definition 1.1 (Program scheduling)**.** Given a network schedule defining when can entanglement generation be attempted with other nodes in the network and a set of programs that should be executed on a particular node, define the order in which individual blocks of instructions should be executed on the classical processing unit (CPU) and the quantum processing unit (QPU).

The output of program scheduling is called a *node schedule* and it specifies when should each block be executed on a given node. We give a very simple illustration of how a node schedule looks like in Figure 1. We define the first research question of this project to investigate how to judge the quality of a node schedule. Knowing the answer to this question will help us evaluate different approaches to program scheduling.

Research Question 1. How can we evaluate the quality of a node schedule?

Finding a suitable node schedule can be described by having to satisfy multiple types of constraints. There are resource constraints based on the availability of the processing units, precedence constraints based on the order of blocks within a program that should be respected, network schedule constraints for generating entanglement imposed by the network schedule, and timing constraints resulting from the nature of qubits that have a limited lifetime. Furthermore, there can be deadlines imposed on particular applications given by the users of the quantum network.

Therefore, the problem of problem scheduling can be taken as a constraint scheduling problem. We propose a solution to finding node schedules using the Resource Constrained Project Scheduling Problem (RCPSP) [4] framework. It is important to note that while there might be an optimal node schedule in terms of some success metrics, finding this one optimal schedule is NP-Hard [5, 6]. The complexity of this problem motivates the use of heuristics which is why we investigate the performance of *heuristic-driven* program scheduling.

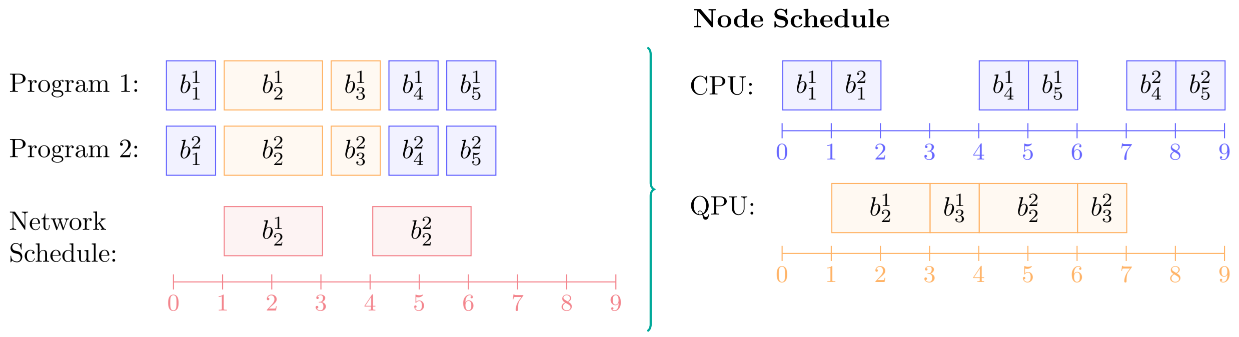

Node Schedule
Program 1:
Program 2:
Network Schedule:
CPU:
QPU:
b_1^1 b_2^1 b_3^1 b_4^1 b_5^1
b_1^2 b_2^2 b_3^2 b_4^2 b_5^2
0 1 2 3 4 5 6 7 8 9

Figure 1: A simple node schedule based on two programs and a network schedule. Two programs define blocks b_i^s where $s \in \{1, 2\}$ and i is the index of the block. Blue squares correspond to classical blocks of instructions, orange squares represent blocks with quantum instructions, and orange rectangles represent blocks in which entanglement generation happens.

Next to heuristic-driven program scheduling, we also define *optimal* program scheduling which looks for a node schedule with a minimised value of an objective function. Furthermore, we introduce *naive* program scheduling in which blocks are scheduled consecutively without any interleaving between sessions. Note that naive program scheduling is not compatible with the constraints imposed by the network schedule, which is why we define the following two research questions:

Research Question 2. How does the performance of heuristic-driven and optimal program scheduling differ when there are network schedule constraints?

Research Question 3. How does the performance of heuristic-driven and naive program scheduling differ in the absence of network schedule constraints?

Execution of quantum applications between nodes in a network inevitably requires both classical and quantum communication. The scheduling of quantum communication operations is constrained by the network schedule given by a prior scheduling workflow [3]. The duration of classical communication has some inherent uncertainty because messages can get delayed in transmission. Entanglement generation (result of quantum communication) or message reception (result of classical communication) might happen earlier or later than expected. There is an extension of RCPSP called *risk-aware RCPSP* which attempts to deal with uncertainty in the duration of scheduled activities and we use it to define the last research question:

Research Question 4. How can risk-aware RCPSP be used to account for the uncertainty in duration of classical and quantum communication?

To put this thesis into the context of related work, we briefly summarise the existing work regarding scheduling of application execution on quantum networks. Quantum application scheduling has been defined in [3]. Program scheduling directly depends on the outputs of network scheduling. While network scheduling considers scheduling operations for distributing entanglement across a quantum network, its focus is not on scheduling blocks of instructions that are related to specific quantum applications.

Scheduling of network resources has been investigated in [7, 8]. This project focuses on scheduling entanglement delivery in a network. It considers heuristic-driven scheduling approaches to construct time-division multiple access schedules, so while the scheduling approach shows some similarities to this project, the granularity and type of scheduled operations are different. For example, scheduling of classical instructions is not considered at all. In terms of goals, it is much closer to the problem of network scheduling.

Individual instructions for a quantum application have been scheduled using RCPSP in [9]. Here, constraint programming models were used to investigate scheduling of instructions for a blind quantum computing application. Similarly to our project, there was an entity similar to the network schedule and success metrics such as makespan and success probability were used to evaluate assignment of start times to a set of tasks. Instead of being focused solely on the blind quantum computing application with a variable number of clients, our project proposes a solution to program scheduling for an arbitrary quantum application. We furthermore investigate the use of heuristics in the RCPSP implementation and include a more realistic representation of resource constraints by including capacities for the processing units.

In this thesis, we make the following contributions:

- We formally define program scheduling. This definition is compatible with the framework of network scheduling being developed within the quantum application scheduling [3] and allows for developing alternative approaches to program scheduling.
- We formalise several classical and quantum success metrics to evaluate the quality of a node schedule. This in turn allows us to evaluate different approaches to program scheduling.
- We develop a method to generate random network schedules satisfying the demands of sessions to be scheduled. We describe the insights gained during development of this method which can be useful in the domain of network scheduling [3]. This random network schedule generation can serve as a baseline for evaluation of network schedule construction.
- We implement program scheduling using the RCPSP framework and the execution of node schedules using a simulator for quantum applications. This allows us to evaluate program scheduling using the proposed classical and quantum success metrics.
- We evaluate the performance of the heuristic-driven and optimal approaches to program scheduling based on randomly generated network schedules. We compare the constructed node schedules and conclude there is no significant difference in the performance of the two scheduling approaches. The heuristic-driven approach, however, requires fewer computational resources. This is a novel contribution to the field of quantum application scheduling.
- We compare the performance of naive and heuristic-driven approaches to program scheduling in the absence of network schedule constraints. We additionally investigate whether heuristic-driven program scheduling performs better with or without network schedule constraints. We motivate the need for network schedules in larger quantum networks.
- We provide insights into how risk-aware extensions of RCPSP can be applied to the domain of quantum program scheduling and suggest concrete steps how this can be further investigated.

The rest of this document is structured as follows: we describe the relevant background for this project in Section 2, we give a formal definition of program scheduling in Section 3, we propose how to evaluate the quality of a node schedule in Section 4, and we explain our method of generating random network schedules in Section 5. We present our implementation of program scheduling in Section 6. The results of our investigation are presented in Section 7 and discussion, further research directions, and summary can be found in Section 8.

2 Background

In this section, we introduce the necessary background to understand the work presented in this thesis. This means explaining what are the building blocks of quantum networks and how is quantum information represented and manipulated. We describe the structure of quantum applications as defined for the purposes of quantum application scheduling. We define the three quantum applications used in this project and how they work. The architecture of quantum program scheduling is explained to illustrate the context of program scheduling. Lastly, we provide background information about the framework of Resource-Constrained Project Scheduling Problem (RCPSP).

2.1 Quantum Networks

To be able to explain quantum networks, we must first establish an understanding of the rudimentary unit of quantum information - the *qubit*. A qubit is a quantum counterpart to a classical bit. In comparison to the classical bit which has a value of either 0 or 1, a qubit can take on the states $|0\rangle$, $|1\rangle$, or any linear combination (a *superposition*) of these two. A very useful visual representation of a qubit state is the so-called Bloch sphere depicted in Figure 2 where the qubit state can be envisioned anywhere on this sphere.

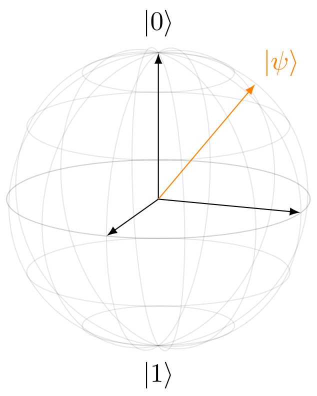


Figure 2: Bloch sphere representation of a qubit with an arbitrary quantum state $|\psi\rangle$.

The ultimate objective of a quantum state is to read out information from it. This extraction, or readout, can only yield the binary results of 0 or 1, leading to the *collapse* of the quantum information. This concept can be visualised using the Bloch sphere, where regardless of the initial state of the qubit, upon measurement, it invariably points to either the north or south pole. The likelihood to which one of the two poles the state collapses is determined by the quantum state, while the axis on which the states collapse is defined by the measurement basis.

However, the capacity of quantum states to assume a superposition is not the only property making them unique. Another uniquely quantum mechanical property is known as *entanglement*. When two quantum states are entangled, the measurement result of one of the states is intrinsically (anti-)correlated with the other state's measurement outcome. This property holds even across large distances, rendering it instrumental in various quantum applications.

The status of a qubit can be manipulated through the application of gates. In general terms, these gates modulate the state of a qubit; for single-qubit gates, one can conceptualise a gate as a rotation on the Bloch sphere. There are also multi-qubit gates, which act on the *target* qubit depending on the state of the *control* qubit. For the purposes of this thesis, it is important to understand that gates are local operations that modify the state of qubits, require a finite time for execution, and their functionality is not always flawless. More specifically, in relation to tangible quantum hardware, gate operations may not always succeed in their intended effect on the quantum state. We can measure the quality of a quantum state using a metric called *fidelity* which defines how indistinguishable two quantum states are: higher fidelity means that the actual state is very similar to the ideal desired state.

A quantum node is essentially a piece of quantum hardware facilitating the storage and manipulation of qubits. Generally, we differentiate between *communication* qubits, which facilitate quantum communication over distances, and *storage* or *memory* qubits, which are used to store quantum states. The characteristics of qubits can be delineated by two parameters, T1 and T2, signifying the influence of time on the quality of quantum states. Generally, quantum states lose information over time which is called *decoherence*. For more details, see Nielsen and Chaung [10] for a comprehensive overview of the field of quantum information and quantum computation.

A quantum network is a network of interconnected nodes; each node has access to a classical processing unit (CPU) and a quantum processing unit (QPU) which operates on qubits. Users of this network can implement quantum applications across nodes. Further discussion about quantum applications will be covered in the following section.

2.2 Applications for Quantum Networks

Before we delve into specific applications for quantum networks, we describe the structure of a quantum application. We assume all applications require interaction of two nodes in the quantum network. This definition is also used in the context of the entire quantum application scheduling workflow [3] defined below.

To explain the structure of an application, we start by looking at the smallest building block. This smallest building unit of an application is an instruction. An instruction is an order given to the processing unit of a computer — in the context of this project, the processing unit can be either the classical processing unit (CPU) or the quantum processing unit (QPU). There are four types of instructions: classical local (CL), classical communication (CC), quantum local (QL), and quantum communication (QC). Some examples of instructions are arithmetic operations (CL), receiving a classical message (CC), executing a gate on a quantum state (QL), or generating entanglement with another node (QC). The classical instructions are executed on the CPU and the quantum instructions are executed on the QPU.

In summary, a list of instructions is called a block. A list of blocks to be executed on a particular node is called a program. Corresponding programs on two nodes form a session. A session is an independent run of a quantum application; one or more sessions therefore comprise a quantum application. The formal definitions are given below and a visual representation of this structure is given in Figure 3.

Definition 2.1 (Block)**.** A *block* is a series of instructions which are executed sequentially without an interruption once they are initialised. The instructions can be either quantum or classical and either communication or local–a block consists of a grouped list of instructions of one type. It is parameterised as $B = (\texttt{node}, \texttt{instr}, M)_B$ where `node` is the node on which the list of instructions `instr` are executed and M is metadata about the block.

Definition 2.2 (Program)**.** A *program* is a set of blocks on a particular node. This set is either strictly ordered or represented as a directed acyclic graph. It is defined as a tuple $P = (\mathcal{B}, M)_P$ where $\mathcal{B}$ is the list of blocks, and M is the program metadata.

Definition 2.3 (Session)**.** A *session* is a pair of programs being executed on different nodes; these programs include communication between the nodes. A session is defined as $S = (S_{ID}, P_A, P_B)_S$

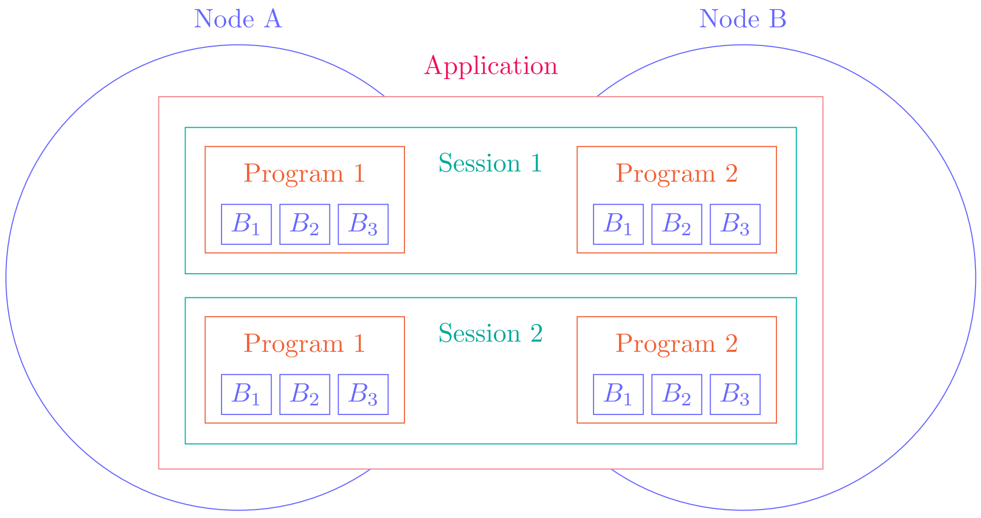


Figure 3: An overview of the structure of an application.

where S_{ID} is the session identifier, and P_A, P_B are the two programs residing on nodes involved in this session. A session is a construct used by the network scheduler but the individual nodes are not aware of the program details of other nodes. Note that exactly one program is being executed on each node.

Definition 2.4 (Application)**.** An *application* consists of the programs to be executed on different nodes. From the perspective of the network scheduler, an application forms an individual session or it can be split up into multiple sessions if those sessions can be executed independently.

Before we give concrete examples of these constructs, we introduce the three applications we consider for program scheduling: quantum key distribution (QKD), blind quantum computing (BQC), and the so-called ping-pong (PP) application consisting of two teleportation operations. There are two main approaches to measuring a quantum state — the *measure directly* paradigm in which a quantum state is measured shortly after its creation to mitigate the effects of decoherence, and the *create and keep* paradigm in which the quantum state is kept in memory before it is measured. BQC and PP are examples of create-and-keep applications, whereas QKD can be implemented as either of the two; in our project we implement QKD as a measure-directly application. The difference between BQC and PP is the demand on entanglement generation. Together, these three applications encompass structure of most known quantum applications.

Quantum Key Distribution. A secret key shared between two parties is often required if these two parties wish to privately communicate over a network. The goal of quantum key distribution is to distribute a shared secret key between two parties. A possible way to implement QKD is to repeatedly generate entanglement between Alice and Bob such that each of them has one half of the entangled pair, and then let Alice and Bob measure their entangled state in a randomly chosen measurement basis. Alice and Bob can afterwards publicly communicate which measurement bases they used without revealing the actual measurement outcomes. Whenever Alice and Bob measure in the same basis, they get correlated measurement outcomes and can use this to create a shared secret key. The most well-known protocols for implementing QKD are the BB84 [11] and the E91 [12] protocols. A commonly used metric of evaluating the performance of a QKD implementation is the rate at which bits of secret key can be generated between Alice and Bob (called the *secret key rate*).

Blind Quantum Computing. It is to be expected that the availability of advanced quantum hardware will be limited in the near-term future. This means that different nodes in a quantum network will likely have different computational powers. Blind quantum computing allows a *server* node to perform computations on behalf of a *client* node. A client with limited quantum resources can therefore delegate the computation to a server with access to more resources. An important consideration in this application is that the server is not aware of which computation it is performing (hence the *blind* quantum computing). The application usually starts with generating multiple entangled pairs between the client and the server. The client then measures their halves of the entangled pairs and becomes aware of what quantum states there are at the server's side. The client can then instruct the server on what kind of computation should be performed using classical communication. To ensure the server is honest (and performing the computations it should be), the client can execute *test* rounds in which it knows what kind of results to expect [13]. A single test round can then be said to be either successful or not. For more details about BQC, see for example [14].

Ping-pong: Quantum teleportation is a crucial ingredient in applications for quantum networks [15, 16]. Quantum states cannot be copied [10, p. 532], but it is possible to reconstruct a quantum state at a distance (provided the original copy is destroyed) using quantum teleportation. Assume that Alice has a quantum state she wants to send to Bob; quantum teleportation then consists of entanglement generation between Alice and Bob, Alice making some local operations on her two quantum states, and sending her measurement outcomes to Bob who can then use it to reconstruct Alice's original state. We define an application in which two teleportation operations send a qubit back and forth between Alice and Bob. This "ping-pong test" has been formulated to test the ability of a quantum network to execute multi-round quantum protocols [17]. The goal of the ping-pong application is that Alice reconstructs the same state she initially teleported to Bob.

An important consideration when scheduling quantum applications is that any unnecessary delays between generating entanglement, performing local operations, and measuring the quantum state decrease the fidelity of the state as well as the likelihood of getting useful information. By considering these three applications within program scheduling, we hope to gather insight applicable to other applications as well.

To refer back to the structure of quantum applications we defined earlier, we give examples of what programs and sessions correspond to. One session can be for example one round of computation in BQC or generating x bits of secret shared key in QKD. A program would then be what each of the participating nodes needs to execute. An application can comprise of one or more of these sessions.

Note that while the details presented here offer some information about the quantum applications, understanding how these applications achieve their goal is not necessary for understanding the rest of the thesis. The crucial concept for the rest of this project is that entanglement generation (a necessary part of a quantum application) needs to be synchronised and scheduling of the surrounding classical and quantum operations can greatly influence the usefulness of executing quantum applications.

2.3 Quantum Application Scheduling for Quantum Networks

Having explained how quantum applications look like, we now turn to describing the process of executing applications on quantum networks. This process starts by two users of the quantum network deciding they want to execute an application and ends when the nodes are aware when do they need to execute the blocks of instructions pertaining to that application. This *quantum application scheduling* workflow consists of several steps and was designed at the start of this project in collaboration with other researchers [3]. In this section, we summarise quantum application scheduling to give more context for program scheduling and to be aware of where are the inputs to program scheduling coming from.

The goal of application scheduling is to support the execution of quantum applications on quantum networks. To achieve this, we propose an architecture which is summarised in Figure 4. In the rest of this section, we will look at the detail at the processes depicted in this diagram.

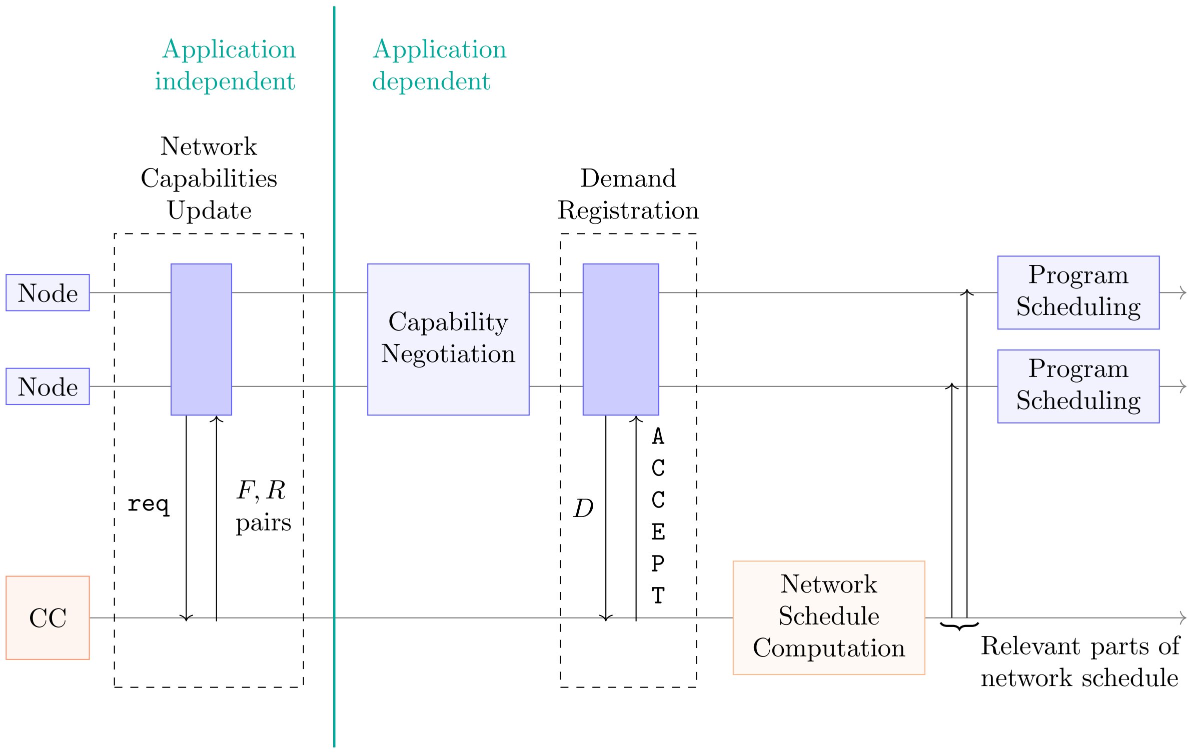


Figure 4: The architecture of quantum application scheduling. This shows the interaction between two nodes that want to execute a quantum application and the central controller (CC) which has an overview of the entire quantum network.

First, it is important to describe all the relevant parties participating in application scheduling. We have a quantum network which consists of interconnected nodes. When there are two users on two different nodes in the network that want to execute an application, we call these nodes *end nodes*. These nodes do not have to be neighbouring nodes and the other nodes in the network can serve as intermediary *repeater* nodes. Each node in a quantum network only has information about itself, its capabilities, and which application(s) it wants to execute. For the purposes of application scheduling, we also require an entity which has an overview of the entire network. We call this entity the *central controller*. The central controller makes scheduling decisions based on aggregated data from all nodes in the network.

Application scheduling starts with the ***network capabilities update***. This update is essentially a round of communication between a pair of nodes and the central controller, and is entirely application independent (i.e. no information about the application which the nodes wish to execute is required at this stage). As soon as two nodes realise they wish to execute any quantum application, they send a request to the central controller. The central controller in turn evaluates the state of the quantum network and sends back information on what kind of entanglement generation possibilities there are for this particular pair of nodes. At the end of the network capabilities update, the two nodes are aware of what kind of performance they can expect from the network. In particular, the performance refers to the fidelity of entanglement that can be generated and the rate at which this can be done; generally there are different fidelity-rate pairs that be achieved.

At this point, the workflow of application scheduling becomes application dependent. This means that all following steps need to be executed for each application that is being scheduled on the network.

When two users of a quantum network make a decision on which quantum application they want to execute, the nodes at which the users reside engage in ***capability negotiation***. The goal of capability negotiation is to define a single demand for the quantum network. A very general form of this demand (taken from [3]) is

$$D = (k, (\mathfrak{p}, R)_j, \mathcal{T}) \tag{1}$$

where k is the number of entangled pairs required, $\mathfrak{p}_j$ describes the properties of entanglement required at a rate R_j for each acceptable path through the network j, and $\mathcal{T}$ is a set of timing constraints between attempts to generate the entanglement, in conjunction with an expiry time for the demand. The structure of the packet $\mathfrak{p}_j$ could be in the format $(w, F)_j$ where w is the maximum time between the first and last of the k pairs, or it may be some description of the fidelity requirements for each individual pair. Both nodes have possibly different parameters of available resources and they have to agree on what they require of the network. The way capability negotiation is performed is not specified as this can be different for each application; it would for example look different in peer-to-peer applications (such as QKD) and client-server applications (such as BQC).

The demand that the two nodes construct is then submitted to the central controller during ***demand registration***. The central controller reserves the right to reject submitted demands; it sends back an `ACCEPT` message only if it agrees to consider this demand in the next iteration of network schedule computation.

After a sufficient number of demands has been registered at the central controller, it begins the process of ***network schedule computation***. The goal of this task is to compute a network schedule that satisfies the demands submitted to the central controller by the different users of the network. The central controller computes the network schedule at two different levels of granularity: the high-level network schedule only specifies the timeslots during which entanglement generation can be attempted between pairs of nodes in the network, and the low-level network schedule also specifies which operations needs to be performed at the intermediary nodes in the network. From the perspective of program scheduling, it is the high-level network schedule that is crucial; we provide its formal definition below.

Definition 2.5 (Network Schedule)**.** A network schedule is a list of timeslots (defined as start times and durations) with information about which specified pair(s) of nodes can attempt entanglement generation.

The computation of network schedules is an open problem, currently under investigation [3]. Having a defined network schedule is, however, necessary for program scheduling, which is why we propose a method to generate random network schedules in Section 5.

When the network schedule is computed, the relevant parts are then communicated to the nodes. By relevant part of the network schedule, we mean a list of timeslots in which the particular node is involved; this means each node finds out when it can schedule entanglement generation with other nodes in the network.

The final stage of application scheduling is ***program scheduling*** performed by a scheduling entity called the *local scheduler*. At this point, each of the nodes has a list of programs it needs to execute and a part of the network schedule pertaining to this list of programs. Note here the distinction between programs and sessions: while we talk about a network schedule being constructed for a set of sessions (because it considers both nodes), the local scheduler on a single node is only aware of the programs it will be executing. The goal of program scheduling is therefore to come up with a way to schedule all the blocks the particular node needs to execute in a way that conforms to the constraints imposed by the network schedule. Program scheduling is defined in detail in Section 3.

2.4 Resource-Constrained Project Scheduling Problem

In this section, we give context to scheduling problems by explaining what are the frameworks used to solve them. Before we look at the framework of the Resource-Constrained Project Scheduling Problem (RCPSP for short), it is important to highlight the difference between constraint satisfaction and optimisation problems. A constraint satisfaction problem (CSP) is defined by (adapted from [6]):

- a finite set of variables $X = x_1, ..., x_n$,
- a finite domain D_i for each of the variables x_i, and
- a finite set of constraints restricting the values for variables.

The objective of CSP is to find a solution in which a value $a_i \in D_i$ is assigned to each variable x_i and all the constraints are satisfied. The constraint optimisation problem (COP) additionally introduces an objective function; the solution to COP is therefore a solution with either a minimal or maximal objective value.

The resource-constrained project scheduling problem can be taken as a particular instance of the constraint optimisation problem put into the context of project scheduling. There is a lot of research on project scheduling problems because they often pose interesting computational challenges; the first unified notation for RCPSP was proposed by Brucker in 1999 [4]. Simply put, RCPSP is the "problem of determining start times of activities which need to satisfy precedence and resource constraints" [18, p. 714].

RCPSP is defined in terms of activities which need to be scheduled, precedence constraints between these activities, and a set of resources on which the activities are executed. Each activity consumes a subset of the resources for given time; the resources can be either renewable or non-renewable. Precedence constraints can be for example defined as a set of immediate predecessors for each activity [18]. Parameters are assumed to be integer-valued [4], and interruption (*preemption*) of an activity while it is being executed is not allowed. Optionally, modes with processing alternatives for each activity can be defined (this is called *multi-mode RCPSP*), but that is not considered within this project.

The solution to an RCPSP instance is a list of start times $S = [S_1, ..., S_n]$ for all the activities $\{1, ..., n\}$ which satisfies the precedence constraints and the constraints imposed by the availability of resources. The vector $S = (S_i)_{i=1}^{n}$ defines a schedule for the project and it is called feasible if all constraints are fulfilled [6]. RCPSP is often posed as a makespan minimisation problem [6, 18] where the makespan is the schedule completion time. For a more detailed mathematical formulation, see for example [4] or [18].

Note that two modes of schedule execution can be distinguished [19]: a *static* execution where the construction of a schedule is done separately from the execution, and a *dynamic* execution where the scheduling actually modifies an existing schedule; the construction and execution of the schedule are in a sense interleaved. The static mode is only applicable when there is a sufficient backlog of activities to be scheduled — we assume this to be the case in the context of quantum application scheduling.

The general RCPSP framework can be extended to include more expressive timing constraints. There is the general precedence constraint $i \rightarrow j$ between two activities i and j for start times S_i, S_j and a processing time p_i defined as $S_i + p_i \leq S_j$. This can be generalised to a *start-start* relation of the form $S_i + d_{ij} \leq S_j$ with an arbitrary integer $d_{ij} \in \mathbb{Z}$ [6]. We say d_{ij} is the time-lag between two activities i and j, and it can be further specified into the minimum allowed time-lag $d_{ij}^{\min}$ and the maximum allowed time-lag $d_{ij}^{\max}$.

RCPSP belong to NP-hard problems [5, 6]. When the minimum and maximum time-lags are added, it can result in problem instances that cannot be solved — this moves the RCPSP into a class of NP-complete problems [5, 20, 21]. Small instances of RCPSP can still be solved optimally (using an objective function that e.g. minimises the makespan), but the complexity also motivates use of heuristics when solving scheduling problems.

2.4.1 Risk-aware RCPSP

Whenever we want to execute the constructed schedules in real-life, it is inevitable that some uncertainties will occur. There can be a sudden change in the availability of resources or activities can take shorter or longer than expected. With this in mind, the *risk-aware* extension of RCPSP has been investigated. The risk-aware RCPSP attempts to deal with uncertainty in the scheduling environment.

There are several general approaches that one can take when tackling uncertainty [22]: **reactive scheduling** which revises the initial schedule during execution when an unexpected event occurs, **stochastic scheduling** which models activity durations using stochastic variables and uses scheduling policies to make decisions during the execution, **proactive scheduling** which creates robust schedules with built-in flexibility, and **scheduling under fuzziness** which uses fuzzy numbers (rather than stochastic variables) for modelling activity durations. In the rest of this section, we explain the reactive and proactive approach to scheduling in a more detailed fashion. This will be later used to address how uncertainty in program scheduling could be addressed.

Reactive scheduling, as the name suggests, is based on the idea of reacting to unforeseen changes in the scheduling environment. A baseline schedule is constructed the usual way, but upon execution schedule repair actions can be taken to restore its consistency [22]. The simplest schedule repair action is called the *right-shift rule* and consists of postponing execution of scheduled activities if the activity currently being executed takes longer than anticipated. Reactive scheduling covers any schedule repairs which can either perturb the schedule as little as possible or can mean full rescheduling.

On the other hand, proactive scheduling focuses on creating robust baseline schedules that can deal with the stochastic nature of activity durations [21]. A robust schedule is defined as a“schedule that is able to absorb some level of unexpected events without rescheduling” [23]. In a sense, this means creating a fault tolerant schedule — this can be achieved if redundancy in the schedule is introduced, such as including time buffers which prevents propagation of delays or disruptions.

In the context of quantum application scheduling, uncertainty concerns mainly blocks with communication instructions. Whenever nodes send classical information around the network or try to generate entanglement, some degree of uncertainty is inevitable involved in the execution of these tasks. We describe in Section 6.4.3 how risk-aware RCPSP can be used to mitigate the risks associated with unpredictable changes to activity execution.

3 Formal Definition of Program Scheduling

Program scheduling refers to the process in which the local scheduler considers all the blocks which are to be executed on a particular node and creates a *node schedule.* To formally define program scheduling, we present the definition of inputs and outputs of the entire process. This definition is a "black-box" definition in that it does not describe the internal mechanism of the scheduling process; rather it only provides a high-level overview of the overall process. The implementation of the proposed scheduling algorithm is presented in Section 6. The information and definitions presented here are aligned with the entire architecture for quantum application scheduling [3].

3.1 Inputs to Program Scheduling

Program scheduling works with blocks (see Definition 2.1); the inputs to the scheduling process should therefore describe all the relevant information about each of the blocks to be scheduled. Some of the blocks perform so-called quantum communication, i.e. they attempt to generate entanglement. These are operations that need to be scheduled and executed by each of the involved nodes at the same time. The times at which the blocks containing quantum communication operations (QC blocks) can be scheduled are given by the network schedule.

The information required in program scheduling comes from multiple sources: for each program to be scheduled, the local scheduler is given a list of blocks involved in the program, block metadata for each of the blocks, and program metadata for the overall program. Next to this, the local scheduler also makes use of information about the node itself which is retrievable from the node lookup table and a network schedule defined for a set of sessions to be scheduled. Note that although the local scheduler is scheduling execution of programs on the local node, the network schedule is constructed for a set of sessions.

More formally, each block $B = \texttt{instr}$ is simply a list of instructions `instr` to be executed on a particular node. The block metadata is defined by the compiler and takes the form

$$M_B = \left(\#Q_{\{C,S\}}, k, \#I_{\{QC,QL,CC,CL\}}, \texttt{remote\_node}, CS_B\right) \tag{2}$$

where:

- $\#Q_{\{C,S\}}$ is the number of communication (C) and storage (S) qubits required by the quantum operations,
- k is the number of entangled pairs required,
- $\#I_{\{QC,QL,CC,CL\}}$ is the number of quantum ($Q\_$) or classical ($C\_$), communication ($\_C$) or local ($\_L$) instructions,
- `remote_node` is the other node in a given application, and
- CS_B is an identifier for the critical section to which a given block belongs to.

Handling quantum states requires careful consideration of timing constraints because qubits lose information with time. For example, it is important that after entanglement is generated, any local operations and a measurement of the qubit are performed as soon as possible. Until now, we defined a program as simply a list of blocks, but this does not allow for specifying any time dependencies. To this end, we introduce the notion of *critical sections.* If two blocks are in the same critical section (i.e. the critical section field in the block metadata holds the same identifier), they need to be executed sequentially without any delays. In this way, the scheduler becomes aware of some timing dependencies between individual blocks.

Next to the information about the particular block, there is some information the scheduler needs to know about the entire program. Note that in Definition 2.2, we define a program as a list of blocks and the program metadata. The order of blocks within that list defines the order in which the blocks should be executed. Program metadata for a program P is defined as

$$M_P = \big((M_{B_i})_{i=1}^{|P|}, F, D_A, S_{ID}\big) \tag{3}$$

where:

- $(M_{B_i})_{i=1}^{|P|}$ is a list of block metadata from all blocks corresponding to the program,
- F is a fidelity requirement,
- R is a rate requirement, and
- D_A is a deadline for the entire application (i.e. all programs belonging to the application needs to be executed by this time), and
- S_{ID} is an identifier for the session to which this program belongs to.

Because the block metadata is included in the definition of the program metadata, the local scheduler needs to get a list of the blocks and the program metadata for each program it is supposed to schedule. Furthermore, the local scheduler also needs to be aware of the local parameters. This is given in the form of a lookup table residing on each of the nodes. The contents of this lookup table are depicted in Table 1.

Parameter	Description	Units or range
$\#Q_C$	Number of available communication qubits	$\in \mathbb{N}$
$\#Q_S$	Number of available storage qubits	$\in \mathbb{N}$
$[T1, T2]$	Maximum lifetime of quantum memory	s
F_G	Minimum gate fidelity	$\in (0, 1]$
T_G	Minimum gate duration	s
T_{CC}	Expected maximum communication time between the two involved nodes	s

Table 1: Contents of a lookup table of an end node in the network.

The last necessary input for the local scheduler to be able to perform program scheduling is the network schedule, an outcome of a prior scheduling workflow [3]. The network schedule that each node receives is only a subset of the entire network schedule, and contains a list of timeslots in which a given node can participate in entanglement generation.

In summary, the input to program scheduling is therefore a list of programs that need to be scheduled, the node's lookup table, and the network schedule for a given set of sessions corresponding to the programs. Each program defines a list of blocks to be executed and the program metadata. This is summarised in Table 2.

3.2 Outputs of Program Scheduling

The output of program scheduling is a so-called *node schedule*. Both nodes involved in executing some sessions of possibly multiple different applications have their own node schedules. That

Source	*Parameter*	*Description*
Block metadata	$\#Q_{\{C,S\}}$	the number of communication (C) and storage (S) qubits required by the quantum operations
	$\#I_{\{QC,QL,CC,CL\}}$	the number of quantum ($Q\_$) or classical ($C\_$), communication ($\_C$) or local ($\_L$) instructions
	`remote_node`	the other node in a given application
	CS_B	an identifier for the critical section of given block
	k	the number of entangled pairs required
Node lookup table	$[T1, T2]$	the worst-case lifetime of a node's quantum memories
	T_G	the worst-case time duration of gates
	F_G	the worst-case fidelity of gates
	T_{CC}	the worst-case classical communication time between the two involved nodes
Program metadata	S_{ID}	session ID this program belongs to
	D_A	deadline for the entire application

Table 2: Information available about each block which needs to be scheduled.

is because it is possible that what Alice and Bob need to do in order to execute the session are different things and they might be scheduling different blocks. However, both Alice and Bob receive the same network schedule to construct their nodes schedules, so the times at which Alice and Bob perform entanglement generation is synchronised.

Both the network schedule and the constructed node schedules work by assigning timestamps for operations. These timestamps are defined on a scale of nanoseconds. It is important that this is unified across the scheduling workflows in application scheduling.

A node schedule is a mapping of blocks to their scheduled start times. Assume there is a set of blocks $\mathcal{B} = \{1, ..., n\}$ that should be executed on a particular node (these blocks can belong to $[1, n]$ sessions) and are either classical (i.e. consume the Classical Processing Unit (CPU) resource) or quantum (i.e. consume the Quantum Processing Unit (QPU) resource). The node schedule is then defined as follows:

Definition 3.1 (Node Schedule)**.** A node schedule is a set of start times $\mathcal{S} = \{s_1, ..., s_n\}$ satisfying the following constraints:

1) **resource constraints**: no two blocks i, j consume the same resource k concurrently where k can be either the CPU or the QPU of a node,

2) **precedence constraints**: the order of blocks within a session is respected. That is, if $i < j$ for any two blocks i, j in the same session, the start times s_i and s_j are assigned such that $s_i < s_j$,

3) **network schedule constraints**: blocks with quantum communication operations respect the network schedule. That is, the network schedule defines a timeslot for entanglement generation starting at $t_{S_n,i}$ for each QC block i belonging to a session S_n, the start time s_i of each QC block to be scheduled is then set such that $s_i = t_{S_n,i}$,

4) **critical section constraints**: if blocks belong to the same critical section, they are scheduled consecutively. That is, for blocks i, j with $CS_i = CS_j$ and $i < j$, given an execution

time p_i for block i, the start time s_j is set such that $s_j = s_i + p_i$. Note that this constraint must be relaxed for QC blocks as the above network schedule constraint has a higher priority; instead of being scheduled immediately, a QC block in a critical section must be scheduled at the first possible opportunity,

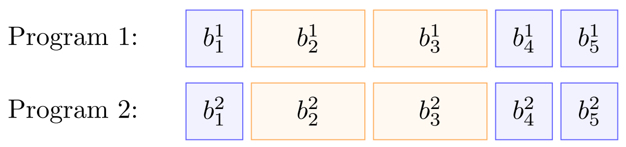
5) **application deadline constraints**: if an application deadline D_A is defined, any block i associated with application A will finish executing before the deadline. That is, if p_i is the execution time of block i, $s_i + p_i \leq D_A$.

4 Evaluating Node Schedule Quality

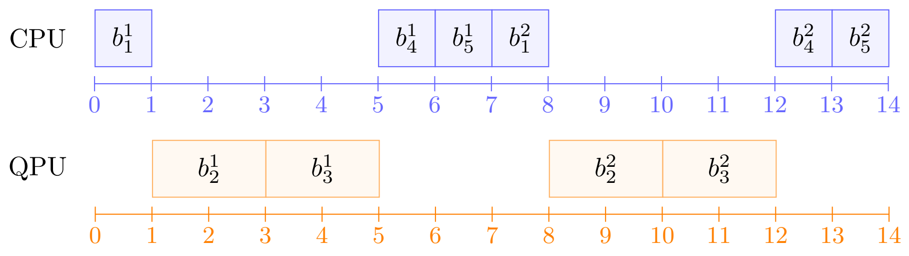
In this section, we propose a method of evaluating the quality of a node schedule. This is necessary in order to evaluate the performance of program scheduling. Node schedule quality can be evaluated using classical and quantum success metrics. We delve deeper into their definitions in the following two subsections. By doing so, we answer the first research question of this project.

By definition, the form of a node schedule highly depends on the network schedule which is used to construct it. The problem of finding an "optimal" network schedule is outside the scope of this project, and so we use a set of randomly generated network schedules to see how the success metrics behave. For more details about the network schedule generation, see Section 5.

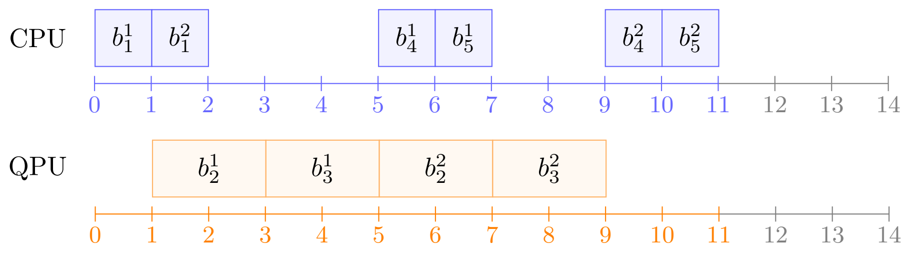
For illustrations of how the success metrics would be calculated, we use two example node schedules created based on the same input, as defined in Figure 5. Note that in this section, we abstract away the network schedule constraints and specific timing issues and consider two simplified node schedules.



(a) Input for program scheduling. Two programs define blocks b_i^s where $s \in \{1, 2\}$ and i is the index of the block. Blue squares correspond to blocks of classical instructions and orange rectangles to blocks of quantum instructions.



(b) An example of a *naive* node schedule. The blocks are scheduled according to their strict order within a session and the sessions are not overlapping.



(c) An example of an *optimal* node schedule. Here, classical instructions are executed along quantum instructions and execution of blocks from different sessions is overlapping.

Figure 5: Two possible node schedules given input from two programs. Note that that construction of these node schedules corresponds to a scenario without network schedule constraints.

4.1 Classical Success Metrics

To evaluate a node schedule using classical success metrics, it is sufficient to consider a given node schedule without executing it. Classical success metrics depend purely on the static form of the schedule. As such, we can evaluate a single node schedule with these classical metrics. In this section, we define two main metrics one can look at when evaluating a quality of a schedule: *makespan* and *processor utilization factor (PUF)*. We further illustrate how these are calculated using the example node schedules given in Figure 5.

Scheduling problems are often posed as makespan-minimization problems [20, 21] because it is a good reflection of the quality of schedules in the classical context. Makespan refers to the total time it takes to execute a given schedule. If a node schedule defines start times $s_0, ..., s_n$ for blocks $b_0, ..., b_n$ with durations $d_0, ..., d_n$, the makespan is given by

$$\text{makespan} = \max_{0 \leq i \leq n} (s_i + d_i) - \min_{0 \leq j \leq n} s_j$$

The processor utilisation factor (PUF) is defined as the fraction of time spent executing a given set of tasks compared to the makespan. It has been used as a success metric for evaluating scheduling in the quantum domain [7]. Generally, a higher value of PUF means a more efficient schedule. For our project, we distinguish between two different processor utilisation factors: (1) PUF of the CPU in which the set of tasks are all the classical blocks, (2) PUF of the QPU in which the set of tasks are all the quantum blocks. Given the above definition of a node schedule with the addition of $t_0, ..., t_n$ being types of blocks, we define the two metrics as:

$$\text{PUF}_{\text{CPU}} = \sum_{\substack{0 \leq i \leq n \\ t_i \in \{CC, CL\}}} \frac{d_i}{\text{makespan}}$$

$$\text{PUF}_{\text{QPU}} = \sum_{\substack{0 \leq i \leq n \\ t_i \in \{QC, QL\}}} \frac{d_i}{\text{makespan}}$$

Table 3 shows calculations of all three metrics for the example node schedules from Figure 5. While the values of the processor utilisation factors depend on the makespan, distinguishing between PUF for the CPU and the QPU allows us to to see possible differences between the utilisation of the different processing units. Having a success metric which is focused solely on the QPU allows for optimisations focused on the quantum side of program scheduling. Note that generally we cannot expect the PUF values to reach the upper threshold of 1 — this is because there are some dependencies between the classical and quantum blocks, but also because consideration of network schedule constraints will inevitably introduce some additional delays in the node schedules.

Schedule	**makespan**	**PUF$_{\text{CPU}}$**	**PUF$_{\text{QPU}}$**
Example naive schedule (Fig. 5b)	14	$6/14 \approx 0.43$	$8/14 \approx 0.57$
Example optimal schedule (Fig. 5c)	11	$6/11 \approx 0.55$	$8/11 \approx 0.73$

Table 3: Calculation of classical success metrics using example node schedules from Figure 5.

There are also several common success metrics that depend on defined deadlines for the individual activities being scheduled, such as the maximum lateness, the total tardiness or the number of late activities [6]. In the context of program scheduling, we can define deadlines for applications and therefore programs, but not for the individual blocks of instructions, and so we do not consider success metrics related to deadlines in this project.

None of the classical success metrics mentioned here fully capture the subtleties of quantum systems — quantum states lose information over time and while having a higher value of PUF_{QPU} can mean there is not a lot of delay between quantum operations, it does not track individual quantum states. This gives us the motivation to also look at quantum success metrics which can hopefully reflect more accurately the intricacies of working with quantum states. Lastly, it is worth mentioning that since the PUF metrics correlate with makespan, makespan can be taken as the primary classical success metric.

4.2 Quantum Success Metrics

To fully evaluate the quality of a node schedule, it is also vital to consider the quantum perspective. This means using a success metric that takes into account the effect of the unique properties of quantum systems. One such metric is the success probability of sessions executed according to the node schedule. In this project, we use simulations to execute sessions on a pair of nodes. While there have been many developments on the side of quantum hardware, it is not yet possible to run experiments on a scale which this project requires.

For the sake of generality, we explain how to obtain a success metric using a general simulation-based approach. We specify which tool for simulating quantum networks is used in this project in the section on implementation (see Section 6.5). We assume that the simulator can simulate execution of both classical and quantum instructions, and that one can define the input in terms of blocks of instructions that need to be executed and the order in which this should be done. We can then use the order given by a node schedule to evaluate how a particular schedule performs in terms of whether the sessions are executed successfully.

Assuming a minimal quantum network of two nodes, the simulator is given a node schedule created for each of the nodes and executes the instructions. The success probability of one node schedule is then taken as the number of sessions successfully executed compared to the total number of sessions scheduled in the node schedule, i.e.

$$\text{success probability} = \frac{\#\text{ of successful sessions}}{\#\text{ of scheduled sessions}}$$

The behaviour of quantum simulations is not deterministic, and the simulated execution of a node schedule must be repeated many times to obtain an *average success probability.* Note that this parameter will be the same for both node schedules inputted to the simulator; the average success probability is evaluated for both nodes schedules at once because their execution depends on each other.

We need to define what a "successful execution of a session" means. This looks different for each application. We define how we evaluate success of the applications being considered in this project in Section 6.5.

Having defined how a success is evaluated for each of the applications we consider in this project, we can obtain the value for the average success probability of a pair of node schedules using simulations. Unlike the classical metrics which can be evaluated for a single schedule, the average success probability depends on the interplay of both node schedules. This quantum success metric allows us to observe an effect of the quantum properties which cannot be easily captured using classical success metrics.

5 Network Schedule Generation

In this section, we describe the iterations of generating random network schedules we have gone through in hopes of providing some insight into future work done on network schedules. The program scheduling in this project depends on network schedules, an outcome of a prior scheduling workflow called *network scheduling* [3]. The problem of network scheduling has not been solved yet, thus we instead come up with a method to generate random network schedules.

While the form of a network schedule is defined and followed, in this project we include a few minor adjustments to the requirements of how a quantum application is structured. In the formally defined network scheduling, each session of an application contains only one block with quantum communication but can pose different requirements for entanglement generation, i.e. the timeslots in a network schedule do not have to have the same length. Instead of such a setup, in this project we allow for multiple QC blocks within a session with the caveat that they always require one entangled pair and therefore the timeslots in a network schedule are of a constant length. Note that these two approaches are essentially equivalent and one can easily translate between them.

Before we start generating network schedules, we need to answer two questions: (1) how long is the network schedule and (2) how long is one timeslot in the network schedule? The length of a network schedule clearly depends on the sessions being scheduled, and so we start with setting the length of the network schedule to twice the duration of all blocks that are being scheduled. We later elaborate on how this decision affects generating network schedules. Furthermore, the length of a network schedule timeslot should correspond to some realistic parameters achievable on current hardware. We pick a length of 2 ms (milliseconds) based on the results of entanglement delivery experiments [24, Figure 3b]. Note that we keep the same order of magnitude as presented in this work, but due to computational time we decrease the number of entanglement generation attempts this time allows for. While implementing the solution, these values are not hard-coded and can be easily modified in future investigations.

5.1 Randomly Assigning Network Schedule Timeslots

When the network schedule length and the length of the timeslots are fixed, we can determine the number of timeslots in the network schedule. Our first idea was to randomly assign each of these timeslots to the sessions that are being scheduled where each timeslot corresponds to fulfilling the entanglement generation for one quantum communication block. We begin by calculating a probability with which a timeslot should be assigned to a given session.

This can for example be done by defining a probability of assigning any timeslot in the network schedule to session S_i such as

$$p_{S_i} = \frac{\text{\# of QC blocks in } S_i \cdot \text{\# of scheduled sessions } S_i}{\text{\# of timeslots in the network schedule}}$$

Then for each of the timeslots in the network schedule, that timeslot is assigned to session S_i with a probability of p_{S_i}. This can then be naturally extended to network schedules with multiple session types by scaling the probabilities according to the number of session types being scheduled. This was our initial implementation of the random network schedule generation. In theory, this results in random network schedules that on average schedule the required number of timeslots for each of the sessions.

Let us define a *feasible network schedule* as a network schedule that allows for construction of feasible node schedules, i.e. all the blocks can be scheduled according to the constraints given. The caveat of this initial implementation comes with the fact that it "on average" results in feasible network schedules. There are, however, a lot of network schedules being generated that do not contain enough timeslot assignments to fulfil the demand of the sessions. This means that the entire program scheduling is attempted for a lot of network schedules that are not feasible and that is very computationally intensive.

We gain one insight from this initial implementation. Even though we are attempting to generate network schedules randomly, we need to make sure we schedule the requested number of timeslots. There are some conditions that make network schedules infeasible if they are not being met. It is computationally easier to check for these conditions rather than to attempt to construct node schedules based on an infeasible network schedule.

5.2 Scheduling Exactly the Required Number of Timeslots

The decision to schedule precisely the required number of timeslots for each session also agrees with the direction of current investigation of network scheduling [3]. It would of course also be possible to schedule more timeslots than required in the hopes that if a session is not executed successfully, it can be retried. However, this approach is not scalable because this will not be possible when the demand on the network is higher, e.g. when multiple pairs of nodes request timeslots. Therefore, fulfilling the demand and not over-delivering agrees with how network scheduling will be implemented.

To schedule the required number of timeslots, we can take several different approaches. One could formulate it as a constraint optimisation problem (as is currently being done [3]); this seems like an overly complicated approach to generating network schedules randomly. It is not the objective of this project to come up with the most optimal way of generating network schedules.

Therefore, we stick to randomly assigning timeslots but make sure that the required number of timeslots is always delivered. To do this, we first generate a list of all the timeslots in the network schedule and randomly pick a subset of these. Note that whenever we refer to a timeslot, we mean the combination of a start time of the timeslot and to which session type it belongs to. (The duration of the timeslot is fixed by the duration of quantum communication as motivated above.) Generating a list of all the possible timeslots is therefore trivial whenever only one session type is being scheduled because all the timeslots are assigned to that session type. The starting times of the timeslots are multiples of the length of quantum communication.

Whenever multiple session types are being scheduled, we iterate through the possible start times and for each one randomly pick which session type will be assigned. This allows us to define different probabilities and ensure that session types which contain more quantum communication blocks will have a bigger pool of available timeslots. This provides the first dimension of randomness.

Upon implementing this approach, we again find out that many of the randomly generated network schedules are not feasible. Closer investigation reveals that this is because this approach does consider any timing constraints. We include this in the next iteration of the random network schedule generation.

5.3 Careful Consideration of Timing Constraints

There are several timing considerations we need to take into account when generating network schedules. We categorise them based on an example session presented in Figure 6. This example session does not represent any particular quantum application and is meant to capture all possible timing constraints that need to be considered when constructing network schedules. This is a useful insight for network scheduling.

Before we categorise all the different timing dependencies and how they translate to constraints for network schedules, we need to re-introduce the notion of critical sections. Critical sections are defined for each program individually where one program can include one or more critical sections and each critical section can contain one or more blocks with quantum communication. It is important that the QC blocks within the same critical section are scheduled close to each other in the network schedule. As we previously mentioned, network schedule is generated based on sessions — the information about critical sections of each program therefore has to be aggregated by the central controller to arrive at the information presented in Figure 6.

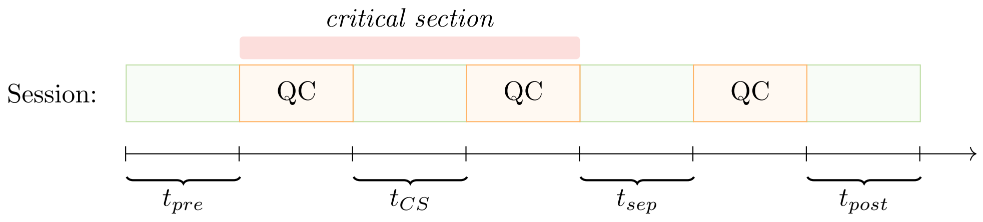


Figure 6: An example of a session which illustrates the four possible types of time variables t_{pre}, t_{CS}, t_{sep}, and t_{post} that need to be considered when generating network schedules. The orange rectangles correspond to QC blocks, while the green rectangles can be either classical or quantum local blocks. The red highlight denotes blocks in a critical section.

We therefore define four time variables that need to be taken into account: pre-processing time t_{pre}, separation time between QC blocks within a critical section t_{CS}, separation time between critical sections t_{sep} (QC block which is not in a critical section is taken as its own separate critical section), and post-processing time t_{post}. If we take $s_{i,j}$ to be a start time of a timeslot for the j-th QC block in session S_i, these time variables impose the following constraints on the network schedule (from the perspective of a node):

- The order of scheduled QC blocks for a session is preserved: for any two QC blocks in a session with indexes j and k where $j < k$, it holds that $s_{i,j} < s_{i,k}$.
- No timeslot is scheduled sooner than the first entanglement generation can be attempted. Therefore for session S_i and any QC block j, it holds that $t_{pre} < s_{i,j}$.
- If two timeslots are assigned to QC blocks belonging to the same critical section, any timeslots inbetween these two timeslots cannot belong to a different critical section. Assume that $CS(i, j)$ returns an identifier of the critical section the j-th QC block in session S_i belongs to. Therefore if we have $s_{i,j} < s_{x,y} < s_{i,k}$ where $CS(i, j) = CS(i, k)$, then $x = i$ and $CS(i, j) = CS(x, y)$.
- Two timeslots for the same session are scheduled at least t_{CS} or t_{sep} apart, depending on whether they belong to the same or different critical sections respectively. More specifically, for any two QC blocks with indexes j and k belonging to session S_i where $j < k$, we have $s_{i,k} - s_{i,j} \geq t_{CS}$ if $CS(i, j) = CS(i, k)$ and $s_{i,k} - s_{i,j} \geq t_{sep}$ if $CS(i, j) \neq CS(i, k)$.
- For any two timeslots from two different sessions where the pre-processing time of session S_m is t^m_{pre} and the post-processing time of session S_n is t^n_{post}, we have that $s_{m,k} - s_{n,j} \geq (t^n_{post} + t^m_{pre})$ for any QC block indexes k, j.

These constraints can in theory be used to define a constraint satisfaction problem. However, a more trivial implementation can include a feasibility check. We pick a subset of timeslots from all the available timeslots that fulfils the demand for number of timeslot assignments. Then we execute the feasibility check: if the proposed network schedule passes, we found a random network schedule, if it does not pass, we simply move on to picking a different subset of the timeslots.

The feasibility check depends on the actual sessions being scheduled. However, because of the classification of all timing considerations that need to be taken into account, this is easily adaptable to any session with an arbitrary internal structure. Our proposed final implementation method outputs feasible network schedules and is summarised in the next subsection.

5.4 Final Implementation

We present the method used for generating random network schedules in this section. The complexity of this approach is motivated by the need to create network schedules that result in feasible node schedules; a trivial implementation of random network schedule generation is possible but computationally ineffective because more resources are then required to check the feasibility of the network schedule through construction of node schedules.

An overview of the proposed approach can be seen in Figure 7. An input to the generation of random network schedules consists of the network schedule length (defined in nanoseconds) and a set of the sessions which need to be scheduled. Each of the sessions has a number of QC blocks for which timeslots need to be assigned and some timing constraints as defined previously. An additional remark about the network schedule length will follow after an explanation of the network schedule generation.

The implementation presented here makes use of the `numpy.random` module. For reproducibility purposes, one can define a seed used in the module to ensure it is possible to arrive at the same "random" outcomes. Whenever we make a probabilistic choice in the implementation, we use a seed which then defines the identifier of a network schedule. A network schedule with ID x therefore used `seed=x` in all its random choices. In this way, we enable intuitive iteration through the probabilistic choices and an easy reproducibility of our results.

The first step in the network schedule generation is to create a random domain for all possible timeslots. Given the length of the network schedule and the duration of entanglement generation, we can easily define start times for all the timeslots. Then we iterate through these timeslots and randomly assign them to the sessions — the probabilities of assigning a timeslot to a particular session are scaled depending on how many QC blocks there are in that session. We arrive at a completely filled network schedule. (As another optimisation step, for sessions with very specific timing constraints, we already further restrict the domain of all possible timeslots. For example, for a QKD session, no two timeslots can be scheduled directly after one another. Therefore we already restrict the initial domain to either even or odd timeslots. This speeds up the process of picking a feasible subset of these timeslots.)

The next step is then to pick a subset of these timeslots to satisfy the exact required number of timeslots for each of the sessions. Note that instead of picking a number of timeslots corresponding to the number of QC blocks, we pick a number of timeslots that is equal to the number of critical sections. This allows us to always schedule the subsequent QC blocks in a critical section at the earlier possible time according to t_{CS}.

Feasibility check is then executed on the suggested network schedule. This feasibility check checks that are the timing constraints defined above are satisfied. In Figure 7, an example of a suggested network schedule that would not pass the feasibility check is if a timeslot for S_2 is picked directly after a timeslot for S_1 because the second QC block for session S_1 could not be scheduled. If a suggested network schedule does not pass the feasibility check, we increase the seed and make a new random choice of a subset of the timeslots.

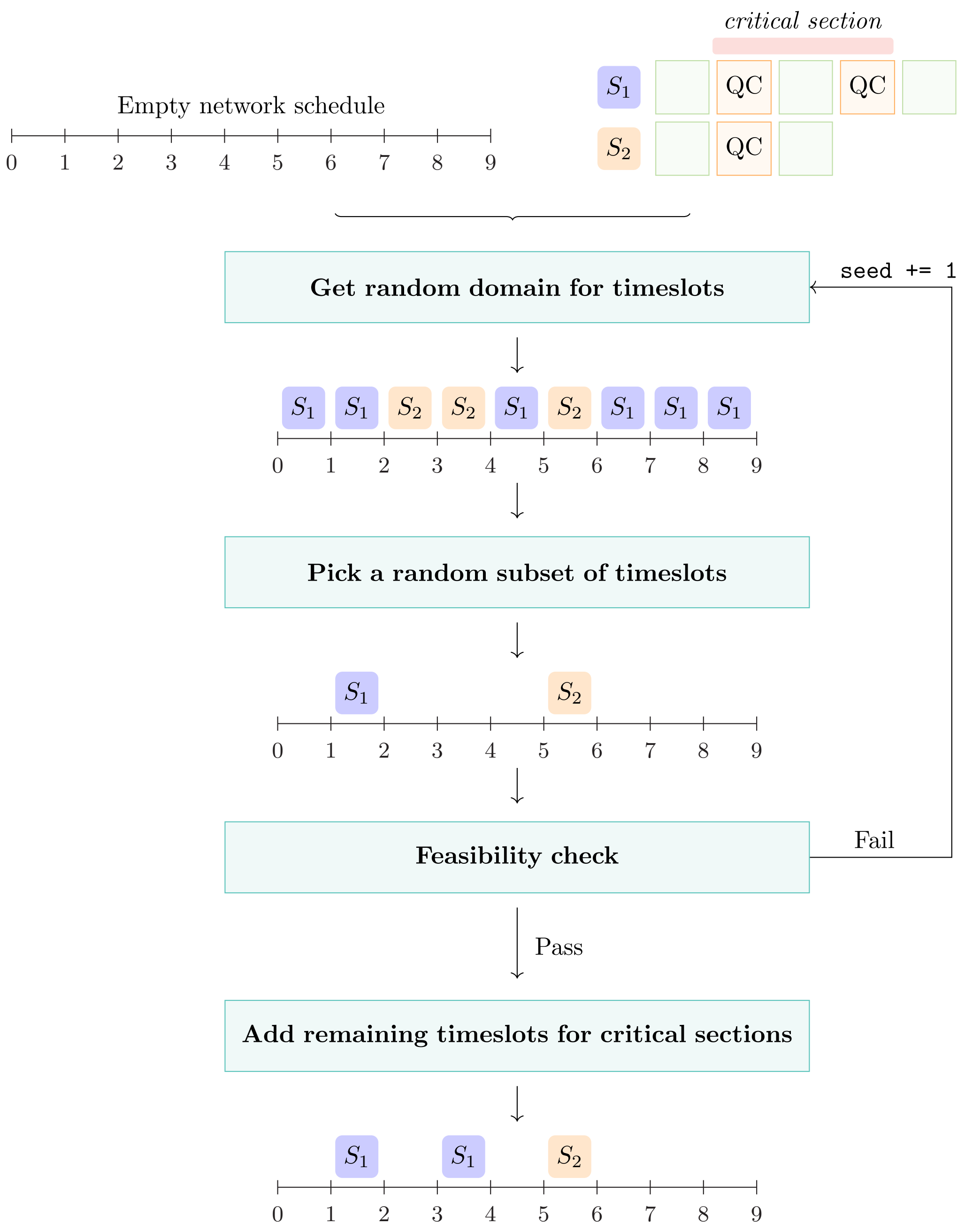


Figure 7: Overview of the implementation of random network schedule generation. The input to the random network schedule generation is a set of sessions for which entanglement generation needs to be scheduled; these are shown in the top right corner where the green squares correspond to quantum local or classical blocks and the orange squares represent QC blocks. In this toy example, we generate a network schedule for one session S_1 and one session S_2.

Once a feasible network schedule is found, we add timeslots for the remaining QC blocks in the critical sections. By scheduling the timeslots for subsequent QC blocks in critical sections like this, we make it impossible to interleave different critical sections. The form of a network schedule does impose constraints on how the node schedules can look like, but in this case, the restriction already exists due to the definition of critical sections which prohibits interleaving. We have now arrived at a randomly generated network schedule that is guaranteed to create feasible node schedules.

At the beginning of this section, we mention that a length of the network schedule is one of the inputs to the random network schedule generation. This length has a great effect on the performance of generating random network schedules. If the length of a network schedule is too short, it is very improbable that a chosen subset of timeslots will satisfy all the timing constraints. This increases the need for regenerating the subset of timeslots and increases the computational time. On the other hand, if the network schedule is too long, the problem size for program scheduling increases because the network schedule length determines the length of the relevant node schedules (more details about the limitations of program scheduling are mentioned in Section 6.4.4). We define a *network schedule length factor* which results in a network schedule length l_{NS} of

$$l_{NS} = \text{network schedule length factor} \times \sum_{i=1}^{N} d_i$$

where the particular network schedule schedules N blocks and d_i is a duration of each of the block being scheduled. We then pick a length factor that allows us to generate a 100 random network schedules for each of the datasets within 5 minutes. For datasets with 6 sessions, we find that this condition is satisfied by a length factor of 3, whereas for datasets with 12 sessions, we need a length factor of 5.

6 Implementation

In this section, we present our proposed solution to the problem of program scheduling as defined in Section 3. We implement program scheduling using the Resource-Constrained Project Scheduling Problem (RCPSP) framework. To evaluate the quantum success metrics, we provide a script that executes the constructed node schedules and evaluates their performance. First, we motivate the choice of software tools used for the implementation, we describe the input data used in this project and the general code flow of program scheduling. Then we explain in detail how both the program scheduling and the simulations of node schedule execution are implemented.

6.1 Existing Tools

We need to both construct and execute node schedules. Here we summarise the available software tools and motivate the choice of software used in this project. For the program scheduling, a software library that allows for RCPSP problem formulation is required; we look at the RCPSP implementation of "Link Scheduling" used in [7, 8], the *MiniZinc* constraint programming language used in [9], and the *PyCSP*^{3} Python library for modelling constrained problems. (For clarity, we will refer to PyCSP^3 as PyCSP3 for the rest of this document to prevent any confusion with footnotes.)

Link Scheduling [25]. The software used for investigating allocation of links in a quantum network in [7, 8] implements a block extension to RCPSP [18]. This scheduling problem works with network resources rather than resources on quantum nodes (and as such is perhaps closer to network scheduling rather than program scheduling). Even though it implements RCPSP, it was implemented with link allocation in mind and would need to be heavily rewritten to be suitable for program scheduling.

MiniZinc [26]. MiniZinc is a constraint modelling language used to specify constraint optimisation and decision problems. It is designed to interface with different back-end solvers. It provides a Python interface, but the documentation also states that this interface is still in its early development stage.

PyCSP3 [27]. PyCSP3 is a Python library for modelling "combinatorial constrained problems". It supports modelling of constraint satisfaction problems and constraint optimisation problems, providing the same range of functionality as MiniZinc. The RCPSP framework has been implemented as one of the example uses of this library.

Before making any decision on the software used for implementing program scheduling, we investigate available tools that can be used to simulate the execution of node schedules. We look at several quantum simulators to see which one fits our needs best; we consider *NetSquid*, *SquidASM*, *QuNetSim*, and *Qoala*. To be able to simulate execution of node schedules, the tool needs to provide control over both classical and quantum instructions.

NetSquid [28]. NetSquid is a discrete-event quantum simulator and allows for modeling and simulation of scalable quantum networks. Its main aim is to accurately model the effects of time on quantum systems. While this is exactly what is needed for evaluation of node schedules using the quantum success metrics, NetSquid is too low-level and using purely NetSquid would require writing a lot of abstractions on top of it.

SquidASM [29]. SquidASM is a simulator built specifically for execution of quantum applications. It is build on top of the NetSquid simulator and abstracts away some of the unnecessary details. It is written in Python. This seems to be more suitable for the execution of node schedules, but a key functionality of having control over the timing of (both classical and quantum) instructions is lacking.

QuNetSim [30]. QuNetSim is a Python-based framework for simulating applications on quantum networks. It is similar to SquidASM in the sense that its focus lies on high-level quantum applications, but it lacks the guarantees of precise modeling of the effects of time. Furthermore, it does not provide any classical control and focuses on the execution of quantum instructions.

Qoala [31]. Qoala is a comprehensive software library allowing the simulation of both the software and hardware of nodes in a quantum network. It is written in Python and internally makes use of the NetSquid simulator. This simulator allows for specification of instructions for the software of quantum nodes, enabling to evaluate classical control as well as quantum performance.

In the end, we implement the software for this project using PyCSP3 and the Qoala simulator. Both of these are written in Python and therefore their interface should be easily implementable. The choice between MiniZinc and PyCSP3 boiled down to the programming language they are written in. The Qoala simulator was the only software tool allowing for scheduling of both classical and quantum operations.

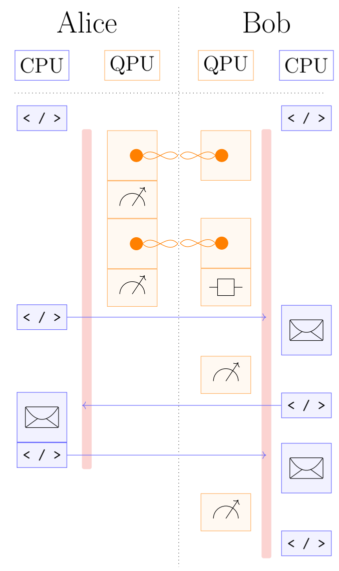
6.2 Input Data

There are three different applications we consider for program scheduling within this project. These are blind quantum computation (BQC), the so-called ping-pong test (PP) implementing two consecutive teleportation operations, and quantum key distribution (QKD). Each of these is defined through a `qoala` program (see folder `configs` in [32]). For illustration, Figure 8 below shows one session of each of these applications, which blocks they consists of and how we define critical sections.

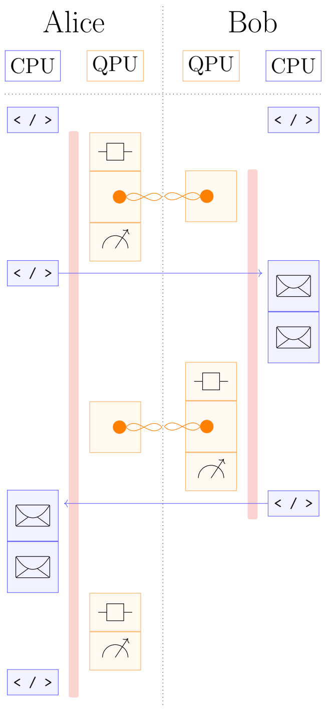
Note that each of these sessions has a fairly different structure. Both BQC and PP have one longer critical section while QKD contains three shorter critical sections. This means that it is for example easier to interleave QKD sessions with other sessions, rather than to interleave either BQC or PP. This can affect how program scheduling performs and it is to be expected that program scheduling heavily depends on the programs to be scheduled.

The actual structure of these sessions is limited by the number of available communication and storage qubits. For this project, we choose to limit the number of communication qubits to one while the number of storage qubits is not limited. This corresponds to the hardware of qubits implemented with color centers in diamonds [33]. This means that if a session needs two entangled pairs, a move operation needs to be performed to move the first generated entangled state onto a storage qubit.

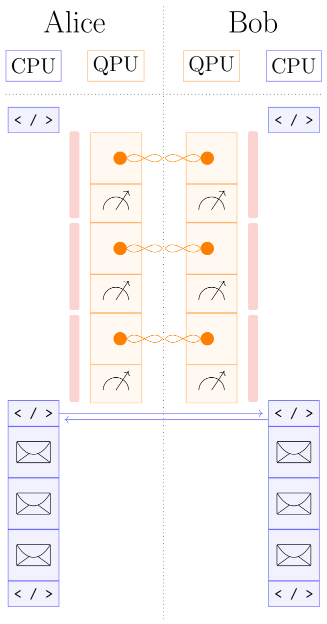
We define seven different datasets for all the possible combinations of sessions; this is depicted in Table 4. We run program scheduling and Qoala simulations with datasets that have either 6 or 12 sessions, to observe whether any trends we see in the results hold across different sizes of input.



(a) A BQC session defined in terms of blocks, their types, and critical sections. Alice has the role of the client and Bob of the server.



(b) A Ping-pong session defined in terms of blocks, their types, and critical sections.



(c) A QKD session defined in terms of blocks, their types, and critical sections. Note that this session has three separate critical sections for both Alice and Bob.

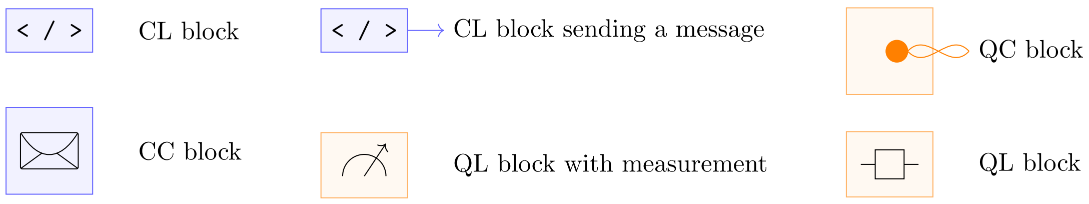


Figure 8: Illustration of how sessions are structured with regards to block types and critical sections. Blue rectangles are classical blocks, orange rectangles are quantum blocks. Critical sections are highlighted in red. Corresponding `iqoala` and `yaml` configuration files can be found in the `configs` folder in the project repository [32]. Note that the length of the different types of blocks presented in this figure is not proportional to the actual duration of blocks.

Dataset	Shorthand name	Number of sessions of		
		BQC	PP	QKD
0	BQC	N		
1	PP		N	
2	QKD			N
3	BQC & PP	$N/2$	$N/2$	
4	BQC & QKD	$N/2$		$N/2$
5	PP & QKD		$N/2$	$N/2$
6	BQC & PP & QKD	$N/3$	$N/3$	$N/3$

Table 4: Combinations of sessions in a dataset if a given dataset consists of N sessions in total. The abbreviations refer to the three applications considered in this project: blind quantum computing (BQC), the ping-pong test (PP), and quantum key distribution (QKD).

6.3 General Code Workflow

In this subsection, we present a high-level overview of all the code logic involved in implementing program scheduling and evaluating the quality of constructed node schedules. There are four main steps which are in detail described below. The implemented code is available in a GitHub repository [32].

Step 1: Data preparation. Using Qoala programs defining program configuration (specific for each application and for each node), we create YAML configuration files used for program scheduling. For this, we use the `setup_configs.py` script. Note that after using this script, the critical sections must be manually defined and an application deadline can be optionally specified. We furthermore prepare datasets (a specific combination of session types and a given number of total sessions, defined in Table 4) using the methods in `datasets.py`.

Step 2: Network schedule generation. Given a dataset, a random network schedule is generated using the `NetworkSchedule.generate_random_network_schedule()` method. The exact implementation of randomly generating network schedules is explained in Section 5.

Step 3: Node schedule construction. A pair of coupled node schedules (one for each of the involved nodes) is constructed for each combination of dataset and network schedule using the `create_schedules.py` script. The construction of a node schedule automatically saves the resulting classical success metrics. Details on the RCPSP model used for the program scheduling can be found in Section 6.4.

Step 4: Node schedule execution. Given two coupled node schedules, the dataset, and the relevant Qoala programs, the node schedules are executed using the Qoala simulator with the `execute_schedules.py` script. Here, we evaluate the quantum success metric of the node schedule execution. Further details on this step are given in Section 6.5.

6.4 RCPSP Model

The framework of Resource Constrained Project Scheduling Problem (RCPSP) is used when a list of activities with precedence relations and resource constraints need to be scheduled. In the case of application scheduling, the activities are blocks and the goal is to schedule their execution on either the *classical processing unit* (CPU) or *quantum processing unit* (QPU) depending on the nature of the instructions contained in the block. The processing units are then the renewable resources. Precedence constraints for this problem arise from the block ordering within a session, though there is no order defined between different sessions. There are further timing constraints based on the deadlines for application execution as well as the nature of quantum states which lose information over time.

To fully define the RCPSP model used in this project, we describe inputs to the model and the implemented constraints, and explain the three possible approaches to constructing a node schedule. We then investigate how the risk-aware extension could be implemented to deal with uncertainty in activity durations, and we comment upon the limitations of PyCSP3.

6.4.1 Model Specification

In this section, we describe how the inputs for the RCPSP model implemented in PyCSP3 are defined. First, we consider what kind of information about the blocks is directly used by program scheduling. We also describe in detail all the constraints used in the model and how they relate to the constraints defined for a feasible node schedule in Definition 3.1.

As mentioned earlier in Section 6.3, instead of manually defining all the information about each block as outlined in Table 2, we make use of the Qoala simulator to pre-process the input data. For the Qoala simulator, we define the programs to be executed on each of the nodes including specific instructions for both the CPU and QPU (these can be found in the `configs` folder in the project repository [32]). Using the `setup_configs.py` script, we automate extracting the type and duration of each of the blocks being scheduled. This is exported into a YAML file: after this step it is necessary to manually define the critical sections and the session ID, and optionally include an application dealine. In this way, we do not make use of the parameters defined in Table 2 coming from either the block metadata or the node lookup table; this would be different if another simulator was used.

An input to the PyCSP3 model for program scheduling is therefore a list of blocks, a list of their types, a list of their durations. For the RCPSP implementation, we also define a list of successors, resource requirements, and maximum time lags. PyCSP3 works with lists because it creates a variable array for start times of all the blocks which it continuously fills in. Instead of working with blocks as objects, the model works with a list of each of the parameters.

Having the input data, the PyCSP3 model requires a definition of all the constraints that must be satisfied. The constraints used in our implementation are as follows:

- **Resource constraints**: each block defines the resources which it is using; through the use of the `Cumulative` constraint, we restrict that the capacity of the available resources (the CPU and the QPU) is not exceeded at any point. Note that this implementation easily allows to extensions where we allow parallel executions on the CPU or even the QPU, but in this project we keep a capacity of a single CPU and a single QPU without ”multi-threading”. See Section 8.2 for more details on how this could be used.

- **Precedence constraints**: if a block j belongs to a set of successors of block i, we constrain its starting time s_j by $s_i + d_i \leq s_j$ where d_i is the processing time of block i. The set of successors for each block is constructed based on the order of blocks in a program. Because there is no specific order defined for the programs, the set of successors for the last block in a program is always empty.

- **Maximum time-lag constraints**: whenever a maximum time-lag constraint $d_{ij}^{\max}$ is defined between blocks i and j, the difference in the start times is restricted by $s_j - (s_i + d_i) \leq d_{ij}^{\max}$. The maximum time-lags are used to enforce the critical section constraints; whenever a block i and j belong to the same critical section and block i must be executed before block j, we define $d_{ij}^{\max} = 0$. Note that when a network schedule is defined, we do not consider any maximum time-lags for QC blocks because the start time defined by the network schedule takes precedence.

- **Network schedule constraints**: a j-th QC block of a program i must be scheduled at the start of its respective timeslot $t_{(i,j)}$ defined by the network schedule. We guarantee this by two constraints in the PyCSP3 code: we constrain that each QC block of program i starts on a timeslot assigned to session i and each j-th QC block within any session starts on a timeslot assigned to a j-th QC block. In conjunction, these two constraints guarantee the desired condition.

6.4.2 Finding a Node Schedule

We consider three different approaches to constructing a node schedule: naive scheduling in which programs are simply scheduled in a consecutive manner, heuristic-driven scheduling which relies on the use of heuristics to find a node schedule satisfying the constraints, and optimal scheduling which makes use of an objective function and guarantees optimality.

For constructing naive schedules, we add another constraint to the RCPSP model. This constraint states that if $i < j$ for any two blocks i, j, the start times s_i, s_j are assigned such that $s_i < s_j$. Notice the difference to the precedence constraint defined in Section 3.2: in that constraint we enforce the order of blocks within a session, while here we impose an order for all the blocks to be scheduled. This constraint guarantees no blocks will be scheduled in parallel on the CPU and the QPU, therefore constructing a naive node schedule.

By default, PyCSP3 uses the `ACE` solver [34, 35]. This solver comes with a default set of settings as defined in its `Control.java`[1] class. There are two heuristics the solver makes use of: a variable ordering heuristic and a value ordering heuristic. The default for the variable ordering heuristic is the `dom/wdeg` heuristic that selects variables based on constraint weighting [36]. The default heuristic for ordering values is the `min` heuristic which selects the minimal value in the domain of the currently selected variable. PyCSP3 allows the user to specify more detailed settings for the solvers, including which heuristics to use [37]. We start out by using the default heuristics to see how they perform and later reflect on whether other heuristics would be more suitable.

Lastly, PyCSP3 allows for specification of objective functions. Whenever an objective function is specified, the solver iterates through all possible solutions to find the optimal one. We use this to construct an optimal node schedule where we define optimality as minimised makespan. It is not possible to define an objective function based on the average success probability, because that is a metric we obtain only after executing the node schedule. Note that finding the optimal node schedule using an objective function is computationally more expensive because it means solving a constraint optimisation problem instead of a constraint satisfaction problem.

[1]`https://github.com/xcsp3team/ace/blob/main/src/main/java/dashboard/Control.java`

6.4.3 Risk-aware RCPSP Extension

There are two sources of uncertainties in the schedule that is being created with the RCPSP workflow. These are the blocks with quantum communication operations and blocks with classical communication (in the case of classical communication, the uncertainty only applies to receiving messages). Uncertainty in activity duration can be mitigated by extending the RCPSP framework to be so-called "risk-aware". We describe the two main approaches to risk-aware RCPSP in Section 2.4.1.

Throughout the research into risk-aware RCPSP, it became obvious that the domain of classical systems makes some underlying assumptions about the context of the schedules and this might not directly translate to the quantum domain. Firstly, it is note-worthy that due to the inevitable randomness of quantum applications, a failure in execution of one session is not highly-critical — it is very common that it would need to be rerun until successful execution. Secondly, we assume that the current implementation of quantum hardware and the interface with classical control means the time scale of quantum operations is much lower than the communication delay between the quantum hardware and the classical control unit. This limits the possibility of reactive behaviour during the schedule execution.

Reactive scheduling as an approach to risk-aware RCPSP is not applicable to program scheduling precisely because of this communication delay between quantum hardware and the classical control unit. It might not be feasible to make ad-hoc decisions whenever an activity duration deviates from the schedule; the use of the static mode of node schedule execution (in which we first construct the schedule and then execute) means reactive scheduling is not suitable due to the interactions between the CPU and the QPU.

Proactive scheduling and the construction of robust schedules relies on including time buffers. In classical systems, the guaranteed robustness may outweigh the extra cost of delays due to these buffers, but in the domain of quantum program scheduling, an introduction of any unnecessary delays might have a detrimental effect on the quality of quantum states due to the effects of decoherence. For this reason, proactive scheduling is not suitable for mitigating the consequences of uncertainty in program scheduling.

It seems that neither reactive or proactive scheduling lends itself to be useful in program scheduling. However, as our project also introduces a way of executing node schedules, we do propose a "risk-aware" addition to the Qoala simulator. Note that this very much depends on the actual implementation of the simulator and we cannot guarantee this will be possible on actual quantum hardware.

The proposed solution is to implement a *left-shift rule* for the schedule execution. An implementation of the left shift rule means that whenever a block executes earlier than expected (i.e. entanglement is generated faster than expected or a classical message arrives earlier than expected), any blocks scheduled to execute in the future are moved to execute earlier. (Note that there is no guarantee this will always be possible due to other constraints, such as the resource constraints.)

To be able to investigate this risk-aware extension, we need to be able to adjust the duration of individual blocks with classical or quantum communication in the Qoala simulations. Unfortunately, while the current version of the Qoala simulator (0.2.2) provides a way to do this for quantum communication blocks (by adjusting the `prob_success` for the `Link` configuration), it is not implemented for blocks with classical communication. Nonetheless, if this is implemented in the future, the investigation of a risk-aware extension to execution of node schedules should be fairly straightforward. We elaborate on this as part of the possible future research directions in Section 8.2.

6.4.4 Limitations of PyCSP3

It is important to mention that PyCSP3 and its solvers have limited computational capabilities. With problem instances that are large enough, the solver cannot find a solution even though the problem is feasible to solve. The documentation of PyCSP3 does not offer any official information on where these limits reside, so the limits were being continuously explored during this project.

To increase the domain of problems PyCSP3 could solve, we implemented an additional step to program scheduling. When we formulate program scheduling within the RCPSP framework, the solver is solving a problem of assigning a start time to each of the block. Each assignment of a start time has a domain that is as long as the node schedule. This means that the longer the node schedule, the more possible values there are for each start time and the more complex the problem becomes.

Each of the blocks to be scheduled has a certain duration on a time scale of nanoseconds. There is no block that only takes 1 nanosecond — in fact, the shortest block our implementation works with is a classical local block which takes 100000 ns (0.1 ms). We therefore implement *scaling* of all the variables in the RCPSP formulation by the greatest common divisor of all the block durations. This greatest common divisor is usually of the order of magnitude of 100000 nanoseconds, and therefore this additional step in our implementation helps us reduce the problem size by several orders of magnitude.

Despite this optimisation, the computational powers of PyCSP3 are still a limiting factor in some of our design decisions. For example, when deciding on the number of programs that should be scheduled, we found out that 18 programs were already not feasible to solve by PyCSP3. Furthermore, the difference in orders of magnitude between the shortest and longest block determine how much is the above-mentioned optimisation useful — therefore the choice of some hardware parameters mentioned in the next subsection was also motivated by the limits of PyCSP3.

6.5 Evaluating Quantum Success Metrics

As mentioned in Section 4, some of the success metrics of program scheduling can only be evaluated after an execution of the constructed node schedules. Because of that, we include simulations of the node schedule execution in the workflow of program scheduling. To simulate how the node schedules would perform, we use the Qoala simulator [31]. In this section, we describe how the simulations in Qoala are set up and what kind of data is necessary to run them.

To run the Qoala simulations, it is necessary to provide the Qoala programs describing the blocks of instructions both nodes need to execute, as well as the constructed node schedules for Alice and Bob. The node schedules are then used to create what Qoala calls a `TaskSchedule` — a schedule of when are the particular blocks executed. In the Qoala simulations, it is possible to define an entry in the `TaskSchedule` only using its predecessors in which case Qoala automatically executes it as soon as possible. To ensure the node schedule execution satisfies the constraints imposed by the network schedule, we use both the order and the start times of the blocks in the constructed node schedules. Because quantum simulations are probabilistic, an execution of the coupled node schedules is then run many times to get an average of the quantum success metrics.

It is also important to note that the behaviour of the Qoala simulations depends on the parameters one sets for the hardware of the nodes and of the links in the quantum network. For this project, we chose a set of parameters that corresponds to realistic hardware parameters for colour centres in diamonds [24]. The choice of some of the software parameters was motivated by discussions with engineers at QuTech [38] but also by the limitations of PyCSP3 mentioned in the previous section. An overview of all the parameters defined in Qoala can be seen in Table 5.

The average success probability depends on the result of execution of every individual session. To evaluate this quantum success metric, we need to be able to determine whether a session has

Group	*Parameter*	*Value*	*Description*
Node parameters	Communication T1	1 s [39]	T1 time of communication qubits
	Communication T2	1 s [39]	T2 time of communication qubits
	Storage T1	1 s [39]	T1 time of storage qubits
	Storage T2	1 s [39]	T2 time of storage qubits
	Single-qubit duration	20 μs [24]	Duration of single-qubit gates
	Single-qubit noise	0.013 [39]	Noise in single-qubit gates
	Multi-qubit duration	500 μs [39]	Duration of multi-qubit gates
	Multi-qubit noise	0.04 [39]	Noise in multi-qubit gates
Link parameters	`cycle_time`	20 ms [24]	Duration of a QC block
	`prob_success`	1	Probability of successful entanglement generation
	`prob_max_mixed`	0.33	Probability of generating a mixed state
	`state_delay`	0	Delay of delivering states
Software parameters	`HOST_INSTR_TIME`	100 μs [38]	Duration of a classical instruction
	`HOST_PEER_LATENCY`	10 ms [38]	Duration of a classical communication block
	`QNOS_INSTR_TIME`	100 μs [38]	Duration of a classical instruction in a QL block

Table 5: An overview of all the parameters necessary to set up Qoala simulations. The hardware parameters related to nodes are set in the `qoala_topology_config.yaml` file, the parameters related to the quantum link are specified in the `qoala_link_config.yaml` file, and the software parameters are defined as global variables in the `execute_schedules.py` script [32].

executed successfully or not. We therefore define the success/fail metric for each application separately:

BQC: One session of the BQC application corresponds to one round of computation that the server (Bob) performs on behalf of the client (Alice). In the BQC sessions executed within this project, we can set all the input parameters the client is using and therefore we are aware of the expected measurement outcome the server should obtain. The success condition for a BQC session is therefore whether the server gets the expected measurement outcome.

Ping-pong: One session of the ping-pong application consists of two teleportation operations: Alice prepares a state (randomly chosen between $|0\rangle$ and $|1\rangle$) and then two teleportation operations move the state to Bob and back to Alice. We evaluate whether this happened successfully by a measurement of Alice's state. Note that a better evaluation method would be to inspect the state Alice ends up with and calculate its fidelity compared to the state she meant to prepare. However, this is not possible because we cannot examine the quantum state while the simulation is running.

QKD: Execution of quantum key distribution can be evaluated based on the number of bits of secret key shared between the two involved parties. For the purposes of this project, we define a QKD session as generation of a shared key of length 3. Using simulations, we can set the measurement bases in which Alice and Bob measure and therefore we know the expected number of bits of key Alice and Bob should share. A QKD session is successful if Alice and Bob indeed retrieve a key with the expected number of bits.

In this section, we have described what kind of data we use in this project and how is the program scheduling and execution of node schedules implemented. The entire implementation can be found in the GitHub repository for this project [32]. In the next section, we look at the results from our simulations and answer the research questions posed in Section 1.

7 Results

In this section, we show and discuss the results of evaluating the RCPSP implementation of program scheduling. Firstly, we comment on the consequences of classical communication not being synchronised (unlike the quantum communication based on the network schedule) and what this means for the evaluation of program scheduling using the proposed success metrics. Then we compare the performance of heuristic-driven and optimal program scheduling. Lastly, we consider a scenario in which there are no network schedules and compare the performance of naive and heuristic-driven approaches to program scheduling.

The entirety of the code used to obtain these results is available on GitHub [32]. This repository also contains all network and node schedules derived during this investigation, as well as all the processed data and scripts used to create plots shown in this section.

7.1 Effect of Asynchronous Classical Communication

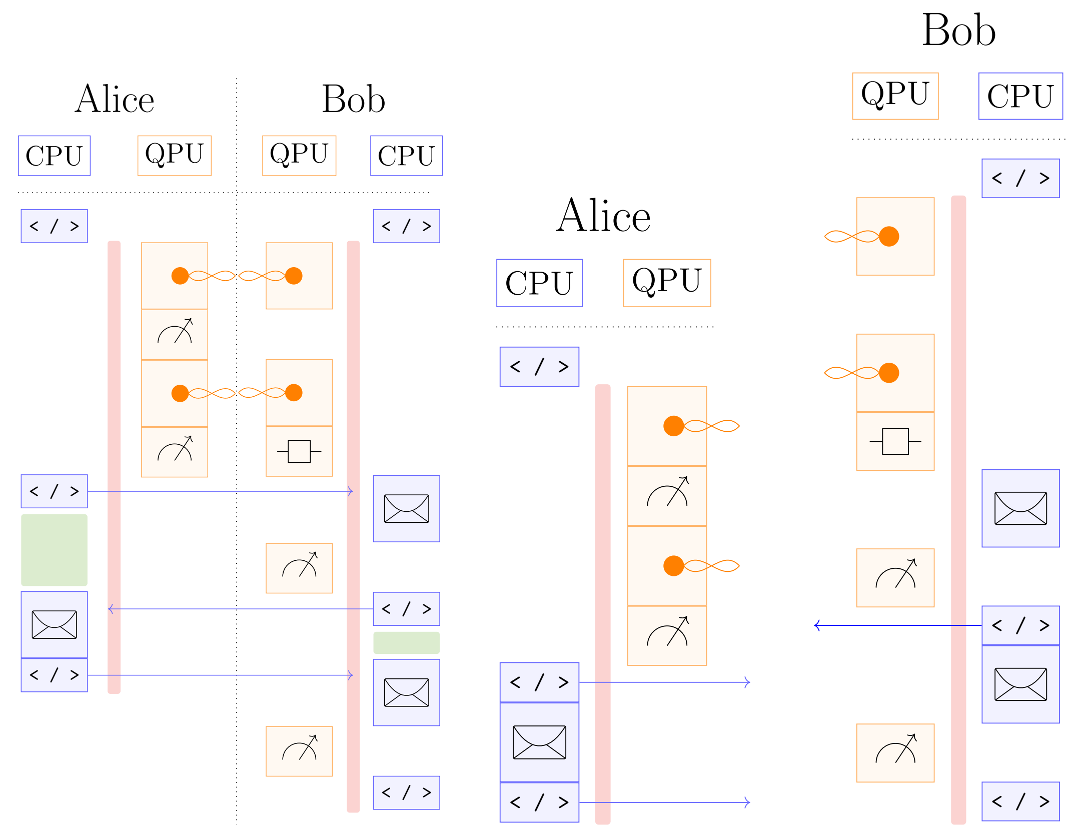
Before we show the results of different approaches to program scheduling, it is necessary to mention a limitation of the simulations and of program scheduling as a whole. One of the key assumptions in program scheduling is that nodes have no information about the programs the other nodes in the network are executing. This ensures the security guarantees of certain applications are not compromised. It is also the motivation for having the network schedule — a central controller must decide when entanglement generation can be attempted using the aggregated data from all the nodes. The network schedule constraints ensure that quantum communication is scheduled at the same time. However, no such equivalent concept exists for classical communication.

This means that nodes cannot know when to expect classical messages from other nodes. Program scheduling is by its definition limited to schedule receiving of a classical message at a point in time when the message might not have been sent yet. To illustrate this point, in Figure 9 we contrast how a BQC session looks (taken from Figure 8a) with how such a session would be scheduled locally for Alice and Bob.

This has several consequences for program scheduling: it limits the possibility to optimise construction of node schedules and renders classical success metrics retrieved from static node schedules potentially inaccurate. Upon execution of the node schedule, receiving a classical message becomes a blocking operation — if the local scheduler knew when it could expect the classical message to arrive, it could have scheduled other blocks in the meantime. The makespan of a static node schedule does not necessarily correspond to a makespan of the node schedule execution because some delays might be caused by a classical message arriving later than expected. It is therefore anticipated that the makespan retrieved from Qoala simulations will be lower bounded by the makespan of static node schedules.

In Figure 10, we compare the static and dynamic makespan for node schedules constructed based on one random network schedule per each dataset. The static makespan is retrieved from the constructed node schedules, whereas the dynamic makespan is retrieved from executing the node schedules using the Qoala simulator. We can observe that indeed the dynamic makespan is at least as large as the static one, and in some cases it is larger. This is caused by the unforeseen delays in classical communication. It is important to highlight that it is expected that the node schedules of Alice and Bob have different makespan values because they might be executing different programs.

We do not have precise information about how long each of the classical communication blocks take — this information is only available during the runtime of Qoala. It is therefore difficult to evaluate the total duration of all blocks executed on the CPU and the QPU; for that



(a) A BQC session defined in terms of blocks, their types, and critical sections (copied from Figure 8a).

(b) How a BQC program would be scheduled in Alice's node schedule.

(c) How a BQC program would be scheduled in Bob's node schedule.

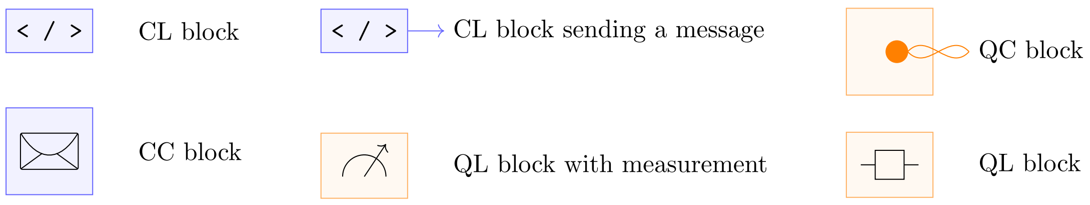


Figure 9: Illustration of asynchronous classical communication. Note that the quantum communication blocks are restricted by the network schedule and are therefore always scheduled at the same time. The green rectangles highlight where the local scheduler schedules receiving a classical message earlier.

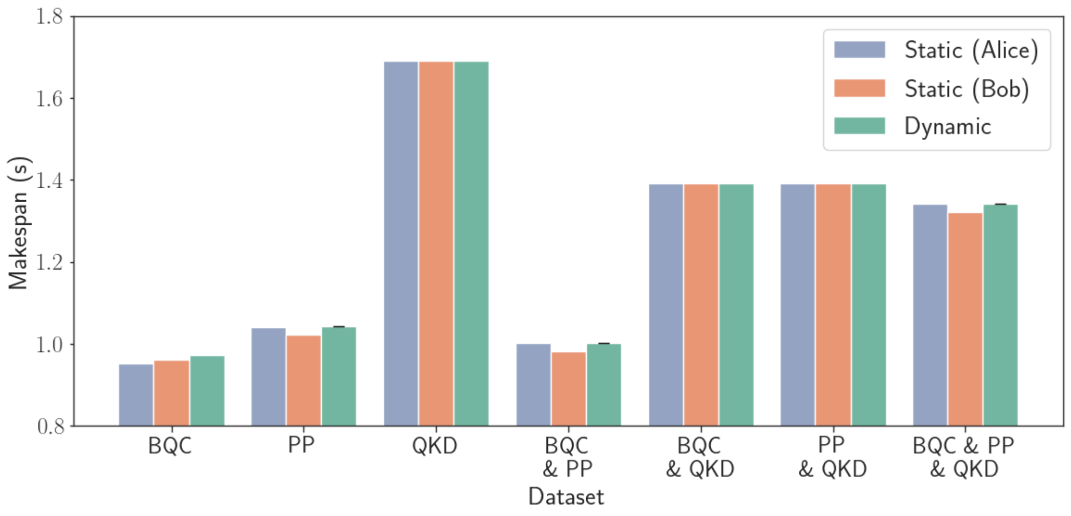


Figure 10: Comparison of makespan retrieved from static node schedules and from the executed pair of node schedules using the Qoala simulator. The node schedules are constructed using the heuristic-driven approach based on datasets with 6 sessions (see which datasets the short-hand notation corresponds to in Table 4). For each of the dataset, we pick one random network schedule and show the makespan values for node schedules constructed based on this network schedule; this means that all the static makespan values that are shown are from a single data point and therefore do not have any error bars. The dynamic makespan is the average makespan retrieved from 200 simulation runs — here we can see some standard deviation for a few of the datasets. It is not clear where precisely this deviation comes from, but we attribute it to how the Qoala simulator resolves execution of instructions. If there is no error bar shown on the dynamic makespan, it means it is the same in all the Qoala simulations.

reason we do not report on the values of the processor utilisation factors defined in Section 4. Note that the PUF values are directly dependent on the values of makespan, and since we show results for the values of makespan, these results should also be representative of the trends in the PUF values.

The question of how to mitigate the unpredictability of classical communication is left as an open question. It is not immediately obvious whether there is anything that can be done to synchronise classical communication without introducing unnecessary delays in the schedules. Based on the results shown in Figure 10, the difference in static and dynamic makespan values is not significant, but it is nevertheless an interesting insight into the capabilities of program scheduling given its current definition.

7.2 Heuristic-Driven and Optimal Program Scheduling

In this section, we look into comparing the performance of heuristic-driven and optimal program scheduling. We first explain how we represent uncertainty in the data based on an example with 6 sessions of BQC. Then, we compare the classical success metric (makespan) and the quantum success metric (average success probability) for all the datasets. Lastly, we look at the time it takes to construct node schedules using the heuristic-driven and optimal approaches to compare their computational requirements.

To answer the research question on heuristic-driven and optimal program scheduling, we first generate a 100 random network schedules for all datasets with 6 and 12 sessions using the method described in Section 5. These random network schedules are used in place of network schedules that would otherwise be constructed by network scheduling. The form of a network schedule can greatly influence what kind of node schedules can be constructed, which is why we generate numerous network schedules and evaluate the averages of the success metrics. For the plots shown in this section, we use the short-hand notation introduced in Table 4 to describe which sessions the datasets consist of. Note that when we say a node schedule is constructed for a given dataset, we mean the dataset to consist of programs and not sessions.

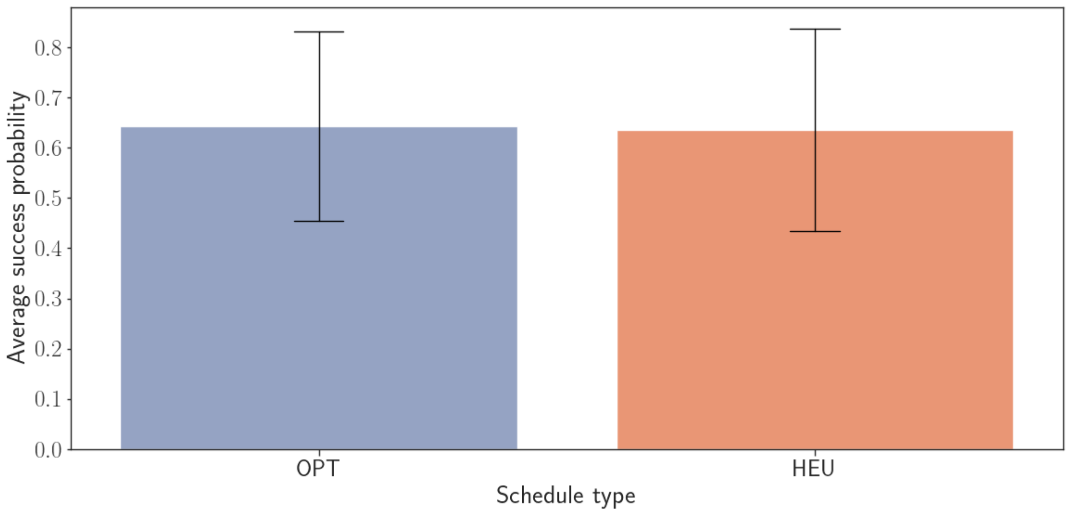
First, we show a comparison of the average success probability for dataset consisting of 6 BQC sessions based on one randomly generated network schedule in Figure 11a. The value of the average success probability is an average over 200 runs of the Qoala simulations (this is the default for any node schedule execution in this project). The error bars shown in this figure represent the standard deviation of the success probability. The standard deviation is quite high — that is because when scheduling 6 sessions, the success probability of executing a pair of coupled node schedules can only take on 6 discrete values (multiples of 16.6 %). The variance in this data is intrinsically large and running the simulation more times would not decrease the size of the error bars.

Next we consider the average success probability for one dataset but averaged over a 100 randomly generated network schedules. In Figure 11b, we show the average success probability for the dataset with 6 BQC sessions. We show two kinds of error bars: the larger red error bars show the standard deviation over all the data points (i.e. each individual simulation run), while the smaller black error bars show the standard deviation of the average success probabilities for each random network schedule (i.e. we first calculate the average success probability for each network schedule separately, and then calculate the standard deviation of these averages). While we cannot remove the intrinsic variance in the data, the smaller error bars should be representative of the program scheduling performance.

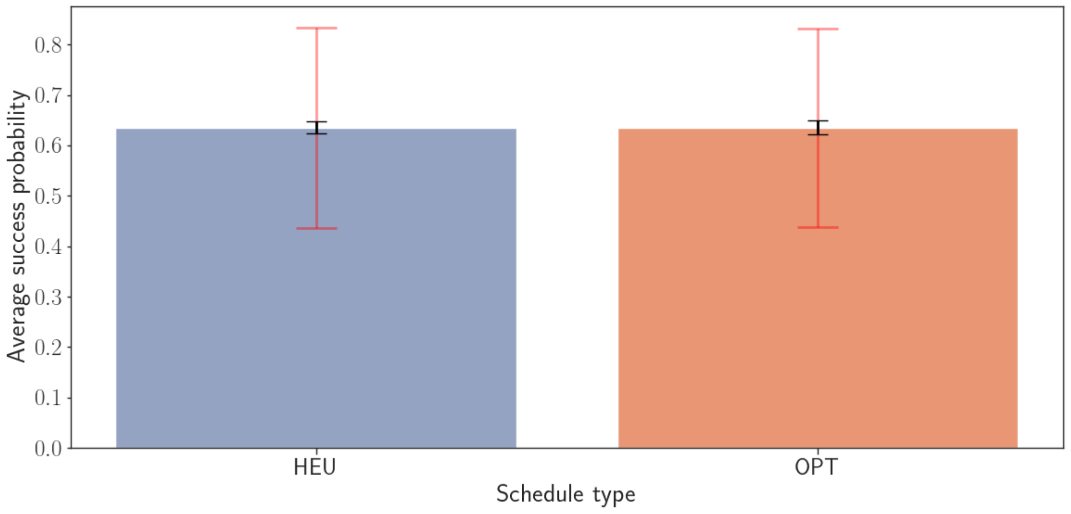
For each of the seven datasets with either 6 or 12 sessions, we generate a 100 random network schedules (that is in total 14 000 different network schedules). Based on each of these network schedules, we construct a pair of node schedules for Alice and Bob using both heuristic-driven and optimal program scheduling (in total, we have 28 000 pairs of node schedules, i.e. 56 000 node schedules). Afterwards we execute each pair of node schedules 200 times using the Qoala simulations. We calculate the average success probability for each dataset of length 6 or 12 and show the results in Figure 12. This allows us to investigate the performance of heuristic-driven and optimal program scheduling.

To compare the performance of heuristic-driven and optimal program scheduling, we can observe whether the achieved success metrics are any different for the two approaches. First, let us comment on the obtained values of makespan: we find that the dynamic makespan values are very similar for the heuristic-driven and optimal approaches. For each of the datasets, we calculate the average makespan over all the random network schedules run 200 times and we report the relative difference in Table 6. The relative difference is very small. Interestingly enough, in a few cases when the relative difference is positive, the makespan of the heuristic-driven node schedules is actually shorter than the optimal node schedules; this might be due to some uncertainty in the data because this is averaged out over many random network schedules.

Next we investigate the behaviour of the average success probability and whether it differs for heuristic-driven and optimal program scheduling. Figure 12a shows that the success probability is statistically similar for both the heuristic-driven and optimal approach. Unfortunately, the large intrinsic variance in the data prohibits us from making any stronger claims about the



(a) Average success probability for node schedules constructed for dataset with 6 BQC sessions based on **one** randomly generated network schedule (with ID 8). The values shown are averaged from 200 runs of the Qoala simulations, and the error bars show standard deviation of these values. The variance of the data is intrinsically high due to the discrete values for the average success probability.



(b) Average success probability for node schedules constructed for dataset with 6 BQC sessions based on a **100** randomly generated network schedules. The values shown are averaged from 200 runs of the Qoala simulations. The red error bars show standard deviation of all the individual success probabilities, while the black error bars show the standard deviation of averaged success probabilities per individual network schedules.

Figure 11: Comparison of the average success probability for node schedules constructed for a dataset with 6 BQC sessions (dataset 0), based on one random network schedule (11a) and a 100 randomly generated network schedules (11b.

Number of sessions	**Dataset**						
	BQC	PP	QKD	BQC & PP	BQC & PP	PP & QKD	BQC & PP & QKD
6	0.0	$-1.5 \cdot 10^{-7}$	0.0	$1.2 \cdot 10^{-7}$	0.0	$-2.2 \cdot 10^{-9}$	$-5.3 \cdot 10^{-8}$
12	0.0	$1.4 \cdot 10^{-9}$	0.0	$-4.0 \cdot 10^{-8}$	0.0	$-8.5 \cdot 10^{-9}$	$-1.0 \cdot 10^{-8}$

Table 6: Comparison of the makespan values for heuristic-driven and optimal program scheduling for datasets with 6 and 12 sessions. The relative difference shown in this table is calculated as $(\mu_{\text{OPT}} - \mu_{\text{HEU}})/\mu_{\text{OPT}}$ between the average makespan values μ_{HEU} and μ_{OPT} for heuristic-driven and optimal program scheduling respectively. If the value shown is negative, it means the heuristic-driven makespan is higher.

differences. Nevertheless, it is reasonable to say that we can achieve comparable performance with the heuristic-driven approach compared to creating optimal node schedules.

This is supported by the fact that the dataset with 12 sessions in Figure 12b show the same trends. Note that the total standard deviation (the red error bars) is smaller for the datasets with 12 sessions because then there are in fact 12 distinct possible values for the resulting average success probability. Nonetheless, the average success probability does not seem statistically different for the heuristic-driven and optimal program scheduling even for the datasets with 12 sessions. The relative differences in the achievable average success probabilities also behave similarly across the datasets with 6 and 12 sessions. In conclusion, neither the makespan nor the average success probability show that heuristic-driven program scheduling performs worse than the optimal approach.

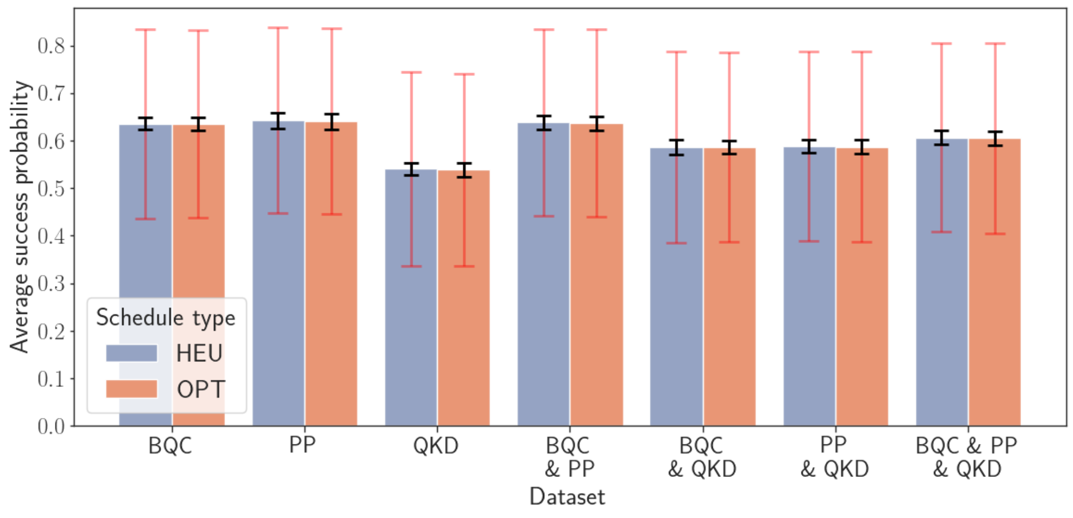
A noticeable difference between the heuristic-driven and optimal program scheduling can be seen in its computational time. In Figure 13 we show the average time taken to construct a node schedule for the different scheduling approaches and different numbers of sessions in a dataset. We can see that a node schedule for a dataset with more sessions takes longer to construct. We can also see a statistically significant difference between the heuristic-driven and optimal approach: heuristic-driven approach performs better.

To investigate the trends more closely, we define an improvement factor of the average time taken to construct a node schedule as

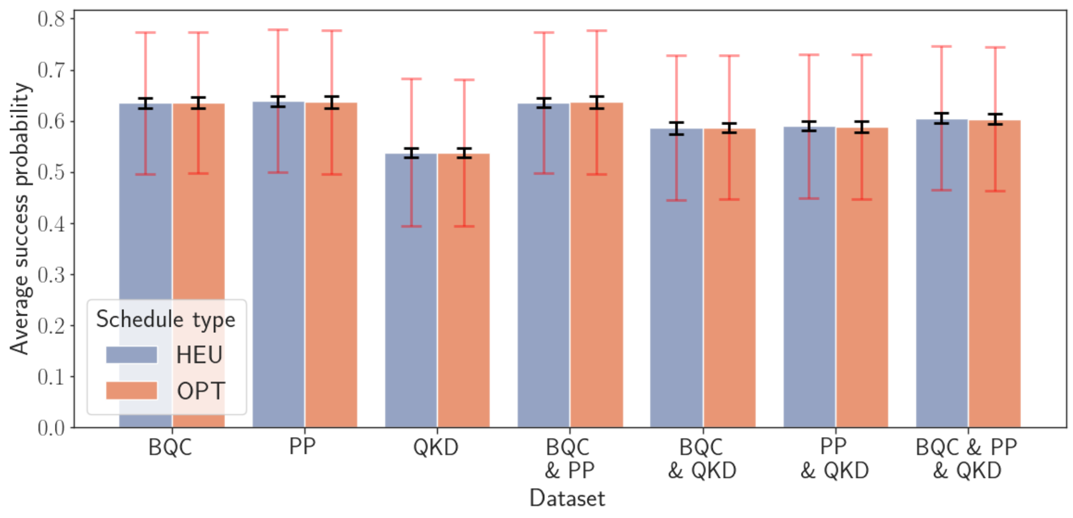
$$\text{Improvement Factor} = \frac{\text{Average computational time using heuristics}}{\text{Average computational time using optimal approach}}$$

This is plotted in Figure 14. The lower the value in the figure, the less time it takes for the heuristics to construct a node schedule compared to the optimal approach. We can see all data points except for one are below the value of 1, therefore showing heuristic-driven program scheduling is faster than optimal scheduling. We also observe a greater improvement with larger datasets; use of heuristics is more beneficial with larger problem instances.

To summarise our investigation into heuristic-driven and optimal program scheduling based on randomly generated network schedules, we can conclude that while performance of both approaches does not seem to differ in terms of the makespan or the average success probability, heuristic-driven program scheduling has lower computational time. This is a useful result for any future investigations into program scheduling using RCPSP as computational resources can be saved by using heuristics. Because we achieved comparable performance to optimal program scheduling using the default heuristic in PyCSP3, we do not investigate using different heuristics in this project.



(a) Average success probability for all datasets with **6 sessions** based on a 100 randomly generated network schedules and 200 runs of Qoala simulations.



(b) Average success probability for all datasets with **12 sessions** based on a 100 randomly generated network schedules and 200 runs of Qoala simulations.

Figure 12: Comparison of the average success probability for heuristic-driven and optimal program scheduling based on datasets with 6 and 12 sessions. The red error bars show standard deviation of all the individual success probabilities, while the black error bars show the standard deviation of averaged success probabilities per individual network schedules.

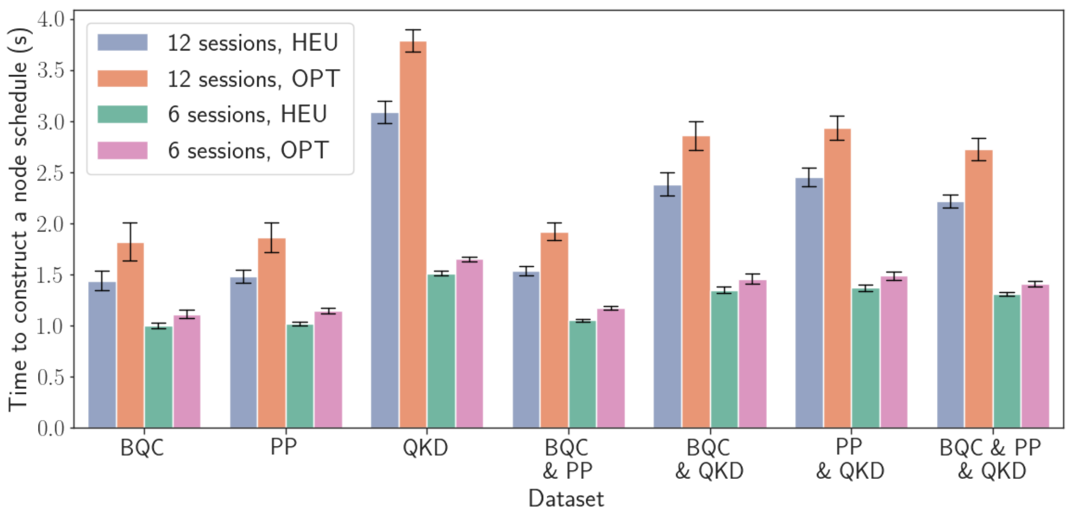


Figure 13: Comparison of the time it takes to construct a node schedule using either the heuristic-driven or optimal program scheduling based on datasets with 6 or 12 sessions. This is averaged across the construction of 100 pairs of node schedules for each combination of dataset size and scheduling approach. The error bars show the standard deviation of the computational time.

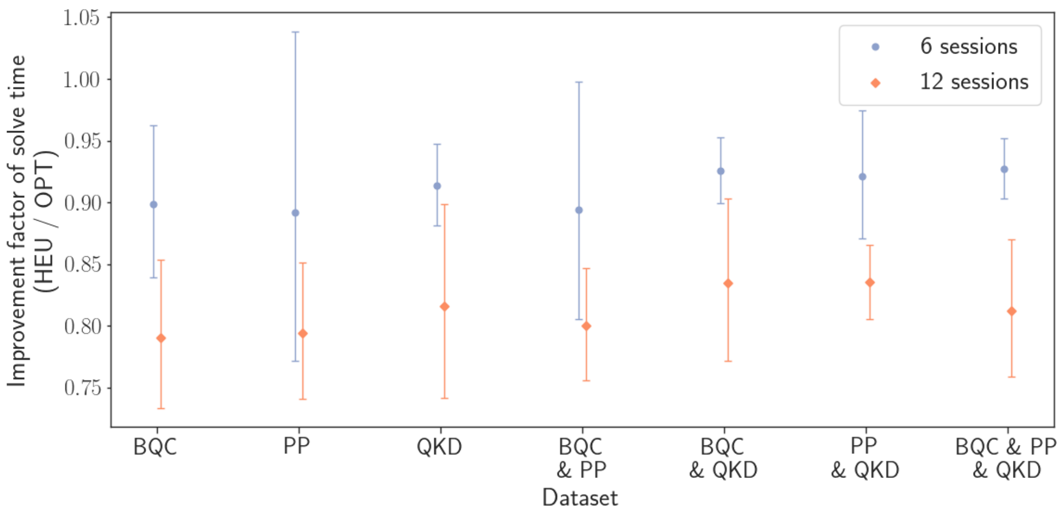


Figure 14: Improvement factor of time taken to construct a node schedule between the heuristic-driven approach and optimal approach. The error bars shown here represent the maximal and minimal improvement factor calculated based on the average solve time μ and its standard deviation σ (shown in Figure 13). More specifically, the minimum and maximum improvement factors are calculated as $IF_{\min} = {}^{\mu_{\text{HEU}}-\sigma_{\text{HEU}}}/_{\mu_{\text{OPT}}+\sigma_{\text{OPT}}}$ and $IF_{\max} = {}^{\mu_{\text{HEU}}+\sigma_{\text{HEU}}}/_{\mu_{\text{OPT}}-\sigma_{\text{OPT}}}$.

7.3 Node Schedules Without Network Schedule Constraints

The RCPSP implementation allows for construction of node schedules without network schedule constraints. This scenario does not represent a situation that would be aligned with the quantum application scheduling workflow [3], but offers insights into a case where nodes in a network have access to some on-demand entanglement generation device. In this section, we analyse the performance of node schedules constructed without any network schedule constraints and evaluate whether such an approach results in better values for the success metrics.

We introduce a new approach to program scheduling called *naive*. This approach schedules programs sequentially without any interleaving. The likelihood that our proposed method of generating random network schedules (see Section 5) creates a network schedule which allows for naive scheduling is very low. In this section, we therefore investigate the performance of naive and heuristic-driven program scheduling in the absence of network schedule constraints.

If there is no network schedule dictating precise timeslots for entanglement generation, the local scheduler is free to choose any start time for its QC blocks. Note that nodes cannot be aware of any scheduling decision undertaken by the other nodes for security reasons (e.g. being aware of the other node's expected computation time in the context of a BQC application can compromise the security guarantees). However, the structure of programs being scheduled on two nodes might be different and it is possible that both nodes will schedule entanglement generation to start at different times. The Qoala simulator only simulates entanglement generation once a request for generating entanglement is submitted by both nodes; this is a blocking operation and whichever node request the entanglement generation sooner cannot execute anything else on the QPU in the meantime.

Now that there are no network schedule constraints, there is a unique pair of node schedules for each combination of a dataset and scheduling approach. When we compare the average success probability for naive and heuristic-driven program scheduling in the absence of network schedule constraints in Figure 15, we cannot conclude any statistically significant results because of the large intrinsic variance in the data. This is again a consequence of having only a limited number of discrete values for the success probability. Additionally, there is not a clear trend in the time it takes to construct a node schedule using either of these scheduling approaches nor any clear difference in the obtained makespan values.

Lastly, we want to consider whether a comparison of performance of program scheduling with and without network schedule constraints provides any insights into the usefulness of network schedules. To that end, we compare the success metrics of heuristic-driven program scheduling based on either randomly generated or no network schedules in Figure 16. In Figure 16a, we see that the average success probability is sometimes lower for the node schedules constructed with network schedule constraints, however, the size of the error bars prevents us from making any statistically significant conclusions.

It is in fact possible that node schedules constructed in the absence of network schedule constraints might perform better. The reason for that might be the values of the makespan: for node schedules without network schedule constraints, the makespan values are around three times smaller than for node schedules based on random network schedules (see Figure 16b). Nonetheless, having no network schedules is not a feasible solution for when different pairs of nodes want to execute quantum applications on a quantum network and the resources to generate entanglement must be shared. We conclude that network schedules bring a guarantee of utility of the network resources and do not cause an observable decrease in the performance of program scheduling.

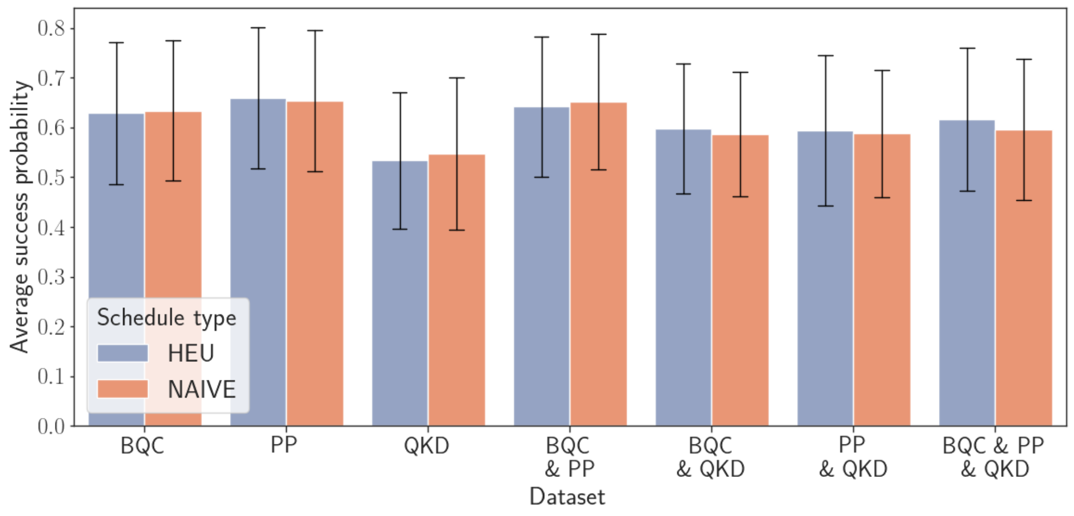
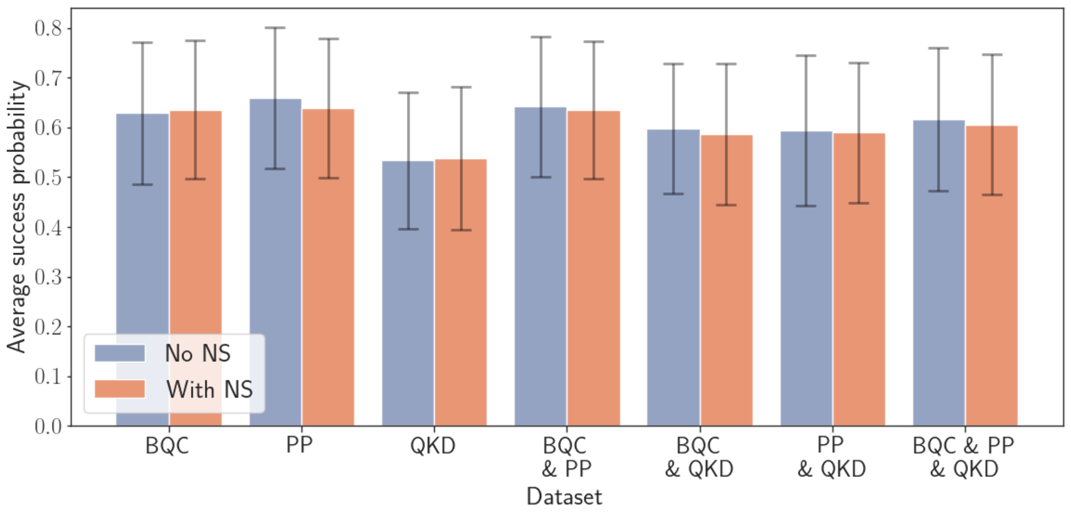
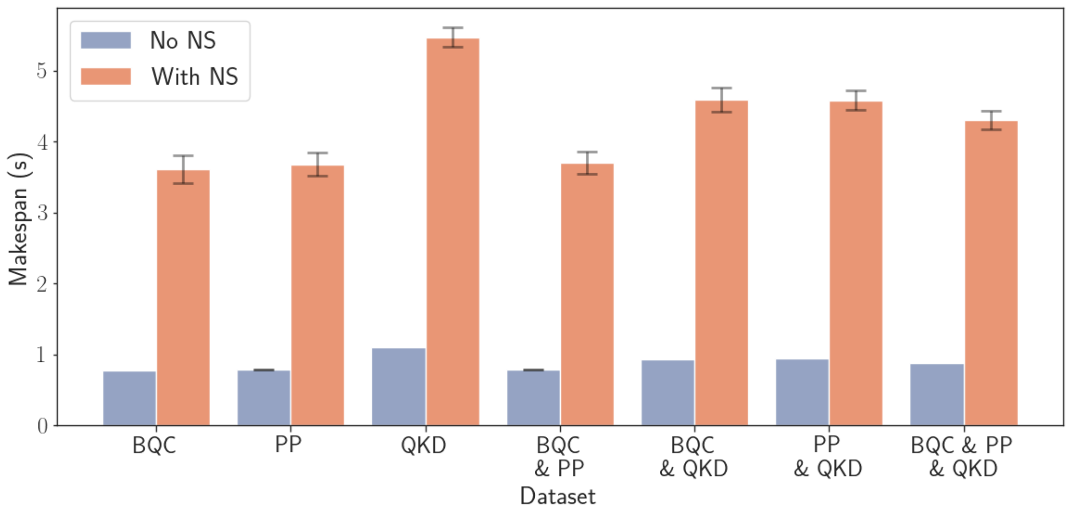


Figure 15: Comparison of the average success probability of node schedules constructed using naive and heuristic-driven node schedules in the absence of network schedule constraints. This is averaged over 200 runs of Qoala simulations, and the error bars show the standard deviation.



(a) Comparison of the average success probability for heuristic-driven program scheduling run with and without network schedule constraints. The average success probability is calculated based on 200 runs of Qoala simulations, and the error bars show the standard deviation.



(b) Comparison of the makespan for heuristic-driven program scheduling run with and without network schedule constraints. The makespan values are averaged over 200 runs of Qoala simulations. Note that in the case of no network schedule constraints, we only have one pair of node schedules and so some of the datasets show no deviation in the obtained makespan. The small error bars for datasets with PP and BQC & PP arise from the Qoala simulator. The error bars for node schedules based on network schedules reflect the fact that different network schedules result in different makespan values.

Figure 16: Comparison of success metrics for heuristic-driven program scheduling run with ("With NS") and without ("No NS") network schedule constraints.

8 Conclusion

In this section, we discuss the limitations of results shown in this thesis and propose directions in which a future research into program scheduling could venture in. We conclude this thesis by a high-level summary of the findings obtained in this project.

8.1 Discussion

Firstly, it is insightful to question the motivation for investigating program scheduling in the first place. The current state of quantum networks and quantum hardware does not require program scheduling yet. However, since program scheduling and the entire architecture for running applications on a quantum network can be developed in parallel to quantum networks through the use of simulations, there is value in investigating this even though it is not yet practically realisable.

Program scheduling is highly dependent on the previous outputs of network scheduling. Because network scheduling is still under development, it is hard to design a realistic input. In this project, we circumvented this by generating random network schedules. Unfortunately, using random network schedules introduces an extra layer of uncertainty in the evaluation of performance of program scheduling. While we can compare how the different approaches to program scheduling behave based on different inputs, it is not possible to reach any absolute conclusions. The problem of finding optimal network schedules is non-trivial to solve because the quality of network schedules can only be evaluated through program scheduling. The evaluation of both network and program scheduling is therefore closely intertwined and must be undertaken simultaneously.

The actual implementation of program scheduling using RCPSP as proposed in this project has several limitations. The scheduling algorithm is fairly complex and its implementation in PyCSP3 imposes strict limits on the size of problems that can be solved. The complexity of RCPSP also affects the computational time — in fact, generating a node schedule takes longer than its subsequent execution. This can still be acceptable if a node schedule is executed multiple times in succession. Nevertheless, in order to mitigate these limitations, it might be worthwhile to consider investigation of other and perhaps simpler scheduling algorithms. We elaborate on this in the next section.

The evaluation of program scheduling and constructed node schedules is also limited by the large intrinsic variance of obtained values for the average success probability. This mainly stems from the small problem instances and a small number of discrete values for the quantum success metric. We can see that the standard deviation is smaller for larger datasets. Unfortunately, it was not computationally feasible to increase the datasets in order to have more precise values of the average success probability. Perhaps looking into a different scheduling approach will allow for larger problem instances for program scheduling, which will in turn allow us to more effectively evaluate the performance.

The last identified limitation to program scheduling is the asynchronous classical communication. As previously remarked upon, the fact that nodes are unaware of what operations are being executed at other nodes means that the classical communication is not synchronised. We do not see a clear way how to circumvent this issue. The consequence of asynchronous classical communication is that program scheduling can only be evaluated by executing the constructed node schedules because only during execution will the effects of asynchronous classical communication become visible.

8.2 Future Research

In this subsection, we outline potential directions for future research. Throughout the course of this project, we encountered many interesting questions that were beyond the scope of our investigation. Here, we provide a summary and insights into each of these questions on each of them, with the hope of inspiring further study of program scheduling scheduling.

Soft deadlines. Hard deadlines are often used in classical problems, but within the quantum field, the use of soft deadlines might be more suitable because the quality of a quantum state decreases with time. Soft deadlines could be implemented by defining a time-utility function such that e.g. a depolarising noise is applied after a certain deadline (or perhaps since the state preparation instructions).

Unordered blocks. For now, we work with the assumption that the list of blocks to be scheduled is ordered. However, the case when such a list would be only partially ordered is more realistic as some blocks can be in theory happening in parallel (i.e. the order of execution does not matter). The question then becomes: how does the application scheduler deal with programs that do not have a clearly defined order of blocks? (Note that if one wants to stay in the realm of RCPSP, it might require some extension such as multi-mode, because "Several heuristic procedures for (approximately) solving project scheduling problems require a *strict order* $\prec$ in node set V." [4, p. 20].)

Increasing capacity of CPU and QPU. In this thesis, we make the assumption that neither the CPU nor the QPU can run two different instructions simultaneously. The limit on CPU does not reflect any realistic scenario because we could assume much larger classical computational power. Furthermore, an on-going research into concurrent execution of programs on the QPU [40] suggests that it could be possible to eliminate the concept of critical sections: assuming sufficient number of storage qubits, it is possible to interleave execution of different critical sections, although it is not evident how much this would affect the average success probability. The capacity of CPU and QPU is therefore another line of direction that can be investigated; in our code, the capacities are given by a parameter and can be very easily adjusted, we therefore make it possible to further investigate this using our proposed implementation. (Note that for implementing swap operations as suggested in [40], some extensions to the code would be required.)

Investigating the left-shift rule. We suggest how to investigate the risk-aware extension to node schedule execution in Section 6.4.3. To be able to do this, the Qoala simulator must be extended to allow for variable durations of blocks with classical communication. A suitable setup of duration of classical communication blocks then becomes crucial. We ideally arrive at a scenario where a delay in classical communication regardless means a very low success probability of a correct execution and as such, the execution can be aborted. This means setting up the durations of blocks for a "worst-case" scenario and using the left-shift rule for improvements during execution.

Different scheduling paradigm. The RCPSP framework provides a lot of flexibility in defining constraints, but it might be too complex for the domain of program scheduling. One could look into utility accrual scheduling methods [41, 42] or simply the earliest deadline first scheduling approach [43] (including a time-dependent utility function as suggested above). Such a simpler method could potentially generate similarly performing node schedules at a lower computational cost.

8.3 Summary

In this thesis, we have investigated an RCPSP implementation of program scheduling. Program scheduling was formally defined within the architecture designed to support execution of quantum applications on quantum networks, and several classical and quantum success metrics were devised to allow for evaluation of the performance of program scheduling.

Our implementation of program scheduling makes use of the Resource-Constrained Project Scheduling Problem (RCPSP) framework. We furthermore provide a way of executing the constructed node schedules using a quantum simulator. Both of these contributions enable an investigation of different approaches to program scheduling: we find that heuristic-driven approaches show no decrease in performance but have the advantage of lower computational costs. Having no network schedule constraints results in having shorter node schedules and potentially a better performance, nevertheless the need for network schedules is motivated by the need to allocate resources between all the users of a quantum network.

Besides its quantitative results, this work provides valuable insights into translating scheduling problems onto the quantum domain. We establish the limits of program scheduling due to the asynchronous nature of classical communication, and clarify some subtleties of timing considerations within the context of network scheduling. We furthermore provide a method of randomly generating network schedules which can be a useful baseline for the on-going research into network scheduling.

This work delivers novel contributions to the area of scheduling quantum applications by contributing to the definition of an architecture for running applications on quantum networks and establishing the first comprehensive implementation of program scheduling. By making the proposed solution open-source and emphasising the possible future research directions, we pave the way for future investigations into quantum program scheduling.

References

[1] Stephanie Wehner, David Elkouss, and Ronald Hanson. "Quantum internet: A vision for the road ahead". en. In: *Science* 362.6412 (Oct. 2018), eaam9288. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aam9288.

[2] H. J. Kimble. "The quantum internet". en. In: *Nature* 453.7198 (June 2008), pp. 1023–1030. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature07127.

[3] Thomas Beauchamp and Hana Jirovská. "An Architecture for Running Applications on Quantum Networks. In preparation". 2023.

[4] Peter Brucker et al. "Resource-constrained project scheduling: Notation, classification, models, and methods". en. In: *European Journal of Operational Research* 112.1 (Jan. 1999), pp. 3–41. ISSN: 03772217. DOI: 10.1016/S0377-2217(98)00204-5.

[5] Sönke Hartmann and Dirk Briskorn. "A survey of variants and extensions of the resource-constrained project scheduling problem". en. In: *European Journal of Operational Research* 207.1 (Nov. 2010), pp. 1–14. ISSN: 03772217. DOI: 10.1016/j.ejor.2009.11.005.

[6] Peter Brucker and Sigrid Knust. *Complex Scheduling*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-23928-1. DOI: 10.1007/978-3-642-23929-8. URL: https://link.springer.com/10.1007/978-3-642-23929-8.

[7] Matthew Skrzypczyk. "Dynamic Time-Division Multiple Access in Noisy Intermediate-Scale Quantum Device Networks". EN. Master. TU Delft, 2020.

[8] Matthew Skrzypczyk and Stephanie Wehner. *An Architecture for Meeting Quality-of-Service Requirements in Multi-User Quantum Networks*. arXiv:2111.13124 [quant-ph]. Nov. 2021. URL: http://arxiv.org/abs/2111.13124 (visited on 11/07/2022).

[9] Bob Dorland. "Instruction Scheduling for Blind Quantum Computing". EN. Master. TU Delft, 2023.

[10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th anniversary ed. Cambridge; New York: Cambridge University Press, 2010. ISBN: 978-1-107-00217-3.

[11] Charles H. Bennett and Gilles Brassard. "Quantum cryptography: Public key distribution and coin tossing". In: *Theoretical Computer Science* 560 (Dec. 2014). arXiv:2003.06557 [quant-ph], pp. 7–11. ISSN: 03043975. DOI: 10.1016/j.tcs.2014.05.025.

[12] Artur K. Ekert. "Quantum cryptography based on Bell's theorem". en. In: *Physical Review Letters* 67.6 (Aug. 1991), pp. 661–663. ISSN: 0031-9007. DOI: 10.1103/PhysRevLett.67.661.

[13] Dominik Leichtle et al. "Verifying BQP Computations on Noisy Devices with Minimal Overhead". In: arXiv:2109.04042 (Sept. 2021). arXiv:2109.04042 [quant-ph]. URL: http://arxiv.org/abs/2109.04042.

[14] Anne Broadbent, Joseph Fitzsimons, and Elham Kashefi. "Universal Blind Quantum Computation". In: *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. Atlanta, GA, USA: IEEE, Oct. 2009, pp. 517–526. ISBN: 978-1-4244-5116-6. DOI: 10.1109/FOCS.2009.36. URL: http://ieeexplore.ieee.org/document/5438603/.

[15] Charles H Bennett et al. "Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels". In: *Physical review letters* 70.13 (1993), p. 1895.

[16] Dik Bouwmeester et al. "Experimental quantum teleportation". en. In: *Nature* 390.6660 (Dec. 1997), pp. 575–579. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/37539.

[17] Victoria Lipinska et al. "Certification of a functionality in a quantum network stage". In: *Quantum Science and Technology* 5.3 (July 2020), p. 035008. ISSN: 2058-9565. DOI: 10.1088/2058-9565/ab8c51.

[18] Aristide Mingozzi et al. "An Exact Algorithm for the Resource-Constrained Project Scheduling Problem Based on a New Mathematical Formulation". en. In: *Management Science* 44.5 (May 1998), pp. 714–729. ISSN: 0025-1909, 1526-5501. DOI: 10.1287/mnsc.44.5.714.

[19] E. F. Codd. "Multiprogram scheduling: parts 1 and 2. introduction and theory". en. In: *Communications of the ACM* 3.6 (June 1960), pp. 347–350. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367297.367317.

[20] Klaus Neumann and Jürgen Zimmermann. "Methods for Resource-Constrained Project Scheduling with Regular and Nonregular Objective Functions and Schedule-Dependent Time Windows". en. In: *Project Scheduling.* Ed. by Jan Weglarz. Vol. 14. International Series in Operations Research Management Science. Boston, MA: Springer US, 1999, pp. 261–287. ISBN: 978-1-4613-7529-6. DOI: 10.1007/978-1-4615-5533-9_12. URL: http://link.springer.com/10.1007/978-1-4615-5533-9_12.

[21] *Resource-constrained project scheduling: models, algorithms, extensions and applications.* eng. Control systems, robotics and manufacturing series. London: ISTE [u.a.], 2008. ISBN: 978-1-84821-034-9.

[22] Willy Herroelen and Roel Leus. "Project scheduling under uncertainty: Survey and research potentials". en. In: *European Journal of Operational Research* 165.2 (Sept. 2005), pp. 289–306. ISSN: 03772217. DOI: 10.1016/j.ejor.2004.04.002.

[23] Andrew J Davenport, Christophe Gefflot, J Christopher Beck, et al. "Slack-based techniques for robust schedules". In: *Proceedings of the Sixth European Conference on Planning (ECP-2001).* 2001, pp. 7–18.

[24] Matteo Pompili et al. "Experimental demonstration of entanglement delivery using a quantum network stack". In: *npj Quantum Information* 8.1 (Oct. 2022). arXiv:2111.11332 [quant-ph], p. 121. ISSN: 2056-6387. DOI: 10.1038/s41534-022-00631-2.

[25] Matthew Skrzypczyk. *An Architecture for Meeting Quality-of-Service Requirements in Multi-User Quantum Networks.* Version 1. Nov. 2021. URL: https://github.com/mdskrzypczyk/LinkScheduling.

[26] *MiniZinc.* URL: https://www.minizinc.org (visited on 01/24/2023).

[27] Gilles Audemard, Christophe Lecoutre, and Nicolas Szczepanski. *PyCSP3.* Version 2.1. 2022. URL: https://github.com/xcsp3team/pycsp3.

[28] Tim Coopmans et al. "NetSquid, a NETwork Simulator for QUantum Information using Discrete events". en. In: *Communications Physics* 4.1 (July 2021), p. 164. ISSN: 2399-3650. DOI: 10.1038/s42005-021-00647-8.

[29] QuTech. *SquidASM.* Version 0.11.0. 2023. URL: https://github.com/QuTech-Delft/squidasm.

[30] Stephen DiAdamo et al. "QuNetSim: A Software Framework for Quantum Networks". In: *IEEE Transactions on Quantum Engineering* (2021). DOI: 10.1109/TQE.2021.3092395.

[31] Bart van der Vecht. *Qoala Simulator.* Version 0.2.2. 2023. URL: https://pypi.org/project/qoala/.

[32] Hana Jirovská. *Program Scheduling.* June 2023. URL: https://github.com/hjir/program-scheduling.

[33] Maximilian Ruf et al. “Quantum networks based on color centers in diamond”. en. In: *Journal of Applied Physics* 130.7 (Aug. 2021), p. 070901. ISSN: 0021-8979, 1089-7550. DOI: 10.1063/5.0056534.

[34] Christophe Lecoutre and Nicolas Szczepanski. *PyCSP3: Modeling Combinatorial Constrained Problems in Python.* 2022. arXiv: 2009.00326.

[35] Christophe Lecoutre. *ACE 2.0: A generic constraint solver.* Version 2.1. 2022. URL: https://github.com/xcsp3team/ace.

[36] Frédéric Boussemart et al. “Boosting systematic search by weighting constraints”. In: *ECAI.* Vol. 16. 2004, p. 146. ISBN: 9781586034528.

[37] Christophe Lecoutre and Charles Prud’homme. *Piloting PyCSP^3 Solvers with General Options.* 2022. URL: https://github.com/xcsp3team/pycsp3/blob/master/docs/optionsSolvers.pdf.

[38] Ingmar te Raa-Derckx. Personal Communication. May 2023.

[39] Guus Avis et al. “Requirements for a processing-node quantum repeater on a real-world fiber grid”. In: (2022). DOI: 10.48550/ARXIV.2207.10579. URL: https://arxiv.org/abs/2207.10579.

[40] Anabel Ovide González. Personal Communication. June 2023.

[41] E Douglas Jensen, C Douglass Locke, and Hideyuki Tokuda. “A time-driven scheduling model for real-time operating systems.” In: *Proc. IEEE Real-Time Systems Symp.* Vol. 85. 1985, pp. 112–122.

[42] Peng Li et al. “A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints”. In: *IEEE Transactions on Computers* 55.4 (2006), pp. 454–469.

[43] Laurent George, Paul Mühlethaler, and Nicolas Rivierre. “Optimality and non-preemptive real-time scheduling revisited”. PhD thesis. INRIA, 1995.