Evaluating Multi-Metric Scheduler Performance in a Quantum Hub Network with NetSquid

A Method for Selecting a Scheduler in a Centralized Quantum Network

Master Thesis Project

Christiaan Korbee

Delft University of Technology

TU Delft

Evaluating Multi-Metric Scheduler Performance in a Quantum Hub Network with NetSquid

A Method for Selecting a Scheduler in a Centralized Quantum Network

by

Christiaan Korbee

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Tuesday May 6, 2025 at 9:00 AM.

Student number:	5307449	
Project duration:	September 2, 2024 – April 22, 2025	
Thesis committee:	Prof. dr. S. D. C. Wehner,	TU Delft, supervisor
	Prof. dr. A. F. Otte,	TU Delft
	Dr. T. J. Coopmans,	TU Delft
Daily supervisors:	S. S. Gauthier,	TU Delft
	M. K. van Hooft,	TU Delft

Cover: What's the Difference Between the Web and the Internet? at `https://askleo.com/`.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

As technology for implementing quantum networks advances, the challenge of efficiently managing entanglement generation under resource constraints becomes critical. In this thesis, a systematic methodology for selecting scheduling strategies in a centralised quantum network is proposed, and applied to a specific case for a quantum hub network utilising an entanglement generation switch. Using the NetSquid simulator, the first in, first out, on-demand, MaxWeight, earliest deadline first, earliest feasible deadline first and round-robin scheduling algorithms are evaluated across a wide range of metrics compiled in this thesis.

These metrics include measures of throughput, fairness, responsiveness, demand completion, and resource utilisation, among others. Varying network load conditions are simulated and two application-level use cases, quantum key distribution and blind quantum computing, are considered. The results offer detailed insight into how each scheduler performs under different demand patterns and operational contexts.

Based on these results, the earliest deadline first performs better in most metrics than the other schedulers within the context of this thesis. Additionally, the results indicate that classical optimality of a scheduler does not always translate to superior performance in quantum network scenarios. The findings presented and the framework described here can provide practical guidance for network operators seeking to balance multiple performance goals in near-term quantum networks.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
AATF	Average active throuhgput per flow
ART	Average response time
AUF	Average utilisation factor
BQC	Blind quantum computing
BT	Busy time
EDF	Earliest deadline first
EDS	Entanglement distribution switch
EFDF	Earliest feasible deadline first
EGS	Entanglement generation switch
FIFO	First in, first out
MW	MaxWeight
OD	On-demand
PGA	Packet generation attempt
PCD	Proportion of completed demands
PCJ	Proportion of completed jobs
PGT	Packet generation task
QKD	Quantum key distribution
RFI	Resource fairness index
RNG	Random number generation
RR	Round-robin
SEM	Standard error of the mean
STD	Standard deviation

Symbols

Symbol	Definition	Unit
$C_l(t_1, t_2)$	Number of jobs completed in (t_1, t_2] in flow f_l	[a.u.]
$C_j^\dagger(t_1, t_2)$	Number of jobs from demand d_j completed in (t_1, t_2]	[a.u.]
$\mathcal{D}$	Total number of demands submitted to the hub	[a.u.]
D_l	Set of (indices of) demands submitted in flow f_l	[a.u.]
$E_{j,p}$	Execution time of job $J_{j,p}$	[s]
$E_{j,p}^\dagger$	Estimate of execution time of job $J_{j,p}$	[s]
$\mathcal{F}$	Total number of flows in the network	[a.u.]
F_k	Set of (indices of) flows that connect to node n_k	[a.u.]
$F_{\text{min},j}$	Minimum fidelity requested by demand d_j	[a.u.]
I_d	Idleness factor	[a.u.]
$J_{j,p}$	p-th job from demand d_j	[a.u.]
$\mathcal{J}_R$	Resource fairness index	[a.u.]
$\mathcal{J}_X$	Throughput fairness index	[a.u.]
$M(t_i)$	Schedule for t_i	[a.u.]
$\overline{M}(t_i)$	Incoming schedule for time t_i	[a.u.]
$M^*(t_i)$	Residual schedule for time t_i	[a.u.]
$M_l(t_i)$	Number of jobs from flow f_l to be executed at t_i	[a.u.]

Symbol	Definition	Unit
$\mathcal{N}$	Total number of user nodes in the network	[a.u.]
N_l	Set of the nodes that are linked by flow f_l	[a.u.]
$N_{\text{inst},j}$	Requested number of packets by demand d_j	[a.u.]
R	Number of resources in the hub	[a.u.]
$\overline{R}(t_i)$	Number of free resources in the hub at time t_i	[a.u.]
$R^*(t_i)$	Number of occupied resources in the hub at time t_i	[a.u.]
$S_{j,p}$	Start time of job J_{lp}	[s]
T	Total running time of the network	[s]
T_{exec}	Average execution time	[s]
$T_i^\dagger$	Wall time duration of computations for events at t_i	[s]
T_l	Active time of flow f_l	[s]
T_{resp}	Average response time	[s]
$T_{\text{resp,max}}$	Worst response time	[s]
T_{wait}	Average waiting time	[s]
U_{avg}	Average utilization factor	[a.u.]
U_{full}	Full utilization factor	[a.u.]
W	Total weight of a schedule	[a.u.]
$W_l(x)$	Weight function for flow f_l	[a.u.]
X	Throughput of the network in completed jobs per second	[s^{-1}]
$X^\dagger$	Average active throughput of the network	[s^{-1}]
X_l	Throughput of flow f_l	[s^{-1}]
$X_l^\dagger$	Active throughput of flow f_l	[s^{-1}]
c_k	Number of communication qubits in node n_k	[a.u.]
$\overline{c_k}(t_i)$	Number of free qubits in node n_k at time t_i	[a.u.]
$c_k^*(t_i)$	Number of occupied qubits in node n_k at time t_i	[a.u.]
d_j	j-th demand in the network	[a.u.]
f_l	l-th flow in the network	[a.u.]
l_j	Index of flow f_{l_j} which submitted demand d_j	[a.u.]
n_k	k-th node in the network	[a.u.]
$q_l(t_i)$	Queue length of flow f_l at time t_i	[a.u.]
$r(t_i)$	Number of resources active at time step t_i	[a.u.]
r_l	Average number of resources allocated to flow f_l	[a.u.]
s_j	Requested number of links by demand d_j	[a.u.]
t_0	First time step at which something may happen in the network	[s]
$t_{\text{expiry},j}$	Expiry time of demand d_j	[s]
t_{expiry}	Expiry time of a packet generation task	[s]
t_i	Discrete time step i at which an event may happen	[s]
t_L	Last (relevant) time step in the network	[s]
$t_{\text{minsep},j}$	Minimum separation time of demand d_j	[s]
t_{minsep}	Minimum separation time of a packet generation task	[s]
$t_{\text{total}}^\dagger$	Computational wall time	[s]
w_j	Window to generate the links from demand d_j	[s]
Γ_j	Set of completed jobs from demand d_j	[a.u.]
Δ_j	Set of unfinished jobs from demand d_j	[a.u.]
Δt	Difference in submission times between demands	[s]
Λ_j	Set of rejected jobs from demand d_j	[a.u.]
Φ	Proportion of completed demands	[a.u.]
Ψ	Proportion of failed demands	[a.u.]
α	Job completion ratio	[a.u.]
β	Job acceptance ratio	[a.u.]
γ_j	Total number of completed jobs from demand d_j	[a.u.]

Symbol	Definition	Unit
δ_j	Total number of unfinished jobs from demand d_j	[a.u]
ζ	Unfinished job ratio	[a.u]
η	Job rejection ratio	[a.u.]
$\theta(x)$	Unit step function	[a.u.]
λ	Total number of rejected jobs from demand d_j	[a.u.]
μ_j	Requested average rate by demand d_j	[s^{-1}]
$\nu_j(t_i)$	Number that denotes whether demand d_j has finished by time t_i	[a.u.]
ξ	Job failure ratio	[a.u.]
σ_j	Submit time of demand d_j	[a.u.]
τ_j	End time of demand d_j, either by expiration or completion	[a.u.]
$\mathbb{N}_0$	Non-negative integers, including zero	[a.u.]

1 Introduction

For the past decades, much research has gone into realizing a quantum internet [1, 2]. A quantum internet enables applications which are fundamentally not possible with classical computer networks alone, such as quantum key distribution [3, 4, 5], blind quantum computation [6, 7, 8] and many others [2]. As technology advances, quantum connections have been shown to be possible on metropolitan scales [9, 10, 11], which brings us one step closer to achieving a quantum internet bridging long distances. This can be interesting to many potential users, because the possibility of secure, secret communication with other parties sets a quantum internet apart from its classical counterpart. To this end, entangled links between the user nodes are required, as remote entanglement of quantum states is the foundation of long-distance quantum applications [2, 6, 3]. To obtain the entangled links required for the application they want to run on their local nodes, these physically separated users must demand entangled links from the network. The demand for entanglement submitted by users reflects the requirements of the application that they wish to execute. For example, the demand may be for multiple entangled pairs delivered within a finite window of time. However, many different users may make these demands at the same time, meaning that in a network with a limited number of resources and independent paths connecting end nodes, it may not be possible for the network to provide service to all demands at once. The problem of how to best allocate service to the various demands is a scheduling problem. It may be resolved by the creation of a network schedule, indicating when the network will attempt to serve each demand. To make sure the maximum number of users are satisfied in their demands, it is therefore important to make sure that such a network can service all its users to its best ability under any scenario.

One way to implement a quantum network is by using a quantum hub network, in this report also referred to as a metropolitan hub. The specific network topology is that of a “hub and spoke network” or “star network”, which is a network where the only connections are from user nodes directly to the central hub itself. In a central hub network, the hub allocates resources for entanglement generation, which allows a quantum connection to be established between two nodes connected to the hub. In this report, we use an “entanglement generation switch” as the hub [12], and it is a “generate-when-requested” network [13]. The central hub has to make decisions regarding the scheduling of the incoming requests. There are many different ways to do make this decision, and many different reasons to consider one way over the other. This is known as the “network scheduling problem” [13], where the goal is to decide how to assign limited network resources to user pairs for the generation of entangled links. Scheduling algorithms allow for a systematic way to choose which request should be considered next by the central hub. These scheduling algorithms generally optimize a certain metric, to allow a network owner to choose a scheduler[1] based on whatever it is they may prioritize. In reality however, a network owner may value multiple different metrics and want a scheduler that performs well on all of these. Most recent studies on schedulers in quantum networks have focused on their performance with respect to mostly just the optimized metric and a few related metrics however [14, 15, 16, 17], so a comparison

[1] While a scheduler and scheduling algorithm can be understood to be different concepts, in this report they will be used interchangeably.

between different schedulers is difficult to do in a fair manner that considers all these aspects.

In this thesis, a method for selecting a scheduler for centralised quantum networks is introduced, which is described generally in Algorithm 1.

Algorithm 1 Scheduler Selection Process for Centralised Quantum Network

Step 1: Choose schedulers for consideration
Step 2: Choose metrics for consideration
Step 3: Set up scheduling problem to represent the quantum network
Step 4: Simulate/compute results
Step 5: Use results to select scheduler as desired for the network

The selection process for a certain quantum network consists of 5 main steps. The first is to make a choice of which schedulers will be considered in the first place. This should take into account schedulers being studied in relevant literature, as well as schedulers that perform well in classical systems analogous to the quantum system considered here. The second step is choosing and defining which metrics will be looked at. While important metrics will be based on the purpose of the network, i.e. a network which has to give fair access to its users should have fairness metrics to measure some sense of fairness, this should also include a broader spectrum of metrics based on relevant literature and classical analogues. This is because it is important to understand how well each scheduler performs outside of just the main metric to make a well-informed decision. After this, a problem case has to be set up to represent the quantum network in consideration. This could either be a test set of demands as input for each scheduling algorithm, or a complete simulation of a small/large scale network. Then, the results of the problem or simulation have to be computed. These can then be compared in the last step, so a network operator has a better understanding of the difference between the schedulers in consideration based on the chosen metrics. This should then allow them to make a selection of scheduler that best fulfills their expectations and desires.

This selection process will be tested in this thesis by simulating multiple schedulers in a quantum hub network using NetSquid [18]. The results of these simulations will then be used to anwser the following questions:

1. Which scheduler performs best regarding the successful completion of demands overall?
2. Which scheduler performs best in demand completion based on different network loads?
3. Which scheduler performs best in demand completion based on different use cases for the network?
4. Does the classical optimality for schedulers transfer into better performance than other schedulers in a quantum network? Specifically: Does the MaxWeight scheduler, that is classically considered throughput optimal, also perform better than the other schedulers in throughput?
5. Can an earliest deadline first algorithm reach better performance as an earliest feasible deadline first scheduler by adding a form of admission control?
6. When using a specific way of demand formatting and processing [13], is online or off-line scheduling preferable?
7. Can network loads be emulated in different ways?
8. Can networks that serve multiple use cases be approximated by only considering networks that each serve a single one of these use cases?

In the first section of this thesis, the necessary background information along with the chosen schedulers for this thesis will be explained in Chapter 2. Then, the metrics that will be used in this thesis will be introduced, as well as how the simulation was implemented in Chapter 3. Afterwards, the results of the simulations will be discussed in Chapter 4 and finally summarised in Chapter 5.

2

Theory

In this section, the necessary background information will be explained. First, a small introduction will be given to relevant quantum mechanics, as it is important to understand what makes a quantum network fundamentally different from a classical one. Then, relevant use cases that will be considered in this thesis will be explained, specifically quantum key distribution and blind quantum computing. After that, there will be an explanation of the architecture of the network used in this thesis, so it is clear what type of network the results apply to. Then, a way of demand formatting introduced by Beauchamp et al. [13] will be introduced, as it is used in the processing of applications to individual jobs in this thesis. Lastly, the schedulers that are used in this thesis will be given.

2.1. Qubits

A quantum internet is not a simple upgrade to the classical internet. Instead, its purpose is to allow users to perform specific applications in a way that is impossible on the classical internet [2], often by providing security that does not rely on computational assumptions [3, 4, 5, 6, 7, 8] or by exploiting long distance correlations between two nodes [19, 20, 21, 22, 23, 24, 25]. The fundamental reason why a quantum internet differs so much from the classical internet is because of the units of information in each case. In the classical internet, information is transferred using classical bits, objects that can be either 0 or 1. In a quantum internet, qubits are the units of transferring quantum information. These qubits have three main properties that fundamentally change them from their classical counterparts, when considering their use in a quantum internet [2].

2.1.1. Quantum superposition

Qubits are two state quantum systems, meaning that the corresponding quantum system has two available and orthogonal states for the qubit to be in. These are represented in Dirac bra-ket notation as $|0\rangle$ and $|1\rangle$ [26]. Unlike classical bits, which can only be in either the 0 or 1 state, qubits can be in a superposition of $|0\rangle$ and $|1\rangle$. What this means is that a qubit can be in any state $|\psi\rangle$ of the form $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α, β are complex numbers with the constraint that $|\alpha|^2 + |\beta|^2 = 1$, so that the state is normalized. The Born rule states that a quantum state measured in a specific basis will be projected upon one of the basis states with a probability dependent on the coefficients in the superposition [26]. Specifically, the probability to get outcome 0 when measuring $|\psi\rangle$ is $|\alpha|^2$ and likewise, the probability of 1 is $|\beta|^2$. Similarly, choosing a basis such that $|\psi\rangle$ is one of the basis states yields a definite result, since the corresponding coefficient of that state is 1. This has the important implication that measurement results of a superposition state in an arbitrary basis are fundamentally random, whereas measuring a state in the basis in which it is a definite state will yield a definite result. While this may seem like using superpositions would be inconvenient regarding the transfer of information, it has important implications for using qubits in real applications, as the superposition of qubits is what allows entanglement and quantum parallelism to be leveraged in quantum networking and computing [26]. For example, because quantum superpositions are linear combinations of the basis states, it means that any linear operation f applied on $|\psi\rangle$ results in $\alpha f(|0\rangle) + \beta f(|1\rangle)$, meaning both results are calculated at the same time while

the operation was only applied once.

2.1.2. Quantum entanglement

Entanglement is an important concept for quantum applications, and understanding how it is generated will explain why it is so useful. Let's say that Alice has two qubits, one in the superposition state $|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and another in the basis state $|\psi_2\rangle = |0\rangle$ and wants to form an entangled pair with them. Their combined state can be written as $|\psi_1\psi_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$. To entangle them, she can use a C-NOT gate with $|\psi_1\rangle$ as the control and $|\psi_2\rangle$ as the target. This will perform the following operation [26]: if the control is 0, nothing happens to the target, but if the control is 1, the target will be flipped. Since the C-NOT gate is a linear operation, it can be applied to each basis state of the superposition individually, resulting in the final state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, often called the $|\Phi^+\rangle$ Bell state. Now, the qubits are said to be entangled, meaning that you cannot describe the quantum state of either of the individual qubits without considering the other anymore. Furthermore, many entangled states, such as the Bell states, are maximally correlated, meaning that measuring one of the qubits will give you all the information of the other qubit as well. Among other uses [3, 4, 5, 6, 7, 8], entanglement can be used to do quantum teleportation, transferring a quantum state from one qubit to another without physically moving them [27], which is an important concept for quantum networks, and superdense coding, where two classical bits are encoded into a single qubit which is entangled with another. This allows the transfer of two bits of information by sending just one qubit [28].

2.1.3. No-cloning theorem

An important theorem derived regarding quantum communication and computation is the no-cloning theorem [29, 30], which states that an arbitrary quantum state cannot be copied. If it is known in which basis a quantum state is encoded, it is possible to copy it by measuring it in that basis. This has two important implications, especially for quantum communication. First of all, it allows the secure communication of quantum information between two people. Since the qubit that is being sent cannot be copied, an eavesdropper can only attempt to gain this information by measuring it, which alters the state if the basis is not known. This concept is used in many protocols, such as the BB84 and E91 protocols [3, 4] to allow for secure communication. Another implication is that quantum signals cannot be amplified, so to combat decoherence and signal loss a different type of repeater, called a quantum repeater, is needed to transfer quantum information over long distances [31]. So while the no-cloning theorem makes it more difficult to send quantum information successfully over long distances, it also allows a specific type of eavesdropping detection that secure communication protocols are based on.

2.2. Use cases for quantum internet

There are many different types of use cases for a fully developed quantum internet, all with their own minimum functionality requirements for the quantum network [2]. Examples of these uses include, but are not limited to, secure leader election in anonymous networks [23], clock synchronization [19], distributed quantum computing [32, 33], quantum sensing [20], extending the baselines of telescopes [21], fast coordination of decisions by remote parties without any real-time exchange of information [24, 25] and quantum fingerprinting [22]. Two use cases that will be looked at more closely in this thesis are quantum key distribution (QKD) [3, 4, 5] and blind quantum computing (BQC) [6, 7, 8].

2.2.1. Quantum key distribution

Quantum key distribution is the process of generating key for encryption purposes by utilizing quantum mechanics. Generally, it is a type of Measure Directly application, where many entangled pairs are generated and measured immediately after being generated, requiring no quantum memory [34]. The earliest protocol for such a process is the BB84 protocol [3] by Bennett and Brassard. This protocol uses superpositions and the no-cloning theorem to allow two persons to generate key in a secure manner. Here, secure means that an eavesdropper cannot try to intercept and read the quantum string of qubits without being detected.

Alice and Bob want to generate key for encryption, while a possible eavesdropper Eve may be eavesdropping on both the quantum and classical channels. The protocol they can use, ignoring other sources of noise and interference other than eavesdropping, as described by Bennett and Brassard [3]

consists of the following steps:

1. Alice randomly chooses a sequence of random bits and a sequence of random bases of the same length. The basis of the qubit can be either the Hadamard (X) or Computional (Z) basis. She sends the resulting string of qubits to Bob, while keeping the original string of bits and bases to herself.
2. Bob measures each incoming qubit in a random basis and writes down the result and the basis used.
3. Alice and Bob communicate classically which parts of the qubit string were successfully received and measured in the same basis as it was generated. They discard all other qubits. Because these qubits are measured in the same basis, Alice and Bob expect them to have a perfect match for these measurements if they were transferred without outside interference.
4. Alice and Bob test for eavesdropping by taking a subset of the string they are left with and comparing their results for that subset. If the results match perfectly, they can conclude there was no significant eavesdropping and can use it for encryption.

They can detect eavesdropping by Eve in this way, because if Eve tried to eavesdrop on the quantum channel, she would also have to guess the basis randomly. 1/2 of the time, she would guess the basis wrong, meaning that she alters these qubits by projecting them on a new basis. Afterwards, if Bob measures them in the original basis again, they will get projected onto a new basis again, where they have a 1/2 chance of being projected back to the original state. This means that, if Eve attempts to eavesdrop on the entire quantum string, 1/4 of the results Bob compares with Alice will be wrong [3]. For this reason, a QKD protocol is usually specified together with a maximum tolerable error rate, based on what is detected in the testing phase. If the detected error is higher, the protocol is aborted.

2.2.2. Blind quantum computing

The goal of blind quantum computing is to allow a person without access to a set of universal quantum gates to be assisted by another person who does have access to such a set, without that person being able to find out what the inputs, outputs or function being computed are [6, 7, 8]. This type of protocol is also interesting for a user who only has access to a small number of quantum registers or qubits, but wishes to complete an application that requires many quantum registers. It is a type of Create and Keep application, where multiple entangled pairs may be required at once, and joint operations may be performed on them [34]. One of the protocols to do blind quantum computing is described by A.M. Childs [8]. There, he describes how Alice, who does not have access to a set of universal quantum gates or qubit measurements, can be assisted by Bob, who does. This way, Alice can now perform both measurements and gate operations securely. The general idea behind each specific protocol is the encoding and decoding of the qubits Alice generates before sending them to Bob. If she wants to send a qubit to Bob, she first generates a random key kj, and applies the gates $Z^k X^j$ to the qubit. The density matrix[1] of the qubit Bob receives is maximally mixed, so he cannot learn anything from this qubit [8]. After receiving the qubit back from Bob, Alice has to decrypt the qubit. This process of decryption depends on both the key kj she used for encryption and the gates performed by Bob. To increase secrecy further, Alice can also ask Bob to perform each available gate on repeat one after the other for a certain number of times. Now, she sends junk qubits to Bob if that gate is not needed and the real qubit if it is, ensuring Bob cannot even find out the function being performed, despite being the one who is applying the gates in the first place [8]. This still does not prevent Bob from sabotaging the process, he could simply use a different gate than the one asked. For this, the A.M. Childs offers two solutions [8]. If finding a solution is difficult, but verifying the answer is not, i.e. an NP problem, then Bob's honesty can be verified by simply checking the result. If this is not possible, the computation of real questions should be interleaved with small tests of which the answer is already known. This allows

[1] A density matrix allows you to represent mixed ensembles of quantum states in single matrix [26] and can be used to represent missing information about an incoming quantum state. For example, the superposition state $\frac{1}{\sqrt{2}}(|0\rangle + \beta|1\rangle)$ is still a pure state, even though the measurement may yield both 0 or 1 with equal probability if measured in the computational basis. A mixed state would be the case when you know the incoming qubit is either with a 50% chance generated in the $|0\rangle$ state or with a 50% chance generated in the $|1\rangle$ state. Despite measurement also resulting in either 0 or 1 with equal probability, this is not the same, which can be easily shown by measuring in the Hadamard basis instead. Doing so in the first superposition case will always result in the the $|+\rangle$ being measured, whereas in the second case will still result in either 0 ($|+\rangle$) or 1 ($|-\rangle$) with equal probability.

Alice to be sure with a probability below some threshold that Bob is not lying. This way, Alice can still perform complex quantum computing problems without having access to a state-of-the-art quantum computer herself.

2.3. Quantum hub networks

The quantum networks that are of interest for the establishment of quantum internet can be implemented in very different structures, each with its own (dis-)advantages. A simple and naive implementation would be to connect each user directly, but this comes with the drawback that the number of required links between the users scales with $\binom{N}{k}$, where N is the number of users/nodes. This becomes very expensive for large N, so an alternative implementation is desirable. One of these is the use of a quantum hub, which controls the network from one single special node. Here, all nodes are now connected to this hub, which in turn connects the nodes dependent on both what is required at that moment and which algorithm is being used by the hub. In this scheme, the number of connections instead scales with N, which is much more favourable, at the cost of operating a quantum hub. Two types of quantum hubs are the entanglement generation switch (EGS) [12] and the entanglement distribution switch (EDS) [35]. While they are similar, the main difference between them is that an EDS has access to qubits and/or quantum memory, whereas the EGS does not [12]. This thesis will focus on the EGS implementation of a hub. Another distinction to be made is whether a network is pre-loaded network or generate-when-requested network [13]. In a pre-loaded network, entanglement is repeatedly created and buffered at nodes, so that it can be distributed upon request, whereas a generate-when-requested network will only start attempting entanglement generation once a flow requests it [13]. For this thesis, a generate-when-requested network will be considered. This is relevant for near term networks, as, for example, NV centers have currently shown lifetimes possible up to the order of 1 s [36], which is still relatively small compared to classical systems, so buffering may not be the most effective strategy there.

2.3.1. Entanglement generation switch components

The EGS is a quantum hub usually equipped only with some number of resources and no memory [12]. Three possible protocols to create entanglement using a network hub are detection-in-midpoint, sender-receiver and source-in-midpoint protocols [37]. Detection-in-midpoint describes the situation where the outer nodes send photons to be detected by Bell state analysers to do an entanglement swap [38]. Here, the resources are Bell state analysers. In the source-in-midpoint protocol the photons are generated at the hub, entangled and then sent to the two nodes that want entanglement. The nodes then entangle the incoming photons with their communication qubit and measure the photon to end up with a state of entangled communication qubits. Here the resources are bell state generators. The sender-receiver protocol does not have a hub that actively interacts with the photons, but instead it simply routes incoming photons from the sender node to the receiver node. The results of this thesis will primarily relevant for the first two protocols, detection-in-midpoint and source-in-midpoint [37]. An EGS has 3 main components:

1. The aforementioned pool of resources, which allows the EGS to create entanglement between connected nodes.
2. The switch that allows the EGS to connect 2 specific nodes to a resource for entanglement generation.
3. The classical processor, which is capable of scheduling resource allocation, controlling the switch and sending/receiving messages.

A schematic diagram has been added as Figure 2.1 to illustrate how these components are connected to each other.

2.4. Demand format

Different applications will send different requests to the hub and these should all be handled correctly. To streamline this, a demand format can be used to standardize all incoming requests. A demand format was proposed by Beauchamp et al. [13], which can format a demand for any arbitrary quantum network application.

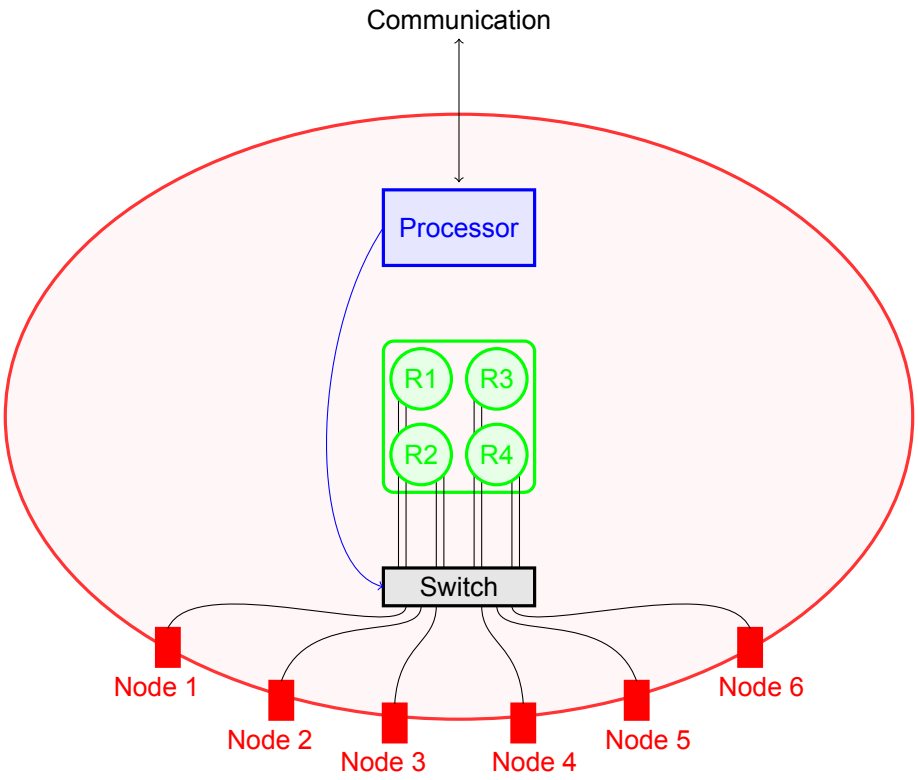

Figure 2.1: A schematic diagram of a quantum hub. The red shaded area represents the quantum hub node, and everything inside it is a part of the hub. There are two ways for information to enter and leave this node. There can be classical channel for classical communication connected to the processor, which can be connected to the other nodes [12] here represented by a two-sided arrow between the inside and the outside, as the processor may both receive and send information over the classical channel. The quantum channels, whose connections to the hub are represented by the red rectangles, allow photons to be transferred between the hub and the nodes. The quantum channels are directed to a switch, which connects the nodes with the resources. The switch is controlled by the central processor, which uses a scheduler to decide which nodes can make use of the resources now. The green circles surrounded by the green border represent the resources - in this diagram four, but there may be any number of them - which are used for the creation of entanglement between two nodes. They may be either Bell state analyzers, Bell state generators, or simply a connection that routes the photon from the sender to receiver node, in accordance with three possible protocols to generate remote entanglement [37].

To translate an application to a demand, the authors of [13] make use of a packet of entanglement, which is a tuple that contains information about how the entanglement should be generated. Specifically, it contains s, the number of requested links/entangled pairs to generate, w, the window within which these links have to be generated and F_{min}, the minimum fidelity for each of the links. The packet of entanglement then becomes the tuple (w, s, F_{min}) [13]. This does not capture all of the information however, as this only specifies the details of a single packet of entanglement, without specifying how many the application wants or when they are needed. A full demand therefore also contains the following information: the average rate r at which packets of entanglement should be produced, the number of packets of entanglements they want to generate N_{inst}, the expiry time of the demand t_{expiry} and the minimum separation time between two packets of entanglement t_{minsep}. The minimum separation time can be useful for applications that need to first operate on the generated entangled pairs before a new packet can be processed, so a minimum time break can be given to prevent the pairs from being generated before they can be used. Together, these form the full demand $\mathcal{D} = (\{w, s, F_{\text{min}}, r\}; t_{\text{minsep}}, t_{\text{expiry}}; N_{\text{inst}})$ [13]. This, in principle, provides a clear way of translating an application to a format that can be communicated between the nodes, though it also makes explicit an assumption for this thesis: demands, and thus applications, are only defined for a single flow. This makes sense in practice, because you would not want to switch with which other node you are executing an application during the execution itself. This does not prevent multi-node applications in the original definition [13], but they will not be considered in this thesis. In their definition, the authors also included an option for a demand to have multiple suitable packets of entanglement that would fit in a demand, which is the reason why the rate

is included within the packet of entanglement in the demand. However, as this notion of suitability of multiple types of packets is not used in this thesis, it won't be discussed further.

To adapt the definition by Beauchamp et al. [13] to this thesis, a couple of adjustments will be made. First of all, it makes sense within the context of this thesis to not just talk about a demand format $\mathcal{D}$, but individual demands d_j that are the j-th demand generated for submission to the hub. Furthermore, the associated flow, which represents the link between the nodes that want to execute a demand, will be included within the demand itself to make it instantly clear for which nodes this demand is relevant. To do so, a flow and a node itself must first be defined:

Definition 1 *Let n_k be the k-th node in the network and let there be $\mathcal{N}$ nodes in the network in total, excluding the hub node in case of a hub-controlled network.*

Definition 2 *Let f_l be the l-th flow in the network, which represents the link between nodes n_{k_1} and n_{k_2} such that $N_l = \{n_{k_1}, n_{k_2}\}$ denotes the set of nodes linked by flow f_l. Let $\mathcal{F}$ denote the total number of flows in the network.*

While these definitions allow us to go from a flow to its associated nodes using N_l, it is not yet entirely clear how to do so the other way around. For that, we introduce the following definition:

Definition 3 *Let F_k be the set of indices such that for each $l \in F_k$ we have that $n_k \in N_l$, and $n_k \notin N_m$ for any $m \notin F_k$.*

This definition means that F_k is a set that contains each flow that connects to node n_k.

Now, we can define the adapted demand, with each symbol changed such that it is clear where it belongs to:

Definition 4 *Let d_j be the j-th demand submitted to the hub. Let l_j be the index such that f_{l_j} is the flow from which demand d_j is submitted. Let s_j be the number of links to generate associated with demand d_j, w_j the window of time within which these links should be generated, $F_{min,j}$ the minimum fidelity for each link, μ_j the requested average rate at which the packets of entanglement should be generated, $t_{minsep,j}$ the minimum separation time between each generated packet of demand d_j, $t_{expiry,j}$ the expiry time of the demand and $N_{inst,j}$ the number of packets associated with demand d_j.*

Then, the full demand is given by $d_j = \big(f_{l_j}; \{w_j, s_j, F_{min,j}, \mu_j\}; t_{minsep,j}, t_{expiry,j}; N_{inst,j}\big)$.

To understand which demands come from which flows, we can define the following:

Definition 5 *Let D_l be the set of indices of demands that are submitted by flow f_l, meaning that for each $j \in D_l$, we have that $l_j = l$, so the flow associated with demand d_j, f_{l_j} is the same as flow f_l.*

Lastly, we have the total number of demands:

Definition 6 *Let $\mathcal{D}$ be the total number of demands submitted in the network. This is related to the total number of flows $\mathcal{F}$ by*

$$\mathcal{D} = \sum_{l=1}^{\mathcal{F}} \sum_{j \in D_j} 1 \tag{2.1}$$

Now, we have introduced a standardized format that enables clear communication of the demands between the nodes and the hub. How the demands will actually be processed in the hub will be discussed in Section 3.2.

2.5. Scheduling

Scheduling in various types of classical network systems has already been extensively researched and there are many available resources related to this [39, 40]. Scheduling in quantum systems can be inspired by classical scheduling, so many terms may be borrowed or copied from the classical case. The scheduling algorithms themselves may also be based on their classical counterparts, often with the only major difference being that non-deterministic execution times may commonly arise in the quantum case. This does make a difference, however, as derivations for optimality may not hold for schedulers if there are non-deterministic execution times. So while they will be described based on the classical

algorithms, for setting expectations it will always be kept in mind that quantum networks have their unique scheduling problems.

For the purposes of this thesis, all scheduling will be considered online, i.e. jobs may enter the system at any time and the schedule will have to be updated, as opposed to off-line, where a schedule may be computed beforehand and is then computed in completion afterward [39]. They are also dynamic, because all schedulers need to at least know if a resource is free or not before making a decision [39].

In this section, the schedulers that will be used later in the analysis will be introduced, though to do so, first the concept of schedules will be made clear.

2.5.1. Schedules

The hub in the metropolitan quantum network has to make a decision regarding which nodes get access to an entanglement generation resource at any moment. To do so systematically, scheduling algorithms are used. However, these have precise definitions, and to understand the algorithms, it is first necessary to understand which mathematical representations are being used. Specifically, the schedules themselves can be represented by many different mathematical objects. In this thesis, they are defined as follows:

Definition 7 *Let R be the number of available resources in the hub, c_k the number of communication qubits available in node n_k, N the total number of nodes connected to the hub and F_k is the set of indices of flows that connect to node n_k. A schedule $M(t_i)$ is represented by a vector with elements $M_l(t_i)$, where $M_l(t_i)$ represents the number of jobs from corresponding flow f_l scheduled for execution at time t_i. Here t_i is a discretized value where $i \in \mathbb{N}_0$, with $\mathbb{N}_0$ being the set of non-negative integers including 0, that can go from 0 to t_L, t_L being the last time step in the network. These time points are chosen such that at least for each moment the network hub has to perform an operation, there is a t_i that corresponds to that moment. To make $M(t_i)$ a schedule that is possible to be executed, the following conditions have to be met for any $M(t_i)$:*

1. $\forall i \sum_l M_l(t_i) \leq R$,
2. $\forall i \sum_{l \in F_k} M_l(t_i) \leq c_k \quad \forall k \in \{1, ..., \mathcal{N}\}$,
3. $\forall i, l \;\; M_l(t_i) \in \mathbb{N}_0$.

The conditions put constraints on the schedule to make it one that is physically possible to be executed. Specifically, the first constraint ensures no more jobs are scheduled than available resources. The second constraint ensures that no node has more jobs running than it has available communication qubits. The last constraint just specifies that resources are discrete objects and cannot be divided up. This means that any schedule M is from the space $\mathcal{M} = \{M \in \mathbb{N}_0^{\mathcal{F}} : 1. \wedge 2. \wedge 3.\}$, where 1., 2. and 3. are the conditions previously specified, which also means $\mathcal{M} \subseteq \mathbb{N}_0^{\mathcal{F}}$. Here, $\mathcal{F}$ is the total number of flows in the network. Alternatively, a matrix representation matching the flows with the resources could be used.

This definition does not capture the full situation, however, since jobs are not necessarily done within one time slot $(t_i, t_{i+1}]$ (where the time slot includes t_{i+1}, but not t_i). Instead, there may be carryover from the previous schedule. Specifically, the scheduling algorithm needs to be online, meaning that the schedule is recomputed when either a job finishes execution or possibly when a new job enters the system [39]. This is because the quantum hub network will get many new demands sent over time as new users log on, changing the parameters for the schedule computation. Schedules are already defined with respect to a specific point in time t_i, to capture the fact that they are only relevant from that specific moment. The requirement for the scheduler to be online brings a new implication: a schedule starting at t_{i+1} is computed when the result of the schedule starting at t_i is known. To do so, we first have to define jobs, which are related to the packets of entanglement. While there will generally be one job related to a specific packet of entanglement, jobs are actually instantiated in the sense that they include the start and execution times:

Definition 8 *Let $J_{j,p}$ be the p-th job from demand d_j with $p \in \mathbb{N}_0$ and with start time $S_{j,p}$ and execution time $E_{j,p}$.*

The execution time $E_{j,p}$ itself is not known by the schedulers before a job has finished due to the nature

of quantum mechanics, but schedulers do have access to an estimate of that time, $E^{\dagger}_{j,p}$. Now, a residual schedule can be defined:

Definition 9 *Let $M(t_{i-1})$ be the schedule for the previous time step t_{i-1} and Γ_j the set of indices such that for every $p \in \Gamma_j$, $J_{j,p}$ is a job that was successfully completed before their deadline and spawned from demand d_j and that this is not true for any $q \notin \Gamma_j$. Then, the components of the residual schedule vector $M^*(t_i)$ are defined as:*

$$M^*_l(t_i) = M_l(t_{i-1}) - C_l(t_{i-1}, t_i)\,, \tag{2.2}$$

where $C_l(t_1, t_2)$ is a function that gives the number of completed jobs for flow f_l in time slot $(t_1, t_2]$, requiring $t_2 \geq t_1$. This can be defined as

$$C_l(t_1, t_2) = \sum_{j \in D_l} \sum_{p \in \Gamma_j} \theta(t_2 - S_{j,p} - E_{j,p}) - \theta(t_1 - S_{j,p} - E_{j,p})\,, \tag{2.3}$$

where $\theta(x)$ is the unit step function, defined as

$$\theta(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.4}$$

$\sum_{p \in \Gamma_j} \theta(t_2 - S_{j,p} - E_{j,p}) - \theta(t_1 - S_{j,p} - E_{j,p})$ can be identified as a function that gives the number of jobs completed before their deadline for a specific demand in a certain time slot, so we can define that as $C^{\dagger}_j(t_1, t_2)$ with

$$C^{\dagger}_j(t_1, t_2) = \sum_{p \in \Gamma_j} \theta(t_2 - S_{j,p} - E_{j,p}) - \theta(t_1 - S_{j,p} - E_{j,p})\,. \tag{2.5}$$

Now, the function C_l can be written as

$$C_l(t_1, t_2) = \sum_{j \in D_l} C^{\dagger}_j(t_1, t_2)\,.$$

This defines the part of the schedule for jobs that were still running and have carried over into the new schedule. Now, a new definition similar to that of $M(t_i)$ is needed to schedule jobs using the remaining free resources. This is because the originally defined schedule $M(t_i)$ does not take into account the fact that some resources are already being used by jobs still being executed. We define $R^*(t_i)$ as the number of resources still in use due to the jobs still being executed and, similarly, $c^*_k(t_i)$ as the number of communication qubits still in use at node k. Now, the schedule for the free resources at t_i can be defined in the following way:

Definition 10 *Let $\overline{R}(t_i) = R - R^*(t_i)$ be the number of resources that are free at t_i, $\overline{c_k}(t_i) = c_k - c^*_k(t_i)$ the number of available communication qubits for node n_k at t_i and F_k the set of indices of flows connected to node n_k. The incoming schedule $\overline{M}(t_i)$ is represented by a vector with vector elements $\overline{M}_l(t_i)$, where $\overline{M}_l(t_i)$ represents the number of newly added jobs at time t_{i-1} to the resources from flow f_l. The incoming schedule then has the following conditions:*

1. $\forall i \sum_l \overline{M}_l(t_i) \leq \overline{R}(t_i)$,
2. $\forall i \sum_{l \in F_k} \overline{M}_l(t_i) \leq \bar{c}_k(t_i) \quad \forall k \in \{1, ..., K\}$,
3. $\forall i, l \;\; \overline{M}_l(t_i) \in \mathbb{N}_0$.

For these schedules a new space, similar to $\mathcal{M}$, can be defined with $\overline{\mathcal{M}} = \{\overline{M} \in \mathbb{N}_0^F : 1. \wedge 2. \wedge 3.\}$, and now also $\overline{\mathcal{M}} \subseteq \mathcal{M}$. Using these partial schedules, an updated definition for the complete schedule $M(t_i)$ can be made:

Definition 11 *Let $M^*(t_i)$ be the residual schedule for t_i and $\overline{M}(t_i)$ the incoming schedule. Then, the complete schedule for t_i is given by $M(t_i) = M^*(t_i) + \overline{M}(t_i)$.*

Even with the schedule now being a sum of the residual and the incoming schedules, it can be seen from the new definition that $M(t_i)$ still adheres to the previously used conditions for the complete schedule.

2.5.2. First in, first out scheduling

The first in, first out (FIFO) scheduling algorithm, also known as first come, first serve, prioritizes the jobs with the earliest arrival times [41, 42]. FIFO is a very low cost algorithm, as it only needs a single queue with no additional information; the jobs that arrived first are at the front of the queue and the newer ones are placed behind, so the arrival dates don't need to be saved. A FIFO scheduler is not preemptive by implication, since a job cannot arrive earlier than any job already in the queue. Most schedulers that have a queue will immediately start a new job if a resource is empty and the queue is not, which FIFO does as well. This makes it work-conserving, as opposed to non-work-conserving schedulers which may leave resources idle for some time despite jobs being available. While its simplicity and low cost to implement are attractive, it does not tend to perform well in various metrics such, making it an unattractive scheduler if performance is important [41, 42].

Schedulers often optimize certain metrics, usually because scheduling algorithms are solutions to specific scheduling problems [39], and are used for some specific cases where those metrics matter. It might seem like a FIFO scheduler would minimize the waiting time, as it always prioritizes the job that arrived first, but this is not actually the case. Consider a simple single flow situation where 3 jobs have to be executed. The first job, J_1, arrives at t=0 ms and takes 2 ms to complete, the second job, J_2, arrives at t=1 ms and takes 10 ms to complete and the last, J_3, arrives at t=2 ms and takes 1 ms to complete. To minimize the waiting time, a scheduler would first schedule J_1, then J_3 and J_2 as the last one. This gives an average waiting time of $(0+0+2) = \frac{2}{3}$ ms. A FIFO on the other hand, would schedule them in order, leading to an average waiting time of $(0+1+10)/3 = \frac{11}{3}$ ms, which is much longer than the optimal case. This problem is generally called the convoy effect [42]. FIFO does introduce some sense of fairness, as it does not discriminate between jobs based on their parameters and if a job is already in the queue, no other jobs submitted later will get priority over it, so there should be no starvation of jobs when using a FIFO scheduler [41].

A FIFO algorithm can be described as in Algorithm 2, based on a discrete event-like system where this algorithm gets called at each time step.

Algorithm 2 FIFO scheduler

```
while there are submitted jobs do
    enqueue(job)
end while
while at least one resource is free do
    activate(oldest job in queue)
end while
```

There are no unique difficulties in implementing a FIFO scheduler in a quantum network instead of a regular one, as in both cases only the job arrival time matters, and non-deterministic execution times do not present a problem for implementing the FIFO scheduler.

2.5.3. On-demand scheduling

While there are many different variations of schedulers that use certain measures to make decisions or preempt certain jobs, the most simple case is the one that immediately schedules an incoming job if the required resource is free, and otherwise blocks it [43]. Such an algorithm is the on-demand (OD) algorithm, an example use case of which are the old telephone switching centers that had to connect users manually. Because this algorithm does not store any queues nor make decisions based on complex considerations, it has a very low complexity. It also optimizes the average response time/responsiveness. This is because, if a job is accepted, it is immediately executed and the response time is equal to the computation time (or 0, depending on the definition), which is the lowest time possible for a completed job. It is expected not to perform as well for other metrics like the global throughput or resource utilization, since this algorithm lacks a queue. This means that after finishing a job, resources may stay idle until the next job arrives, even if multiple requests were blocked while it was performing the previous task, especially in the high traffic regime. For that reason, it can be considered non-work-conserving. The job fulfillment is also predicted to be high, since accepted jobs are immediately executed in OD scheduling, which means that accepted jobs have the largest probability of being completed within the deadline (if there is one). The flip side is that the the expected average number of demands that have

to be sent before a job gets accepted is also high, as, depending on the load condition, many demands may be blocked.

The on-demand algorithm can be described as in Algorithm 3.

Algorithm 3 OD scheduler

```
while there are submitted jobs do
    if at least one resource is free then
        activate(job)
    end if
end while
```

Here, it can be seen that blocking does not have to be an action itself, it can simply be the inaction of the scheduler, so long as it is not required to let the submitter of the job know that his job has been blocked.

The probability that a request gets blocked when using the OD algorithm in the classical case is given by the Erlang Loss formula (also known as the Erlang B formula) [44], but this formula does not hold in the quantum case. Instead, a different formula was derived for the quantum hub implementation of OD scheduling by Gauthier, Vasantam and Vardoyan [43]. The derived blocking probability is given by the probability that a job from a certain flow arrives at the hub while all the resources are active, divided by the probability that this request arrives in the first place, but the actual derivation is outside of the scope of this thesis.

2.5.4. Earliest deadline first scheduling

If the jobs concerned have deadlines, the Earliest Deadline First (EDF) algorithm executes the job that has the earliest deadline first and preempts already executing tasks if a job with an even earlier deadline arrives [39].[2] Unlike FIFO and OD, EDF is generally a preemptive scheduler, meaning that some tasks may be halted before they have finished so that a different job can be executed in between. However, this algorithm does not guarantee that a deadline is reached. In the case of high traffic, a long backlog may cause all of the queued jobs to be late. This is because EDF blindly schedules all jobs with the shortest deadline, even if doing so causes all other jobs to miss their deadline. This is often called the domino effect [39]. Similarly, if a job gets submitted with a deadline that is later than most others, it may get starved by other jobs arriving with earlier deadlines until it is too late to complete that job. Since the scheduler will immediately activate a job when there is one available, it is a work-conserving scheduler. This algorithm has to keep track of all of the queued jobs, and needs to take into account the deadline for each job, making it slightly more complex than the FIFO scheduler, but it is still simpler than other algorithms that utilise queues that will be introduced later.

This algorithm optimises the maximum lateness, meaning that the maximum time a job will be late is minimised. Moreover, it is also optimal in the sense of feasability. This means that if there is a feasible schedule where all jobs finish before their deadline, EDF is able to find it [39]. Despite not being much more complex, the optimality of this scheduler in the case of feasibility introduces a clear advantage of EDF over FIFO.

The EDF algorithm can be described as in Algorithm 4.

Even the simplest EDF scheduler requires deadlines, so now the algorithm also needs to handle job expiry. The current algorithm assumes either hard or firm deadlines: it does not make sense to execute jobs that have already expired [39]. If the deadlines considered are soft, these lines may be removed, since even late jobs are useful then.

While the optimality in the classical case is only guaranteed with preemption, it may not always be helpful to use preemption in the quantum case. Due to decoherence, if the duration for which a job is preempted is long enough, i.e. in the order of the window supplied, all progress is lost and the job must restart completely. In this thesis, jobs will therefore not be preempted in any scheduler, including EDF.

[2] Alternatively, the decision to not preempt jobs could be made, but then the optimality may not be guaranteed.

Algorithm 4 EDF scheduler

```
for job in queue do
    if job is expired then
        dequeue job
    end if
end for
while there are submitted jobs do
    enqueue(job)
end while
while at least one resource is free do
    activate(job with earliest deadline in queue)
end while
```

2.5.5. Earliest feasible deadline first scheduling

Admission control for the EDF scheduler has already been extensively researched in classical scheduling [45, 46, 47, 48]. The earliest feasible deadline first (EFDF) scheduler described here is also a variation of the EDF scheduler that uses admission control to discriminate which jobs get scheduled. Here, the feasible does not refer to the feasibility of the EDF scheduler overall, but rather to the process of checking whether a certain job can still feasibly be scheduled when considering the deadline.

A problem that can occur with the EDF scheduler mentioned earlier was the domino effect, where the scheduling of the job with earliest deadline causes all other jobs after it to fail to meet their deadline. The EFDF scheduler as described in this thesis will therefore de-prioritize scheduling jobs / demands if they are not expected to be completed within their deadline. This attempts to prevent the domino effect, similar to the algorithm proposed by Salmani et al. [48]. The optimality for maximum lateness and feasibility do not automatically transfer to this scheduler, though the trade off between this and the reduction in scheduling of infeasible jobs may still be worth considering in some cases. Additionally, it is also more computationally complex, as it needs to check for feasibility, and it may have to do so for each job in the queue before it finds out there are no feasible jobs. There are many ways to define and check for feasibility, but this thesis will only consider a job infeasible in the case where the (expected) execution time exceeds the time left before the deadline.

An algorithm for an EFDF scheduler may look like Algorithm 5.

Algorithm 5 EFDF scheduler

```
for job in queue do
    if job is expired then
        dequeue(job)
    end if
end for
while there are submitted jobs do
    enqueue(job)
end while
while at least one resource is free do
    for job in queue do
        if job is feasible and deadline for job is earlier than for current next job then
            next job = job
        end if
    end for
    if no feasible job found then
        next job = job with earliest deadline in queue
    end if
    activate(next job)
end while
```

In this implementation, infeasible jobs are may still be scheduled, whenever no feasible job is in the queue. This can be relevant classically if the deadlines are soft and to ensure it is still a work-conserving scheduler. From a quantum scheduling perspective, this also makes sense. The hub will not have access to an execution time beforehand and has to use an estimate of this value, $E^{\dagger}_{j,p}$ to make this decision, since execution times are non-deterministic. A job may always end up being shorter simply by being lucky, so even if they seem infeasible, jobs should still be scheduled if there are no feasible jobs left.

The specific implementation of the EFDF scheduler in this thesis will not use individual job deadlines, but the deadlines of the demand instead, since all jobs are associated with a demand d_j. This means it will not check if a single job is still feasible, but if the number of jobs required to complete the demand are still feasible within the time left.

2.5.6. MaxWeight scheduling

MaxWeight (MW) Scheduling uses individual queues for each user/flow, and in the simplest implementation prioritizes the oldest job from the longest queue [49]. This means that if there is a network of 3 flows, there will be 3 queues. If flow A has 2 queued jobs, flow B has 1 queued job and flow C has 3 queued jobs and there is only 1 free resource at that moment, the first job from queue C will be selected for execution. In a more general implementation, a MW scheduler prioritizes the queue with the highest weight, where the weight $W_l(q_l(t_i))$ for a flow f_l may depend on the queue length $q_l(t_i)$, but can take different forms. MW schedulers are throughput optimal, so long as their weight function $W_l(x)$ is valid, meaning it satisfies the following 3 conditions [49]:

Definition 12 *Function $W(x)$ is defined as a valid weight function if the following conditions hold:*

1. *function $W(x)$ is increasing and differentiable for $x \geq 0$, and $W(0) = 0$;*
2. *as $x \to \infty$, $W(x) \to \infty$;*
3. *for any given $\delta > 0$, there exists a constant $B_\delta \geq 0$ such that, for any $x \geq 0$,*

$$(1-\delta)W(x+1) - B_\delta \leq W(x) \leq (1+\delta)W(\max(x-1,0)) + B_\delta\,,$$

where the third condition is a way of stating that $W(x)$ does not change significantly when x in-/decreases by 1.

Now, the MW scheduler can be defined mathematically in the following way:

Definition 13 *Let $\overline{M}^{(j)}(t_i)$ be the j-th possible incoming schedule for time t_i. A MaxWeight scheduler chooses a schedule $\overline{M}(t_i)$ such that*

$$\overline{M}(t_i) \in \arg\max_{\overline{M}^{(j)}(t_i)} \sum_{l=1}^{F} W_l(q_l(t_i))\overline{M}_l^{(j)}(t_i)\,, \tag{2.6}$$

meaning that it finds a schedule that maximizes the sum of the weights of the queues corresponding to the scheduled jobs. $W^{(j)} := \sum_{l=1}^{F} W_l(q_l(t_i))\overline{M}_l^{(j)}(t_i)$ is called the total weight of the j-th schedule.

The classic case has no preemption, meaning that resources will only be freed once the job being executed is finished. This scheduling algorithm optimizes the throughput. In asymptotic circumstances, like one queue experiencing light traffic while another experiences heavy traffic, the throughput of the lightweight flows is expected to suffer as the many incoming jobs in the busy queue outweigh the few queued jobs in the light traffic ones. This can lead to starvation of flows with light traffic, meaning that certain scenarios could have bad responsiveness when averaged over all the queues. Resource utilisation in this algorithm is predicted to be good, as a resource will be utilized so long as there are enough jobs queued, since this is a work-conserving scheduler. The need for F queues, where F is the number of flows, and the additional need for the algorithm to keep track of the length of each of the queues does make this algorithm more complex than other simpler algorithms.

A ”simple” MaxWeight scheduling algorithm can be described as in Algorithm 6.

This way of implementing it means that it iteratively decides from which queue to schedule, i.e. it checks which queue is the longest each time it needs to schedule a new job. A naive way of implementing

Algorithm 6 MW scheduler

```
for flow in network do
    for job in flow queue do
        if job is expired then
            dequeue from flow(job)
        end if
    end for
    while there are submitted jobs in this flow do
        enqueue in flow(job)
    end while
end for
while at least one resource is free do
    activate(job from flow with the longest queue)
end while
```

this "simple" MaxWeight scheduler for the quantum hub would use Definition 13 with $W_l(q_l(t_i)) = q_l(t_i)$. This, however, would actually produce unexpected results, that deviate from the desired behaviour. This is because the quantum hub allows each flow to use multiple resources and we expect the "simple" MaxWeight scheduler to schedule these iteratively. To see why this would not always correspond to expectations, an example will be illustrated here. Let's say this MaxWeight scheduler is used for a system with 3 flows, f_1 with 5 jobs queued, f_2 with 3 jobs queued and f_3 with 2 jobs queued, 4 resources and 4 communication qubits for each node.[3] The desired behaviour of this scheduler would be to first take 2 jobs from f_1, after which it has 2 resources left and 2 queues with the same queue length, so it takes 1 additional job from both f_1 and f_2. Looking at the definition instead shows a different result. The weight for each queue is defined for a whole time slot, not taking multiple steps to fill the schedule into account. This means that, instead, the optimal schedule is simply taking as many jobs as it is allowed to from the longest queue in 1 go, which would be 4 jobs from f_1. To fix this, a "simple" MaxWeight scheduler should instead use

$$W_l(q_l(t_i)) = q_l(t_i) - \frac{(\overline{M}_l(t_i) - 1)}{2}\,, \tag{2.7}$$

which takes into account the fact that queue lengths decrease when scheduling iteratively. The derivation of this is as follows. Let us take a network with only one flow that has a queue with queue length q, and a total of R resources, and an empty residual schedule M^*. The schedule is now a 1D vector, or simply a scalar, M. Now, M simply represents how many jobs get scheduled in the network and M can be limited by either R or q. Both of the nodes in this flow have R communication qubits. We already know what the total weight W for each M should be when doing MW scheduling iteratively according to Algorithm 6: simply take the queue length each time you schedule a job and sum them all up. It is trivial to see that the weight for this flow with $M = 1$ would be q. If $M = 2$, the first scheduled job would add a weight of q, but the second would add a weight of $q - 1$ since in the iterative process one job has been removed from the queue already, decreasing the queue length. Now, the total weight is $q + q - 1 = 2q - 1 = qM - 1$. For $M = 3$, the total weight then becomes $q + q - 1 + q - 2 = qM - 3$ and for $M = 4$ this becomes $q + q - 1 + q - 2 + q - 3 = qM - 6$. For any M, it seems then that the weight is given by $W = \sum_{m=0}^{M-1} q - m = q + q - 1 + q - 2 + ... + q - M + 1 = qM - 1 - 2 - ... - M + 1 = qM - \sum_{k=1}^{M-1} k$. Here the last part can be identified as the triangular number T_{M-1}, which has an analytical expression given by $T_n = \frac{n(n+1)}{2}$. Putting this together, the total weight can be written as $W = qM - T_{M-1} = qM - \frac{M(M-1)}{2} = (q - \frac{M-1}{2})M$. This shows that the weight function in this case should then be $W(q) = q - \frac{M-1}{2}$. This was derived for a single flow, but it holds for a network with multiple flows too. This is because it can be derived for every single flow on that network, by using M_l instead of M and then taking all the results together. Also, this can be applied to every single time slot, as it is independent of the time of the system. This transforms it into the earlier shown eq. (2.7).

[3] The communication qubits are not important for this example, but choosing it this way means that no matter what, no node can need more qubits than the number of communication qubits it has available, since at most 4 resources can be used in a flow connected to that node.

Now, it is important to ask if this weight function adheres to the previously described conditions for valid weight functions. A problem with this is that the weight function depends on the final chosen schedule, making it more difficult to find out what the weight of a queue would be by simply looking at the queue lengths; $M_l(t_i)$ will be limited by how many jobs get scheduled in other flows. What can be seen more easily is that this weight function always satisfies condition 2, so long as $\overline{R} < \infty$, meaning every $\overline{M}_l < \infty$. For the first condition, it seems that there may be cases where $W_l(0) \neq 0$. In fact, if you assume $q_l \geq \overline{M}_l$, then you will always find $W_l(0) = \frac{1}{2}$. This can be fixed by instead choosing a new weight function $W_l^\dagger(q_l) = W_l(q_l) - \frac{1}{2}$, but since this just adds a constant, it will not affect the ordering of the schedules. Now the new total weight for each candidate schedule is $W^{j,\dagger} = \sum_l W_l^\dagger(q_l)\overline{M}_l^{(j)}$ = $\sum_l (W_l(q_l) - \frac{1}{2})\overline{M}_l^{(j)} = \sum_l W_l(q_l)\overline{M}_l^{(j)} - \frac{1}{2}\sum_l \overline{M}_l^{(j)} = W^{(j)} - \frac{1}{2}\overline{R}$.[4] This is just the same weight with a constant subtracted, which means the order by weight of the schedules should stay the same. Since we only care about picking the flow with the highest weight, any different weight function that provides the same order can be chosen for the MW scheduler.

Because the scheduler is not explicitly forbidden from picking jobs from an empty queue we have to be sure that it will never pick an empty queue over a filled one that still has unscheduled jobs, to ensure it is work-conserving. This does not happen and can be proven in the following way. Let $q_0 = 0$ represent an empty queue with weight $W_0(q_0) = \frac{1-\overline{M}_0}{2}$. Let $q_l > 0$ represent a non-empty queue f_l, with weight $W_l(q_l) = \frac{1-\overline{M}_l}{2} + q_l$. If there are $\overline{R}$ free resources available, it will always maximize the total weight by increasing $\overline{M}_l$, because assuming $M_l \leq q_l$ implies $W_l(q_l) > 0$ and since $W_0(q_0) < W_l(q_l)$ always when $\overline{M}_0 = \overline{M}_l$, it will schedule from f_l before f_0 for any q_l. As expected, only once the queue from flow f_l is empty will the weights added by scheduling a job from their queues become equal. However, one can wonder if the scheduler would benefit from scheduling more jobs than available now that it has reached this point, where not a single queue has any jobs. To do this, we can write the difference in total weight due to the addition of 1 job from a flow f_l with m jobs already scheduled and an initial queue length q_l as $\Delta W_l(m, q_l) = W_l(m+1, q_l)(m+1) - W_l(m, q_l)m = (q_l - \frac{m}{2})(m+1) - (q_l - \frac{m-1}{2})m = q - m$. This shows the total weight will not increase anymore if $m \geq q$, so there is no point in scheduling empty queues.

For a practical implementation of a MaxWeight scheduler, most of these technical considerations will not be a problem if it follows the algorithm as described by Algorithm 6.

2.5.7. Round robin scheduling

There are multiple ways to implement Round Robin (RR) in a network with multiple flows. All implementations have in common that a type of fairness is optimised, as RR rotates the jobs so that no starvation takes place. Two less strict implementations will be detailed here. The first assigns a set timeslice for each job currently queued, and when the timeslice is done, the job is preempted if it has not finished by then [42]. This allows the next job in-line to be executed, which will generally result in low waiting times, as jobs don't have to wait on the completion of other jobs before starting execution. If a job finishes before the timeslice is over, the next job in line can be executed, reducing idle time. This method optimises the fairness in time per job, but not time per user, as many different jobs in line may be from the same user. It may also fail to execute a single job if the computation time is longer than the time slice available and the rate of jobs added to the queue is higher than the frequency of the timeslices (so the period of jobs added is shorter than the timeslice duration). This means that with this algorithm responsiveness could suffer in high load scenarios. Moreover, using RR will also cause the average response time to be increased. While the jobs start earlier because they have to wait less, they may end up waiting longer for the execution because this is not necessarily completed in one timeslice [42]. Another problem that can occur is an increase in overall computation time due to the context switching that needs to occur to preempt one job and start another. Increasing the timeslice duration will lessen this effect.

An alternative way of implementing Round Robin is by scheduling a job for each flow in turn, switching flows once a job has finished. This way can be useful if preemption is undesirable, as is the case in quantum hub scheduling. If all jobs are equal in complexity, the variability in the throughput per

[4] This is true for all $M^{(j)}$'s, because it is not explicitly forbidden by this scheduler to schedule empty flows, which actually means that $\sum_l \overline{M}_l^{(j)} = \overline{R}$ in every case.

flow should be small, making this algorithm fair in this sense. This does mean that instead of a single queue containing all jobs, again each flow has its own queue. This increases the memory complexity of this algorithm because of the increased number of required data structures to maintain. There may be cases where this algorithm is expected to have some downsides, like again a case of asymptotic traffic, where most jobs arrive from one flow. Here, while all the other flows should have relatively good responsiveness, the heavy traffic flow could get a long backlog of jobs that have to wait very long (on average $n\mathcal{F}E_{\text{avg}}/2$, where $\mathcal{F}$ is the number of flows, n is the number of jobs in the heavy traffic flow and E_{avg} is the average time it take to execute a job, compared to just $\mathcal{F}E_{\text{avg}}$ for the light traffic queues), which means a poor responsiveness for that flow, which can cause adverse effects if there are deadlines to be considered. Alternatively, if one flow has a similar number of jobs, but significantly longer jobs, then for each round every flow gets one turn, but averaged over time the flow with time-consuming jobs gets more access to the resources.

A round robin scheduling algorithm can be described as in Algorithm 7.

Algorithm 7 RR scheduler

```
for flow in network do
    for job in flow queue do
        if job is expired then
            dequeue from flow(job)
        end if
    end for
    while there are submitted jobs in this flow do
        enqueue in flow(job)
    end while
end for
while at least one resource is free do
    activate(job from flow that is next in line and has a non-empty queue)
end while
```

Round robin can be considered in quantum networks if fairness is a major factor, but how well a RR scheduler will perform in a fairness metric depends very much on how this fairness metric is defined, which should be taken into account.

3

Methods

In this section, the implementation of the simulation will be explained. First, all the metrics that have been used in this thesis and the project code will be explained. Then, it will be explained how demands from the nodes are processed into executable jobs, so it is understood what a individual job in this thesis actually is. Then, there will be a short introduction to the programming and simulation environment used for this thesis, NetSquid, and a description of how the network is implemented therein. After this, the implementation of the actual demands themselves in NetSquid will be given, followed by the simulation parameters that are used for this thesis.

3.1. Metrics

The main focus of this thesis is on measuring the performance of various schedulers to compare their performance. To do so, many different metrics can be used and there are many variations that give similar, but not the same information. In this section, these performance metrics are described and defined mathematically. They are separated in different categories, where variations of the same top-level idea are bundled together. Some of these metrics may be more interesting than others for different people or in the context of different use cases. For example, one might want their scheduler to prioritise throughput, prioritise fairness, minimise idle time of the resources, or any combination of these. To make sure there is no confusion about what each metric represents and how they can be useful, it is important to have clear definitions of these metrics to avoid misunderstandings. In the end, since a lot of metrics give similar information, usually only one per category will be used. Defining all of them may nonetheless be helpful for understanding them. No prior work was found that compiles such a complete set of metrics in the context of a quantum network. Therefore, many of the definitions presented in this section were adapted or formulated based on classical analogues.

3.1.1. Demand Completion

Demand completion is a very high-level metric. Rather than being influenced directly by the performance of a scheduler, it can be understood as being a function of the other lower-level metrics such as throughput and response time. For example, a scheduler that has a short waiting time, but also a low job success rate may find a similar number of completed demands as one that has a long waiting time and a high job success rate.

Demands are formatted by applications [13], which means that to successfully execute an application, the associated demand in the network has to be completed successfully. For that reason, demand completion is considered the most important metric to optimise in a network; optimising other metrics such as throughput and waiting time can be seen as steps to get there, though they can still have individual importance outside of demand completion. For example, achieving a high fairness in the network will not automatically be reached by optimising for demand completion, unless all demands can be completed successfully.

Proportion of completed demands

The proportion of completed demands gives the number of completed demands divided by the total number of demands submitted to the network. The definition is as follows:

Definition 14 *Let $C_j^\dagger(t_1, t_2)$ be the function that gives the number of jobs from demand d_j completed before their deadline in time slot $(t_1, t_2]$, $\mathcal{D}$ the total number of demands submitted to the hub, $N_{\text{inst},j}$ the number of jobs required to complete demand d_j, t_0 the first relevant time step in the network and t_L the last. Then, the proportion of completed demands, Φ, can be defined as*

$$\Phi = \sum_{j=1}^{\mathcal{D}} \nu_j(t_L)\,, \tag{3.1}$$

where $\nu_j(t_i)$ is a time-dependent number $\in \{0, 1\}$ that denotes whether a demand was completed (1) or not (0) at time t_i, given by

$$\nu_j(t_i) = \begin{cases} 1, & \text{if } C_j^\dagger(t_0, t_i) \geq N_{\text{inst},j}, \\ 0, & \text{otherwise.} \end{cases} \tag{3.2}$$

Proportion of failed demands

The proportion of failed demands does not give different information from the proportion of completed demands since a demand can only either fail or complete as the simulation finishes, but it can put the information in a different context that highlights the failing schedulers instead of the succeeding ones. It can be defined as follows:

Definition 15 *Let Ψ denote the proportion of failed demands. It is defined as*

$$\Psi = 1 - \Phi \tag{3.3}$$

3.1.2. Throughput

Generally speaking, throughput in a network measures the average rate of information transfer in a flow. This is often measured in bits per second or data packets per second. Another way to represent it is in number of jobs completed per second [40]. Since a quantum hub network doesn't send classical information bits, it makes more sense to measure it in jobs/entanglement packets generated per second, to represent the throughput of the system [13].[1] It is a metric that is often desired to be high, and may be the main focus of optimisation in a network [40]. It should be intuitive that a higher throughput is a direct way of increasing demand completion: the more jobs get completed per second, the higher the likelihood a demand gets all requested jobs completed in time. It is, however, not a function of just a processor and its capabilities. The throughput also depends on the number of jobs submitted in the first place and cannot exceed the rate at which they are submitted [40].[2] This fact means it is important to understand that throughput is not just a measure of network capabilities, but also network demand. This will also come back in possible ways of measuring this metric, as it is an important consideration for defining the throughput.

Throughput per flow

The simplest metric to define is the throughput of the individual flows themselves. The motivation behind this metric is to get an idea of the performance a user experiences while using this network. Inspecting each individual flow this way may reveal certain bottlenecks in the network or otherwise identify flows that are starved for other reasons. The individual flow throughput also serves as an in-between calculation for other metrics, which makes the following definition more practical too:

Definition 16 *Let t_L be the last time step and t_0 the first in the network and let $C_l(t_1, t_2)$ be the total number of jobs executed before their deadline within time slot $(t_1, t_2]$ in flow f_l as previously defined. The throughput of flow f_l, denoted X_l, is given by*

$$X_l = \frac{C_l(t_0, t_L)}{T}\,, \tag{3.4}$$

[1] It does send classical information to communicate classically, but here we are interested in a networks ability to perform entanglement generation, for which the only interesting aspect is how fast quantum information, i.e. packet generation attempts, are performed

[2] In fact, in steady state, the throughput is equal to the rate at which jobs are submitted [40].

where $T = t_L - t_0$ is the total running time of the network. This will give the throughput of flow f_l in jobs/packet generation attempts per second (jobs/s).

Global throughput

While the individual throughput of a flow can be of interest to a user, the network operator may be more interested in the global throughput. This metric can be seen as the scheduling efficiency of the scheduler, because a scheduler that creates tight but densely packed schedules will have a higher global throughput than a sparse schedule that has the same number of jobs in a longer period of time. It is not the exact same though, as this is also dependent on the given load to a network, so to compare efficiency the same load has to be supplied, especially when open networks are considered [40]. The global throughput is the total number of jobs completed per second in the network, which means it can be defined as the sum of each individual flow's throughput:

Definition 17 *The global throughput, X is the sum over the flows of the throughput X_l, which means that we can write:*

$$X = \sum_{l=1}^{\mathcal{F}} X_l \,. \tag{3.5}$$

Using (3.4)*, we can write this out as:*

$$X = \frac{1}{T} \sum_{l=1}^{\mathcal{F}} C_l(t_0, t_L) \,. \tag{3.6}$$

Active throughput per flow

The previously given definitions are simple to calculate and easy to relate to each other, but they have a shared flaw: a flow may not be active for the entire time the network is operated, if at all. The implementations of the previous definitions can be adapted to not take into account inactive flows, i.e. flows that had no demands or jobs submitted to them. However, that still does not solve the problem of poorly representing the scheduling efficiency of flows that submitted less demands than others. To solve this, a slightly different metric will be defined: the active throughput per flow, which is the throughput of the flow calculated with only the time at which a demand was active. The definition is as follows:

Definition 18 *Let D_l be the indices of demands submitted in flow f_l, σ_j the submit time of demand d_j and τ_j the end time of demand d_j, either by expiration or completion. Then, the active throughput for flow f_l, $X_l^\dagger$ is given by*

$$X_l^\dagger = \frac{C_l(t_0, t_L)}{T_l} \,, \tag{3.7}$$

where T_l is the active time of the flow given by

$$T_l = \sum_{j \in D_l} \tau_j - \sigma_j \tag{3.8}$$

Average active throughput

Similar to the global throughput, a network operator may be more interested in global metrics than individual ones. While summation for a global throughput does not make sense anymore due to the different denominators of the active throughput, they can still be averaged to get a single summary metric:

Definition 19 *Using the individual active throughputs $X_l^\dagger$, the average active throughput, $X^\dagger$, is given by*

$$X^\dagger = \frac{\sum_{l=1}^{\mathcal{F}} X_l^\dagger}{\mathcal{F}} \tag{3.9}$$

3.1.3. Resource utilisation

In some cases, it is preferable to maximise the efficiency of resource usage, maintaining maximum resource utilisation as long as possible. Running a network, even if idle, can consume a lot of power [40,

43]. Especially in a quantum hub network, where the hub has to allocate the resources between many different flows, which will often outnumber the available resources, it is important that a scheduler does not waste time and power by unnecessarily letting resources stay unused. For example, the cooling of superconducting quantum computers is very costly, as they need to be cooled to a temperature on the order of K to mK to be able to operate [26, 50]. While an EGS hub does not have memory qubits that would need cooling [12] and the BSA analysers can be realised with linear optical elements of which only detectors consume power [38], it will still incur a cost to maintain the network while the resources are idle. Moreover the quantum processing nodes of the users may involve such costly elements. On the other hand, utilisation is, just like throughput, a measure of both scheduling efficiency and the network load. There is an upper bound for the utilisation above which scheduling a task set becomes impossible [39], so while it may be undesirable to have resources stay idle unnecessarily, it is important to not mistake a low load on the network for a scheduler that leaves resources idle.

Idleness factor

The easiest implementation of a metric quantifying resource utilisation is to measure how long all resources are idle, by measuring the idle time as a fraction of total running time [43]:

Definition 20 *Let $r(t_i)$ give the number of resources in use between time t_i and t_{i+1}, excluding at t_{i+1} itself, let t_L be the last time step in the network and let $\theta(x)$ be the unit step function as defined in (2.4). Using this, the idleness factor is given by*

$$I_d = \frac{1}{T}\sum_{i=0}^{L-1} \theta(-r(t_i))(t_{i+1} - t_i)\,. \tag{3.10}$$

Here $I_d = 1$ is the case where not a single resource was being used during the running time of the network, whereas $I_d = 0$ is the case where there was always at least 1 resource busy executing a job.

Full utilisation factor

Idleness captures how much time all the resources spend idle, but gives no clue as to how much all the resources are used versus only some. For that, we can use the full utilisation factor, again as a fraction:

Definition 21 *Let R be the total number of resources available in the hub. Then, the full utilisation factor is given by*

$$U_{\text{full}} = \frac{1}{T}\sum_{i=0}^{L-1} \theta(r(t_i) - R)(t_{i+1} - t_i)\,. \tag{3.11}$$

Here $U_{\text{full}} = 0$ is the situation where at all times the number of resources in use is less than the maximum number possible, and $U_{\text{full}} = 1$ corresponds to the situation where all resources are in use all of the time.

Average utilisation factor

While these previous metrics capture specific parts of the resource utilisation, it may also be helpful to get a general average value to compare resource utilisation from different schedulers. This can be done by averaging the resources used per time slot [40]. To make it more general and easily comparable between networks of different sizes, this value can be normalised with the available number of resources to get a value between 0 and 1 in all cases.

Definition 22 *The average utilisation of a scheduler is given by*

$$U_{\text{avg}} = \frac{1}{RT}\sum_{i=0}^{L-1} r(t_i)(t_{i+1} - t_i)\,. \tag{3.12}$$

Here $U_{\text{avg}} = 0$ is the situation where no jobs were executed at any time, $U_{\text{avg}} = 1$ corresponds to the situation where all resources are in use all of the time and $U_{\text{avg}} = 0.5$ corresponds to cases such as all the resources being busy half of the time, or half the resources being busy all the time.

3.1.4. Fairness

For some use cases it may be of interest to see what the fairness is of a certain scheduler. While some systems are expected to not be fair, for example in cases where a priority class based system is used to differentiate low priority from high priority jobs, others systems are expected to be fair. Usually fairness is an expectation when the different nodes in the network are individual users of that network with the same rights [39]. To capture the fairness of a scheduler and see if it is applicable in such cases, we can look at different types of metrics.

Throughput deviation

While the throughput per flow could be used for a sense of fairness, as it shows the distribution of jobs over different flows, it does not quantify this yet in a single easy to understand number. To do so, we can make use of the throughput standard deviation:

Definition 23 *Let $\mathcal{F}$ be the total number of flows in the network. Using the previously defined individual throughput X_l and the global throughput X, we find for the throughput deviation X_{std} the following relation:*

$$X_{std}^2 = \frac{\sum_{l=1}^{\mathcal{F}} (X_l - X/\mathcal{F})^2}{\mathcal{F} - 1} \,. \tag{3.13}$$

Throughput fairness index

A different metric that tries to capture the fairness of a network is Jain's fairness index, as introduced by Jain, Chiu and Hawe [51]. This index number goes from the worst case $1/\mathcal{F}$, to the best case 1. The best case occurs when all users have equal access to the resources, which in this case will be defined as having the same throughput, though other variables are also possible as we will see later. The worst case is then instead when only 1 flow has access to all the resources, meaning that the global throughput is equal to that individual flow's throughput. The definition of Jain's fairness index based on throughput is as follows [51]:

Definition 24 *The throughput fairness index $\mathcal{J}_X$ can be calculated using the following equation:*

$$\mathcal{J}_X = \frac{\left(\sum_{l=1}^{\mathcal{F}} X_l\right)^2}{\mathcal{F} \sum_{l=1}^{\mathcal{F}} X_l^2} = \frac{X^2}{\mathcal{F} \sum_{l=1}^{\mathcal{F}} X_l^2} \,, \tag{3.14}$$

where X and X_l are again the global and individual throughput respectively and $\mathcal{F}$ is the total number of flows in the system.

Resource fairness index

While the previous definitions of fairness have focused on using throughput as a measure to capture the experience a user has while using a network, it does not capture other aspects of the network that can still be important to consider for network fairness, i.e. aspects which may be important from a operators perspective. For example, if there are two flows in the network and they are given equal access to the resources based on achieved throughput fairness, a user with much more intensive jobs will get a lot more access to the resources in the hub. Instead, a definition more closely related to the original introduction by Jain, Chiu and Hawe can be based on the resource allocation directly [51]:

Definition 25 *Let r_l denote the number of resources allocated to flow f_l averaged over the whole runtime of the network, where $\sum_{l=1}^{\mathcal{F}} r_l = U_{avg}$, meaning that the average utilisation factor is the sum of the average number of resources being used be each flow separately and let $\mathcal{F}$ be the total number of flows in the system. The resource fairness index $\mathcal{J}_R$ is given by the following equation:*

$$\mathcal{J}_R = \frac{\left(\sum_{l=1}^{\mathcal{F}} r_l\right)^2}{\mathcal{F} \sum_{l=1}^{\mathcal{F}} r_l^2} = \frac{U_{avg}^2}{\mathcal{F} \sum_{l=1}^{\mathcal{F}} r_l^2} \,. \tag{3.15}$$

Just like the throughput fairness index in Definition 24, the worst case scenario is $\mathcal{J}_R = \frac{1}{F}$ where only one flow gets access to the resources, and in the most fair case $\mathcal{J}_R = 1$ and each flow gets resources allocated equally.

Alternatively, fairness could be defined such that the waiting times are equalised for each job, no matter the execution time of the job [40]. Additional definitions of fairness are possible, but in this thesis we restrict ourselves to those already stated.

3.1.5. Responsiveness

Another important aspect of a scheduler is to what degree scheduled jobs have to wait before they are executed, which can be captured in multiple ways. Two examples are the response time and the waiting time, with the main difference being that the response time takes into account the execution time of a scheduled job whereas the waiting time does not [39, 40, 42][3]. While the execution time is expected to be mostly independent of the scheduler used, there may still be differences in when one of these metrics is preferred, since the waiting time may include jobs that were submitted and started executing, but never finished. The response time on the other hand, does not take into account jobs that started execution but did not finish in time.

Average response time

First, we will look at the total time a job is in the hub, i.e. time it takes for a job to go from submission to completion [40, 42, 39], the response time:

Definition 26 *Let $J_{j,p}$ denote the p-th job of flow f_l, let $Z_{j,p}$ denote the submission time of that job, let Γ_j denote the set of jobs that were successfully completed as defined earlier in Definition 9 and let $\gamma_j = \sum_{p\in\Gamma_j} 1$ be the total number of successfully completed jobs from demand d_j. Then, the average response time T_{resp} of a scheduler is given by*

$$T_{\text{resp}} = \frac{\sum_{j=1}^{\mathcal{D}} \sum_{p\in\Gamma_j} S_{j,p} + E_{j,p} - Z_{j,p}}{\sum_{j=1}^{\mathcal{D}} \gamma_j}\,, \tag{3.16}$$

where $S_{j,p}$ is the start time of job p of flow f_l and $E_{j,p}$ is its execution time.

Worst response time

In some cases, like when trying to find problems in the network, or to achieve a certain guaranteed response time, it may be interesting to also find the worst response time in the network, which is defined in the following way:

Definition 27 *The worst response time $T_{\text{Resp,max}}$ of a network is given by*

$$T_{\text{resp,max}} = \max_{j,p\in\Gamma_j} S_{j,p} + E_{j,p} - Z_{j,p}\,. \tag{3.17}$$

Average waiting time

The waiting time does not take into account how long jobs take to execute and instead only looks at the time between job submission and the start of job execution. This metric also takes into account the jobs that started execution, but did not finish in time. This may be helpful if many jobs do not finish execution and are thus not included in the response time. The definition is as follows:

Definition 28 *Let Δ_j denote the set of jobs such that for every $p \in \Delta_j$, $J_{j,p}$ is a job spawned from demand d_j that started execution but did not finish execution and that this is not true for any $q \notin \Delta_j$ and let $\delta_j = \sum_{p\in\Delta_j} 1$ be the total number of unfinished jobs from demand d_j. Using this, we can define the average waiting time of a network T_{wait} as*

$$T_{\text{wait}} = \frac{\sum_{j=1}^{\mathcal{D}} \sum_{p\in\Gamma_j\cup\Delta_j} S_{j,p} - Z_{j,p}}{\sum_{j=1}^{\mathcal{D}} \gamma_j + \delta_j}\,. \tag{3.18}$$

Average execution time

Lastly, we can also define the average execution time. The execution time is expected to mostly not depend on the scheduler that is used as this is mostly a (random) variable dependent on the job specifics,

[3] Note that the author of [42] uses response time and turnaround time, whereas this thesis defines them as waiting time and response time respectively.

but this is not necessarily true when taking into account switching times in the case of preemption for example[4]. Another important consideration is that the execution time does not take into account jobs that never finished execution, as mentioned earlier. The average execution time can be defined as follows:

Definition 29 *Using only the the execution time $E_{j,p}$ of jobs that finished execution, the average execution time, T_{Exec}, is defined by*

$$T_{\text{exec}} = \frac{\sum_{j=1}^{\mathcal{D}} \sum_{p \in \Gamma_j} E_{j,p}}{\sum_{j=1}^{\mathcal{D}} \gamma_j} \,. \tag{3.19}$$

3.1.6. Job fulfillment

To get an understanding of how close a schedule was to successfully completing all the submitted jobs, we can look at the job fulfillment. Increasing this metric will make it more likely a demand gets finished so that the application associated with it is executed successfully [13]. Job fulfillment also allows us to take another problem into account: it is possible for a scheduler to achieve an artificially low response time simply by not executing jobs that would cause the average to be raised, since non-executed jobs are not taken into account there. This is not the desired behaviour of a scheduler, because jobs that are still executed with a slightly worse response time are often preferred over jobs that were not executed at all. There are multiple aspects to the process of job completion, so first the job completion ratio will be defined.

Job completion ratio

This number between 0 and 1 gives the ratio of completed to submitted jobs. It is defined as follows:

Definition 30 *Let γ_j and δ_j be as the number of jobs successfully executed and the total number of unfinished jobs respectively from demand d_j as defined before, let Λ_j be the set of indices of the jobs that were rejected/blocked, i.e. were submitted to the hub but did not start execution from demand d_j, such that for every $p \in \Lambda_j$, $J_{j,p}$ is a job spawned from demand d_j that did not start execution and that this is not true for any $q \notin \Lambda_j$, let $\lambda_j = \sum_{p \in \Lambda_j} 1$ be the total number of rejected jobs from demand d_j, let $\Omega_j = \Gamma_j \cup \Delta_j \cup \Lambda_j$ be the set of all jobs submitted by demand d_j and let $\omega_j = \gamma_j + \delta_j + \lambda_j = \sum_{p \in \Omega_j} 1$ be the total number of jobs submitted by that demand. Now, the job completion ratio α can be defined as*

$$\alpha = \frac{\sum_{j=1}^{\mathcal{D}} \gamma_j}{\sum_{j=1}^{\mathcal{D}} \omega_j} \,. \tag{3.20}$$

This gives the fraction of jobs that were successfully executed before their deadline versus jobs that were submitted.

Job acceptance ratio

To assess the job completion of a scheduler, it is also useful to understand where the jobs get lost in the process. It is possible some do not reach the deadline and are preempted and discarded mid execution if that is enabled, but bugs in the system can cause them to be lost in other steps as well. To track jobs more exhaustively, we can define also the unfinished job ratio, as being the ratio of jobs that have started execution to jobs that have been submitted:

Definition 31 *The job acceptance ratio β is defined as*

$$\beta = \frac{\sum_{j=1}^{\mathcal{D}} \gamma_j + \delta_j}{\sum_{j=1}^{\mathcal{D}} \omega_j} \,. \tag{3.21}$$

A related measure is the unfinished job ratio, ζ, which gives the ratio of the number of jobs that started but never finished execution to the number of jobs that were submitted:

$$\zeta = \frac{\sum_{j=1}^{\mathcal{D}} \delta_j}{\sum_{j=1}^{\mathcal{D}} \omega_j} \,, \tag{3.22}$$

where we can see that $\alpha = \beta + \zeta$.

[4]There may be more subtle ways in which scheduling differences can affect execution times of individual jobs, for example simultaneous multithreading in CPUs [39].

Job rejection ratio

This does not capture all jobs that failed, since jobs can also be blocked or rejected upon submission to the hub, which is particularly interesting for the On-Demand scheduler [43]:

Definition 32 *The job rejection ratio η, the ratio of jobs that were submitted but immediately or eventually rejected by the hub, is defined as*

$$\eta = \frac{\sum_{j=1}^{\mathcal{D}} \lambda_j}{\sum_{j=1}^{\mathcal{D}} \omega_j} \,. \tag{3.23}$$

Lastly, for convenience, we can now define a job failure ratio, i.e. the ratio of jobs that either did not finish or were never accepted in the first place, ξ:

$$\xi = \zeta + \eta = 1 - \alpha \tag{3.24}$$

3.1.7. Pair success ratio

While the previous metrics will give a good understanding of how many jobs were successfully completed or accepted, this still does not capture all aspects of job fulfillment. Because of the random nature of quantum mechanics, some jobs that require for example the successful generation of multiple entangled qubit pairs in a short window can be particularly demanding for the scheduler. To ensure application success is maximised, it is useful to also maximise the generated number of requested pairs, as those are needed for application success [13]. For the network operator it can also be useful to know how many pairs had been successfully generated before the deadline was missed if the job did not finish execution. To this end, the pair success ratio can be defined as follows:

Definition 33 *Let $s^{\dagger}_{j,p}$ be the number of entangled qubit pairs that were successfully generated within the window w_j for job $J_{j,p}$ and let s_j be the number of entangled qubit pairs/links that were requested, where $s^{\dagger}_{j,p} \in \mathbb{N}_0$ is an integer such that $0 \leq s^{\dagger}_{j,p} \leq s_{j,p}$. Now, the pair generation rate can be defined as*

$$\pi = \frac{\sum_{j=1}^{\mathcal{D}} \sum_{p \in \Gamma_j \cup \Delta_j} s^{\dagger}_{j,p}}{\sum_{j=1}^{\mathcal{D}} \sum_{p \in \Gamma_j \cup \Delta_j} s_{j,p}} \,. \tag{3.25}$$

Here π is a number between 0 and 1, 0 being the case where no pairs where generated and 1 being the case where all requested pairs were generated successfully. It should be noted that $s^{\dagger}_{j,p} = s_{j,p}$ if $p \in \Gamma_j$.

3.1.8. Complexity

For these schedulers, it may be important to understand the complexity of the scheduling algorithms with respect to time, because a schedule that is to slow to compute will delay the system. This consideration may be especially relevant in the context of online scheduling. It is also possible to look into memory usage to understand the memory complexity, but in most cases the memory of the classical processor will not be the limiting factor in these quantum networks. Thus this section is restricted to just temporal and operational complexity, which is generally the number of instructions an algorithm performs based on the given input size in big O notation [39].

Computational wall time

First, we will focus on a heuristic definition instead of the theoretical derivation, which will use measurement of the real time simulation duration, which is related to the number of instructions, to estimate the time complexity relative to other schedulers. Since this is a simulated system, we have to use wall time instead of simulated time for this metric. The results of these measurements will not be used to make claims about the exact theoretic time complexity of the algorithms. Instead, they will only be used for comparison between algorithms, requiring the assumption of simulation under similar load conditions on the same hardware. For that reason, this heuristic measure may not be as useful, but nonetheless it is easy to measure and can give some insight in the time complexity of the schedulers. Now, we will give the definition that will be used to measure the computational wall time for a single simulation:

Definition 34 *Let $t^\dagger$ denote a point in real time / wall time. Let $T_i^\dagger = t_{i,f}^\dagger - t_{i,s}^\dagger$ be the real time it took to have the scheduler algorithm perform its necessary calculations at simulated time step t_i, where $t_{i,s}^\dagger$ and $t_{i,f}^\dagger$ are the moments in real time where the algorithm started and finished operation respectively. Now, the computational wall time $t_{\text{total}}^\dagger$ can be defined as*

$$t_{\text{total}}^\dagger = \sum_{i=0}^{L} T_i^\dagger \,. \tag{3.26}$$

Operational complexity

In this section, the derivations of the time complexity of all the algorithms will be discussed based on their description in Section 2.5. An important point of consideration is that the algorithms described here are online, meaning that it may differ from derivations of the schedulers in off-line cases, where the whole queue is emptied at the end of the computation, instead of here, where the queue may be left filled for the next step dependent on the number of free resources.

The first algorithm is the FIFO scheduler as described by Algorithm 2. Let us consider each part separately.

1. First, a loop iterates over all new submitted jobs. If there are n new jobs, this will be $O(n)$.
2. Second, another loop iterates until either all resources are filled or there are no jobs left in the queue. Either way, this would mean a worst case scenario here is that the loop runs R times, the total number of resources in the hub, leading to a complexity of $O(R)$, but since R is a constant, this is just $O(1)$. Since jobs are already sorted by insertion order, this does not increase the complexity.

While the scheduler would also have to check for the available resources for the second loop, this is also just a constant $O(R)$, so it was immediately left out here and will be in the next schedulers. Now, we can add both contributions together. If we consider both the $O(n)$ and the $O(1)$ from both loops, we find that $O(n)$ will dominate, leading to a total complexity of $O(n)$, where n is the number of submitted jobs.

The second algorithm is the OD scheduler given by Algorithm 3, which is expected to be very low complexity, since it only has the bare minimum features required of a scheduler.

1. There is only 1 loop in this algorithm, which loops over the number of submitted jobs, so long as there are free resources. Worst case scenario is thus again $O(R)$, as it will not be able to activate more than R resources in one time step.

It only has 1 item here, because there is only 1 step for this algorithm if blocking is not an active action. This means we are left with a constant time complexity, $O(1)$, as the number of instructions of this algorithm does not scale with the number of jobs submitted to it.

The next algorithm will be the EDF scheduler as described in Algorithm 4.

1. First, a loop dequeues all expired jobs. If n denotes queue length, then the worst case scenario means that all jobs have to be dequeued, leading to $O(n)$ complexity.
2. The second loop submits all new jobs to the queue, same as for the earlier described FIFO scheduler, so using m for the number of new submitted jobs, we have complexity $O(m)$.
3. The last loop also exists in the FIFO scheduler, so we now we expect a $O(R)$ at least here, but this loop includes an extra step, where the job with the earliest deadline has to be taken from the queue. Assuming a FIFO queue[5], finding the earliest deadline per job will have a worst case of $O(n)$, leading to a total complexity of $O(Rn)$.

Taking all this information together, we can find the final complexity for the EDF scheduler. First though, we can say that $m \leq n$ since any job that gets submitted also gets enqueued, but may stay in the queue for more than one computation. Now, the worst case for the first two loops combined is $O(2n)$. Adding the final loop leads to $O((R+2)n)$, but since $R+2$ is just a constant, it can be discarded.

[5]Related to, but not to be confused with the FIFO scheduler.

The final complexity is thus $O(n)$, with n now being the queue size instead of the number of submitted jobs as with the FIFO scheduler. It should be noted that, if the number of free resources is very large, such that the number of times the third loops iterates is bounded by the queue length and not by the number of free resources, this would lead to a complexity of $O(n^2)$ instead, but since there are only a few resources considered here, this is not the case. This situation is then almost the same as a off-line EDF scheduler that would have to schedule all jobs in the queue. A min-max heap or other type of sorting would lead to a preferable $O(n \log n)$ in that case [52].

The EFDF scheduler is very similar to the EDF, but has additional steps in the last loop, as can be seen in Algorithm 5.

1. The first loop is the same as with the EDF, so we have a $O(n)$ complexity here.
2. The second loop is also the same as with the EDF, so using all the findings from there we also have $O(n)$ here.
3. Again, we have a loop over the resources with worst case $O(R)$, but now inside we have a loop that has to check for feasibility for each job to find the earliest feasible deadline. This leads to $O(n)$ for the first inner loop. However, if there is no feasible job in the queue, there is another search for an the earliest deadline without the check, so the worst case for both loops combined is $O(2n)$. This leads to a total of $O(R2n)$ for the third loop.

Taking this all together, we find a total of $O(2(1+R)n)$ complexity for all loops together, so a $O(n)$ complexity without the constants. If there are many resources however, it once again becomes preferable to sort the queue, leading to $O(n \log n)$ complexity [48].

Next, we will look at the MW scheduler as described in Algorithm 6.

1. The first loop is a loop over all the flows, which itself contains a loop over all jobs in that flow's queue. This means the outer loop gives a complexity of $O(\mathcal{F})$. The inner loop goes over the jobs in each flow queue, let us call this m, so we get $O(m)$ from there. Next, we have another loop that enqueues each newly submitted job again, so we use the previous analyses to also get $O(m)$ here. This gives $O(2m\mathcal{F})$ for the first loop. If we call $n = m\mathcal{F}$ the number of jobs across all flows, we can simplify this to $O(2n)$.
2. Now, the second loop itself again has a worst case of $O(R)$, but inside of it the flow with the longest queue has to be found. This has a complexity of $O(\mathcal{F})$, since it will have to check each flow, leading to a total of $O(R\mathcal{F})$ for the second loop.

$O(2n)$ can be simplified to $O(n)$ again. $O(R\mathcal{F})$ only contains constants, so finally we find $O(n)$ as a complexity for this scheduler.

The last scheduler is the RR scheduler as described by Algorithm 7.

1. This first loop is the same as with the MW algorithm, so we find a complexity of $O(n)$ here.
2. The second loop again loops over all resources in the worst case, and has an inner loop that activates the first job from the queue that is next in line. The tracking of which queue itself is next in line can be considered to be $O(1)$, since a simply integer that gets incremented would do the trick, but it also has to check if if the queue is non-empty, so a worst case scenario would be the scheduling having to check all flows. This leads to a total again of $O(R\mathcal{F})$.

Similarly to the MW scheduler, we now find a total complexity of $O(n)$.

3.2. Demand processing

While Section 2.4 made clear how demands were communicated, it does not actually explain how these are translated to individual jobs that are schedulable. This is an important step, and not necessarily trivial, as there are many choices that have to be made. For example, it is not clear if each packet of entanglement should run for as long as it has to until it is done or if it should be abandoned after some time. While this is certainly possible, without preemption this can create some situations where difficult packets have unfair access to the resources, since packet generation success is not guaranteed for a finite execution time. To do so, these demands can be translated to an internal representation for the scheduler: a packet generation task (PGT), as introduced by Beauchamp et al. [13]. This PGT

will the be divided in individual and independent packet generation attempts, which are attempts at generating a packet of entanglement and are the ”jobs” that are actually being scheduled [13]. While the components of both will be explained afterwards, the definition given by the authors is shown here:

$$\tau = \left(\left\{ (E, R_{\text{attempt}}, \rho)_r \right\}_{r \in \mathcal{R}}, t_{\text{minsep}}, t_{\text{expiry}} \right) . \tag{3.27}$$

Here, t_{minsep} and t_{expiry} are simply taken directly from the associated demand that is translated into it. τ itself is the tuple that represents a packet generation task. The tuple within this PGT is a possible realisation of a demand, which means it is a particular choice of how the PGAs will be executed [13]. $\mathcal{R}$ is the set of all possible realisations, and $(E, R_{\text{attempt}}, \rho)_r$, is a single realisation $r \in R$. So, a packet generation task has demand specific information, i.e. the expiry time of the demand and the minimum separation between each packet of entanglement, and all possible realisations of the demand, though this can simply be a set of one realisation in the simplest cases [13].

3 types of variables make up a single such realisation: E, the execution time that will be allocated to a PGA, R_{attempt}, the average rate at which PGAs are scheduled and ρ, the set of resources which are required to execute each PGA [13]. How E and R_{attempt} are calculated requires more information, which will be explained further on. While ρ can be more relevant in other contexts, it is not so much in this thesis, as the number of resources a single job can use is 1 at most, all the resources are identical, and, as will be explained in Section 3.3.2, since there will be no preemption of jobs, we do not expect jobs to change resources mid-execution. For E, it is important to mention that it will be used as if this E is actually the estimate of the execution time, $E^{\dagger}$, for example in the earliest feasible deadline first scheduler to see if a demand is still feasible. This E is, however, not the estimate of the average execution time of a job, it is the maximum time a job can take before being abandoned, but this means that it also can be seen as a worst case estimate.

3.2.1. Parameter selection process

To determine E and R_{attempt} for a PGA, a p_{packet} has to be known first. This value can either be set at some value, or determined via an algorithm [13]. For a p_{packet}, there is a trade-off between R_{attempt} and E: the longer the execution time, the lower the the required rate can be and vice versa, which can be a factor in selecting a p_{packet} [13]. For this thesis, they will be set at 0.8, as leaving it up to an algorithm caused this value to sometimes be unexpectedly small or large for some input parameters. Once a p_{packet} is selected, the execution time and the rate can be determined.

For the execution times it will use the window, w_j of the demand, the requested number of links s_j and the chance of generating a link between these nodes known to the hub, p_{succ}, to calculate the shortest time needed to generate such a link with probability p_{packet} [13]. This then becomes the maximum time a job gets access to the resources.

The calculation of the rate can go two ways. If a fixed rate, μ_j, was requested by the demand, R_{attempt} could simply be $\frac{\mu_j}{p_{\text{packet}}}$. If this was not the case, or $\mu_j = 0$, the hub can calculate using some algorithm that uses the supplied $N_{\text{inst},j}$, $t_{\text{expiry},j}$, the current time and t_{expiry} to determine some N_{min} such that scheduling N_{min} PGAs leads to $N_{\text{inst},j}$ packets being generated with an internal probability of $1 - \epsilon_{\text{service}}$. For this thesis, $\epsilon_{\text{service}} = 0.05$.

3.3. NetSquid

NetSquid is a ”NETwork Simulator for QUantum Information using Discrete events” [18]. It is optimised for simulating quantum network systems using a discrete event simulation engine to allow evaluation of quantum network and modular quantum computing systems [18]. It is a package for python[6] and was used in this thesis for the simulations of the quantum hub networks. NetSquid allows for modelling of many features of the network, such as the error models for the links and the quantum devices at the nodes, whose parameters will be discussed in Section 3.5.2 [18]. The modular components of the network can also be assigned protocols, which will be the main interest for this thesis, as the quantum hub’s scheduling algorithm will be implemented with such a protocol.

[6] Available at `https://netsquid.org/`.

As was already said, NetSquid uses discrete events to simulate quantum systems. What this means specifically, is that an even timeline exists, to which all events that have to be scheduled are added [18]. The time starts at 0 and jumps to the next point on the timeline at which there is an event scheduled. This event gets handled in an appropriate way, and often the handling of some events will schedule new events for handling further down the timeline. The simulation engine then goes going until all events have been handled. An example of this can be a scheduler for a quantum hub network with 1 flow, 1 resource and 1 submitted job. At $t_0 = 0$, the simulation starts. Then, a event is scheduled by a node for job submission for that job. Let's say that it is scheduled at t_1. Now, the time jumps to t_1 to handle the first-in-line event, which is the job submission. During handling of this event, a new one is created, as the hub now takes the job and puts it into an active resource, where events are scheduled for the entanglement generation. Each entanglement attempt is now a separate event and simulated with random variables based on the parameters of the link, and for each event there is a corresponding t_i to which the simulation jumps. After the entanglement has been generated, another event is scheduled at t_2 to signify the post-generation handling or processing for this situation. For example, this event could be communication from the hub telling the nodes that the entanglement was successful, so they can schedule more events to measure the entangled pair. This way, NetSquid becomes a dynamic timeline where constantly new events are being scheduled and handled [18].

3.3.1. Network Implementation

The network in this thesis is a hub network, specifically a simple star network, where each node is directly connected to the hub via a link. This means the network has 4 basic buidling blocks: a central hub node, the user nodes, classical communication channels and quantum communication channels. All links between the central hub and the user nodes have both a classical and quantum channel. This network was built with netsquid-netbuilder[7], which is a framework for making it easier to implement networks in NetSquid [53]. There are no repeater chains in this network.

Both the central hub node and the user nodes are quantum devices, but an important difference between them is that they are running different protocols, because they both have completely different purposes. The protocol for the user nodes focuses on sending out demands and receiving the results of these requests to the hub. While entanglement generation is a two-node process which does not need a distinction between sender and receiver depending on the protocol, for example the source-in-midpoint and detection-in-midpoint protocols for which this thesis is relevant [37], Netsquid-netbuilder always separates them, meaning there is always a division of roles necessary between the nodes. Only the sender node now has to submit a demand to the hub. Apart from this, all user nodes will be identical. The hub then has to schedule the incoming jobs and maintain this schedule according to the scheduler that is being simulated. The links between the nodes are all identical, as will be discussed in Section 3.5.2, except for their length.

3.3.2. Implementation restrictions

While Netsquid and netsquid-netbuilder allow the simulation to be done in the first place, there are a couple of restrictions on the implementation of the network and the simulation due to limitations and/or bugs in them.

The first restriction is that a single flow may not multiplex / multitask multiple entangled pairs at once. In practice, what this means is that the nodes may still create entanglement with multiple nodes at the same time, limited only by their communication qubits, so long as it is not with the same node.

The second restriction is that nodes may not send and receive entanglement requests at the same time. To avoid the problems that this created, the nodes are now either a sender node or a receiver node. This means that, if there are 10 nodes in the network, instead of 90 possible flows there are only 25, since senders cannot send to other sender nodes and receiver cannot receive from other receiver nodes. This does assume that there are the same number of senders as receivers, which will be true for this thesis. Another workaround could be considered, where each node now is split in two separate quantum devices, one of which is active when it needs to be a sender and one of which when it needs to be a receiver. This, however, was not implemented, because that benefits of this solution were not deemed worth its implementation and overhead when a sender-receiver model could also work.

[7] Available at `https://gitlab.com/softwarequtech/netsquid-snippets/netsquid-netbuilder`.

The last important restriction is that demands were not actually processed at the hub, but at the nodes instead. This is because netsquid-netbuilder does not natively support submitting a whole demand instead of separate jobs to the hub. While this could be implemented, it makes little difference where the translation of a demand into jobs happens if this is independent of the flow that scheduled it. An additional reason that this was not done, is because it makes sense to do so for the on-demand scheduler. Since it is the jobs that get scheduled, it is also the jobs that get blocked in the OD scheduler. This means that here it is the nodes who actually send individual jobs or PGAs to the hub, not entire demands. Since there is no obvious downside to this for the other schedulers, this approach was taken. In practise, this means that requests are already generated before the start of the simulation.

3.4. Demand parameters

In this thesis, two types of applications will be considered for the network. The types were introduced in Section 2.2, being quantum key distribution and blind quantum computing. These will be utilised in 3 types of use cases for the network:

1. Pure QKD
2. Pure BQC
3. A mixed case of QKD and BQC

These use cases are what specify what type of demands get submitted to the hub. In the pure QKD case, only QKD demands will be submitted. Similarly, in the pure BQC case only BQC demands get submitted. In the mixed case, $\sim$50% of the demands will be of QKD and the other $\sim 50\%$ will be of BQC. These demands themselves are randomly generated, based on certain parameters specific to each demand that will be explained in their corresponding sections in further detail.

The demand generation process itself is an iterative process, where demands are generated until a certain number of requested jobs is divided up into the type(s) of demands that were given by the use case. It does so by first calculating how many demands are required based on the use case to achieve this number of jobs. Then, it uses random number generation (RNG) to assign a number of required jobs to each demand. Since these are random, the total number of required jobs for all demands may differ from the total requested, so afterwards the requested number of jobs per demand is scaled with the ratio of total requested jobs by total generated jobs to get the desired total. To fill in all the other parameters for the demands, the following algorithm is used:

Algorithm 8 Demand generation process

while there are demands left to generate **do**
  generate a new Δt using an exponential distribution
  $t = t + \Delta t$ is the proposed demand submission time
  select a random flow from the available flows
  if there is no available flow **then**
    skip to next iteration
  end if
  Generate demand format based on RNG distribution and parameters, specific to each format entry
end while

The step for selecting a random flow is itself a process that has to take multiple factors into account. It shuffles a list of all flows in the network and then it has to check if none of the following is true:

1. Flow is not already busy with another demand.
2. Both nodes in the flow have at least one available communication qubit.
3. Flow is not among a list of some length of the last flows that were used.

The first check exists, because as explained in Section 3.3.2, a flow can only generate a single entangled pair at a time. The time a flow is considered busy here is the time from demand submission to demand expiry, the worst case scenario. The second check reflects the fact that nodes are responsible

for ensuring that there are communication qubits available for entanglement generation and not the hub. The third check is less exact, as the length of the list can be changed depending on what is needed, but its purpose is to prevent the same flows from getting all demands. Because the demand generation is an iterative process, this can happen if the time between submissions Δt is very small, meaning you reach a point where there are no available flows, because there are no more available communication qubits in at least one sender and receiver node. In this simulation, the nodes themselves are identical and the number of senders and receivers is equal, so If now a single flow finishes its demand, it is the only one that will be available for the new demand. Only if 2 or more demands are completed in between such a step will new flows get the chance to generate a demand. To alleviate this problem, 2 approaches could be taken. Either each flow gets a cooldown after a demand's expiry time is reached, or the last flows to generate entanglement are tracked in a list, and the flows in that list will not be available for scheduling, meaning that only as new flows get added to this list will it get pushed out so it is available again. The upside of option 2 here is that it forces the selection of new flows, but the downside is that it affects the random flow selection process in a way that is more difficult to quantise. Option 2 was chosen in this thesis with this trade off in mind, and to reduce the number of parameters that affect the random selection, as option 1 also introduces a new cooldown time that will have to be decided upon.

While the process of demand generation has now been explained, the parameters used for demand generation have not yet been. For that, the network capabilities will be explained first. Afterwards, the actual values for the used distributions and parameters will be given. It should be noted that while those parameters are not necessarily realistic for a specific instance of such an application, they are chosen to represent the difference between the two such types of applications, i.e. between a measure-directly and a create-and-keep application [34]. The specific values shown in this thesis were chosen based on the total simulation time and the usability of execution times[8].

3.4.1. Network capabilities

Users are not necessarily free to request any rate or fidelity that they may want, as the network can be limited in what it can offer to the users [13]. For this thesis, the most important factor here is the fact that nodes have different distances to the hub, and since a longer distance implies more photon loss, flows that have nodes further away from the nodes will have lower rates and fidelities that they can request from the network. Therefore, the hub needs to keep track of what combinitation of rate and fidelity it can offer to a flow at most [13]. If all these values for each flow are stored in a table, the user nodes can request them when creating demands for application execution, so they understand the maximum combination of rate and fidelity they can request to prevent their application from being impossible to successfully execute if these values are too demanding. This means that, for each demand d_j with a requested fidelity $F_{\text{min},j}$, the requested rate μ_j of jobs per second must be such that $\mu_j s_j \leq \mu_{\text{max}}$, where s_j is the number of pairs required per job for this demand and μ_{max} is the maximum rate of pair generation that a flow can request for this fidelity and vice versa for a certain requested μ_j there will be a certain F_{max}, the maximum fidelity it can now request.

For this thesis, only a single pair of fidelity and rate were generated for each flow, though one could generate much more values to fill in the table with many more options.

3.4.2. QKD demand parameters

To generate a QKD demand, the following distributions and parameters are used:

1. For the gap between two possible submissions, Δt, a exponential distribution was used with a rate parameter of $\lambda = 0.005$ s^{-1}.
2. For the number of pairs per packet, s_j, it is always 1 in QKD.
3. There is no window in QKD, as there is only 1 entangled pair needed per job.
4. The minimum separation for QKD is also not needed, as the generated pair is measured directly after being generated. Nonetheless, it has been set to a time of 10 ns to prevent unexpected problems in the simulation.

[8] Usability here does not refer to accuracy, but to the fact that sometimes execution times could be exceedingly long or impossibly small, and the combination of factors here should prevent such things from occuring.

5. The number of jobs per demand is given by a normal distribution with a mean of 1000 and a standard deviation of 200.
6. For the requested rate of a QKD demand, a random exponentially distributed variable is requested with a $\lambda = 10$. This number is then subtracted from 1 and this resulting scaling factor (with a minimum of 0.1) is multiplied with the maximum rate from the network capabilities table.
7. The fidelity is generated similarly to the rate, but now with $\lambda = 20$ and a minimum of 0.5, as requesting a fidelity lower than 0.5 would not make sense.
8. The expiry time is not randomly generated. Instead, it is calculated by $2N_{\text{inst},j}/\mu_j$, where $N_{\text{inst},j}/\mu_j$ is a rough estimate of the duration of this QKD demand and the factor of 2 gives more elasticity for the execution of this demand.

3.4.3. BQC demand parameters

For BQC demands the distributions are the same as the QKD demands if they were specified there, but now with these parameters:

1. Δt now uses a rate parameter of $\lambda = 0.01$ s^{-1}.
2. s_j is now given by a uniform distribution between 2 and 10.
3. The window w_j for BQC is given by a normal distribution with a mean of 400 ms and a standard deviation of 10 ms. This value is then multiplied with s_j.
4. The minimum separation time is now also normally distributed, with a mean of 1 ms and a standard deviation of 0.1 ms.
5. The number of jobs per demand now has a mean of 100 and a standard deviation of 20.
6. The requested rate of a BQC is the same process as QKD, except now it is divided by s_j at the end to reflect the fact that a demand's rate is for the number of jobs per second, whereas the rate of the network capabilities does not specify a single type of jobs and is thus given in pairs per second.
7. The fidelity is exactly the same as for QKD.
8. The calculation of expiry time is now slightly different. For BQC it is calculated by $3N_{\text{inst},j}(t_{\text{minsep}} + \frac{1}{\mu_j})$, to take into account the fact that now there is a small waiting time in between two jobs. The factor in front is now slightly longer to reflect the fact that due to the need for multiple pairs to be generated within a time window, it may take longer to complete the number of required jobs.

3.5. Simulation

As said previously, each single simulation will use 10^4 jobs per simulation. However, there will actually be more jobs submitted to the hub, since in the step of translating a demand to a packet generation task, the minimum number of instances that need to be submitted to reach $N_{\text{inst},j}$ successes within the expiry time gets calculated, which leads to a total number higher than 10^4. Each simulation consists of 100 runs. Based on the total number of jobs that will be submitted and the demand parameters given in the previous section, QKD will on average have about 10 demands per run, whereas BQC will have about 100.

The process of demand generation was shown in the previous section with Algorithm 8. Afterwards, these are translated into PGTs before the simulation starts, as this translation process does not actually need any information available only after the simulation starts. These PGTs are than translated into individual jobs, also before the simulation starts and they are now ready for submission. For most schedulers, these jobs then get submitted to the hub at the same time to simulate the arrival of a single demand, where the scheduler has complete control over if and when these jobs are scheduled. This is not the case for the on demand scheduler, however, as that scheduler does not have a queue. Submitting all jobs at the same time would lead to all but one being blocked, and does not reflect accurately how a real implementation of a on demand scheduler in such network would work. Instead, nodes in a on demand simulation keep submitting jobs until their demand has been completed or the demand expires, with each submission being a random number generated with the exponential distribution based on the requested rate, such that the arrivals model a poisson process. This does

mean that on demand already has a disadvantage compared to the other schedulers, other than the fact that it has no queue to store jobs: if a job finishes earlier in another scheduler, the next job for that demand can start early, but in the on demand scheduler this job may not have been submitted yet, so this demand will be idle until the next job gets submitted.

The schedulers were validated by simulating small scale scheduling problems with a known outcome, such as a case were all jobs were submitted from the same flow, or all flows submitted a different number of jobs at certain times. These were used to ensure the schedulers behaved as expected.

3.5.1. Network load

In addition to the type of scheduler and use case, the network load was also varied. To emulate the business of a network, 3 types of network load were defined and varied across the simulations. Each of these load parameters were varied to either ”high” or ”low” while the other 2 were kept at ”medium” for a total of 7 network load conditions. The process for defining what a medium load was for these, was by inspecting the demand completion rate and the job queue length. The parameters were set such that the job queue length barely didn't converge and the demand completion rate was just barely lower than 100 %[9] and this was repeated 10 times for a set of parameters. Due to the batch submission, the relatively short simulation of a simulation[10] and the random nature of the simulation, determining this exact point is imprecise, if not impossible since such a single point may not exist and be more accurately be described as a region instead. With this in mind, these estimates allow some degree of network load control to analyse the performance of the schedulers in different load conditions. The medium load parameters are all specified in Section 3.4.

The first load parameter is the submission load. In a busy network with a high load, demands are submitted relatively close to each other. How close these demands can be is limited by how many flows there are and the number of communication qubits there are available for each node, since no demands can be submitted if there are no flows available. The high load in this case is a decrease of Δt by half, and a low load is the doubling of it. A high (low) load scenario for this parameter may be referred to as a high (low) submit load in this thesis as a shorthand.

The second load parameter is the number of jobs per demand. If the number of jobs is increased while the submission time is kept the same, the scheduler will have to schedule more jobs in the same time, meaning it is expected to achieve a similar effect as the first load parameter. The high load here is a doubling of the number of jobs per demand, whereas low load is dividing it by 2. A high (low) load scenario for this parameter may be referred to as a high (low) request load in this thesis.

The third load parameter is the expiry load. In a busy network, pressure is put on the hub by demands that are reaching their expiry time. If these aren't met in time, the demand completion rate will drop as a result. Unlike the other parameters, it is not expected that this parameter behaves in such a way that queue lengths start diverging, as instead queues get shorter if a demand expires, since the expired jobs are taken out. This parameter doesn't actually increase the network load in the sense that it has to handle an increased number of jobs, but rather it could be more accurately described as increasing the pressure on the scheduler to complete the demands before expiration. Therefore, it is expected that this load parameter may act differently from the others. Here, a high load is a halved expiry time, and a low load is a doubling of the expiry time. A high (low) load scenario for this parameter may be referred to as a high (low) expiry load in this thesis.

Later on, these loads will be represented by a single tuple that gives the used network load for a single simulation: (x, y, z). Here, each variable x, y and z can be either 'l' (low), 'm' (medium) or 'h' (high). The first variable x represents the request load parameter, y the submit load parameter and z the expiry load parameter.

This gives a total of 12600 runs based on the 126 simulations due to the different schedulers, use cases and load conditions.

[9] Due to random chance in quantum mechanics, a 100 % success rate can not be guaranteed. For larger simulations, it becomes very unrealistic to expect it. The parameters were chosen such that it was the point just before the demand completion rate dropped below ~ 95 % on average.

[10] A simulation of QKD with 10000 jobs has 10 demands, while there are 25 flows.

3.5.2. Physical parameters

The physical parameters for the nodes and the links are based on work by Pompili [54] and Avis et al. [55]. All the nodes and links have the same parameters, except for the distance between the nodes and the hub, so the network is homogeneous. These parameters are for the qlink.heralded_single_click and qdevice.nv in the netsquid-netbuilder package respectively.

The parameters for the single click quantumlinks are as follows:

1. p_loss_init - probability that photons are lost when entering the fibre: 0.01.
2. p_loss_length - attenuation coefficient: 0.2 dB/km.
3. speed_of_light - Speed of light in the fibre: 2/3 c = $2 \cdot 10^5$ km/s.
4. dark_count_probability - Probability of a dark count: $1.5 \cdot 10^{-7}$.
5. visibility - Measure of photon distinguishability, based on Hong-Ou-Mandel visibility [56].

For the nodes, or quantum devices, the following parameters were used using a diamond nitrogen vacancy center model:

1. num_qubits - number of communication qubits: 2[11].
2. eletron_init_depolar_prob - Error probability during initialisation of the electron: 0.01.
3. prob_error_0 - Probability of measuring a 1 instead of 0 :0.07.
4. prob_error_1 - Probability of measuring a 0 instead of 1 :0.005.
5. carbon_init_depolar_prob - Probability of error during initialization of carbon memory qubit:0.02.
6. carbon_z_rot_depolar_prob - Probability of error during a single gate operation on the carbon atom:0.002.
7. ec_gate_depolar_prob - Probability of error during a two qubit gate operation between the electron and the carbon atom:0.002.
8. electron_T1 - Longitudinal relaxation time of the electron: 1 h.
9. electron_T2 - Transverse relaxation time of the electron: 0.1 s.
10. carbon_T1 - Longitudinal relaxation time of the carbon atom: 10 h.
11. carbon_T2 - Transverse relaxation time of the carbon atom: 0.1 s.

All other paramaters not listed here were left to their default value, generally because they were not relevant or used in this thesis.

[11] In the code, 3 communication qubits are used, but during the demand generation it acts as if there are only 2 per node. This is because there seems to be a problem where a qubit may get stuck, halting future jobs as well, so an extra qubit is introduced in a way that does not allow the nodes to use 3 at once.

4

Results

In the introduction, Algorithm 1 introduced at a high level a method for selecting a scheduler in a quantum network with a centralised control structure. Based on the previous chapters, Algorithm 1 can be updated to be more specific. In its original formulation, step 3 was quite vague, as it left many aspects undefined which need to be specified for a potential scheduling problem or simulation to be executed. In Algorithm 9, a more detailed version is stated.

Algorithm 9 Scheduler Selection Process for Centralised Quantum Network

Step 1: Choose schedulers for consideration
Step 2: Choose metrics for consideration
Step 3: Set up scheduling problem to represent the quantum network.
1. **Step 3.1** Choose a network architecture that best represents the desired network, if it can practically not be implemented in the problem or simulation as a whole directly.
2. **Step 3.2:** Choose a method of spawning individual jobs, either by spawning jobs directly or processing relevant use cases as demands into them.

Step 4: Simulate/compute results
Step 5: Use results to select scheduler as desired for the network

This algorithm enables a method for scheduler selection in centralised quantum networks, to facilitate network operators in choosing a scheduler for their network. Note that the category of quantum networks with centralised control encompasses many possible implementations, hence Algorithm 9 has a broad domain of applicability. In the rest of this section, we narrow the domain of application to focus on an implementation of the Quantum Hub Network, as introduced in Chapters 1, 2 and 3. This chapter exemplifies a use of Algorithm 9. In the evaluation results from simulation, discussed in the rest of this chapter, we demonstrate that it can be used to make a well-motivated selection for the scheduler of a centralised quantum network.

4.1. Simulation results

In Sections 3.1 and 3.1.1 of Chapter 3, we motivated that the proportion of completed demands is the most important metric for evaluating the performance of a quantum hub scheduler. In Section 3.4, we demonstrated that the demand parameters may differ substantially between demands for different types of applications, hence the applications corresponding to a demand may affect the proportion of demands completed. In Section 3.5.1 of Chapter 3, we further argued that the network load has a large impact on the performance of the network, and that the specific implementation of the network load may cause these performance results to vary. For the purpose of assessing the various schedulers implemented, we first compare them based on the proportion of demands completed under a variety of load conditions and application use cases. In Figures 4.1, 4.2 and 4.3, the proportion of completed demands is plotted against the network load condition, separated for each use case, BQC, QKD and the mixed use case respectively. The standard error of the simulations is included as error bars, but

the error is often too small to be visible in these graphs, indicating these estimates of the mean have a high precision. The first thing that becomes clear from these graphs is that the demand completion of on-demand schedulers is always lower than for the the other schedulers, except for one single load condition. In the case of high network load regarding the expiry load parameter, the case denoted by the tuple ('m', 'm', 'h'), the demand completion rate even reaches 0 for the QKD on-demand simulation, meaning not a single application was executed successfully. A high level feature of these results is that as network loads decrease, i.e. any network load with a 'l' or low load parameter, the demand completion increases, whereas increasing the load, i.e. any simulation with 'h' or high, decreases the overall demand completion. This trend is consistent with the expectation that schedulers perform better when the network is not overloaded.

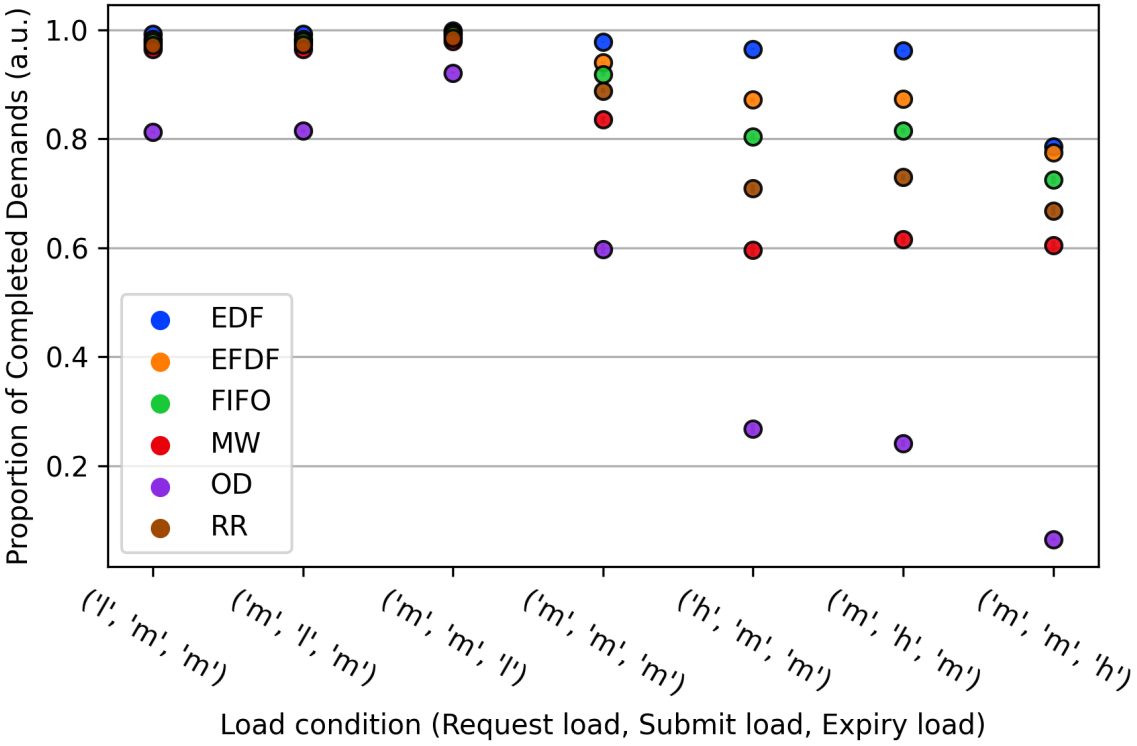

Figure 4.1: Graph showing the proportion of completed demands for each BQC simulation, where the x-axis shows the different load conditions for the network, described by a tuple of the three network load parameters. 'l' stands for low network load for that parameter, 'm' for medium load and 'h' for high load. The standard error is plotted as error bars, but it is too small on this scale to be easily visible.

The graph for the BQC simulations, Figure 4.1, shows that the EDF scheduler always has the highest demand completion. This may be difficult to see in the low network load conditions, because in that regime the results for the different schedulers are so close together. However, this regime is the least interesting for assessing the performance of a scheduler, since schedulability is not a problem in the low load regime. In the medium and high network load regimes, there is clearly a distinct and set order of the schedulers: EDF performs best, followed by the EFDF scheduler, then the FIFO scheduler, the RR scheduler and the MW scheduler. As stated earlier, the OD scheduler performs worst each time. In this figure (Figure 4.1), the spread of the schedulers according to the proportion of demands completed also becomes larger as the load increases. Under low load, ignoring the OD scheduler, all data points are within a range of 0.1. Under high load, however, this range is increased to between 0.2 and 0.4. This indicates that as the load of the network increases, the difference in which scheduler is being used starts to matter more, as differences become more pronounced. In this figure, the simulations that vary request load and submit load, are nearly indistinguishable from each other and vary only slightly from each other. The expiry load on the other hand seems to have a more extreme effect on the demand completion. Relaxing it increases the proportion of demands completed more than in the other low load

scenarios and making this load more demanding decreases the demand completion more than it does in the other scenarios.

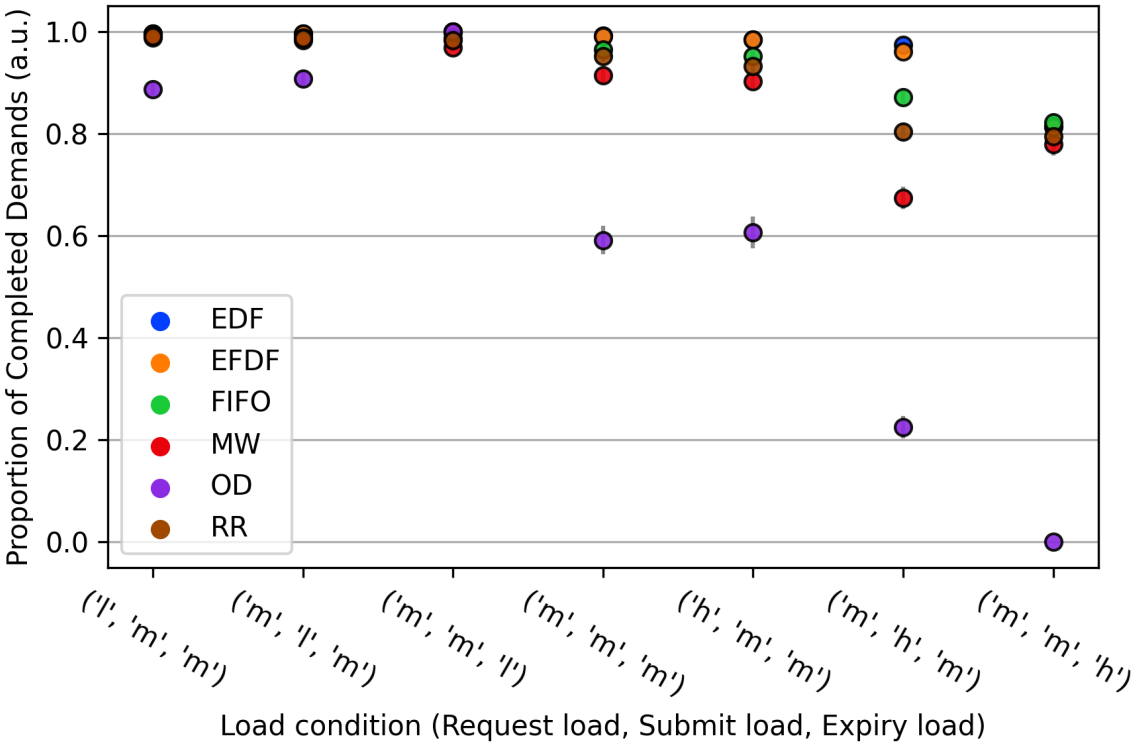

Figure 4.2: Graph showing the proportion of completed demands for each QKD simulation, where the x-axis shows the different load conditions for the network, described by a tuple of the three network load parameters. 'l' stands for low network load for that parameter, 'm' for medium load and 'h' for high load. The standard error is plotted as error bars, but it is too small on this scale to be easily visible.

In Figure 4.2, which demonstrates these results for QKD applications, many schedulers are clustered together in the proportion of completed demands; in all but one load condition the schedulers are spread in a range less than 0.1, apart from the OD scheduler. This figure also shows the only network load condition where the EDF scheduler did not have the highest demand completion rate. In the case of high expiry load, ('m', 'm', 'h'), the FIFO scheduler actually has the highest demand completion. It is not entirely clear from the figure alone which scheduler has the highest demand completion in the other cases because of the overlap, but based on the numerical data, it is the EDF scheduler in all other cases. However, since the differences here are not very large apart from the high submit load regime, it suggests the distinction between the schedulers may not be as important for QKD use cases. This graph also contains the only load case where the OD scheduler did not perform worse than all the other schedulers regarding demand completion. This occurred in the case of low expiry load. In the QKD simulations, increasing the request load from medium to high does not impact the proportion of completed demands significantly. However, like the BQC use case, the order of best to worst scheduler in QKD still seems to be preserved in some cases.

Figure 4.3 compares the performance of the different schedulers in terms of the proportion of completed demands against the network load conditions for the mixed use case, where 50 % of the demands are QKD and the other 50 % are BQC. Once again, OD performs the worst across the board in these results and EDF the best. The order of EDF, EFDF, FIFO, RR, MW and OD for best to worst scheduler holds in most cases here, though like the QKD use case, many load conditions show small ranges within which the proportion of completed demands under the various schedulers lie. Comparing all three figures (Figure 4.1, 4.2, and 4.3) with each other, we can see that the high load conditions of the mixed load resemble a mix of the differences between QKD and BQC. In BQC, a high request load was not distinguishable from a high submit load, whereas in QKD the high request load was not distinguishable

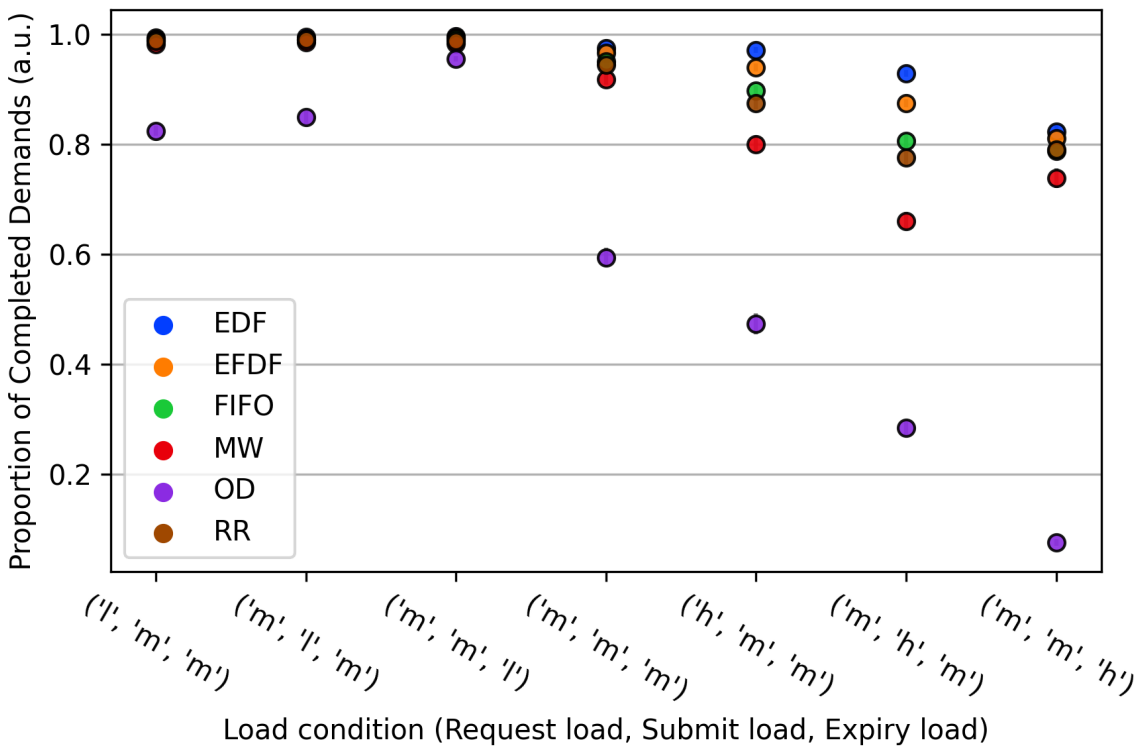

Figure 4.3: Graph showing the proportion of completed demands for the simulations with 50 % QKD and 50 % BQC demands, where the x-axis shows the different load conditions for the network, described by a tuple of the three network load parameters. 'l' stands for low network load for that parameter, 'm' for medium load and 'h' for high load. The standard error is plotted as error bars, but it is too small on this scale to be easily visible.

from the medium network load. In the mixed case, the spread of schedulers resembles a mix of the results from the BQC and QKD cases, because in that case the score of proportion of completed demands for each scheduler in the high request load is in between the medium network load and the high submit load scores. Taking these results together then answers the first three research questions that were posed in Chapter 1. Regarding the third question, we have seen that for each use case, the EDF scheduler performs the best in demand completion. Similarly within each section, for all but one scenario did the EDF scheduler perform the best in demand completion, answering the second question as well. This means EDF performs the best overall as an answer to the first question. The only scenario where the EDF does not beat the others is the high expiry load with a QKD use case.

While the demand completion is considered the overall most important metric, the other lower-level metrics allow us to understand the results more fully and may be important to users or network operators in certain contexts. Radial plots allow the comparision of many such metrics at once, hence in Figures 4.4, 4.5 and 4.6 radial plots are shown to summarise the simulations. Each figure corresponds to one combination of load condition and use case, and each radial plot represents a scheduler's performance within those conditions. The standard error is again shown with error bars. All of these figures contain a high load scenario, though the use case and which specific parameter is in high load changes between them. Each radial plot contains 7 metrics to gauge scheduler performance: the proportion of completed demands (PCD), the average active throughput per flow (AATF), the resource fairness index (RFI), the average response time (ART), the average utilisation factor (AUF), the proportion of completed jobs (PCJ) and the busy time (BT). The average response time and busy time are the only metrics that are desired to be low in high load scenarios. The metrics themselves are normalized with the highest value for that metric observed under any scheduler plus its standard error within that figure, to scale all metrics to a range from 0 to 1. Indices and factors that are already normalized are left unchanged. Each figure has its own normalization, so comparisons made in between figures need to take that into account. For this purpose the normalization factors are quoted in the figure captions.

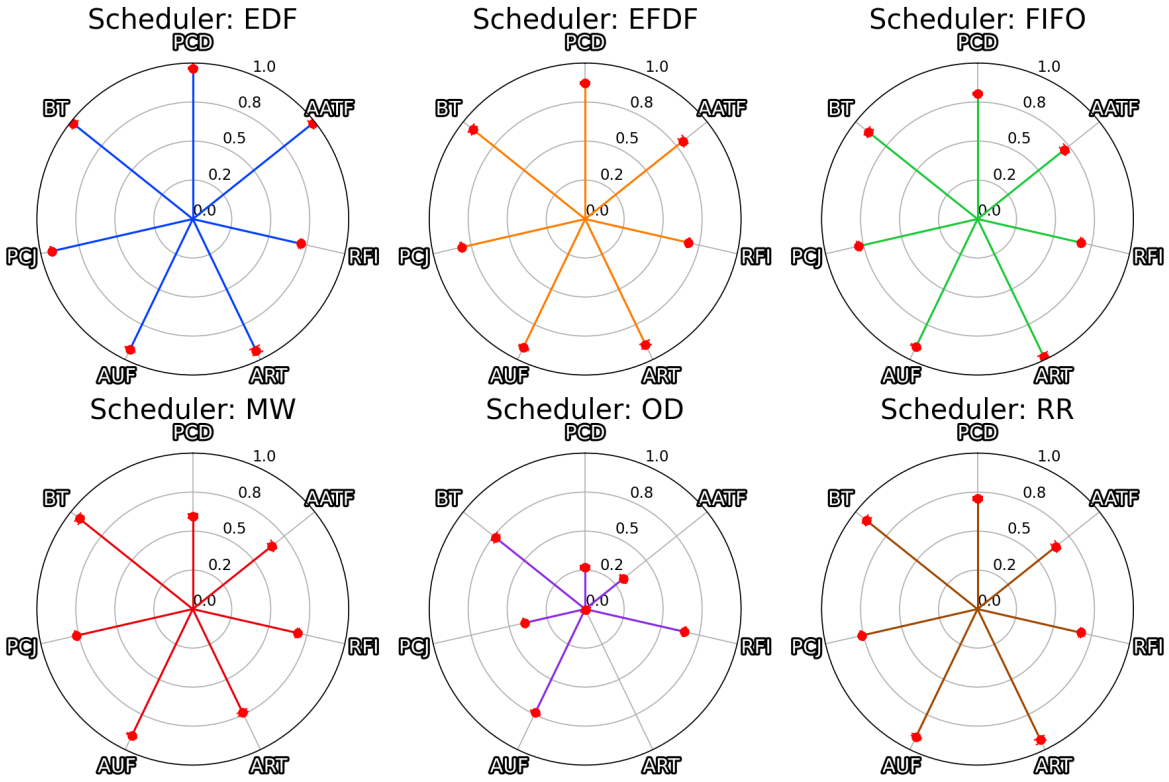

Figure 4.4: Radial plots comparing schedulers according to multiple metrics in separate directions. Each radial plot corresponds to a single simulation of scheduler type, use case, and load condition. This figure is for BQC simulations in the high request load regime and shows all the schedulers in a separate radial plot. Going in clockwise order and starting at the top, the metrics are the proportion of completed demands (PCD), the average active throughput per flow (AATF), the resource fairness index (RFI), the average response time (ART), the average utilisation factor (AUF), the proportion of completed jobs (PCJ) and the busy time (BT). The values are normalized with 0.519 jobs/s for the AATF, 398 s for the ART and 17.4 s for the BT. All other metrics are already normalized between 0 and 1. The graphs include an error bar for the standard error.

In Figure 4.4, the radial plots show the performance of the different schedulers with a BQC use case in a high request load scenario. From the graph, it becomes immediately clear that the EDF scheduler in this load still has a high (>0.9) demand completion, despite the other schedulers starting to struggle more to keep up with the increased network load. As expected, the OD scheduler performs much worse in this regard, and has a demand completion of about 0.2. The EDF scheduler also performs clearly better in the active throughput per flow, which may be an important reason why it performs so well in the demand completion. Regarding resource fairness, there are no large differences, as all schedulers have a resource fairness index within 0.05 of 0.8, including the OD scheduler. The response time is the lowest in the OD scheduler, as expected, since is it has no queue, so the only contribution here comes from the execution time, which does not deviate strongly between the schedulers. This also indicates that the waiting time is a much more significant portion of the response time than the execution time in high request load. Interestingly, despite perfoming not as good as the other schedulers in the other metrics, the MW scheduler does perform the best in the response time if you exclude the OD scheduler. The FIFO scheduler has the longest response time. The average resource utilisation factor is high (>0.9) for all schedulers but OD. The job completion reflects the trend in the throughput and the demand completion: EDF scores the highest, all other schedulers with a queue perform somewhat worse and the OD scheduler performs much worse than the others.

Figure 4.5 shows the radial plots for the QKD use case in the high submit load scenario. Again, we see that EDF scores the highest in the demand completion, though now EFDF is close behind, differing less than 0.05 with the EDF scheduler. When comparing the two radial plots, EFDF and EDF perform very similarly in all metrics in this scenario, including the throughput and the demand completion, where they differed more in Figure 4.4. All schedulers have a higher active throughput in these QKD simulations than in the BQC simulations from Figure 4.4, not just relative to the top performer of their respective

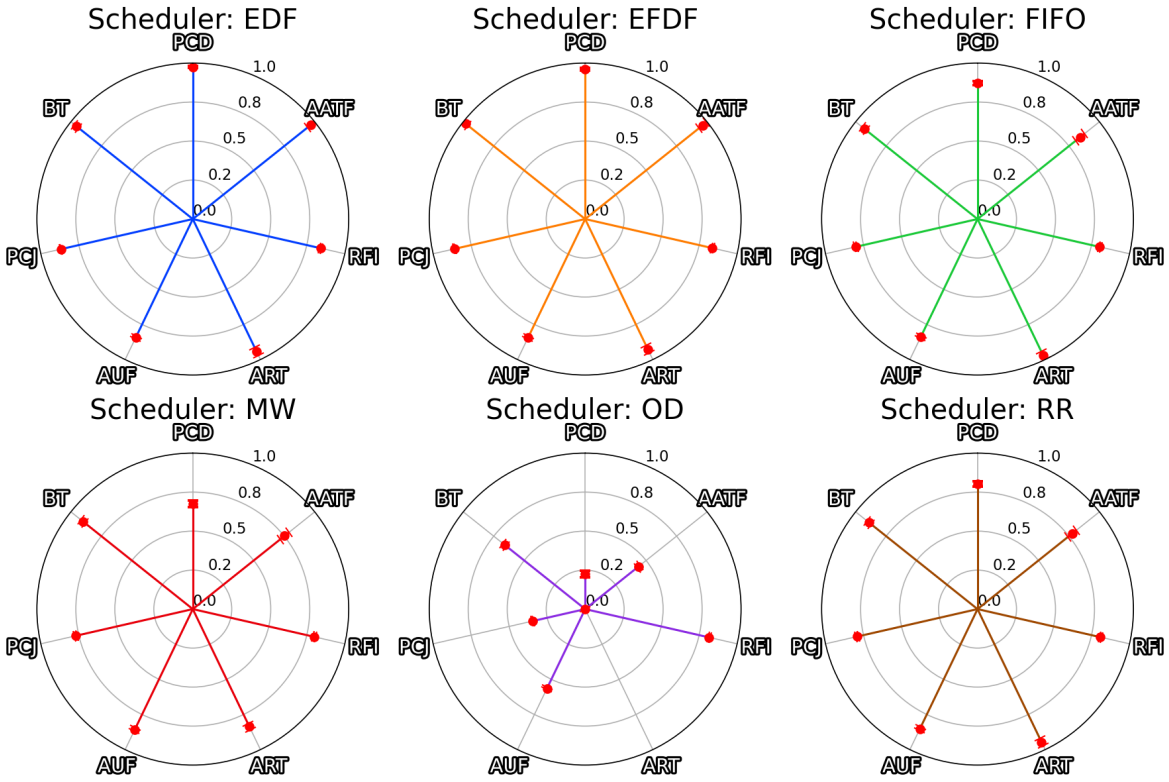

Figure 4.5: Radial plots comparing schedulers according to multiple metrics in separate directions. Each radial plot corresponds to a single simulation of scheduler type, use case, and load condition. This graph is for QKD simulations in the high submit load regime and shows all the schedulers in a separate radial plot. Going in clockwise order and starting at the top, the metrics are the proportion of completed demands (PCD), the average active throughput per flow (AATF), the resource fairness index (RFI), the average response time (ART), the average utilisation factor (AUF), the proportion of completed jobs (PCJ) and the busy time (BT). The values are normalized with 3.78 jobs/s for the AATF, 247 s for the ART and 25.0 s for the BT. All other metrics are already normalized between 0 and 1. The graphs include a error bar for the standard error.

sets, but also in absolute terms, since the value used for normalisation in Figure 4.5 is 3.78 jobs/s and in Figure 4.4 it is only 0.519 jobs/s. Though that they have a higher throughput in absolute terms makes sense, given that QKD applications have less demanding individual jobs than BQC applications. The resource fairness in QKD simulations is slightly higher than the fairness of the BQC simulations for all the schedulers. Again, the FIFO scheduler has the longest response time of them all. The average utilisation factor is lower in QKD simulations across the board when compared to the BQC simulations.

In Figure 4.6 we can see mixed use case simulations in the high expiry load. Here, it is interesting to note that the MW scheduler performs better than in the case of the individual QKD simulations in high submit load from Figure 4.5 and BQC simulations in high request load from Figure 4.4, despite the other schedulers performing either the same or worse in the mixed case. Except for the OD scheduler, all schedulers now have similar throughputs that fall within a range of 0.05 of each other (normalised). The resource fairness is now slightly lower than the QKD case, but similar to the BQC simulations. The average response times of all but the OD schedulers now differ very little, again within a range of 0.05 of each other (normalised). In the mixed use case with high expiry load, the average utilisation of the resources is lower than the other two cases from Figure 4.4 and Figure 4.5. Barring the OD scheduler again, the values for the job completion are almost identical. Here, there is a large difference between the busy times of the busiest schedulers, as now the OD scheduler has only half the busy time of the EFDF scheduler, who has the highest busy time. While the OD scheduler also had a lower busy time in the other two cases, it only had a time lower by about a quarter in those cases.

Overall, across all the 21 combinations of use cases with network loads and all the 22 metrics that were considered for optimality, EDF is optimal for 57.1 % of the total 462 metrics across all cases. OD is optimal in the second most, with 25.8 %, though these are all regarding metrics such as busy time

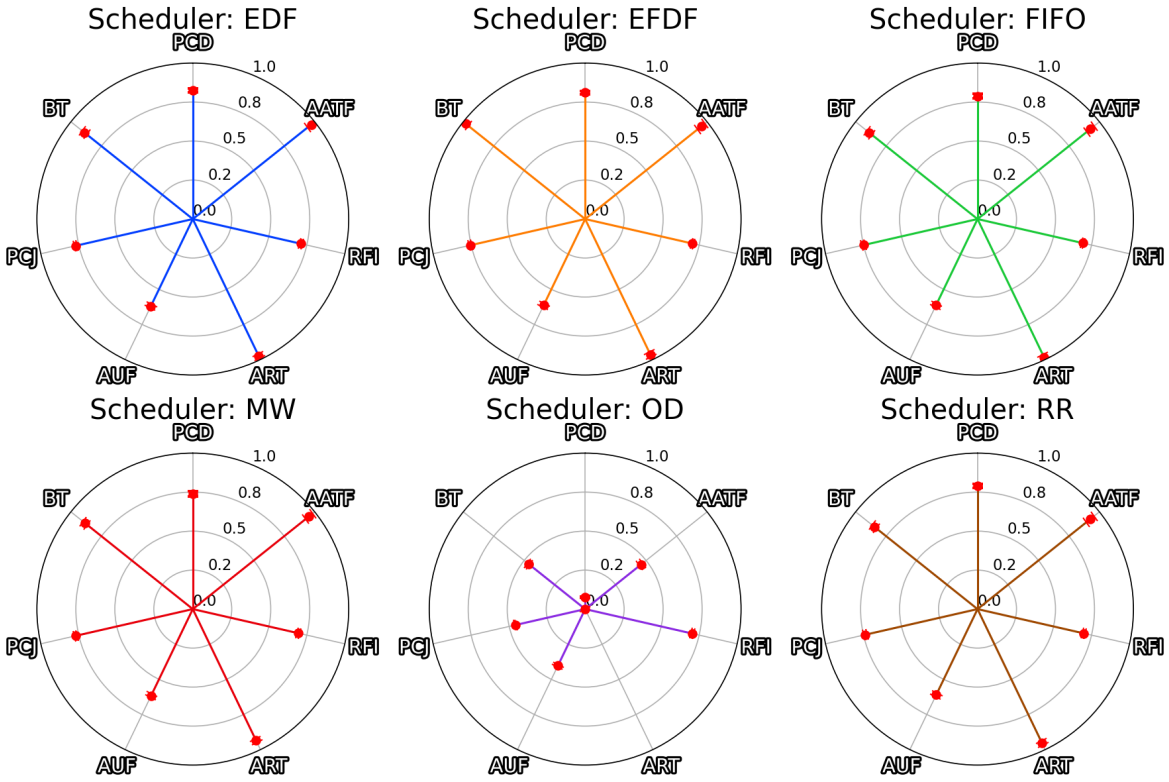

Figure 4.6: Radial plots comparing schedulers according to multiple metrics in separate directions. Each radial plot corresponds to a single simulation of scheduler type, use case, and load condition. This graph is for mixed use case simulations in the high expiry load regime and shows all the schedulers in a separate radial plot. Going in clockwise order and starting at the top, the metrics are the proportion of completed demands (PCD), the average active throughput per flow (AATF), the resource fairness index (RFI), the average response time (ART), the average utilisation factor (AUF), the proportion of completed jobs (PCJ) and the busy time (BT). The values are normalized with 2.73 jobs/s for the AATF, 162 s for the ART and 22.7 s for the BT. All other metrics are already normalized between 0 and 1. The graphs include a error bar for the standard error.

and due to its low complexity and waiting time due to its lack of queues. These results do not show everything, however, as it is also of importance to understand how well other schedulers due if you look at the second most optimal scheduler. A scheduler who performs second best most of the time and best some times may be more interesting than a scheduler who performs best often but also performs worst in other metrics. The EFDF scheduler is second most optimal the most often, as it got 42.0 % of the total, though most of these second places were behind the EDF scheduler. After EDF now comes the MW scheduler, who is second most optimal 24.0 % of the times.

In Figure 4.7 the relation between the average active throughput and the demand completion is shown for the BQC simulations. The on-demand scheduler is not shown here, as it is not within the bounds of this graph. The other use cases have similar graphs, but only the BQC use case is focused on to make the relationship between the two metrics more clear. High network load scenarios are located in around the bottom left, low load scenarios at the top right and the medium load scenario is in the middle. Here, an interesting pattern emerges, as each network load forms a curve, and the order of the schedulers within this curve is always the same: MW, RR, FIFO, EFDF and lastly, EDF. Each step in these curves is usually a step to the right and/or up, but not so for the high expiry load scenario. There, MW has a significantly higher throughput than the RR scheduler, but it has a lower demand completion. It is not clear why these arcs appear in the relationship between the demand completion and active throughput based on just this data. Interestingly, the high expiry load scenario shows a curved arc that first decreases in throughput as demand completion increases and later increases again. This means that despite achieving a higher active throughput in this scenario, the MW scheduler still performs worse than the RR and FIFO scheduler, suggesting that its scheduling algorithm makes decisions that decrease the chance of demand success compared to the other schedulers.

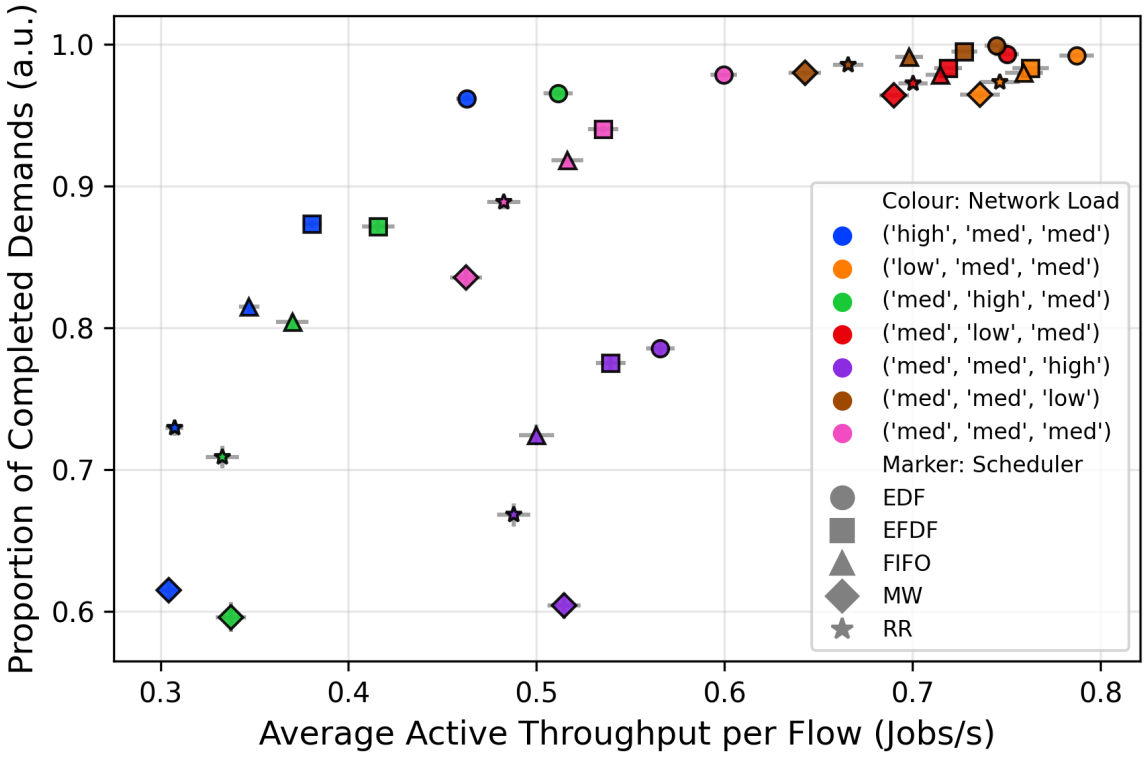

Figure 4.7: Plot showing BQC simulations with the OD scheduler excluded. The x-axis displays the active throughput averaged over the flows and the y-axis shows the proportion of copmleted demands. The network load is given by a tuple of the network load parameters. The graph includes the standard error of the mean as error bar.

4.2. Discussion

4.2.1. Error analysis

It is important to understand how accurate and precise your results are when performing these simulations, since this is a major consideration for how many runs one must do in a single simulation. For this, the standard error can be used, which is an estimate of how precise the estimate of the mean is for a sample. To see if the choice of 100 was enough in this thesis, an error analysis was done with this standard error. The results of this are shown in Figure 4.8. Here, the standard error of the mean is displayed for taking different proportions of the total data available. It shows, as expected, that taking more data decreases the standard error. It also shows a decreasing pay-off from taking more and more data, which is expected as well. The pattern of the gaps in between the data batches reflects the scaling of the SEM with $\frac{1}{\sqrt{n}}$, and the differences between the metrics reflect the fact that the estimate of the standard deviation itself is also not perfect and changes as more data gets added. This indicates that increasing the number of runs with a large amount will have diminishing returns, as the increase in precision gets smaller the more runs are already done. It is worth noting that while the standard error decreases with more runs, the standard deviation of the sample remains considerable, which indicates considerable variability in individual outcomes. This may be due to the lower number of demands in some runs, as always $\sim$10000 requests were divided between the demands based on their parameters, so runs in the QKD use case only had about $\sim$10 demands per run. If the number of demands in a single run is relatively low, its individual parameters are likely to have more impact on the average result, so adding more demands and request to each run may decrease the variability of a single simulation parameter set.

4.2.2. MaxWeight performance

The MaxWeight scheduler is classically optimal for throughput, however in both the case of total throughput and active throughput, EDF outperforms. In fact, even the EFDF and FIFO outperform it multiple

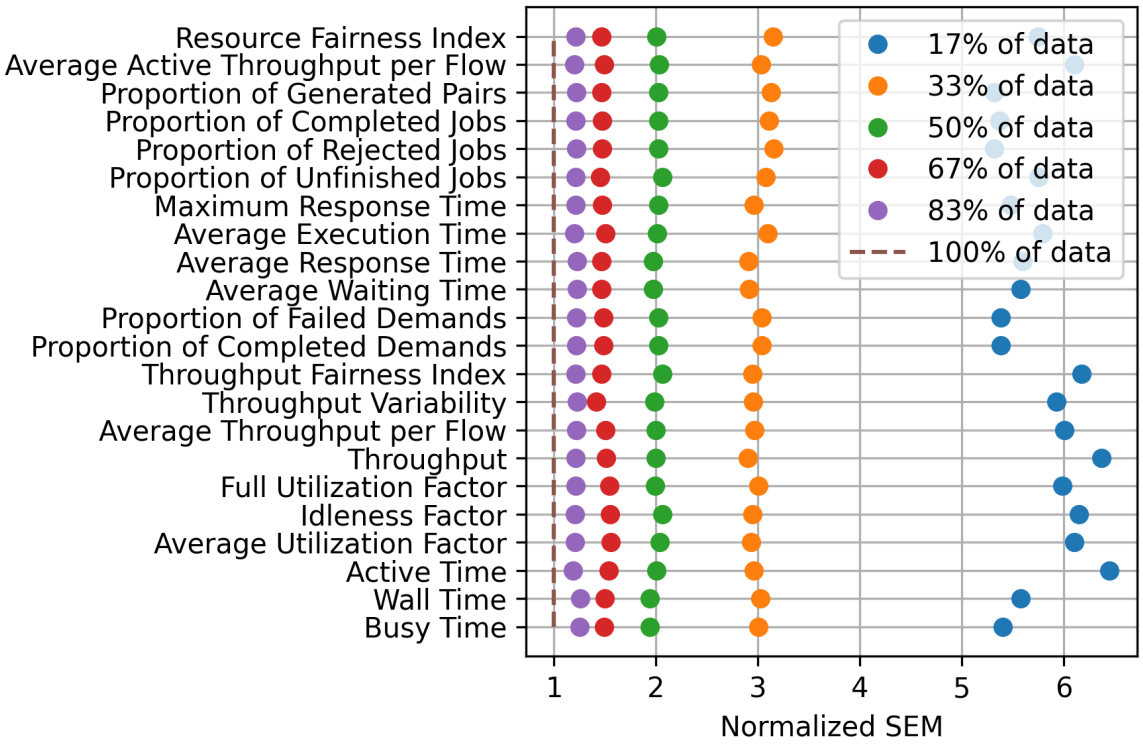

Figure 4.8: Results of the error analysis. The y-axis displays the metrics, and the x-axis displays the standard error for these metrics, normalised with the error if 100 % of the data is used. The line indicates the case of using all data, which is always 1 due to normalisation. Subsets of the data were taken by randomly sampling runs from the simulations and using only these for the averaging of metrics. The resulting standard error of these averages is plotted.

times, and sometimes even the RR scheduler. The only scheduler that doesn't outperform it in throughput is the OD scheduler. This is unexpected, based on the fact that it performs very well in classical scheduling. Recent work by Bhambay et al. [57] shows that the MaxWeight scheduler is not always optimal in quantum networks in case of probabilistic decoherence. Decoherence is a consideration for windows in BQC, so it is possible that this also plays a role here. However, since the MW scheduler also underperforms in QKD use cases, where decoherence should not play a role, this does not explain it whole. This paragraph therefore provides an answer to research question 4, as it seems classical optimality does not necessarily mean better performance than other schedulers in some metric.

The MaxWeight scheduler performed worse than all other schedulers except for OD most of the time in the demand completion metric. It is also the case that the MaxWeight scheduler did not perform as well in other metrics when compared to the other schedulers, so it may be the case that the MW scheduler performs worse in demand completion as a result of the performance in lower level metrics. However, this does not explain why the MW scheduler performed worse in other metrics, so something to consider here is that it is possible the way jobs are submitted may affect the performance of the MW scheduler. If jobs were submitted with some probabilistic intervals, busier flows would have more access to the resources because of their faster growing queues. Since jobs are submitted as batches at the start of the demand, the queues are not growing during the execution of an application. Now, the MW scheduler will first equalise all the queues in length and then proceed as if it was a RR scheduler, scheduling from each demand on at a time. This means that all demands submitted while another is active will finish around the same time, since a queue cannot empty before all other queues have at most 1 job in them. This has important implications for demands that get submitted randomly and have expiration dates. If a demand gets submitted when another is almost done, it will get access to the resources instead while the other demand is almost done. Depending on the exact numbers, this may cause demands that were almost done to fail nonetheless. This could also explain why waiting times for the MW scheduler are shorter than the other schedulers. The waiting time only considers jobs that

at least started execution. Under this explanation, jobs from a demand in MW scheduling are more likely to start execution near the submission of a demand than near the expiry, meaning that most jobs that are included in calculation of the waiting time had a short waiting time. Whether this is really the case and how this affects the other metrics is difficult to say based on these results.

4.2.3. On-demand performance

The on-demand scheduler does not tend to perform well in any metrics that aren't related to waiting or computing time. It consistently has the lowest demand completion and even achieves a demand completion of 0 in the high expiry load QKD simulation. That does not mean that this scheduler brings no value to network hub systems. Of course, since it has no queue, it makes sense that it would perform worse than others in the current setup. However, it is also the case that the way it had to be implemented along the other schedulers meant it stood at a disadvantage. The current implementation in of the schedulers in NetSquid-netbuilder does not allow for jobs to be created during simulation, as they are all generated before. This would benefit a network with an OD scheduler however, since it could then react accordingly if a job gets blocked or is completed and resubmit a new job. Work by Kumar et al. [58] has shown that a on-demand scheduler with exponentially increasing back-off times can lead to better performance of the scheduler. For these reasons, it may be interesting to study the performance of the on-demand scheduling algorithm within a network that allows such improvements.

4.2.4. EDF and EFDF

The eariest feasible deadline first scheduler was introduced as an attempt to prevent a problem with the EDF scheduler in cases where it would keep scheduling infeasible jobs so long as their deadline is first. This was expected to increase the demand completion, as now the EDF scheduler should spend more time on demands that it actually can complete if the network load is high. The results of the simulations have shown however that the EDF almost always outperforms the EFDF scheduler. The results indicate that there is no reason to choose this EFDF scheduler over EDF. It is possible that this unexpected underperformance of the EFDF is simply due to how this specific EFDF scheduler is implemented. There have been multiple attempts at introducing admission control for the EDF scheduler [45, 46, 47, 48], and for this thesis a relatively simple feasibility check was used. A possible problem with this feasibility check may be related to how expiry times are calculated for the demands. These use the number of jobs for that demand and the execution time estimate, which is always the longest time possible a job is allowed to be in execution for, and multiplies them together (though there are some small differences for calculating a BQC expiry time). The EFDF scheduler does the inverse of this, as it uses the execution times supplied by a demand to see if the number of jobs still left to do fit within the remaining time until the deadline is reached. If most jobs finish before the execution time is up, this will be an overestimate, meaning the EFDF scheduler can deprioritise a demand even if there is still enough time left to finish that demand realistically. Characterising these demands as infeasible even though they can still be executed in time may then be the reason why it does not perform better, but worse than the EDF scheduler. A possible improvement here could be to instead calculate a chance that a demand finishes on time based on the execution time supplied by the demand, the average execution time of the jobs that finish, the number of jobs left and the chance that a job finishes in time. Then, all demands with a probability to execute in time successfully lower than some threshold can be deprioritized. Then, to answer the fifth research question posed in Chapter 1, it seems that the specific type and implementation of admission control in this thesis does not allow for better performance for an earliest deadline first scheduler.

4.2.5. Execution time analysis

In the process of generating packet generation attempts to execute based on the demand format, execution times have to be calculated [13]. It is interesting to see then how the execution times that were supplied by the demand compare to the mean execution times of the simulation. This is because, in off-line scheduling, the schedule will be made in whole before execution, meaning it can not reschedule jobs a bit earlier if another before them finishes early. On the other hand, there are communication times associated with online scheduling to let a user know for each job when that gets scheduled. If average execution times are close to the maximum execution time they are given by the demand, off-line scheduling may be beneficial if these small losses are preferable over the overhead of online scheduling. If on the other hand execution times are significantly smaller than the time slot given, it

may become more interesting to consider online scheduling instead to make use of the leftover times.

Taking into account all simulations, the ratio of the average execution time E and the average estimated execution time $E^\dagger$ is $\frac{E}{E^\dagger} = 0.32 \pm 0.01$, where 0.01 is its standard error. For example, if a job has been given a maximum execution time of 3 s based on its demand, it will have an execution time of 0.96 s on average, assuming it finishes. For BQC itself, this is $\left(\frac{E}{E^\dagger}\right)_{\text{BQC}} = 0.453 \pm 0.003$, for QKD it is $\left(\frac{E}{E^\dagger}\right)_{\text{QKD}} = 0.3388 \pm 0.0006$ and the mixed case has $\left(\frac{E}{E^\dagger}\right)_{\text{Mixed}} = 0.157 \pm 0.001$. Interestingly, $\left(\frac{E}{E^\dagger}\right)_{\text{Mixed}} < \left(\frac{E}{E^\dagger}\right)_{\text{QKD}}, \left(\frac{E}{E^\dagger}\right)_{\text{BQC}}$. All ratios are smaller than 0.5, meaning that jobs that finish, finish under half the allowed time on average. While no analysis has been done on the overhead introduced by online scheduling, if this is less than half of the time of these jobs, the results suggest it is more interesting to consider online scheduling in this scenario, as related to research question 6.

Another interesting point to consider here is whether the allotted execution times result in the expected probability of success. The execution time calculated as per the work of Beauchamp et al. [13] is supposed to be the time such that the jobs have a chance of p_{packet} to finish in time. In this thesis, $p_{\text{packet}} = 0.8$, so to verify the calculated execution time, the chance that a job succeeds should be compared with 0.8. Using the proportion of jobs completed, as an estimate of the chance that a single packet succeeds, $p^\dagger_{\text{packet}}$, we find a $p^\dagger_{\text{packet}} = 0.79 \pm 0.01$ for all simulations. This indicates that the calculated execution times are working as intended. For BQC individually we find $\left(p^\dagger_{\text{packet}}\right)_{\text{BQC}} = 0.82 \pm 0.02$, for QKD $\left(p^\dagger_{\text{packet}}\right)_{\text{QKD}} = 0.78 \pm 0.03$ and for the mixed case $\left(p^\dagger_{\text{packet}}\right)_{\text{Mixed}} = 0.76 \pm 0.03$. This indicates that in mixed use cases, the interference of different types of demands with each other results in a less accurate calculated execution time.

4.2.6. Load analysis

The network load has been varied in three different ways in this thesis, and now it becomes interesting to ask if these different ways indeed behave as was expeceted. The purpose of the request and submit load was to increase the number of active demands at the same time, whereas the expiry load only increases pressure on the scheduler to complete these demands in time.

From Figures 4.1, 4.2 and 4.3, we can see that the submit and request load do not behave differently in the BQC use case. In the QKD case, however, the high request load does not differ from the medium load, though the low request load is very similar to the low submit load. The reason for this is not known, but it is possible that overestimates of the demand play a role here: if you increase the number of jobs but also the demand expiry time, whether or not a single demand becomes more difficult to execute successfully depends on how well the estimate of demand expiry scales with increasing jobs. If this is an overestimate, the demand gets disproportionately more time for completing them. This can interact with the effect of more demands being busy at the same time, since they will now take longer to complete, but it is possible that for the QKD use case the demands are overestimated so much that it cancels out the effect of increasing the number of active demands at the same time. The mixed case seems to reflect that it is a 50/50 combination of QKD and BQC.

The demand completion seems to be more sensitive to the expiry load than the submit and request load. Low expiry loads have demand completions >0.9 for each scheduler, whereas high expiry load always has the lowest demand completion, especially for OD which reaches a demand completion of 0 in the QKD use case. Clearly, this load is different from the others. In Figure 4.7, the low load network simulations are all near each other in the upper right. The high expiry load simulations are in the bottom middle however, whereas the other high load simulations are on the left. It seems that this load affects the throughput less, but the demand completion itself more directly. The submit load parameter, for example, reflects more directly how busy a network is by having more applications be submitted in a smaller time. The expiry load on the other hand is decided by parameters set by users themselves. It only reflects a busier network insofar as a network implementation would limit how long applications can be active for on the network if it is busy.

To answer research question 7 from Chapter 1, it is therefore most interesting to consider the submit load parameter as a load condition for a quantum network, especially since it is also the classical definition for network loads in the first place. The other network load parameters can also have a place in simulations, but especially the expiry load could be more interesting as tool for congestion control in

busy networks instead of a model for busy networks.

4.2.7. Mixed use case

Lastly, it may be interesting to see if the mixed case can be approximated by just averaging the BQC and QKD demands. Based on Figures 4.1, 4.2 and 4.3, this is not the case. While most data points of the mixed case lie in between the QKD and BQC scenarios, this is often not an equal average of the two. Furthermore based on Figures 4.4, 4.5 and 4.6, we can see there are cases where a performance metric for the mixed case lies in between the QKD and BQC values: the active throughput of the MW scheduler is higher in the mixed cases than both the QKD and BQC case. Lastly, indications of some type of interference between the two demands arise in comparing the job completion in the previous Section 4.2.6. Here, the job completion of the mixed case was lower than the both QKD and BQC. Regarding research question 8, these findings indicate that a quantum network that serves different types of use cases should not necessarily expect the average performance of a BQC and QKD network.

5

Conclusion

In this thesis, a method to select a scheduler for a quantum network was introduced. This is done first by choosing which schedulers to consider and what performance metrics to rank them by. Then a problem is set up and executed with these schedulers, and the results are used to select the scheduler that fits best within the desires of a network operator and the network users. Applying this process to 3 use cases and 7 different network loads for 6 schedulers in a star network revealed that the earliest deadline first scheduler is a strong contender for these types of networks. This was done by simulation in NetSquid using NetSquid-netbuilder and demand formatting according to the method proposed by Thomas et al. [13]. Additionally, this thesis compiled a comprehensive list of metrics and identified when and to whom these can be interesting to, thereby providing a structured reference that may benefit both users and operators of quantum networks.

For future work, it may be interesting to consider other ways of simulating the network. The on-demand scheduler may perform better in a network where blocking events are communicated to the node so that they can act on it. Similarly, it may be informative to consider networks where jobs are submitted separately at some rate like is done for the on-demand scheduler instead of as a single batch at demand submission and confirm if indeed MaxWeight performance is hampered by the submission of a whole demand at once.

Also, alternative implementations of the earliest deadline first scheduler may be considered that introduce different methods of admission control, to quantify whether such a scheme can deliver better performance than the bare earliest deadline first algorithm. Additionally, it would be informative to assess if some form of optimality for the earliest deadline first scheduler can be proven, and if the earliest deadline first scheduler still performs best when considering alternative networks, use cases or schedulers.

In this thesis, the number of demands ranged between 5-200 per run, based on the use case and network load. Especially for QKD simulations, the number of demands were on the low end near $\sim$10. To ensure steady state is reached and the results are more accurate, individual runs could be done for longer and with more requests, such that randomness across individual demands balances out. This may be a reason for why individual runs can vary a lot from one another.

Overall, this thesis provides a possible framework for a systematic and adaptable scheduler selection method in centralized quantum networks. The scheduler code and simulation setup developed for this thesis can be directly reused or modified by others working with NetSquid-netbuilder, enabling them to apply this method to other schedulers or network scenarios with minimal effort. The current approach targets centralised networks, though the general structure of the method is flexible and may serve as a starting point for developing a similar scheduler selection scheme for distributed quantum networks, taking into account additional challenges like decentralised coordination and limited knowledge of the total network state.

Acknowledgements

I owe thanks to many other people, as this thesis would not have been possible without their advice, insight and support. Scarlett Gauthier provided regular feedback and guidance, and her involvement was essential throughout the project's process. Michał van Hooft offered invaluable assistance in understanding how to work with NetSquid-Netbuilder, and was always ready to respond to any issues that arose. High-level direction and valuable perspective were offered by Stephanie Wehner, whose support was greatly appreciated at key stages of the work. Additionally, Thomas Beauchamp and Janice van Dam gave helpful input during the development of this project, helping to prevent delays and resolve various challenges. Lastly, appreciation is extended to the Wehner Group and QuTech for allowing me to make use of their resources and work in their environment during the project's duration.

In the development of this thesis, OpenAI's ChatGPT was used as a tool assisting with language and phrasing during the drafting process. All content was reviewed and edited by me to ensure academic integrity.

References

[1] H. J. Kimble. “The quantum internet”. In: *Nature* 453.7198 (June 2008), pp. 1023–1030.

[2] S. Wehner, D. Elkouss, and R. Hanson. “Quantum internet: A vision for the road ahead”. In: *Science* 362.6412 (2018), eaam9288.

[3] C. H. Bennett and G. Brassard. “Quantum Cryptography: Public Key Distribution and Coin Tossing”. In: *Proceedings of the International Conference on Computers, Systems & Signal Processing, Bangalore, India*. Vol. 1. New York: IEEE, 1984, pp. 175–179.

[4] A. K. Ekert. “Quantum cryptography based on Bell’s theorem”. In: *Phys. Rev. Lett.* 67.6 (1991), pp. 661–663.

[5] V. Scarani et al. “The security of practical quantum key distribution”. In: *Reviews of Modern Physics* 81.3 (2009), pp. 1301–1350.

[6] P. Arrighi and L. Salvail. “Blind Quantum Computation”. In: *International Journal of Quantum Information* 04.05 (2006), pp. 883–898.

[7] A. Broadbent, J. Fitzsimons, and E. Kashefi. “Universal Blind Quantum Computation”. In: *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. 2009, pp. 517–526.

[8] A. M. Childs. “Secure Assisted Quantum Computation”. In: *Quantum Information and Computation* 5.6 (2005).

[9] A. J. Stolk et al. “Metropolitan-scale heralded entanglement of solid-state qubits”. In: *Science Advances* 10.44 (2024), eadp6442.

[10] C. M. Knaut et al. “Entanglement of nanophotonic quantum memory nodes in a telecom network”. In: *Nature* 629.8012 (May 2024), pp. 573–578.

[11] J. L. Liu et al. “Creation of memory–memory entanglement in a metropolitan quantum network”. In: *Nature* 629.8012 (May 2024), pp. 579–585.

[12] S. Gauthier, G. Vardoyan, and S. Wehner. “An Architecture for Control of Entanglement Generation Switches in Quantum Networks”. In: *IEEE Transactions on Quantum Engineering* 4 (2023), pp. 1–17.

[13] T. R. Beauchamp et al. “A Modular Quantum Network Architecture for Integrating Network Scheduling with Local Program Execution”. In: *arXiv preprint arXiv:2503.12582* (2025).

[14] N. K. Chandra, E. Kaur, and K. P. Seshadreesan. “Network Operations Scheduling for Distributed Quantum Computing”. In: *2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*. 2024, pp. 506–515.

[15] T. Vasantam and D. Towsley. “A throughput optimal scheduling policy for a quantum switch”. In: *Quantum Computing, Communication, and Simulation II*. Vol. 12015. International Society for Optics and Photonics. SPIE, 2022, p. 1201505.

[16] Z. Xiao et al. “Purification scheduling control for throughput maximization in quantum networks”. In: *Communications Physics* 7.1 (2024), p. 307.

[17] R. Zhou, Y. Gan, and Y. Liu. “Towards Flow Scheduling in A Quantum Data Center”. In: *Association for Computing Machinery*. 2023, pp. 45–50.

[18] T. Coopmans et al. “NetSquid, a discrete-event simulation platform for quantum networks”. In: *Commun Phys 4, 164* (2021).

[19] P. Kómár et al. “A quantum network of clocks”. In: *Nature Physics* 10.8 (2014), pp. 582–587.

[20] C. L. Degen, F. Reinhard, and P. Cappellaro. “Quantum sensing”. In: *Rev. Mod. Phys.* 89 (2017), p. 035002.

[21] D. Gottesman, T. Jennewein, and S. Croke. “Longer-Baseline Telescopes Using Quantum Repeaters”. In: *Phys. Rev. Lett.* 109 (7 2012), p. 070503.

[22] H. Buhrman et al. “Quantum Fingerprinting”. In: *Phys. Rev. Lett.* 87 (2001), p. 167902.

[23] S. Tani, H. Kobayashi, and K. Matsumoto. “Exact Quantum Algorithms for the Leader Election Problem”. In: *STACS 2005*. Ed. by V. Diekert and B. Durand. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 581–592.

[24] M. Hasanpour et al. “Quantum load balancing in ad hoc networks”. en. In: *Quantum Information Processing* 16.6 (Apr. 2017), p. 148.

[25] D. Ding and L. Jiang. *Coordinating Decisions via Quantum Telepathy*. arXiv:2407.21723 [quant-ph]. Sept. 2024.

[26] H. Y. Wong. *Introduction to Quantum Computing: From a Layperson to a Programmer in 30 Steps*. 2nd Edition. Cham: Springer, 2024, pp. II, 7.

[27] C. H. Bennett et al. “Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels”. In: *Phys. Rev. Lett.* 70 (13 1993), pp. 1895–1899.

[28] C. H. Bennett and S. J. Wiesner. “Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states”. In: *Phys. Rev. Lett.* 69 (20 1992), pp. 2881–2884.

[29] W. K. Wootters and W. H. Zurek. “A single quantum cannot be cloned”. In: *Nature* 299.5886 (1982), pp. 802–803.

[30] D. Dieks. “Communication by EPR devices”. In: *Physics Letters A* 92.6 (1982), pp. 271–272.

[31] H.-J. Briegel et al. “Quantum Repeaters: The Role of Imperfect Local Operations in Quantum Communication”. In: *Phys. Rev. Lett.* 81 (1998), pp. 5932–5935.

[32] L. K. Grover. *Quantum Telecomputation*. 1997. arXiv: `quant-ph/9704012 [quant-ph]`.

[33] J. I. Cirac et al. “Distributed quantum computation over noisy channels”. In: *Phys. Rev. A* 59 (1999), pp. 4249–4254.

[34] A. Dahlberg et al. “A link layer protocol for quantum networks”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM ’19. Beijing, China: Association for Computing Machinery, 2019, pp. 159–173.

[35] G. Vardoyan et al. “On the Stochastic Analysis of a Quantum Entanglement Distribution Switch”. In: *IEEE Transactions on Quantum Engineering* 2 (2021), pp. 1–16.

[36] C. E. Bradley et al. “A ten-qubit solid-state spin register with quantum memory up to one minute”. In: *Physical Review X* 9 (2019), p. 031045.

[37] H. K. Beukers et al. “Remote-Entanglement Protocols for Stationary Qubits with Photonic Interfaces”. In: *PRX Quantum* 5 (2024), p. 010202.

[38] P. G. Kwiat and H. Weinfurter. “Embedded Bell-state analysis”. In: *Phys. Rev. A* 58 (1998), R2623–R2626.

[39] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd Edition. New York, NY: Springer, 2011, pp. XVI, 524.

[40] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Hardback. Cambridge, UK: Cambridge University Press, 2013, p. 576.

[41] S. Altmeyer, S. M. Sundharam, and N. Navet. *The case for FIFO real-time scheduling*. Tech. rep. Fakultät für Angewandte Informatik, 2016, p. 12.

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

[43] S. Gauthier, T. Vasantam, and G. Vardoyan. “An on-demand Resource Allocation Algorithm for a Quantum Network Hub and its Performance Analysis”. In: *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Vol. 01. 2024, pp. 1748–1759.

[44] A. K. Erlang. “Solution of some problems in the theory of probabilities of significance in automatic telephone exchanges”. In: *Post Office Electrical Engineer’s Journal* 10 (1917), pp. 189–197.

[45] V. Firoiu, J. Kurose, and D. Towsley. “Efficient admission control for EDF schedulers”. In: *Proceedings of INFOCOM ’97*. Vol. 1. 1997, pp. 310–317.

[46] A. Masrur, S. Chakraborty, and G. Färber. “Constant-Time Admission Control for Partitioned EDF”. In: *2010 22nd Euromicro Conference on Real-Time Systems*. 2010, pp. 34–43.

[47] Y. Xie and T. Yang. “Efficient admission control for EDF scheduler with statistical QoS guarantee”. In: *Proceedings of Sixth International Conference on Computer Communications and Networks*. 1997, pp. 242–247.

[48] V. Salmani, M. Naghibzadeh, and M. Kahani. “Deadline Scheduling with Processor Affinity and Feasibility Check on Uniform Parallel Machines”. In: *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*. 2007, pp. 793–798.

[49] R. Srikant and L. Ying. *Communication networks: An optimization, control and stochastic networks perspective*. Cambridge University Press, 2014.

[50] H.-L. Huang et al. “Superconducting quantum computing: a review”. In: *Science China Information Sciences* 63.8 (2020), p. 180501.

[51] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe. “A quantitative measure of fairness and discrimination”. In: *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA* 21 (1984).

[52] C. Ekelin. “Clairvoyant non-preemptive EDF scheduling”. In: *18th Euromicro Conference on Real-Time Systems (ECRTS’06)*. 2006, 7 pp.–32.

[53] M. van Hooft. *netsquid-netbuilder*. GitLab Repo: `https://gitlab.com/softwarequtech/netsquid-snippets/netsquid-netbuilder`. Variation of version 0.13 used in this thesis.

[54] M. Pompili. “Multi-Node Quantum Networks with Diamond Qubits”. PhD thesis. Delft University of Technology, 2021.

[55] G. Avis et al. “Requirements for a processing-node quantum repeater on a real-world fiber grid”. In: *npj Quantum Information* 9 (2023), p. 100.

[56] C.-K. Hong, Z.-Y. Ou, and L. Mandel. “Measurement of subpicosecond time intervals between two photons by interference”. In: *Physical review letters* 59 (1987).

[57] S. Bhambay, T. Vasantam, and N. Walton. *Optimal Scheduling in a Quantum Switch*. 2025. arXiv: `2501.05380 [quant-ph]`.

[58] A. Kumar et al. “New insights from a fixed point analysis of single cell IEEE 802.11 WLANs”. In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 3. IEEE. 2005, pp. 1550–1561.

A

Source Code

While the complete source code is available at `https://gitlab.tudelft.nl/wehner-research/hubschedulerperformance`, this section contains snippets of code detailing how exactly the schedulers were implemented. Specifically, the *_active_next_que_item()* and *_add_to_que* methods that decides which item from the queue gets scheduled next and how they get added to the queue are shown here. The surrounding code is omitted here, as the focus is only on the implementation itself.

Listing A.1: FIFO scheduler implementated as a class in NetSquid showing only the relevant methods.

```
class FIFOScheduleProtocol(TimeslotSchedulerProtocol):
    ...

    def _add_to_que(self, item: QueItem):
        self._que.append(item)
        self.queue_length_tracker.append((ns.sim_time(), len(self._que)))

    def _activate_next_que_item(self):
        """
        If possible, activates next request in queue, according to scheduling rules. Removes
            Que_items from queue if they have expired.
        """
        for que_item in self._que:
            if que_item.demand_id not in self._unavailable_demands:
                self._remove_from_queue(que_item)
                self._activate_request(
                    que_item.node1_id, que_item.node2_id, que_item.create_id, que_item.
                        num_qubits
                )
                break
```

Listing A.2: OD scheduler implementated as a class in NetSquid showing only the relevant methods.

```
class OnDemandScheduleProtocol(FIFOScheduleProtocol):
    ...

    def _activate_next_que_item(self):
        """
        On demand does not have a queue, so this does nothing on purpose.
        """
        pass

    def _add_to_que(self, item: QueItem):
        self._reject_incoming_request(item)
```

Listing A.3: EDF scheduler implemented as a class in NetSquid showing only the relevant methods. It does not show the activiation method that is inherited from the FIFO scheduler as it is left unchanged.

```
class EDFScheduleProtocol(FIFOScheduleProtocol):
    ...

    def _add_to_que(self, item: QueItem):
        insort(self._que, item, key=lambda x: x.deadline)
        self.queue_length_tracker.append((ns.sim_time(), len(self._que)))
```

Listing A.4: FEDF scheduler implemented as a class in NetSquid showing only the relevant methods. It does not show the method to add jobs to queues that is inherited from the EDF scheduler as it is left unchanged.

```
class FEDFScheduleProtocol(EDFScheduleProtocol):
    ...

    def _activate_next_que_item(self):
        """This overrides the method to prioritize "feasible" demands. Only afterward does it
             do infeasible ones. This method requires demands for its functionality, so if
            these are not present it will revert to its parent's method behaviour."""
        if self.params.using_demands:
            infeasible_request = None
            infeasible_demands = []
            for que_item in self._que:
                demand_id = que_item.demand_id
                if demand_id not in self._unavailable_demands and demand_id not in
                    infeasible_demands:
                    demand_log = self.demand_log[demand_id]
                    expected_demand_execution = ((demand_log["packets_required"] - demand_log
                        ["n_successful"])
                                                 * (demand_log["execution_time"] + demand_log["
                                                     min_separation"]))
                    if expected_demand_execution > demand_log["expiry_time"] - ns.sim_time():
                        infeasible_demands.append(demand_id)
                        if infeasible_request is None:
                            infeasible_request = que_item
                        continue
                    self._remove_from_queue(que_item)
                    self._activate_request(
                        que_item.node1_id, que_item.node2_id, que_item.create_id, que_item.
                            num_qubits
                    )
                    break
            else:
                if infeasible_request:
                    self._remove_from_queue(infeasible_request)
                    self._activate_request(
                        infeasible_request.node1_id, infeasible_request.node2_id,
                            infeasible_request.create_id,
                        infeasible_request.num_qubits
                    )
        else:
            super()._activate_next_que_item()
```

Listing A.5: MW scheduler implemented as a class in NetSquid showing only the relevant methods.

```
class MWScheduleProtocol(FIFOScheduleProtocol):
    ...

    def _add_to_que(self, item: QueItem):
        if self.params.using_demands:
            index = item.demand_id
        else:
            index = self._get_flow_from_ids(item.node1_id, item.node2_id)
            self._extend_ques(index)

        self._ques[index].append(item)
        self.queue_length_tracker.append((ns.sim_time(), len(self._que)))

    def _activate_next_que_item(self):
        """
        If possible, activates next request in queue, according to scheduling rules. Removes
            Que_items from queue if they have expired. index = demand id if self.params.
            using_demands is True.
        """
        if self._que:
            weights_unsorted = list({i: len(que) for i, que in enumerate(self._ques)}.items()
                )
            # shuffle to act as tiebreaker, as python uses placement in list as tiebreaker
            rng = get_random_state()
            rng.shuffle(weights_unsorted)
            self._que_order = [k for k, _ in sorted(weights_unsorted, key=lambda item: item
                [1], reverse=True)]

            for index in self._que_order:
                if index in self._unavailable_demands or not self._ques[index]:
                    continue
                que_item = self._ques[index][0]
                self._remove_from_queue(que_item)

                self._activate_request(
                    que_item.node1_id, que_item.node2_id, que_item.create_id, que_item.
                        num_qubits
                )
                break
```

Listing A.6: RR scheduler implemented as a class in NetSquid showing only the relevant methods. It does not show the method to add jobs to queues that is inherited from the MW scheduler as it is left unchanged.

```
class RRScheduleProtocol(MWScheduleProtocol):
    ...

    def _activate_next_que_item(self):
        """
        If possible, activates next request in queue, according to scheduling rules. Removes
            Que_items from queue if they have expired. index = demand id if self.params.
            using_demands is True.
        """
        if self._que:
            for index in self._que_order:
                if index in self._unavailable_demands or not self._ques[index]:
                    continue

                que_item = self._ques[index][0]
                self._remove_from_queue(que_item)

                self._que_order.remove(index)
                self._que_order.append(index)
                self._activate_request(
                    que_item.node1_id, que_item.node2_id, que_item.create_id, que_item.
                        num_qubits
                )
                break
```

B

Best performing schedulers

This section contains the results of the simulations regarding who performed best in each metric that was considered. The table is split in multiple parts for each use case. The columns correspond to the load condition for that simulation and the rows to the metric. The individual cells show the name of the scheduler that performed ”best” in each category, which either means having the highest or lowest of that metric, dependent on the metric itself. This table also includes the wall time, i.e. the real time the whole simulation took and the total active time, which is the time for which the simulation itself was active, and not waiting for some leftover events to fire.

Request Load	**High**	**Low**	**Med**	**Med**	**Med**	**Med**	**Med**
Submit Load	**Med**	**Med**	**High**	**Low**	**Med**	**Med**	**Med**
Expiry Load	**Med**	**Med**	**Med**	**Med**	**High**	**Low**	**Med**
Lowest busy time	OD	OD	OD	OD	OD	EFDF	OD
Lowest wall time	OD	OD	OD	OD	OD	OD	OD
Lowest total active time	MW	MW	MW	MW	MW	MW	MW
Highest average utilisation factor	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Lowest idleness factor	EDF	OD	OD	EDF	EDF	OD	EDF
Highest full utilisation factor	EDF	MW	MW	EDF	EDF	MW	EDF
Highest throughput	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest throughput per flow	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Lowest throughput deviation	EDF	EDF	EDF	EDF	EDF	RR	EDF
Highest throughput fairness index	EDF	EDF	EDF	EDF	EDF	RR	EDF
Highest proportion of completed demands	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Lowest proportion of failed demands	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Lowest average waiting time	OD	OD	OD	OD	OD	OD	OD
Lowest average response time	OD	OD	OD	OD	OD	OD	OD
Lowest average execution time	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Lowest worst response time	OD	OD	OD	OD	OD	OD	OD
Lowest unfinished job ratio	MW	MW	EDF	MW	MW	EDF	MW
Lowest job rejection ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest job completion ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest pair success ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest active throughput per flow	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest resource fairness index	EDF	EDF	EDF	EDF	EDF	EDF	EDF

Table B.1: Table showing the schedulers who performed best in the BQC simulations for each metric by network load.

Request load	**High**	**Low**	**Med**	**Med**	**Med**	**Med**	**Med**
Submit load	**Med**	**Med**	**High**	**Low**	**Med**	**Med**	**Med**
Expiry load	**Med**	**Med**	**Med**	**Med**	**High**	**Low**	**Med**
Lowest busy time	OD	OD	OD	OD	OD	OD	OD
Lowest wall time	OD	OD	OD	OD	OD	OD	OD
Lowest total active time	MW	MW	MW	MW	MW	MW	MW
Highest average utilisation factor	MW	EDF	EDF	MW	EDF	FIFO	EDF
Lowest idleness factor	EDF	EDF	EDF	EDF	EDF	EFDF	EDF
Highest full utilisation factor	MW	MW	FIFO	MW	EDF	MW	EDF
Highest throughput	MW	EDF	EDF	MW	MW	EDF	EDF
Highest throughput per flow	MW	EDF	EDF	MW	MW	EDF	EDF
Lowest throughput deviation	EDF	EDF	EDF	EDF	OD	EDF	EDF
Highest throughput fairness index	EDF	EDF	EDF	EDF	OD	EDF	EDF
Highest proportion of completed demands	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Lowest proportion of failed demands	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Lowest average waiting time	OD	OD	OD	OD	OD	OD	OD
Lowest average response time	OD	OD	OD	OD	OD	OD	OD
Lowest average execution time	MW	EDF	MW	MW	MW	OD	MW
Lowest worst response time	OD	OD	OD	OD	OD	OD	OD
Lowest unfinished job ratio	FIFO	MW	RR	RR	OD	OD	EDF
Lowest job rejection ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest job completion ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest pair success ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest active throughput per flow	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest resource fairness index	EDF	EDF	EDF	EDF	EDF	EDF	EDF

Table B.2: Table showing the schedulers who performed best in the mixed case simulations for each metric by network load.

Request load	**High**	**Low**	**Med**	**Med**	**Med**	**Med**	**Med**
Submit load	**Med**	**Med**	**High**	**Low**	**Med**	**Med**	**Med**
Expiry load	**Med**	**Med**	**Med**	**Med**	**High**	**Low**	**Med**
Lowest busy time	OD	OD	OD	OD	OD	OD	OD
Lowest wall time	OD	OD	OD	OD	OD	OD	OD
Lowest total active time	MW	RR	RR	MW	MW	MW	MW
Highest average utilisation factor	MW	RR	EFDF	MW	EDF	MW	EDF
Lowest idleness factor	FIFO	EDF	EDF	EDF	EDF	EDF	EDF
Highest full utilisation factor	MW	RR	MW	MW	EDF	MW	MW
Highest throughput	MW	EDF	EDF	EFDF	EDF	MW	EDF
Highest throughput per flow	MW	EDF	EDF	EFDF	EDF	MW	EDF
Lowest throughput deviation	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest throughput fairness index	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest proportion of completed demands	EDF	EDF	EDF	EDF	FIFO	EDF	EDF
Lowest proportion of failed demands	EDF	EDF	EDF	EDF	FIFO	EDF	EDF
Lowest average waiting time	OD	OD	OD	OD	OD	OD	OD
Lowest average response time	OD	OD	OD	OD	OD	OD	OD
Lowest average execution time	EDF	EDF	EDF	EDF	OD	OD	EDF
Lowest worst response time	OD	OD	OD	OD	OD	OD	OD
Lowest unfinished job ratio	MW	EDF	OD	OD	OD	MW	EFDF
Lowest job rejection ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest job completion ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest pair success ratio	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest active throughput per flow	EDF	EDF	EDF	EDF	EDF	EDF	EDF
Highest resource fairness index	EDF	OD	EDF	EDF	OD	EDF	EDF

Table B.3: Table showing the schedulers who performed best in the QKD simulations for each metric by network load.