POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Quantum Key Distribution simulation on a distributed infrastructure

Supervisors
Prof. Antonio Lioy
Dott. Ignazio Pedone

Alessandro Pizzorno

April 2022

Contents

Chapter 1

Introduction

In conventional cryptography, the secure communication problem can be represented by two distant parties, traditionally called Alice and Bob, that want to communicate privately over a shared channel even if an eavesdropper, Eve, was present on it.

To address this issue, the One-time-pad (OTP)[1] could be employed. However, to solve the secure communication problem, some prerequisites need to be met. In particular, the key the two communication parties need to share and use to apply the OTP scheme is required to be both: as long as the message to exchange and it should only be used once[2]. Consequently, a whole new problem arises: how to distribute a key of this kind even if an eavesdropper is present on the channel. This issue is known as the key distribution problem and it is a central challenge for all kinds of encryption methods.

Nowadays, public-key systems are usually employed to solve this problem. However, the security proofs of these systems are entirely based on the computational unfeasibility assumption of different mathematical problems such as the discrete logarithm one. As a result, public-key systems are vulnerable to unanticipated advances in hardware and software like the advent of quantum computers for example.

In 2015, the US National Security Agency (NSA)[1] announced a plan for transition to quantum-safe cryptosystems and broadly speaking, there are two approaches to obtain a quantum-safe encryption scheme.

The former consists of employing conventional cryptography and developing alternative public-key encryption schemes, in which known quantum attacks such as Shor's algorithm[3] do not apply. This approach is called Post-Quantum Cryptography and it has the advantages of being compatible with the existing crypto infrastructure and having particularly high key rates that are available over long distances. However, its main drawback is that the conventional algorithms it proposes are only shown to be secure against known quantum attacks. Therefore, there is always a possibility someone might one day come up with a clever and efficient way for breaking the algorithms proposed by Post-Quantum Cryptography. Besides, this would also lead to a retroactive security breach in the future for data that is transmitted today using those schemes.

The latter is to use Quantum Cryptography, a discipline that promises to provide information-theoretical security by using the fundamental laws of quantum physics. As a result, it is possible to obtain security that remains independent of all future advances of algorithm or computational power.

This thesis work aims to explore Quantum Key Distribution (QKD), the branch of Quantum Cryptography that is committed to dealing with the key distribution problem. In particular, an overview of QKD's security proof and an analysis of its practical implementation is given. The most popular protocols will be discussed and the main QKD simulation platforms available today

[1] https://www.nsa.gov/

will be described putting particular stress upon the solution proposed by the TORSEC research group: the Quantum Software Stack QKD Simulator (QSS QKD Simulator).

The purpose of the final part of this work is to provide a complete redesign of this simulator to add new functionalities to it, improve the ones it already provides and make its deployment in a distributed software stack much easier to achieve. The new version of the QSS QKD simulator was developed to allow users to model a full-fledged network topology and to perform point-to-point QKD simulations on any couple of nodes that are directly linked to each other. The performance of a single simulation was improved thanks to the remodelling of the simulator's internal communication following an asynchronous paradigm. Besides, the possibility of configuring and leveraging parallelism to handle multiple QKD simulations concurrently running was introduced. This allowed for a significant improvement of the simulator performance both in terms of overall throughput, i.e. overall bits exchanged per second.

The rest of this document has the following structure:

- chapter 2 describing the current state of the art of Quantum Key Distribution;
- chapter 3 presenting the main different QKD protocols available;
- chapter 4 illustrating QKD Simulation and the main platforms currently available to perform it, QSS QKD Simulator included;
- chapter 5 describing the technologies used for the implementation of the new version of the simulator;
- chapter 6 presenting the architecture and design of the new version of the simulator;
- chapter 7 illustrating the implementation of the new version of the simulator;
- chapter 8 describing the tests run on the new version of the simulator;
- chapter 9 presenting conclusions and possible future works on the new version of the simulator;
- appendix A illustrating the steps to follow in order to set up an instance of the new version of the simulator;
- appendix B presenting the low-level implementation details of the new version of the simulator.

Chapter 2

Quantum Key Distribution

2.1 Quantum Key Distribution Overview

Quantum Key Distribution(QKD) is the approach proposed by Quantum Cryptography to solve the Key Distribution Problem and its main goal is to achieve information-theoretical security and, to do so, it leverages the law of physics, quantum physics to be precise[4, 5, 6, 7].

One of the main physics principles QKD exploits to guarantee security is the quantum no-cloning theorem. Such a theorem clearly states that a random unknown quantum state cannot be cloned reliably, so it is unfeasible to create an identical copy of it without modifying it[8, 9]. As a result, if a key was to be distributed via quantum (e.g. single photon) signals, an eavesdropper absolutely couldn't produce two quantum copies of the transported quantum state. The only way to know the quantum state carried by the signal would be through direct measurement of it. This, however, has some consequences.

Furthermore, another key feature of quantum mechanics QKD exploits is the complementarity of the rectilinear and diagonal basis. Measurements done in the two bases do not commute with each other, so observables cannot simultaneously be measured in both bases without having their state disturbed. Such principle is leveraged by QKD to make eavesdropping unavoidably disturb the quantum signals, resulting in the users communicating with each other noticing there is something wrong.

One last important advantage of QKD is that quantum communication doesn't leave any classical transcripts an eavesdropper could keep. As a result, the communication session can only be broken in real-time.

In the following sections of this chapter, we will go over QKD's security more in-depth, as well as its practical implementation and the different attacks it can be subject to, but before doing that, we need to quickly introduce some key notions of quantum computing that will be needed to fully understand the concepts that will be exposed next.

2.2 Prerequisites

2.2.1 The Qubit

Classical computing's fundamental unit of information is a bit. On the other hand, in QKD, but in quantum computing more in general, the most basic unit of information is the qubit. The difference is that quantum computing manipulates elemental particles obeying the quantum mechanics laws and, to represent the values 0 and 1, measurements are made on some of their physical properties.

Before measurement, a qubit is considered in a superposition of states. Therefore, differently from classical information where a bit can only have as value either '0' or '1', a qubit has the

capability of representing both those states simultaneously. The reason behind this is that the state of a qubit, its quantum state, is much more complex than simple binary values; it provides a probability distribution about the outcomes of all of the possible measurements you can do on a system.

2.2.2 The Bra-ket Notation

Qubits can be represented as vectors and the bra-ket notation, introduced in 1939 by Paul Dirac[10], allows simpler management of them.

In mathematics, a ket represents a vector in the Hilbert complex vector space. It can be represented as a vertical bar | and a right angle bracket >, e.g. $|0\rangle$ and it is used to represent quantum states.

On the other hand, a bra represents a linear map of each vector to a complex number, mathematically speaking, it corresponds to the complex conjugate transpose of a ket.

As far as its representation is concerned, it consists of a left angle bracket < and a vertical bar |. e.g. $\langle 0|$.

In other words, we can see kets as column vectors and bras as row ones.

Now, we are able to define the bra-ket operation symbolizing the inner product of two vectors:

$$\langle 0|1\rangle = \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.1}$$

Instead, the ket-bra notation refers to the tensor product of vectors which is particularly useful when describing the state of more qubits together:

$$|ba\rangle = |b\rangle \otimes |a\rangle \tag{2.2}$$

2.2.3 Superposition

Superposition consists of a linear combination of two quantum states and can be described as follows:

$$|\psi\rangle = \alpha\,|b\rangle + \beta\,|a\rangle \tag{2.3}$$

with α β representing probability amplitudes.

Furthermore, physically speaking, superposition can be seen as two quantum states added together, generating a new one.

A quantum state indeed has its amplitude and two superposed quantum states interfere with each other in either a constructive or a destructive way. Quantum interference is one of the fundamentals of quantum computing. Letting two particles interfere allows swaying the measurement of a qubit toward the desired state. To be clearer, each time a quantum system is solving a problem, it will be presented with an interference pattern to handle. In particular, all of the paths pointing to a wrong solution will destructively interfere and get cancelled out while the ones leading to the correct outcome will constructively interfere, boosting the energy of the state representing the right solution.

It is important to note that, while interference is an induced phenomenon allowing to affect the qubit state in the desired way, decoherence, instead, is what happens when interference with the external environment causes the quantum state to collapse. Decoherence is an unwelcome disturb of the quantum state that causes it to lose its superposition properties.

2.2.4 Entanglement

The entanglement in quantum physics is a complex phenomenon and there aren't actually any limits to the number of particles it can involve, but in practice, for the sake of simplicity, it is usually performed with just two. Quantum entanglement causes two particles to interact from a physical point of view and to bound to each other. In particular, the quantum state of a particle becomes highly correlated to the one of the other causing any action performed on a particle to affect both. The bond that has been created persists even if the two particles get far from each other. As a result, a measurement on a particle will reveal information about the state of the other entangled one.

Mathematically speaking, for example, we could express two entangled states as follows:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{2.4}$$

Eq. 2.4 states that any measurement on any of the two entangled states will result only in either $|00\rangle$ or $|11\rangle$ with 50% probability each. The chance for the interested particles to be found in either $|01\rangle$ or $|10\rangle$ state is completely absent. Therefore, if we happened to measure only one particle to find it in $|0\rangle$ state, we would be sure the other one, with a probability of 100%, is in such a state as well. The same would happen if the measured state was $|1\rangle$.

The concept of entanglement is only related to quantum mechanics and, at first glance, it appears to violate the theory of relativity which states that the speed of light represents the highest boundary of information transmission. In reality, there is absolutely no information transferred during measurement so the previous assertion is false[11].

Moreover, in 1935 Einstein, Podolsky, and Rosen proposed the hidden variable theory to provide a deterministic explanation of quantum mechanics phenomena and therefore, predict quantum measurement results on a given particle as well. However, in 1964, Bell proposed a theorem demonstrating quantum physics was incompatible with the hidden variable theory[12]. In particular, Bell demonstrated that the hidden variable theory couldn't correlation resulting from entanglement, since otherwise, Bell's inequality, which regards measurements made on entangled particle pairs, would be violated. Finally, it is important to mention that Bell's inequality has been generalized by CHSH inequality in 1969[**?**], which lifted all of the prerequisites about the perfect correlation of the entangled particles. This newly introduced inequality represents a reliable indicator to verify two different particles have entanglement properties.

2.2.5 Measurement

Measurement in quantum mechanics represents the process of manipulating a quantum system to obtain a numeric value from one of its properties. In particular, when a qubit is measured it loses its superposition and its state collapses to the measured one.

The Born's rule which allows to calculate the probability that a given quantum state, $|\psi\rangle$ collapses to a generic state $|x\rangle$ is, practically speaking, the core of measurement. This probability is given by[13]:

$$p(x) = |\langle x|y\rangle|^2 \tag{2.5}$$

Any orthonormal basis can be used to perform a measurement and, depending on the chosen one, the result of this process will be one of the two state-vectors composing the base. Measurement maps the quantum state to one of the basis vectors. In particular, we refer to such process as *projection measurement*[14].

Measurement allows highlighting some important properties of qubits. In particular, if a qubit in a known state is measured in an orthogonal basis, the outcomes are completely random. However, after the measurement, the state of the qubit is known and will remain so for any further measurement that uses the same basis. Independently of the selected basis to measure in, this will always be the observed behaviour.

Furthermore, qubit memory is limited and it only keeps track of the last performed measurement. If the same qubit is measured more than once in a different basis, each new measurement cancels out the outcome of the previous one (unless the same basis is subsequently used). Finally, it is important to note that if a qubit is measured in a basis and it assumes a given value with a probability of 50%, a new measurement in an orthogonal basis will outcome one of the two states composing such basis with a probability of 50% again.

2.3 Security Of Quantum Key Distribution

During the years, different security proofs for QKD have been proposed. In particular, as time progressed, each newly proposed security proof helped lift important prerequisites that were needed to prove QKD's security until we reached the current definition of security for QKD.

Ideally, a secure key satisfies two requirements.

First of all, the key bit strings that Alice and Bob possess, need to be identical, i.e. be correct and second, such strings need to be secret. However, practical issues cause this keys to have a small failure probability ϵ. As a result, we consider a QKD protocol to be ϵ-secure with $\epsilon = \epsilon_{cor} + \epsilon_{sec}$, if it is both ϵ_{cor}-correct and ϵ_{sec}-secret[15, 16].

Let's define K_A and K_B (of identical length m) to be the key bit strings obtained Alice and Bob respectively obtained. A quantum state ρ_E, held by Eve, can be correlated to these keys obtaining the joint state ρ_{ABE} the expression of which isthe follwoing:

$$\rho_{ABE} = \sum k_A, k_B Pr(k_A, k_B) \left|k_A\right\rangle \left\langle k_A\right| \otimes \left|k_B\right\rangle \left\langle k_B\right| \otimes \rho_E^{(k_A, k_B)} \tag{2.6}$$

where k_A, $k_B \in \{0,1\}^m$ are the bit values. In particular, an ideal key state held by Alice and Bob is described by the private state:

$$\rho_{ABE}^{ideal} = 2^{-m} \sum k \left|k\right\rangle_A \left\langle k\right| \otimes \left|k\right\rangle_B \left\langle k\right| \otimes \rho_E \tag{2.7}$$

where $k_A = k_B = k$ implies that Alice and Bob hold the same string and ρ_E is independent of k, so Eve has no information on the key string variable K.

A QKD protocol is defined to be ϵ_{cor}-correct[15], if the probability distribution $Pr(k_A, k_B)$ of the final state ρ_{ABE} in eq. 2.6 satisfies:

$$Pr(k_A \neq k_B) \leq \epsilon_{cor} \tag{2.8}$$

A QKD protocol is defined to be ϵ_{sec}-secret[16], if the state ρ_{ABE} is close in trace distance to the single-party private state ρ_{AE}^{ideal}, in mathematical terms:

$$min_{\rho_E} \frac{1}{2}(1 - p_{abort})||\rho_{AE} - \rho_{AE}^{ideal}||_1 \leq \epsilon_{sec} \tag{2.9}$$

where p_{abort} is the probability that the protocol aborts, $\rho_{AE}^{ideal} \equiv 2^{-m} \sum s \left|s\right\rangle_A \left\langle s\right| \otimes \left|s\right\rangle_B \left\langle s\right| \otimes \rho_E$ and $\left\|A\right\|_1 \equiv Tr[\sqrt{A^T A}]$ is the trace norm.

As a result, in general and as clearly stated in Ben-Or et al.'s work[15], a QKD protocol can be defined to be ϵ-secure, if the final distilled state ρ_{ABE} is ϵ-close to the ideal key state ρ_{ABE}^{ideal}, given in eq. 2.7, with a proper chosen ρ_E:

$$min_{\rho_E} \frac{1}{2}(1 - p_{abort})||\rho_{ABE} - \rho_{ABE}^{ideal}||_1 \leq \epsilon \tag{2.10}$$

Finally, it's important to note that the ϵ parameter used shouldn't be mistreated as Eve's guessing probability.

2.4 QKD Implementation

All of the previous security proofs and definitions required some assumptions, some prerequisites to be met in order to be applicable to a certain QKD system. These assumptions are related to different parts of the system:

- the photon source is assumed to be an ideal single-photon source;
- the encoding state is assumed to be done on two dimensions and basis independent;
- the measurement is assumed to be done on two dimensions and basis independent;
- the photon detection is assumed to be an Ideal SPD (Single Photon Detector).

In practice, the security of a QKD system is often related to its implementation.

There are different ways to implement a QKD system but, since photons are the most widely used for communication, given their robustness against decoherence caused by noisy environments and their fast travelling speed, we will mainly focus on the quantum optical realization of them.

A QKD system is composed of three main components:

- the Source;
- the Channel;
- the Detection.

In this section we will first discuss the main encoding and decoding methods employed, then we will briefly introduce the practical realization of the main parts of a QKD system and, finally, we will quickly go over the most used post-processing techniques.

It's important to recall that, in a rigorous security proof, the channel is assumed to be under the full control of Eve. As a matter of fact, in section **??** there were no references on the communication channel exploited by Alice and Bob to communicate. As a result, the security of a QKD system does won't on the physical realization of the quantum channel as well. As far as the quantum source and detection are concerned, on the other hand, a security proof does normally presents some assumptions on their practical realization.

2.4.1 Encoding State

In QKD, the two very popular encoding methods are:

- polarization encoding;
- time-bin phase encoding.

Polarization encoding is probably the most widely used technique of encoding and it exploits the polarization modes:

- the Z basis encoding leverages the horizontal and vertical polarizations of a photon, denoted by $|10\rangle_{HV}$ and $|01\rangle_{HV}$;
- the X basis encoding exploits the linear polarization modes along the $\pm 45°$ direction, denoted by $\{|10\rangle_{HV} \pm |01\rangle_{HV}\}$;
- the Y basis encoding uses the left-handed or right-handed circular polarization modes, denoted by $\{|10\rangle_{HV} \pm i\,|01\rangle_{HV}\}$.

In the decoding process, a polarization controller is needed to specify the chosen basis and a polarization beam splitter (PBS), connected with single-photon detectors, will perform the polarization measurement.

The other most common method is time-bin phase encoding. Here, Alice, i.e. the communication source, selects two pulses: a signal pulse and a reference pulse to use for the two encoding modes s and r. The two time-bin modes form the Z basis, $\{|10\rangle_{HV}$ and $|01\rangle_{HV}\}$. The qubit in the Z basis determines whether the photon stays in the signal or in the reference time bin. The X basis $\{|10\rangle_{HV} \pm |01\rangle_{HV}\}$ states and the Y basis $\{|10\rangle_{HV} \pm i\,|01\rangle_{HV}\}$ states, denote the photons with a relative phase $0, \pi$ and $\pi/2, 3\pi/2$ between the signal and reference pulses, respectively. Finally, the decoding process is done thanks to an interferometer used to extract information about the phase of the signal.

2.4.2 Photon Source

Here, we will mainly discuss the two main practical photon sources used respectively in QKD.

First of all, it is worth mentioning that, in practice, weak light sources, modulated to follow a Fock state mixture, are what is used in practice to approximate a single-photon source. In particular, the state mixture that is used can be expressed as follows: In general, they are modulated to be a Fock state mixture:

$$\rho = \sum_{n=0}^{\infty} P(n)\,|n\rangle\ \langle n| \tag{2.11}$$

with $P(n)$ is the photon number distribution which will differ according to the kind of photon source we desire to model and $|n\rangle$ is the n-photon number state.

The most widely used photon-source used in QKD is the weak coherent-state source and it is realized using the technique we mentioned above thanks to attenuating lasers used as weak light sources. These lasers generate a light that can be considered as a coherent pulse $|\alpha\rangle$, where α represents a complex number, $\mu = |\alpha|^2$ the average photon number and the phase of α reflects the relative phase between different photon number components. In particular, to create a photon source in the form of eq. 2.11, it is sufficient to randomize the phase of coherent pulses emitted by the lasers and make it a mixture of photon number states[17]:

$$\rho_\mu = \frac{1}{2\pi}\int_0^{2\pi} \mathrm{d}\phi\,\left|\alpha \mathrm{e}^{\mathrm{i}\phi}\right\rangle\left\langle\alpha \mathrm{e}^{\mathrm{i}\phi}\right| = \sum_{n=0}^{\infty} P(n)\,|n\rangle\ \langle n| \tag{2.12}$$

If we wanted to realize a source that generates entangled pairs of photons, instead, this would be implementable thanks to the use of the parametric down-conversion process (PDC). The PDC process essentailly consists of a high frequency photon that is converted to a pair of low frequency entangled ones. In particular, a PDC source will generate a superposition state of different number of photon pairs[18, 19]:

$$\Psi = (\cosh\chi)^{-1}\sum_{n=0}^{\infty}(\tanh\chi)^n\,|n,n\rangle \tag{2.13}$$

with χ representing the non-linear parameter for the downconversion process, $\mu = \sinh^2\chi$ being the average photon pair number, and $|n,n\rangle$ symbolizing n photon pairs encoded in two optical modes.

2.4.3 Channel And Detection

As we mentioned at the beginning of section 2.4, there are no assumptions about the way the quantum channel for QKD should be implemented. However, to increase the performance of QKD, two different types of optical communication technology are used:

- optical fiber;

- free-space optics.

A fiber-based QKD implementation is usually adopted to solve several problems, such as chromatic dispersion, polarization mode dispersion, birefringence and so forth[**?**].

A free space optic implementation of the channel also has its advantages. Several atmospheric transmission windows have a low loss with particularly low attenuation and where the decoherence of polarization is negligible. However, the weather conditions heavily influence the losses of free space and the apertures of the sending and receiving telescopes are affected by factors like alignment, movements and atmospheric turbulence, all of which negatively impact the performance of free space QKD.

Finally, as far as the implementation of the detection process is concerned, threshold detectors are used for single-photon detection. These detectors can only differentiate vacuum (zero photons) from single-photon or multi-photon cases and their efficiency ν is not 100% of course. As a result, some non-vacuum signals won't be detected while some vacuum ones will be instead and this affects QKD performance.

2.4.4 Post-Processing

Post-processing allows the two communication parties, Alice and Bob, to distil a secure key from the raw data measured during quantum transmission with the help of classical public discussions.

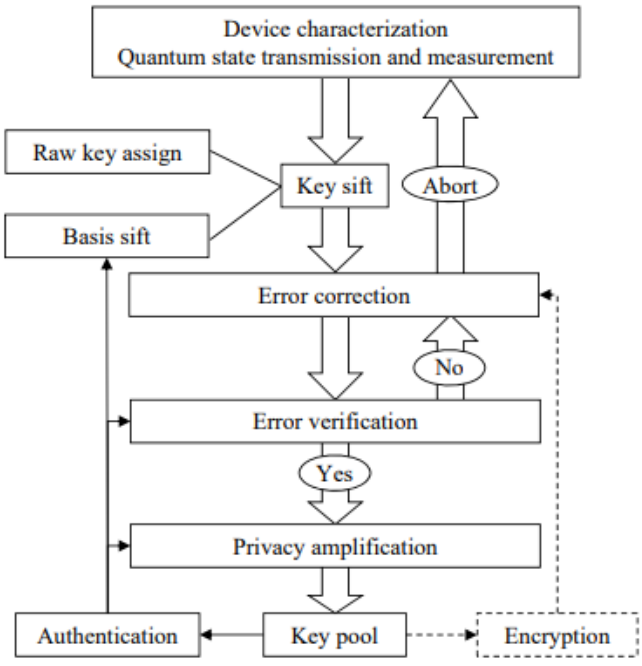


Figure 2.1: Post-processing Flow Chart[20]

The flow chart of QKD post-processing is shown in figure 2.2.

The first step is the raw key assignment, which depends on the adopted QKD scheme. As far as the protocols we will analyse in the next chapter are concerned Alice and Bob usually discard all the no-clicks, randomly assign double-clicks of the source and use the rest to generate a raw key.

The basis sift step is as straightforward as the name says. Alice and Bob will communicate to compare their choices of the basis used to measure the qubits used in quantum communication. Error correction and error verification are the processes carried out in post-processing by Alice and Bob to make sure, at the end of them, they share the same key, always with a certain failure

probability. Finally, privacy-amplification (PA) is the process carried out to extract a secret key from the identical shared one.

As shown in figure 2.2, authentication is required for three steps:

- basis sift;
- error verification;
- privacy amplification.

Encryption for error correction is optional depending on whether later on, more or less PA will be performed. In practice, there is an advantage to using error correction without encryption, since if Alice and Bob abort the QKD procedure after error correction, no pre-shared secret bits will be lost due to encryption.

Now we will just have a brief overview of two of the most used error correction techniques used.

A widely used error correction protocol for QKD is the Cascade protocol[21]. This protocol is simple and highly efficient and it is able to achieve an error correction factor of around $1.1 \sim 1.2$ for a large QBER range from 0% to beyond 11%.

In the Cascade protocol, Alice and Bob divide their sift key bit strings into blocks and compare the parities of each block to look for errors. Whenever the parity of a block is different, they perform a binary search to locate the error. This process repeats a few times with different block sizes and permutations to ensure all of the error bits are corrected.

The Cascade protocol is highly interactive because the binary search requires $1+\log_2 n$ communications and successful error correction often requires several passes As a result, several improved versions of this protocol have been proposed to reduce the interaction rounds[22, 23, 24].

On the other hand, forward error correction is a family of error-correcting codes, where only one syndrome needs to be sent from Alice to Bob. As a result, given its light classical communication load, forward error correction is implemented in many commercial QKD systems. Low-density parity-check (LDPC) codes[25] represent a great example of forward error correction.

They work well for QKD due to their high error correction efficiency and very limited communication requirement. The design and optimization of LDPC codes in QKD postprocessing can be divided into three steps:

1. Finding a good degree distribution for the target error rate[26, 27].
2. Generating a good parity-check matrix[28].
3. Decoding with Bob's key string and the received syndrome[29].

The standard LDPC algorithm is only optimum at its designed rate for the designed QBER. However, in real QKD QBER usually fluctuates from round to round. As a result, the rate compatible version of LDPC codes[30] is employed. In particular, this process leverages puncturing and shorting in order to find the best code rate suitable for the actual error rate.

Besides efficiency, another important factor of error correction is throughput. The main factors affecting throughput performance are the highly interactive communications in the case of the Cascade protocol and the computational cost in iterative decoding as far as LDPC codes are concerned. It was reported the throughput with both the Cascade and the LDPC codes can be higher than $10\,Mbps$[31, 32].

2.5 Quantum Hacking

Finally, this last section of the chapter will be dedicated to the analysis of the most popular attacks QKD systems can be subject to. In particular, an eavesdropper may try to exploit the imperfections in real implementations of QKD systems to perform the so-called quantum hacking. It is important to note that quantum hacking isn't covered by the QKD security proof since it leverages the practical defects of real QKD systems.

As far as the source is concerned, in a standard QKD scheme, we assume Alice, to be placed in a protected laboratory where state preparation can be correctly carried out. Imperfect state preparation may indeed leak information about the secret key and unfortunately, practical implementations of this process can be subject to errors due to imperfect devices or Eve's disturbance[33, 34, 35, 36, 37].

2.5.1 Photon-number splitting attack

The photon-number-splitting (PNS) attack[33] is the first well-known kind of hacking strategy that aims at exploiting an imperfect photon-source. In section 2.4.2 we have seen how weak coherent pulses (WCPs), that are generated by a highly attenuated laser, are widely used in QKD implementations of a photon source. In particular, since the photon number of phase-randomized pulses follows the Poisson distribution expressed in eq. 2.12, multiple-photon pulses, i.e. those pulses containing two or more photons, can happen with a probability $> 0\%$. As a result, exploiting these particular pulses, an eavesdropper, Eve, could perform a PNS attack.

For each WCP, a quantum non-demolition (QND) measurement is used to obtain the photon number information. Depending on the result of such measurement, Eve either blocks the one-photon pulse or splits the multiple-photon pulse in two storing one f the two particles and sending the other to the communication receiver Bob. As a result, thanks to multi-photon pulses, Eve is able to gain information about the secret key exchanged, without introducing any kind of disturbance that could reveal her presence to the communication parties.

This attack alone was enough to make researchers in the field have doubts about the future of QKD and its practical implementation with a WCP source. Fortunately, nowadays, modified versions of popular QKD schemes have been proposed to properly address this attack[17, 38, 39].

2.5.2 Time-Shift Attack

The detection component of a QKD system is much more vulnerable to quantum hacking attacks than the source is. This is due to the fact that an eavesdropper Eve is assumed to be in control of the channel and can send any signals to the receiver Bob who has no choice but to receive them. As a result, a significant amount of attacks have been proposed to hack detectors in QKD systems, in particular single-photon detectors (SPDs).

A time-shift attack[40, 41] may take place when SPDs, in "gated" mode, are used during a QKD process. In particular, these devices are characterized by different parameters but the most important one is *Dark count rate* which represents the false detection of photons on average. A good detector should have a low dark count rate. To minimize this parameter, different techniques can be used and "gated" mode operation is one of these. In gated mode, the detector is only activated during a small window of time whenever a photon is expected to be received. Any photons arriving outside of such a window won't ve detected. As a result, when operating in "gated" mode, synchronization between the two parties of the communication is crucial.

Generally, in QKD schemes, two detectors are used to receive quantum states: one to detect "0" bits and the other for "1" bits. Even if the same signal is used to synchronize the detectors, some electrical imperfection and other factors may cause a different time response of the devices. If the timing mismatch is not negligible, the following situation may arise: there may be some moments in which just one of the two detectors is enabled to receive a pulse.

.

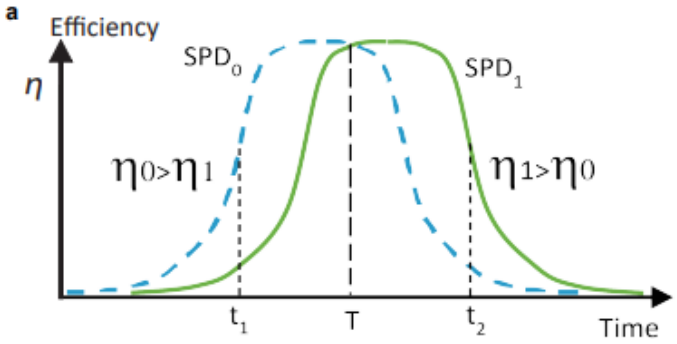


Figure 2.2: Time-dependence efficiency of Single Photon Detectors (SPDs)[20]

Let's consider, as shown in figure 2.2, t1 as the instant of time when the first detector ("0" bits one) is the only one active while t2 represents the moment the second detector is the only one enabled for operation.

Eve can simply shift the arrival time of each pulse sent from Alice to arrive either at t1 or t2. As a result, whenever receiver Bob detects a signal, Eve can be sure for the signal to be a 0, if she chose t1, or a 1 if she chose t2. Moreover, since no measurements were made over the intercepted quantum signals, the two communication parties have no way of detecting an ongoing attack.

However, knowing the main flaws of SPDs, it is possible to develop the right countermeasures to restore the expected security of the protocol. Proposed solutions focus on the difference between the response curves of the detectors and the removal of leaked information through privacy amplification techniques. Besides, it is also possible to randomly switch the bit assignment between the two detectors, to prevent the eavesdropper from assigning bit values to the retrieved information[42].

2.5.3 Man In The Middle Attack

A Man In The Middle attack over a quantum device happens during a process that is carried out before the actual key exchange: the quantum channel calibration process.

As we saw in section 2.5.2, a QKD system usually employs multiple detectors operating in "gate" mode to keep the *dark count rate* low. When it operates in "gate" mode, a detector is activated only for a small window of time when reception of quantum signals is expected, during the rest of the time the detector is turned off. Due to the unique electrical characteristics of each device, activation time may differ. Hence, a calibration step is required. During the calibration process, multiple signals are sent to estimate channel length and delay between the arrival timing of pulses at different detectors. It is important to note that the activation timing of each detector is scanned independently to find out when the count rate is maximum[43].

A single pulse per system cycle is usually used as a calibration signal and this kind of signal does not contain any information. As a result, we have no way of telling whether calibration signals were sent by the actual source of the communication or by an untrusted third party. This allows the possibility for the Man In The Middle attack to take place. In particular, the calibration signal may be substituted with one that contains more than one pulse per cycle to create a disparity between the activation timing of the detectors. an eavesdropper may replace the signal with another one containing more than a pulse per The different activation timings obtained will allow the attacker to perform other attacks to gain information about the exchanged secret key. The time-shift attack explained in section 2.5.2 represents a suitable option.

As far as countermeasures are concerned, software monitoring of the relationship between activation timings of the detectors represents the best prevention strategy against this kind of attack. Besides, it is worth mentioning that measurement device-independent QKD[44] offers a

way to eliminate detector side channels altogether. A brief overview of this branch of QKD will be given in the next chapter.

2.5.4 Additional Attacks

Finally, other attacks worth mentioning are:

Double-click attack. A double-click event occurs when the two different detectors that are used in a QKD system to distinguish "0" and "1" pulses click at the same time, i.e. they both detect signals. As a result, the QBER will be increased by 50% whenever either one of the signals is selected. An easy way to prevent this from happening is to just discard signals detected as a result of a double click. However, this opens the possibility for an attack to take place. In particular, it is possible to flood the polarization beam splitter of the receiver with multiple photons causing a double click to happen. Consequently, if an eavesdropper carries out this procedure whenever the receiver tries to perform a measurement in a different basis from his, he will be able to gain information about the secret key exchanged during the communication while keeping his presence completely undetectable. A simple solution to this attack would be not to discard qubit measured during double-click event even if it increases QBER[45, 41].

Trojan horse attack. An eavesdropper can attack the communication source, Alice, by sending bright Trojan-horse pulses to gain information about the basis she selected. In particular, this information is carried by the back-reflected pulses coming out of Alice[46, 41]. Consequently, a correct design of the system becomes crucial, filters, isolators and privacy amplification methods need to be employed to keep information leakage below a given threshold.

Denial of service attack. This attack consists of preventing the communication parties from generating a key altogether. A very simple implementation of a DoS attack to QKD would be just cutting the optical fiber composing the quantum channel[47]. However, quantum networks development aims to provide the possibility to route quantum signals over different paths just like it happens with routing in current IP networks.

Now, that we have provided an overview of the different attacks real QKD systems can be subject to, it is possible to note how to avoid each attack, different and specific countermeasures have to be taken. To address this issue, different branches of Quantum Key Distribution have originated and a brief overview of them will be given at the beginning of the next chapter.

Chapter 3

QKD Protocols

3.1 Different Types of QKD Protocols

QKD protocols differ from one another according to various criteria. One of these criteria, and possibly the most important one is the way to encode the information you want to send.

Discrete Variable QKD protocols were the first and most popular QKD protocols that were introduced. These protocols, as the name suggests, exploit discrete variables like relative phase but especially polarization to encode the information to be sent.

It's important to note that QKD protocols can be further divided in:

- prepare-and-measure protocols;
- entanglement-based protocols.

In particular, entanglement based protocols leverage a source of entanglement EPR particle pairs, quantum error correction and the CHSH inequality notion within the quantum theory to make sure the two parties involved in the communication can share the same information about the key without, at the same time, allowing any possible eavesdropper on the channel to gain any insight on it.

On the other hand, prepare-and-measure protocols, as the name states already, move the error correction and privacy amplification steps after the measurement of the received quantum particle. This allows the classical information, i.e. a bit, to be extracted first and therefore, the last two steps of QKD won't require any quantum memories anymore since they will correspond to classical error correction and privacy amplification.

Furthermore, over the years, more branches of QKD were developed, each with the goal of coming up with solutions to one of the problems presented by the practical implementation of QKD. In particular, two important branches are:

- device independent QKD (DI-QKD)[48, 49];
- measurement-device-independetn QKD (MDI-QKD)[50].

The main advantage of DI-QKD[49] is that the measurement devices don't need to be characterised. As a result, a much smaller number of assumptions is required to achieve security. MDI-QKD[50] was developed to allow removing detectors side-channel attacks altogether and compared to full DI-QKD, its key generation rate is a lot larger.

This section will mainly analyse and describe both prepare-and-measure and entanglement based DV-QKD protocols since they are still the most popular ones.

3.1.1 BB84 and Six-State Protocol

The best-known QKD scheme is the Bennett-Brassard 1984 (BB84) protocol[51, 20]. The protocol allows two users sharing both a quantum and an authenticated conventional classical channel, to generate a secure key even if an eavesdropper with unlimited quantum computing powers is present.

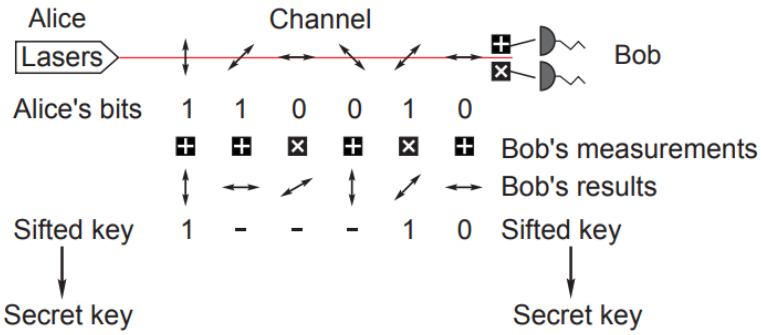


Figure 3.1: BB84 protocol scheme[20]

Figure 3.1 illustrates an example of the way the BB84 protocol works. First of all, for each signal she wants to transmit, Alice randomly chooses between four polarization states expressed by the two complementary bases she chose to adopt for the communication. Then she encodes a single photon using the selected state and she sends it over a quantum channel to Bob. In particular, as far as the choices of which bases to adopt is concerned, many different alternatives are possible but the BB84 protocol usually employs $\{|0\rangle, |1\rangle\}$ (the Z basis) and $\{|+\rangle, |-\rangle\}$ (the X basis) which correspond to: 0°, 90°, 45°and 135°.

Upon reception of each qubit, Bob will also randomly pick between the two employed bases to measure the polarization of the received photon. Once this detection step is over, both Alice and Bob will leverage an authenticated classical channel to publicly announce what basis they respectively chose for each qubit.

After that, they both discard the polarization data that was encoded and detected using mismatching bases and the remaining data represents the so-called *sifted key*. Finally, they select and compare a random sample of the sifted key bits to compute Quantum Bit Error Rate (QBER). If the value of this metric is too high, they abort. Otherwise, they apply classical post-processing techniques such as error correction and privacy amplification to distil a secret key.

In an ideal scenario, any $QBER > 0\%$ value would point to the presence of an eavesdropper Eve, over the communication channel the two parties used. However, in a realistic situation and given the imperfections of practical implementations of QKD systems, noise is always present. As a result, Alice and Bob won't be able to distinguish between genuine errors and errors due to Eve's meddling.

A pessimistic to deal with all of this would be aborting the protocol every time an error is detected. However, this would mean never being able to establish a secure key for the communication parties. Therefore, instead of detecting the presence of Eve, we need to focus on how we can still manage to exchange a secret key given her presence on the channel.

Finally, it is important to note that over the years, different modified versions of the BB84 protocol were developed to achieve various goals. In particular, the so-called six-state protoco[52, **?**] consists of a version of BB84 extended to use six polarization states on three bases: X, Y and Z. This modification enhances the key generation rate and the tolerance of the protocol to noise. Moreover, it creates an obstacle for Eve who, having to guess among three bases rather than just two, will produce a larger rate of error, thus becoming easier to detect.

3.1.2 Decoy-State Protocol

As we have seen, photon-number splitting (PNS) attacks represent a serious threat to practical implementations of QKD systems. They essentially allow an eavesdropper Eve, to leverage multi-photon pulses emitted by the practical realization of the communication source to gain information about the secret key exchanged.

The decoy-state method[53, 41] is essentially a modified version of the BB84 protocol using a variety of different intensities of photons to encode information. Taking into consideration a practical setup, Bob's gain consists of a weighted average of all detected photons, including empty pulses too and it can be mathematically expressed as follows:

$$Q_m = \sum_{i=0}^{\infty} Y_i exp(\mu) \frac{\mu^i}{i!} \tag{3.1}$$

with Y_i being the probability that Bob successfully detect the i-photon pulse sent by Alice. Besides, the contribution multi-photon pulses have on QBER can be written as:

$$E_m = \frac{1}{Q_m} \sum_{i=0}^{\infty} Y_i exp(\mu) \frac{\mu^i}{i!} \tag{3.2}$$

the values of Q_m and E_m are the only ones that can be determined by the communication parties in a practical implementation, they have absolutely no information about the values of Y_i. However, if we were to use different values of the light intensity, μ, then two systems of linear equations based respectively on eq. 3.1 and eq. 3.2 can be deduced. In particular, this two system will allow obtaining the following solution sets: $\{Y_0, Y_1, ...\}$ and $e_0, e_1, ...$ It is now a rather straightforward matter to determine Q_1.

As a result, Alice could send to Bob photon pulses with varying intensities of light while intending for just a specific one to be preferred for key bits while the others act as a decoy, i.e. decoy states. Since these are randomly chosen, Eve won't be able to know which photons were used for key purposes while the communication parties, leveraging linear algebra, will deduce such values without any issues.

In particular, it has been shown that even just two decoy states are enough to properly implement the decoy-state protocol[54].

3.1.3 SARG-04

SARG04[55, 56, 41] is a modified version of the BB84 protocol that, differently from the decoy-state one, employs a different classical communication process.

In particular, the first step of the protocol where Alice randomly chooses and sends one out of four polarization states from two complementary bases and Bob measures it, remains the same.

In the second step, however, when Alice and Bob determine for which bits their bases matched, Alice publics a pair of non-orthogonal states, one of which corresponds to what she used to encode her bit.

As a result, the next step of the protocol will be Bob's trying to deduce the state of the pair that Alice used for information encoding.

In SARG04[41], this process consists of the application of the 'unambiguous state discrimination" (USD) between states in the announced pair. Let's introduce an example to gain a better understanding of what happens.

Let's suppose that Alice transmits $|0\rangle$, Bob measures it in the X basis and that the pair of states published by Alice is the set $\{|0\rangle, |+\rangle\}$. Bob, depending on the result of the measurement he performed, will be able to tell the state Alice sent.

Alice's disclosed state	**Bob's X measurement value**
$\|0\rangle$	$\|+\rangle$ with 50% probability, $\|-\rangle$ with 50% probability
$\|+\rangle$	$\|+\rangle$ with 100% probability, $\|-\rangle$ with 0% probability

Table 3.1: Possible Bob's measrement outcomes

As we can see in table 3.1, if the result of Bob's measurement is $|+\rangle$, then he won't be able to infer Alice's state conclusively since the output he obtained could have been equally generated by the measurement of either state of the pair disclosed by Alice.

However, if the obtained result is $|-\rangle$, then can directly store it to later perform post-processing on it since it could have only been generated by the measurement of the $|0\rangle$ state, i.e. the state used by Alice to encode information is $|0\rangle$ for sure.

3.1.4 B92

The B92[57] protocol uses only two states to distribute a secret key between the remote parties. In particular, Alice will prepare a qubit in one of two different quantum states, $|\psi_0\rangle$ and $|\psi_1\rangle$ that respectively correspond to the "0" bit and to the "1" bit. to Bob, who measures it in a suitable basis, to retrieve Alice's bit[41].

The main peculiarity of this is the fact that the $|\psi_0\rangle$ and $|\psi_1\rangle$ states used to perform information encoding are non-orthogonal, i.e. $\langle\phi_0|\phi_1\rangle = s \neq 0$. If they were, Bob would always be able to recover the bit Alice encoded but, an eavesdropper Eve could do the same. In particular, she could intercept the states during transmission over the quantum channel and after having retrieved the information they carry, she could prepare brand new states identical to the ones she just measured and forward them to the appropriate receiver. As a result, the communication parties won't be able to detect the presence of Eve.

On the other hand, in the case of non-orthogonal states, measurements will be bounded by uncertainties but thanks to "unambiguous state discrimination" (USD)[58, 59] the communication parties will be able to overcome this issue and distil a common bit from the data. This essentially represents the principle upon which the whole security of the B92 protocol is based. To understand this process it is sufficient to remember that the state $|0\rangle$ ($|+\rangle$) is an eigenstate of Z (X) and that $|\pm\rangle = (|0\rangle \pm |1\rangle)/\sqrt{2}$.

bit	Alice	Bob(Z)		Bob(X)	
0	$\|0\rangle$	$\|0\rangle$, $Pr = 1$	$\|1\rangle$, $Pr = 0$	$\|+\rangle$, $Pr = 1/2$	$\|-\rangle$, $Pr = 1/2$
1	$\|+\rangle$	$\|0\rangle$, $Pr = 1/2$	$\|1\rangle$, $Pr = 1/2$	$\|+\rangle$, $Pr = 1$	$\|-\rangle$, $Pr = 0$

Table 3.2: Schematic representation of Bob's outcomes and their probabilities (Pr) depending on Alice's encoding state and Bob's chosen basis for measurement[41]

As we can see in table 3.2, depending on what bit of information Alice sent, there is always a state Bob will never be able to obtain after measuring the received data in both of the bases employed by the B92 protocol. Therefore, this specific information can be exploited to infer what state was originally prepared by Alice. In particular, table 3.2 illustrates an example of the B92 protocol that employs X and Z as bases and uses:

$$|\psi_0\rangle = |0\rangle \, |\psi_1\rangle = |+\rangle \tag{3.3}$$

As a result, Bob can deduce that when $|1\rangle$ is detected, Alice must have prepared the state $|+\rangle$, so he decodes the bit as "1", whereas when $|-\rangle$ is measured, Alice must have prepared the state $|0\rangle$ so he decodes the bit as "0". Whenever Bob detects any other state, Bob is unsure of Alice's preparation so users just discard these "inconclusive" events from their records

Finally, it's important to note that, any couple of non-orthogonal states can be used in the B92 protocol, even mixed, disjoint subspaces of the Hilbert space[57, 41]. However, to maximize performance, states are usually derived from two mutually unbiased bases (MUBs).

As far as performance is concerned, since it only employs two non-orthogonal states, B92 proves to be much more loss dependent and it is much less noise tolerable if compared to classical BB84.

3.1.5 E91 and BBM92

E91 was first introduced in 1991 by Artur Ekert[60, 41]and its security is guaranteed by a Bell-like test to rule out the presence of any eavesdroppers.

In particular, the scheme of the protocol assumes to have a single source that emits pairs of entangled particles, each described by a Bell state, in particular, the singlet state $|\Psi\rangle = (|01\rangle - |10\rangle)/\sqrt{2}$. Then, the twin particles consisting of polarized photons, are separated and sent to Alice and Bob, i.e. the communication parties get half of each generated pair.

Upon reception of each particle, both Alice and Bob will randomly choose a basis to measure their half of the pair. The employed bases are selected in accordance with the Clauser, Horne, Shimony and Holt (CHSH) test[61] which will later allow us to verify the presence of Eve over the communication channel. In particular, in E91, the communication parties can choose between three different bases: Z, $(X + Z)/\sqrt{2}$ and X.

Next, as happens in BB84[51], Alice and Bob will disclose in clear the different bases they chose and used for measuring. However, in E91[60, 41], the communication parties will leverage the instances where mismatching bases were used for measurements and thanks to this data, they will be able to check for Eve's presence. In particular, the violation of the CHSH quantity**??** is checked:

$$E = \langle a_1|b_1\rangle - \langle a_1|b_3\rangle + \langle a_3|b_1\rangle + \langle a_3|b_3\rangle \tag{3.4}$$

where $\langle a_i|b_j\rangle$ represents the expectation value when Alice measures using a_i and Bob, b_j . If the inequality $-2 \leq E \leq 2$ holds, it means that the received photons are not truly entangled and this could either be due to Eve's presence or to the measurement device having some issues. On the other hand, if there is no eavesdropper, the expected value of E is the maximal violation $-2\sqrt{2}$. If this is the case, the communication parties will proceed to process the results obtained from instances of communication where the basis choice was the same into a shared secret key.

It is important to note that while QKD usually relies on the no-cloning theorem and the inability to exactly distinguish two non-orthogonal states from one another, the E91 protocol leverages the non-local feature of entangled states in quantum physics and Eve's presence can be considered as an event inducing elements of physical reality that affect the non-locality of quantum mechanics[41]

As far as the BBM92[62, 41] is concerned, this protocol was essentially introduced to operate a critique about E91è91.

Both BB92 and E91 assume to have a source that can provide each party involved in the communication with half of the generated entangled pairs. However, the latter employs just two different mutually unbiased bases (MUBs) instead of three therefore, it can work more efficiently than the former. Besides, the two bases can be chosen to be the same ones BB84[51] uses.

In BB92, after receiving the various halves of the generated entangled pairs, Alice and Bob select the instances where they chose the same basis to obtain correlated measurement results and be able to extract a secret key, a sample of which will be publicly disclosed later to check for errors and Eve's presence as well.

The main idea behind the security of this protocol is that Eve cannot become entangled with Alice's and Bob's qubits without introducing any errors in their measurements and therefore, there would be no need for the communication parties to commit to a Bell test.

It is interesting to note how if we had a BB92 protocol scheme where Alice possesses the source of entangled pairs, her measurement in a random basis would essentially correspond to preparing

the state to send Bob As a result, without a Bell test, we are left with an instance of the BB84 protocol. In particular, there is no way of telling whether Alice started by measuring part of a Bell state or by preparing a qubit state using a random number generator[41].

Employing a QKD protocol that uses entangled pairs or not, has no consequences as far as standard eavesdropping on the main communication channel is concerned. However, it is important to note that a protocol with a Bell test provides a higher level of security because it allows relaxing the assumption that the communication parties have control over the other degrees of freedom of the quantum signals[41]. .

Chapter 4

QKD Simulation

4.1 Overview

In the previous chapter, we analysed what Quantum Key Distribution is, why it is secure, and we went over what it allows to achieve for security if properly implemented.

As we now know, in a practical implementation of QKD, many factors come into play that could compromise security and allow an eavesdropper to gain valuable information about the ongoing communication between two parties.

Deploying QKD in the real world isn't easy nor cheap at all and that is exactly why more and more different simulators are getting developed and implemented.

The goal of QKD Simulation is exactly to allow whoever wants to start utilizing QKD inside of their infrastructures, to be able to evaluate what the performance of the communication would be and to test the many different protocols available, before having to spend any money on all the equipment needed for QKD implementation.

Nowadays, many different simulators, each having a different final goal and features, have been developed and have been proposed to the public. In the following sections, we will briefly describe and go over some of the most famous QKD simulators made available to the public.

4.2 QuISP

The Quantum Internet Simulation Package (QuISP)[1] is an event-driven simulation of quantum repeater networks developed by the Advancing Quantum Architecture (AQUA) research group. QuISP[63] aims at addressing concerns about a future Quantum Internet such as:

1. evaluating the completeness and robustness of paper protocols designed for Quantum Internet;
2. the connection architecture of quantum networks and their performance prediction;
3. dynamic behaviour of quantum networks and protocols as conditions change over time;
4. emergent behaviour. As the Quantum Internet scales up, new and unexpected behaviour may emerge that current models of quantum networks didn't take into account.

As we can see in figure 4.1 QuISP's approach to simulating large-scale quantum networks is based on OMNeT++[64], a modular, component-based architecture simulation environment. The OMNeT++ model consists of modules that communicate via messages and it has the following features:

[1] https://aqua.sfc.wide.ad.jp/quisp_website/

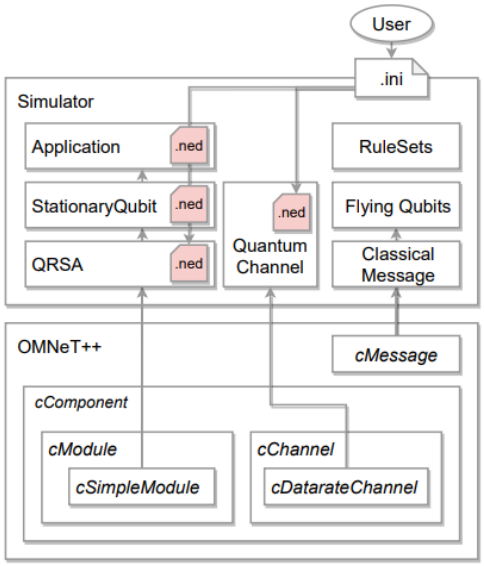


Figure 4.1: QuISP architecture scheme[63]

- the first batch of messages is created during module initialization;
- upon receiving a message, which represents an event, the module at issue processes it and usually creates another batch of messages;
- the simulation ends when there are no messages left to be processed or the simulation reaches the time limit set by the user.

Modules and messages in OMNeT++[64] are provided by the `cSimpleModule` and `cMessage` class, accordingly. In particular, as we can see in figure 4.1, QuISP's software modules like the QRSA and Application one are built on top of cSimpleModule, which supports standard message handlers. The Stationary Qubit module is also implemented as a subclass of `cSimpleModule`, but it keeps its quantum properties as its variables and behaves as a quantum memory. These software components can emit registered classical packets written in a `cMessage`. On the other hand, Flying Qubits represents an emitted message from the Stationary Qubit module. Therefore, its implementation is an extension of a classical packet passed through quantum channels. Finally, the Quantum Channel module representing the carrier of the flying qubit is implemented as a subclass of `cDatarateChannel`. QuISP[63] currently supports end-to-end tomography-based quality analysis of connections as the application. Users can leverage a .INI file to provide the different simulation parameters that will be used to configure the different modules. The simulator, when it's done working, returns the performance statistics at the end of simulation when the target configuration contains tomography. Moreover, it is important to note that QuISP doesn't keep track of the entire quantum state of qubits but it works on the premise that the desired quantum state is known. As a result, it only tracks deviations from this ideal state in the form of errors that are affecting it.

Other simulators like NetSquid[65] are limited by focusing on a physically realistic simulation of a single, small network. QuISP is designed with internetworking in mind while maintaining full physical realism. The long-term the goal for the QuISP's simulator is to be able to handle an internetwork with 100 networks of 100 nodes each, with each network running independent error management protocols, hardware parameters, and topology.

4.3 NetSquid

NetSquid[2]: the Network Simulator for Quantum Information using Discrete events, is a software tool for the modelling and simulation of scalable quantum networks developed at QuTech.

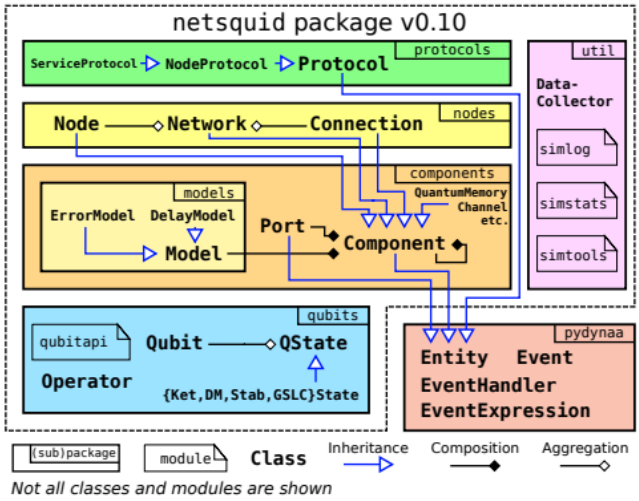


Figure 4.2: NetSquid architecture scheme[65]

As we can see in figure 4.2 NetSquid[65]'s Python package is composed by the following sub-packages:

- pydnaa package;
- qubits package;
- components and nodes packages;
- protocols package.

The `pydnaa` package is a software library to provide NetSquid's simulation engine. This package is separate from the main NetSquid's one, it is inspired by classical networks simulators and, instead of moving forward one little step at a time, it jumps between events. That is why it provides discrete event simulation. In Netsquid, Entities involved in a simulation can dynamically generate these events as time goes by.

The `qubits` package used by NetSquid[65] was developed in-house and it is *qubit-centric* differently from other libraries for quantum computation where the quantum circuit paradigm is used. As a result, it manages to be representation agnostic as far as the quantum state of a qubit is concerned, i.e.. users can choose to represent the quantum state of a qubit in any of the possible ways such as statevector, density matrix...

The *components and nodes* packages offer composable physical models used to create your basic quantum network components.

The `protocols` package offers a way to describe classical control over the quantum network, i.e. specify the communication protocols the nodes of the quantum network should follow.

Simulating a quantum network with this simulator is generally performed in three steps:

1. firstly, the network is modelled using a modular framework of components and physical models;

[2] https://netsquid.org/

2. next, protocols are assigned to network nodes to describe the intended behaviour;
3. finally, the simulation is executed for a typically large number of independent runs to collect statistics with which to determine the performance of the network.

Let's compare NetSquid[65] to other existing quantum network simulators. Differently from SimulaQron[66] the purpose of which is application development, Netsquid tries to provide realistic physical models of channels and devices or timing control. Netsquid is a monolithic solution while Simulaqron is meant to be run in a distributed way on physically-different classical computers.

Similarly to NetSquid, QuISP aims to support the investigation of large networks that consist of too many entangled qubits for full quantum-state tracking. However, QuISP's approach is to track an error model of the qubits in a network instead of their entire quantum state.

4.4 SimulaQron

SimulaQron[3] is a simulator providing an essential tool for software development for a quantum internet. Specifically, SimulaQron simulates several quantum processors, connected by a simulated quantum communication channels.

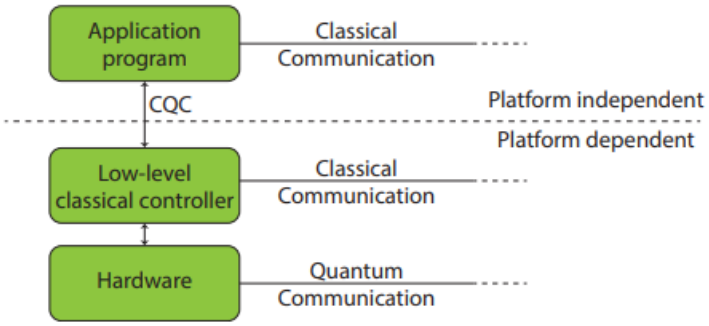


Figure 4.3: High-level schematic of one node in a quantum network[66]

As we can see in figure 4.3, a node in a quantum network is generally composed by three parts:

- application program;
- low-level classical controller;
- hardware.

In the most general case, the *hardware* part represents a quantum processor able to store and manipulate qubits. Moreover, this processor has an optical interface to perform quantum communication with the neighbouring nodes of the network.

The *low-level classical controller* is needed to manage the quantum processor and it possesses an interface to perform classical communication with the neighbouring nodes.

As figure 4.3 shows, the quantum hardware and the low-level control system form the platform-dependent quantum processing system. Since the implementation of quantum networks is still a very modern research topic, SimulaQron[66] represents a viable alternative for anyone who wants to develop applications for quantum networks or implement a quantum network stack. In particular, this simulator acts as a platform-independent substitute for the quantum processor and its control system.

[3] http://www.simulaqron.org/

The SimulaQron backend is composed of a client and server program running on each physically-different classical computer that wants to participate in the simulated quantum network. To be more precise, two servers are running on each node participating in the simulated quantum network:

- the SimulaQron server;
- the CQC server.

The *SimulaQron server* is in charge of handling the simulation of the quantum hardware and communication between multiple instances of this server is needed to simulate the quantum network itself.

The *CQC server* handle the simulation of the platform-dependent quantum processing system and it provides the Classical Quantum Combiner (CQC) interface to applications.

Any application wanting to use SimulaQron[66], will need to use the provided client to access the simulated network through the CQC interface. This interface is universal and exposes all of the "standard" quantum instructions as well as other special features for quantum networks.

SimulaQron[66] itself does not provide a new quantum simulator, but rather builds the distributed simulation on top of an existing one using a modular design. In its initial release, QuTip[67] was used for the underlying simulation.

Finally, SimulaQron does not aim to achieve an efficient simulation of a large scale quantum internet. Such low-level simulation capabilities which are of interest to quantum information protocol designers are the subject of other simulation platforms like NetSquid[65].

4.5 The Qiskit Framework

Qiskit is an open-source software development kit (SDK) that allows working with quantum computers at different levels[4]. In particular, it offers a tool to simulate quantum circuits on a local device. This makes the development of quantum programs much more accessible since nowadays, quantum technology is still expensive and hard to access. This framework has been developed in the Python programming language and it is available as any other Python module. Besides, the Qiskit community is spread worldwide and it offers a lot of support.

Qiskit framework was originally organized into four different components:

- Terra;
- Aer;
- Ignis;
- Aqua.

Terra[5] is the foundation on which the rest of Qiskit is built and it allows modelling quantum circuits using gates and pulses. This component is also made up of different modules and the most important ones are:

- *qiskit.circuit* is used to build and run quantum circuits. In particular, given a register of qubits, it allows applying a sequence of operations, i.e. gates, to it. In the beginning, the qubits in the register are all initialized to the $|0\rangle$ state and as they pass through gates, their quantum state evolves. Finally, measurement is leveraged to extract information about the quantum state of the qubits and map the outcomes onto classical registers;

[4] https://qiskit.org/documentation/

[5] https://qiskit.org/documentation/apidoc/terra.html

- *qiskit.pulse* is used to work with pulses. Pulses can be sent to a channel to carry out experiments to study how to reduce errors (e.g. error mitigation and dynamical decoupling);
- *qiskit.quantum_info* is used for utility purposes such as analysing circuits run on quantum computers. Thanks to this component it is possible to both estimate metrics and generate quantum states, operations and channels;
- *qiskit.visualization* is used to obtain an illustration of the generated circuits as well as plots quantum states and results of circuits. It is a utility module to gain a quick idea about the designed quantum circuit and it can be used as a visual debug tool too.

Aer[6] provides a high-performance simulator framework that can be accessed through different simulator backends. It can be used to run the circuits compiled using Terra. Moreover, it allows building noise models to be used for performing realistic noisy simulations of errors that may occur on real devices. The two main simulator backends, it offers, are:

- *AerSimulator* is used to perform the simulation of noisy quantum circuits. it supports multiple simulation methods and offers configurable options for each of them. In particular, the `method` option allows selecting the kind of simulation to run such as a dense statevector simulation that can sample measurement outcomes from ideal circuits or dense density matrix simulation that may sample measurement outcomes from noisy circuits;
- *PulseSimulator* is used as a pulse schedule simulator. In particular, the continuous-time Hamiltonian dynamics of a quantum system are simulated.

Nowadays, Ignis[7] is deprecated and it has been replaced by the Qiskit Experiments[8] project that allows carrying out different experiments:

- *characterization experiments*: are aimed at measuring parameters in the system such as noise;
- *verification experiments*: are used to verify the performance of gates and small circuits;
- *mitigation experiments*: are designed to generate mitigation routines by running calibration measurements.

Finally, Aqua is the highest level component of the four original ones and it allowed building algorithms for quantum computers, but it is deprecated as well and its functionality has been split out to separate application repositories, but its core algorithm and operator function have been directly included in Terra.

4.5.1 Simulation Of A Quantum Circuit

To describe the quantum circuit we want to simulate, we will use the *qiskit.circuit* module offered by the Terra component. First of all, we need to define our quantum circuits and this can be done by instantiating the QuantumCircuit class[9]:

```
circuit = QuantumCircuit(2, 2)
```

where the input parameters represent respectively, the number of quantum and classical (second parameter) bits the circuit is composed of. In a quantum circuit, we usually have both

[6] https://qiskit.org/documentation/apidoc/aer.html

[7] https://qiskit.org/documentation/apidoc/ignis.html

[8] https://qiskit.org/documentation/experiments/

[9] https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html#qiskit.circuit.QuantumCircuit

quantum and classical registers. Quantum registers are where quantum operations are carried out while classical ones are just needed to collect the results of the circuit.

Now that we have created our circuit, we can define the operations it has to perform by adding gates to it. In particular, the `circuit` object returned after circuit initialization, offers different methods to add gates.

Follows a list of the most basic ones:

- `z(qubit)`: it applies the Pauli-Z gate to the specified qubit of the circuit specified as the parameter;
- `x(qubit)`: it applies the Pauli-X gate to the specified qubit of the circuit specified as the parameter;
- `y(qubit)`: it applies the Pauli-Y gate to the specified qubit of the circuit specified as the parameter;
- `cx(control_qubit, target_qubit)`: it applies the CNOT gate to the specified qubits of the circuit;
- `measure(qubit, cbit)`: it measures quantum bit into a classical one.

Once we have added the desired gates, we can check if the resulting circuit has the structure we want to model by using the `draw` method of the `circuit object`

If everything is as desired, we are ready to perform the simulation on our circuit and to do so, we will use the Aer component. First of all, we need to select the backend that allows us to run the circuit and this can be done by instantiating the `AerSimulator` class and selecting the simulation method we want:

```
backend = AerSimulator(\simulation_method")
```

To perform the simulation, we need to pass our quantum circuit and backend to `execute` method as parameters. This method also allows us to specify the number of times we want our circuit to be executed to build up statistics about the distribution of the bitstrings.

```
job = execute(circuit, backend, shots=2048)
```

After circuit execution, depending on the simulation method that we chose, the results of the simulation will either be:

- the quantum state, represented by a complex vector of 2^n dimensions, with n being the number of qubits of our circuit;
- the $2^n \times 2^n$ matrix representing the gates in the circuit.

The `execute` method returns a `BaseJob` object and we can use it to collect the execution result in a `Result` object:

```
res =job.result()
```

Finally, we can retrieve the aggregated binary results of our circuit thanks to the `get_counts` method:

```
c = result.get_counts(circuit)
```

4.6 QSS QKD Simulator

The QSS (Quantum Software Stack) QKD Simulator[68] was designed and implemented to provide a different approach to the simulation of QKD protocols and it was implemented using the Python programming language.

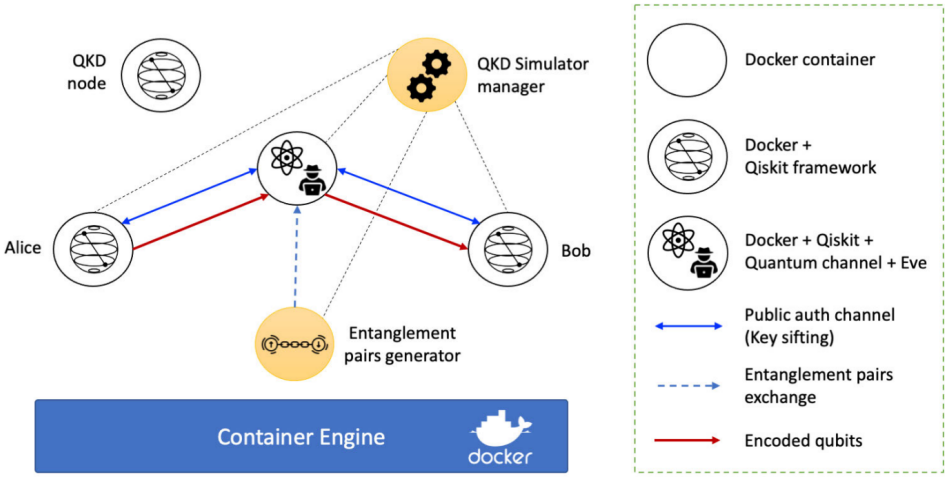


Figure 4.4: QKDSim High-Level Architecture[68]

As we can see in picture 4.4 the components of the simulator are:

- the *QKD node*: this is the component representing a QKD device, in particular, it acts as one of the communication parties trying to exchange a secret key. It has all of the software capabilities needed to simulate QKD and it also provides a serialization mechanism for quantum data and it exploits a TCP/IP network to interact with the other components of the simulator;
- the *Quantum channel and Eve*: this is a peculiar entity that the same quantum simulation software a QKD node has, but its aim is reproducing the effects of noise or an eavesdropper on the quantum channel connecting the communication parties;
- the *Entanglement pairs generator*: it generates maximally entangled pairs of qubits in the desired representation and it is employed in the simulation of entanglement-based protocols;
- the *QKD Simulator Manager*: this is a central unit used to retrieve data about the QKD simulation and it also has both a GUI and a CLI to provide an interface to configure the other components of the simulator and the simulation itself.

As far as the communication aspect of the simulator is concerned, a TCP/IP network is employed to exchange both classical and quantum data. In particular, the data that needs to be exchanged is serialized using the pickle Python framework and embodied inside an HTTP request to send. Each module of the simulator has a Flask web server running inside that exposes the REST APIs needed to make communication possible.

Moving to the quantum simulation aspect of the simulator, the Qiskit Python framework[69] is used to implement the simulation of qubit encoding their manipulation and their measurement as well. In particular, the `Statevector` class is employed for qubit encoding implementation and the `QuantumCircuit` class is used to model the quantum circuit implementing the QKD protocol desired for simulation.

Moreover, the simulator[68] includes the `QKD` abstract class that contains the following overridable methods;

- `begin`: this method is used to initialise a communication with the simulator;
- `startKeyExchange`: this method allows starting a point-to-point QKD simulation;
- `end`: this method gives the possibility to close an existing conncetion to the simulator.

As a result, it is easily possible to implement new QKD protocols to be simulated. Besides, it's important to note that the classical communication part of simulation needs authentication and to address this issue, the QSS QKD simulator provides a choice between two different methods:

- SPHINCS+: a post-quantum algorithm that is highly flexible since it doesn't need pre-shared keys;
- AES with Galois/Counter Mode (AES-GCM): a quantum-resistant authenticated encryption scheme that requires pre-shared keys instead.

As far as the simulated QKD protocols are concerned, the simulator supports BB84[51] and E91[60]. The BB84 scenario implemented by the simulator can be described by the following processes:

- the qubit exchange process;
- the key sifting process;
- the QBER estimation process.

In the *qubit exchange* process, Alice generates and initializes the qubits to send Bob for the secret key exchange. Then, for each qubit, she randomly picks a basis between the computational and the Handmard one to perform the quantum encoding process with and next, she sends the corresponding statevectors to Bob. After receiving the data, Bob for each qubit, will also randomly choose one of the bases mentioned before but it will use it to measure the qubit. This concludes the *qubit exchange* process.

During the *key sifting* process, Alice and Bob exploit the authenticated quantum channel to share information about the basis they respectively chose. Next, they exclude from the key the qubits where a mismatch of the basis choice is detected. This concludes the *key sifting process*.

The *QBER estimation* process happens right after both of the communication parties publish a subset of the bits used for the key. This process is performed by the simulator to be able to detect the presence of Eve or not. In particular, since the QSS QKD simulator doesn't model non-idealities of the quantum channel and noise, an increase of the QBER value over the threshold expected for the BB84 protocol can only indicate the presence of an eavesdropper Eve.

As far as the E91[60] protocol implementation is concerned, the simulator uses the *Entanglement pairs generator* module to obtain maximally entangled pairs. Then, the representation of the pairs is serialized and sent to the quantum channel where all of the quantum operations will be performed without sending any qubits to either of the communication parties. In particular, the *Quantum channel and Eve module* will measure the pairs according to the basis choices provided by the Alice and Bob nodes. After that, the bits are divided into two groups: one for the key and one for the verification of the CHSH inequality. If the latter is successful, the key is kept, otherwise, it is discarded.

The main purpose of the QSS QKD simulator, even if it can be extended, is specifically QKD simulation. Differently from the QuISP and NetSquid simulators, it doesn't aim to support the investigation of large quantum networks with Quantum Internet in mind and unlike SimulaQron it is not after application development for those specific platforms. However, it is worth noting that the previously mentioned simulators could be used to perform QKD simulation but performing this kind of simulation close to the practical case would mean managing a series of details that

are not of interest to those simulators at the moment. In particular, since each QKD system is strongly related to the real devices used to implement it, each of these devices needs to be characterized in a specific manner. For example, the features of a specific source need to be modelled from a mathematical point of view and this is not an easy process to perform with generic simulators like QuISP, NetSquid and SimulaQron. As a result, in order to easily keep track of the previously mentioned mathematical models and perform QKD simulation in an easier way, more special-purpose simulators like the QSS QKD Simulator have been developed.

Chapter 5

Kubernetes & RabbitMQ

5.1 Kubernetes

Kubernetes[70] is an open-source container orchestration tool, so it essentially allows to manage containers, being these docker ones or other technologies.

It helps manage applications made up of hundreds or even thousands of containers in different deployment environments, like physical machines, virtual machines or cloud environments, even hybrid deployment environments.

The rise of microservices caused increased usage of container technologies since they offered the perfect hosts for small independent apps.

This resulted nowadays in applications composed of so many containers that managing them across different environments, using scripts and self-made tools, can result pretty difficult.

That's why tools like Kubernetes are needed. They essentially offer the following features:

1. high availability or no downtime;
2. scalability;
3. disaster recovery, backup and restore.

5.1.1 Kubernetes Components

The main and essential components of Kubernetes are:

Node And Pod[1]. A worker node is simply a server, a physical or virtual machine and the basic component or smallest unit of Kubernetes is a Pod. A Pod is an abstraction over a container, it creates this running environment or a layer on top of the container and the reason for it is that Kubernetes wants to abstract away the container run-time or container technologies so that you can replace them if you want and you are not restricted to work with only one of them. Interaction will only happen with the Kubernetes layer. A Pod usually is meant to run 1 application container inside of it. Kubernetes offers a virtual network out of the box, meaning that each Pod, not container, gets its IP address and each of them can communicate with one another by using that address. It is an internal/private address, not a public one. However, Pod components in Kubernetes are ephemeral, meaning they can die very easily when the application inside crashes or when the node they are operating on runs out of resources. As a result, a new Pod will then be created in its place with a new IP address assigned to it. To solve the issues on communication this may cause, another component of Kubernetes will be used.

[1] https://kubernetes.io/docs/concepts/workloads/pods/

Service And Ingress[2]. A Service is a permanent IP address that can be attached so to say, to each Pod. The advantage is that the lifecycle of the Pod and the service are not connected, so even if the Pod dies, the service and its address will stay. Now, let's suppose we want to make our application accessible through a browser and to this, we would need to create an " external Service". An external Service is used to open the communication from external sources. There are things like your DB for example, you wouldn't want it to be open to public requests and for that, we would create an internal Service to attach to the corresponding Pod instead. If you want the URL that will be used to access your application to be more user friendly with secure HTTP protocol and a proper domain name, there is another component of Kubernetes that can be exploited: Ingress. It will receive the requests and do the forwarding to the appropriate service.

ConfigMap And Secret[3]. As we already explained, Pods will communicate by using Services, let's suppose, for example, e want our application to have a database endpoint called mongo-db-service to use to access the database. Usually, these endpoints or URLs are configured in application.properties files or in some external environmental variable, but they usually end up inside of the built image of the application. Therefore, if the service name of the endpoint would change to mongo-db for instance, you would need to adjust the corresponding URL in the application and so rebuild its image, push it to the repository, pull the new image in your Pod and restart it. To handle this issue, luckily, Kubernetes has a component called ConfigMap. It will act as your external configuration to your application., then you will just have to connect the ConfigMap to the Pod that will use it to get the configuration data contained inside of it. It is important to note that, since values are stored in clear, important credentials should not be put in a ConfigMap. For this purpose, Kubernetes has another component called Secret that is just like ConfigMap but it is used to store secret data as credentials or certificates encoded in a base64 encoded format. The data from ConfigMap or Secret can be used in your application through environmental variables or even a properties file.

Volume[4]. Let's suppose we have some data stored on our database and its associated Pod or container gets restarted. As a result, we will lose all of the data contained in our DB. To fix this issue we can use a specific component of Kubernetes called Volume. It attaches a physical storage on a hard drive to your Pod. That storage could either be:

- local Storage located on the same server node where you host your specific Pod;
- remote Storage outside of the Kubernetes cluster;
- cloud Storage;
- premise Storage that is not part of the Kubernetes cluster.

It's important to note that, unless it is specifically and appropriately configured, the Kubernetes cluster won't explicitly manage any data persistence.

Deployment And Stateful Set[5]. Now, as we said at the beginning of this section, one of the main advantages of tools like Kubernetes is High Availability, that is to say, no downtime. Therefore, to have that we usually have many replicas of our application Pod on different server nodes and all of them are connected to the corresponding and only Service component too. In such cases, the Service component also acts as a Load Balancer so it will catch the request and forward it to whichever Pod replica is the least busy. To create replicas of a Pod, you don't need to create them manually, but instead, it is sufficient to specify the number of desired replicas, when defining the blueprints for the Pod's deployment. The Kubernetes component allowing you to do such a thing is called Deployment. It represents one more layer of abstraction on top of Pods, and in practice, we will mostly be working with this and not Pods directly. However, a Pod running a

[2] https://kubernetes.io/docs/concepts/services-networking/

[3] https://kubernetes.io/docs/concepts/configuration/

[4] https://kubernetes.io/docs/concepts/storage/

[5] https://kubernetes.io/docs/concepts/workloads/controllers/

database container inside of it, cannot be replicated using a Deployment component. The reason for it is that a database has a state which, in such case, is the data it contains. As a result, the different replicas would need all to access the same data storage and therefore, a mechanism to manage concurrent reads and writes would be needed too, to avoid data inconsistencies. That mechanism and other features are offered by a different Kubernetes component called StatefulSet. This component is specifically meant for applications like databases and in general stateful ones. The only small disadvantage is that deploying databases using the StatefulSet component in a Kubernetes cluster is not so easy and that is why database applications are often hosted outside of the Kubernetes cluster.

5.1.2 Kubernetes Architecture

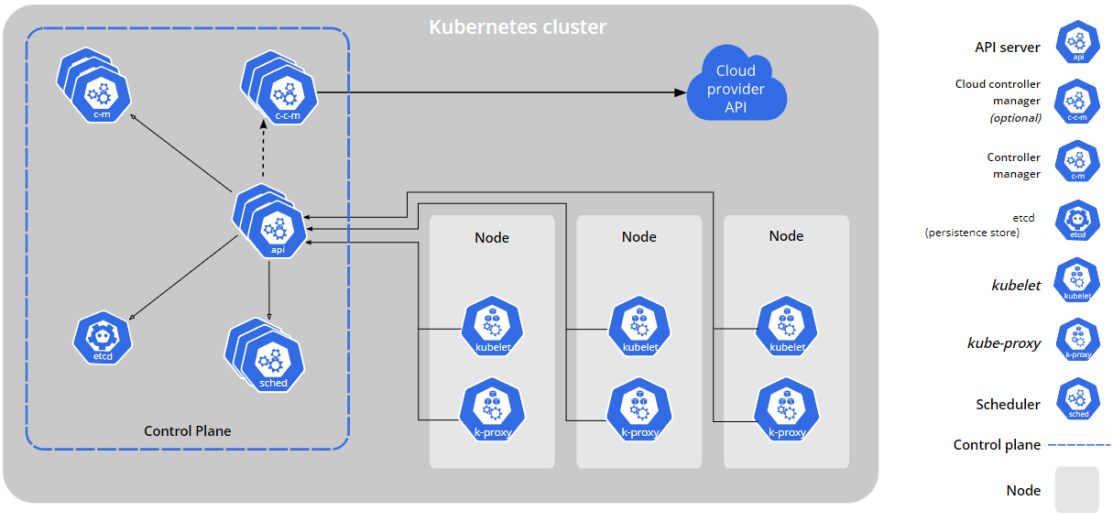


Figure 5.1: Kubernetes Architecture source: https://kubernetes.io/docs/concepts/overview/components/

Figure 5.1 illustrates the architecture of Kubernetes[6]. In particular, we can see how Kubernetes operates on two different types of nodes:

- the Master Nodes;
- the Slave or Worker Nodes.

One of the main components of Kubernetes architecture is its workers, servers or nodes. Each worker node will have multiple application pods with containers running on itself as well as three processes.

These processes are used to schedule and manage the Pods running on the specific node. As we can see, these nodes are the ones doing all of the work and that is why they usually are called Worker Nodes. Now, let's describe more in-depth the processes we just mentioned.

The first process needed on every node is the *Container Runtime*, like Docker for example or other technologies. This process is needed since containers are running in the Pods on the node.

Then, we have the process actually in charge of scheduling the Pods and the containers underneath them and its name is *Kubelet*. It is a process of Kubernetes itself that has interfaces with both the Container Runtime and the node, the machine itself, because, at the end of the day, Kubelet is responsible for taking that configuration and running or starting a Pod with a container inside and then assigning resources from the node to the container.

[6] https://kubernetes.io/docs/concepts/overview/components/

Usually, a Kubernetes cluster is made up of multiple nodes which also must have the two processes we just mentioned installed on them and they will also have multiple Pods inside. As we know, communication between Pods in Kubernetes happens through Service components. They act basically as a load-balancer catching the request directed to a Pod and forwards it to the respective Pod The third process is called *Kube Proxy* and is responsible for forwarding requests from Services to Pods. It has intelligent forwarding logic inside to make sure the communication also works in a performant way with low overhead.

Master nodes allow interacting with the Kubernetes cluster and thanks to them it is possible to start a Pod, perform monitoring, make a new Node join the cluster and so on. In particular, 4 processes are running on every Master Node.

The first process is *API Server*, you can use some client to interact with this process that acts as a cluster gateway, receiving the initial request of any updates into the cluster or even the queries from the cluster and it also acts as a gatekeeper for authentication to make sure only authenticated and authorized requests get through to the cluster. The API Server validates incoming requests and if everything is fine it will forward them to other processes to schedule the Pod or create the requested component. It is important to note that all of this is good for security since you only have one entry point to the cluster.

Another process is the *Scheduler*. If the cluster receives a request to schedule a new Pod, the API Server will forward the request to the Scheduler after validation. The Scheduler has its intelligent way of deciding on which specific worker node to schedule the next Pod or component based on the resources it will need, in particular, the ones you specified it needs. The process that then performs the scheduling, that is to say, it starts the Pod with a container is the *Kubelet* one on the designated worker node the specific Pod will be deployed, it gets the request from the Scheduler and executes it on the node.

The next process is the *Controller Manager*, it is needed to detect whenever Pods die on any node and to reschedule them as soon as possible. It detects state changes of the cluster such as crashing of Pods for example and it tries to recover the cluster state as soon as possible. To do that, it will request the Scheduler to re-schedule the dead Pods and the Scheduler will do its usual things.

Finally, the last process is *etcd* which is a key-value store of a cluster state. It is the cluster brain where every change in the cluster gets saved or updated. The data etcd contains is essential to allow other master processes to work properly and make the right decisions. It is important to note that the actual application data is not stored in etcd.

In practice a Kubernetes cluster is usually made up of multiple Master Nodes, each of them runs its master processes where, of course, the API Server process is load balanced and the etcd store forms a distributed storage across all of the Master Nodes.

The Master Nodes are more important but they generally have less load of work and so they need fewer resources like CPU, RAM and Storage.

5.1.3 YAML Configuration Files

YAML Configuration Files[7] are the main tool to create and configure components in a Kubernetes cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:
  labels:
spec:
  replicas:
```

[7] https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

```
selector:
template:
```

Listing 5.1: Kubernetes YAML Configuration File generic strcture

The code in listing 5.1 roughly shows the general structure of YAML configuration files used for the deployment of Kubernetes components.

The first two lines of the file are used to declare what you want to create:

- `apiVersion`: it has a different value for each type of component there is a different one;
- `kind`: this is used to specify the kind of component you will be creating, such as Service, Deployment and so on...

Every configuration file in Kubernetes has 3 parts:

- **metadata**: the metadata of the component you are creating, for example, the name of it
- **specification**: it is where every kind of configuration you want to apply for that component resides. The attribute of this part will be specific to the kind of component you are creating.
- **status**: this part is gonna be automatically generated and added by Kubernetes. Kubernetes will always compare what is the actual state and what is the desired state of that component and if they do not match, Kubernetes knows there is something to be fixed there and it will try to do so. This is the basis of the self-healing feature Kubernetes offers. The status part will be continuously updated by Kubernetes. The information about the status is taken from etcd by Kubernetes.

Usually, it is a good practice to store these configuration files with your application code.

In the template part of the specification section of a Deployment YAML configuration file, there are again metadata and specification parts, We have a configuration file inside of a configuration file and the reason for this is that this second configuration applies to a Pod. We will have the blueprint of a Pod and so here we will specify the containers it has, the images of them, which port they should open/expose.

Now, to connect the components between them, we will need to use label and selector fields. The metadata section contains the labels and the specification one contains the selectors. In the metadata, you give a component a key-value pair and it can be any key-value pair you can think of. This will just be a label stuck to that component. We give pods the same key-value pair label and through the selector, we tell them to match all the labels of that key-value pair so that we create that connection we want between the deployment and the pods for the first to know which Pods belong to it

A Deployment also has its label and this label is gonna be used by the Service selector when we want to attach a service to the Deployment or better its pods.

In the Service YAML configuration file, or better in its specification section we also have to configure its ports. The Service will have a port where the service itself is gonna be accessible at but it will also need to know to which Pod it should forward the incoming requests and also to which port that Pod is listening. This will be the target Port and it should match the containerPort of the Pod.

Finally, the replicas field in the specification section of a YAML configuration file is used to express how many replicas of the Pod implementing the component we want Kubernetes to create.

5.1.4 Exteneding Kubernetes Using Custom Operators

Operators are mainly used for Stateful Applications. Stateful applications when deployed on Kubernetes need more hand-holding throughout their entire lifecycle. Each stateful application will have its different solution, there is no standard one you can adopt for everything. Typically that is why this kind of application requires manual intervention, no matter if they are deployed on a Kubernetes cluster or a standard server.

Manual work in Kuberntes[70] kinda goes against its main concept and features offered, so stateful applications are usually hosted outside of the Kubernetes cluster.

There are, however, some stateful applications you need in Kubernetes and this is why an alternative exists to manage stateful applications in Kubernetes and this alternative is Kubernetes Operators[8]. This component replaces a human operator with a Software Operator.

All the manual tasks that a dev-ops team would do to operate a stateful application is now gonna be packed into a program having the knowledge and intelligence about how to deploy that specific application, how to create a cluster of multiple replicas of that application, how to recover when one replica fails and so on.

These tasks are all automated and therefore reusable. This means that the more complex and the more environments you deploy your application in and apply Operator to, the more benefits you will obtain from it.

An Operator, at its core, has the same control loop that Kubernetes has. It watches for changes in the application state, so if its configuration changed, it will apply the new one, if a replica died it will create a new one if the application image version got updated it will restart the pods with the new version. We can think of an operator as a Custom Control Loop in Kubernetes.

It uses a CRD (Custom Resource Definition), basically a custom Kubernetes component that allows extending the standard Kubernetes API, by letting you create your own custom Kubernetes component on top of the ones offered by default.

An Operator takes the basic Kubernetes resources and its controller concept as a foundation to build upon and on top of that, it includes the domain or application-specific knowledge to automate the entire lifecycle of the application it operates.

Operators are created for each application by those experts in the business logic of installing running and updating that specific application. Different frameworks are leveraging the Operator SDK that allows us to create our own Custom Operators.

It is important to note that, Kubernetes commands sent to its API server are not guaranteed to always succeed. Kubernetes events, on the other hand, are facts and things that have happened and so they cannot semantically be rejected by the receiver meaning yes you can just ignore them but they have happened still and this fact cannot be rejected.

Usually, in Kubernetes a Controller is observing events regarding 1 root object, then if it has to interact with other objects it can. As a result, we can say that, to realize a Kubernetes Operator, two different things need to be specified:

- A Custom Resource Definition (CRD)[9]. This element is needed to specify the Kubernetes components in the cluster whose lifecycle the Operator is in charge of managing.
- A Custom Controller. It is needed to specify the actions we want the Operator to perform in response to the different events regarding the lifecycle of the components it manages.

Depending on the framework we chose, the implementation of these two elements composing our Custom Operator may vary.

[8] https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[9] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

5.2 RabbitMQ

RabbitMQ[71] is an open-source message-broker written in Erlang, that implements the Advanced Message Queuing Protocol (AMQP).

Messaging is a way to manage the communication between two entities that aims to achieve both the decoupling of the two and improve their scalability. AMQP standardizes messaging by defining Producers, Message Broker and Consumers.

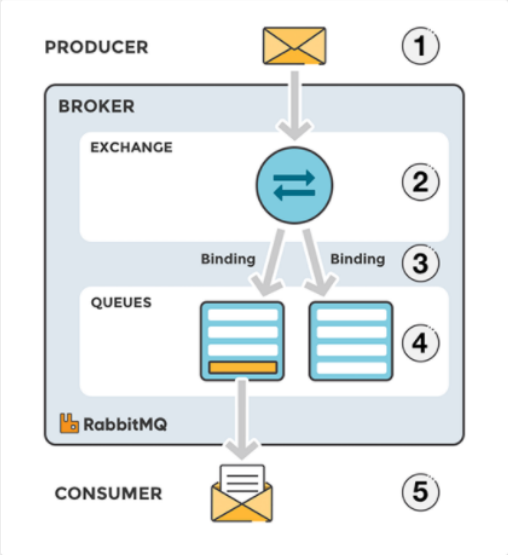


Figure 5.2: AMQP Architecture source: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html

As we can see in figure 5.2, we have a Producer, a Consumer and a queue that lies in between. The producer is in charge of sending messages and to do so, it never uses a queue directly but it leverages an Exchange that will be in charge of forwarding the message to the appropriate queue. The consumer picks up a message from its associated queue and consumes it. An Exchange proves to be particularly useful when we have an application with many different queues since it allows to send a message to selected ones. To connect to a queue, an exchange uses a binding and a binding key. In particular, when sending a message, a producer needs to specify a routing key and the exchange receiving the message will compare it to the binding keys of its connected queues to perform proper routing.

Depending on the characteristics of the comparison it performs, we can identify five different types of exchanges:

- fanout exchange;
- direct exchange;
- topic exchange;
- header exchange;
- default (nameless) exchange.

The *fanout* exchange completely ignores the routing key of the message and sends it to all of the queues it knows about. The *direct* exchange sends messages to queues where the routing key is the same as the binding key. The *topic* exchange allows partial matching of keys and this is achieved thanks to regular expressions and it is often used to implement different publish/subscribe pattern

variations. The *header* exchange uses the message header instead of the routing key. This is done to implement message delivery based on multiple attributes that are more easily expressed as headers of the message than its routing key. The *default* exchange compares the routing key of the message to the name of the queue instead of its binding key. It is pre-declared by the broker and every queue that is created is automatically bound to it with a routing key that matches the queue name.

An exchange can be durable or transient. In particular, the former survives to broker restarts whereas the latter doesn't and it will need to be redeclared every time.

Queues are used to store the messages that are consumed by applications. Some of the most important properties of these entities are:

- the name;
- the durability: this represents whether a queue will survive to a restart of the Broker or not;
- the exclusiveness: this means whether a queue can be used by only one connection or not;
- the auto-delete property: this is used to point out whether a queue is deleted when it has no more consumers or not.

Before a queue can be used it has to be declared, causing it to be created if it doesn't exist already. Besides, to receive messages from an exchange, a queue needs to be bound to it. Bindings are rules that exchanges use to perform proper routing of messages. The Broker keeps track of all of the queues connected to a specific exchange thanks to binding keys. In particular, if we wanted to draw an analogy:

- a queue is like your destination in New York City;
- an exchange is like JFK airport;
- bindings are routes from JFK to your destination. There can be zero or many ways to reach it.

It is important to note that the use of both exchanges and queues, allows the AMQP protocol[10] to introduce a layer of indirection that enables the possibility to model particularly complex routing scenarios and eliminate a certain amount of redundant work application developers may need to perform.

As far as the messages are concerned, their structure can be divided into two parts: a set of attributes and the actual payload. Attributes are set at message publication and some of them are so general that they are defined directly by the AMQP Protocol. Some examples are:

- the content type;
- the content encoding;
- the routing key;
- the message publishing timestamp.

On the other hand, the so-called *headers* represent optional attributes that are similar to X-Headers in HTTP.

Moreover, the data a message carries represents its payload and the AMQP protocol treats it as an opaque byte array. This means the Broker will route messages without performing any

[10] https://www.rabbitmq.com/tutorials/amqp-concepts.html

operations on the data they transport. As a result, it is possible for messages to only contain attributes.

If for any reason, a message cannot be routed to a queue it is either dropped or returned to the publisher, depending on the values of its attributes.

One more feature the AMQP protocol presents is message acknowledgements. This aspect of the protocol was specifically introduced to deal with the unreliability of networks and the possible failure of applications. In particular, upon the reception of each message, the consumer needs to notify the Broker about it. As a result, when acknowledgements are used, the Broker will only remove a message from a queue after it receives a notification for it. Having this mechanism built into the protocol helps developers to build more robust software. The AMQP protocol provides two different acknowledgement modes:

- explicit acknowledgement;
- automatic acknowledgement.
- after broker sends a message to an application;
- after the application sends back an acknowledgement.

The *explicit acknowledgement* mode allows users to choose when to send acknowledgements of messages. On the other hand, in the *automatic acknowledgement* mode, a message is considered to be successfully delivered immediately after it is sent.

AMQP[11] is considered a programmable protocol because it allows applications to directly define the AMQP entities and routing schemes they need. Therefore, the protocol directly provides all of the operations needed to declare queues and exchanges, define bindings between them, subscribe to queues and so on. It is possible to have more than one consumer per queue or to register an exclusive consumer.

To interact with the Broker, applications need a connection to it. AMQP connections are typically long-lived and are built on top of TCP connections to perform reliable delivery. They use authentication and can be protected using TLS and when an application no longer needs to interact with the broker, it should gracefully close its AMQP connection instead of the corresponding TCP one.

Besides, some applications may need to instantiate multiple connections to the broker. However, having many TCP connections open at the same time heavily affects system resources. As a result, channels have been provided to allow the multiplexing of AMQP connections Every protocol operation performed by a client happens on a channel. In particular, the communication happening on a specific channel is unique therefore, both the broker and clients will use a channel ID to keep track of the different channels. Finally, it is important to note that a channel only exists within the context of the connection it multiplexes so, whenever the latter is closed, the same happens for all of the channels on it.

[11] https://www.rabbitmq.com/tutorials/amqp-concepts.html

Chapter 6

QKD Simulator New Design

6.1 Overview

Despite its advantages, the QSS QKD Simulator also has some drawbacks, such as:

- allowing to simulate QKD only between the same two endpoints;
- being rather complex to use;
- being difficult to be deployed to distributed infrastructures;
- having limited scalability, especially because REST APIs were used to manage internal communications of the simulator.

As a result, a complete redesign of the simulator was needed. In particular, the main goal at the basis of the new version of the QSS QKD Simulator was the simplification of its structure while improving its performance and extending its functionalities. The main features of the new version are the following:

- being easy to deploy to distributed infrastructures thanks to the technologies used for its development;
- allowing the user to be able to model an entire network topology inside the simulator;
- allowing the user to start and evaluate the performance of different point-to-point QKD simulations between distinct nodes of the modelled topology through the appropriate communication channels.
- having increased scalability and decoupling between modules thanks to the use of messaging instead of REST APIs to implement internal communications.

In the following sections of this chapter, we will go over the architecture and design of the new version of the QKD simulator.

In particular, we will describe the different Modules composing the simulator, the way they can interact with each other and, as far as the Simulator Manager module is concerned, with any external entity willing to use the simulator too. The different interfaces exposed by the Simulator Manager will be analysed. Furthermore, the data model characterizing the messages the different modules of the simulator exchange between them during its operation will be illustrated and explained.

Finally, the main different workflows characterizing the simulator will be illustrated and appropriately described in-depth to have a good understanding of the way things work, the different operations it offers and how these operations are carried out. It is important to note that all

of these analyses will be done without making any reference to the specific technologies used to implement the simulator.

The description of such technologies will be the main subject for a whole different chapter and the notions that will be presented here will be needed for its complete and thorough comprehension.

6.2 Modules

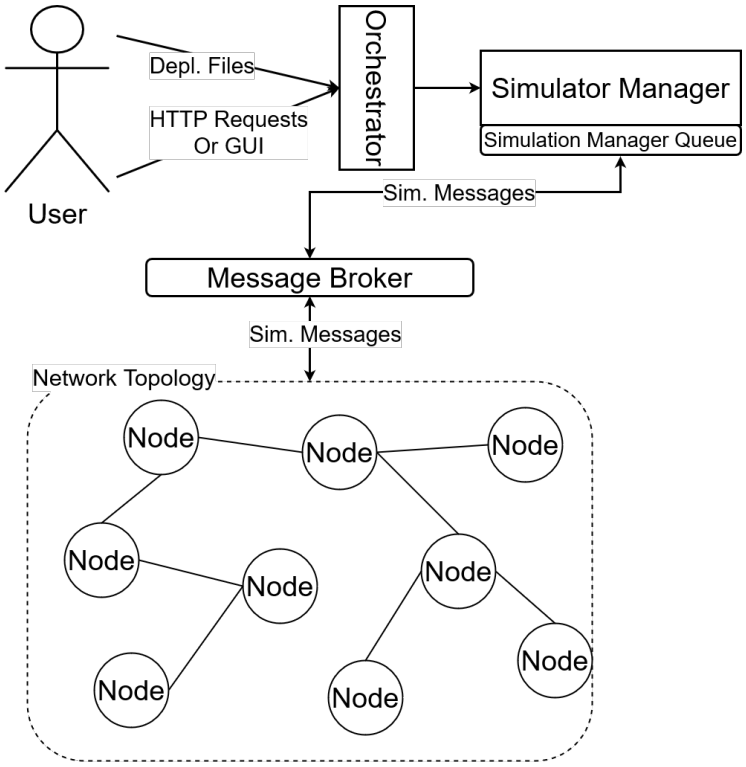


Figure 6.1: QKD Simulator Architecture

As we can see, figure 6.1 shows the architecture of the new version of the simulator. It is important to note that, we simply refer to all of the different messages that are gonna be exchanged between the modules of the simulator through the use of the Message Broker module as "Simulation Messages" because those messages represent the internal communication happening in the simulator during its operations, among which, of course, the QKD simulation is the most complex and important one. The features of those messages will be better explained later in section 6.3 of this chapter.

Besides, the *Orchestrator* component isn't exactly part of the modules composing the new version of the QKD Simulator, but it represents the actor that will be in charge of managing the deployment and lifecycle of the different modules of the simulator, as a consequence of the received "Deployment Files" to describe what parts the user wants to deploy and how he wants to configure them.

On the other hand, two of the main modules of the new QKD Simulator are:

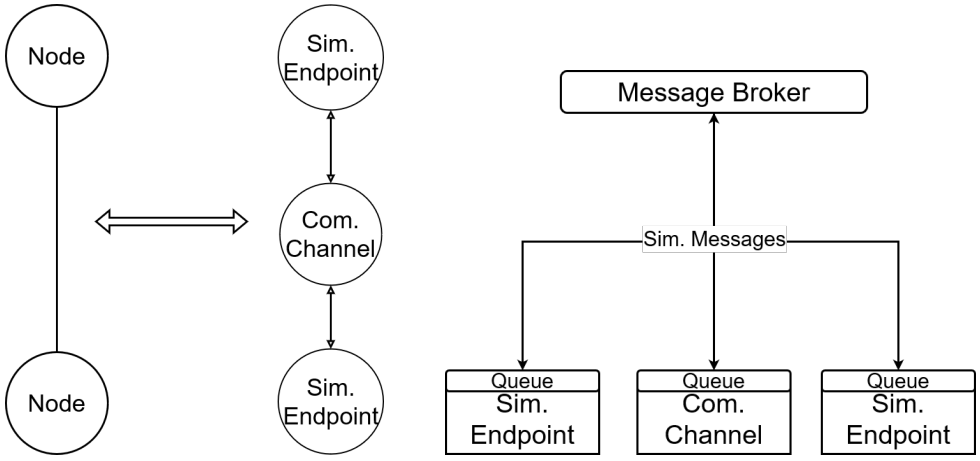
- *Simulator Manager*: this is a module designed to allow the user to interact with the simulator. It allows different operations to be performed:
  - describing the network topology we want the simulator to model:
  - distributing the various network nodes the keys needed for classical communication authentication;

– performing a point-to-point QKD simulation between two selected network nodes through the appropriate link between them and given different configuration parameters for the simulation such as the type of protocol to simulate and many others.

The manager was designed to provide a user-friendly interface to the simulator environment. As we can see, the manager isn't a module that will actively participate during the execution of a single point-to-point QKD simulation, but its presence is key to make sure such simulation has been properly configured as well as the Simulator in general. Furthermore, given the architecture, the manager is also the only module allowing the Simulator to interact and be reachable from any external entities wanting to use it;

- *Message Broker*: It is the module in charge of handling all the internal communications happening in the simulator. A user will operate using the simulator, just by interacting with the Simulation Manager itself. From that moment on, all of the remaining communications that need to happen for the requested operation to be accomplished will be handled and will pass through the Message Broker module.

Now, we can illustrate the way the modelling of a generic network topology is handled by the simulator.



(a) Point-to-point Link Design (b) Point-to-point Link Message Flow

Figure 6.2: Network Topology Design

As we can see, figure 6.2a shows the way a generic point-to-point link of the network topology is implemented in the simulator's architecture.

In particular, there are two different modules used to implement respectively, a single node of the topology and a single link between two nodes. We decided to realize a generic link of the topology as a whole module of its own to have more flexibility in terms of modelling the different events that can happen to messages of communication while travelling on a network link. This design choice easily allows us to implement different kinds of attacks the simulated QKD protocols can be subject to and to introduce various types of noise modelling too.

Figure 6.2b illustrates, on the other hand, the way we chose to model the point-to-point communication between two linked nodes of the network topology modelled. In particular, the communication is realized through the utilization of messages, queues and the Message Broker module. Each message will be delivered to the message broker that will be in charge of putting it on the proper queue for the specific module associated with it to receive and consume it. In our design, each module of the Simulator, except for the Message Broker, will have its associated queue to listen on to receive messages.

Therefore, to recap, the last two modules of the new version of the QKD Simulator, that are used to allow the modelling of an entire network topology, are:

- *Simulation Endpoint*. It represents a specific node of the network topology given the simulator. It acts as both the source or the destination of a given point-to-point QKD simulation depending on the configured parameters. The steps it performs depend on the protocol chosen for the simulation and on the role (source/destination) it has to perform during it and will be one of the objects of interest in the next chapter;
- *Communication Channel*. It represents the communication channel between two specific nodes of the network topology given the simulator. It acts as both a classical and a quantum channel between the nodes forwarding both classical and quantum messages between them during a simulation. It is assumed to be under the full control of an eavesdropper(Eve). The channel has been thought of as a separate block to achieve the following goals:
  - properly modelling the links between the nodes of a given network topology;
  - allowing different kinds of attacks to be performed;
  - introducing distinct kinds of noise modelling.

6.3 Data Model

At the end of the previous section, we briefly explained the main features of a Message Broker. Now that we know what it is, we can explain more in detail the general structure of all the messages the Broker handles inside the Simulator.

Message Structure	
Kind	**Description**
Type	The type of the message
Source	The name of the entity that sent the message
Destination	The name of the entity the message is destined to
Metadata	Contains the configuration parameters of a Simulation
Body	What the source wants sent to the receiver

Table 6.1: General structure of simulator's messages

As we can see, table 6.1 shows the variety of fields a general message handled by the Broker is divided into; starting from the top:

1. the **Type**: it is a field the use of which is allowing, whichever the Simulator module receiving it, to know what kind of message it has to do with to properly handle it and perform the appropriate operations;
2. the **Source**: the role of this field is to uniquely identify the node of the topology modelled by the Simulator, the specific message originated from;
3. the **Destination**: this field, instead uniquely identifies the node of the network topology modelled by the Simulator, the current message is meant for;
4. the **Metadata**: this field contains all of the different configuration parameters for the specific QKD simulation the specific message regards. Some of the parameters contained in such a field could be:
   - key length;
   - protocol;
   - chunk size;
   - authentication method for classical communication part of the simulation.

5. the **Body**: it represents the last field of the message and we can consider it as a generic "Blob" that will contain the appropriate message payload according to the message type that is needed to carry on with the simulation.

As we now know, a Message Broker makes use of queues to deploy messages to their appropriate receivers. Depending on the technology used to implement the Broker and what someone is trying to achieve when using this particular communication method, different choices can be made as far as queues are concerned.

We know Queues allow the delivery of messages to their rightful receivers, but they can vary in number, in role and they can be configured in different ways through the usage of proper configuration parameters. In our case, as figure 6.1 shows, each different module of our simulator that will actively participate in a QKD simulation has its queue associated with it to be able to properly exchange messages. We could have decided to configure and organize Queues in our Simulator in many different ways to the one we just explained, but the main reasons of our choice are mainly related to the specific technology we chose for the Broker's implementation and, therefore, we will better analyse such issue in the following chapter regarding the actual implementation choices behind the Simulator.

6.4 Interfaces

As we saw in the previous section 6.2, all of the interactions with the QKD Simulator happen through the Simulator Manager module.

Therefore, this component will be the only one exposing interfaces to allow communication to the simulator.

In particular our Simulator Manager module will expose two different interfaces:

- the Web Interface;
- the Manager Interface.

In the following subsections, we will describe more in-depth both of these interfaces and their functionalities.

6.4.1 Web Interface

Web interface exposes different REST APIs accessible through HTTP requests by any external agent (user or module) wanting to use the simulator.

It allows performing all of the operations offered by the simulator, as long as the appropriate HTTP request is made.

The interface can be easily summarized by the following table A.1.

Web Interface		
Method Name	**URL**	**Access Method**
netTopology	http://{Simulator URL}/getTopology	POST
distributeKeys	http://{Simulator URL}/distributeKeys	POST
startKeyExchange	http://{Simulator URL}/startKeyExchange	POST
begin	http://{Simulator URL}/begin	POST
end	http://{Simulator URL}/end	POST

Table 6.2

`netTopology` is the method to be used to provide the Simulator Manager module with the description of the desired network topology. It expects to receive a JSON description of the topology to model inside of the request body.

If the received network description is properly formatted the method will then create a JSON description of a YAML file and it will return such a description inside the body of the response with status code 201.

The YAML file can be easily fed to the Orchestrator to deploy the desired topology in case it hadn't already been deployed. If, on the other hand, the provided request body was not properly formatted, a response with status code 400 and a body containing the description of the error will be returned.

`distributeKeys` method is used to give out to each node of the modelled network topology the appropriate parameters needed to be able to apply a specific authentication method during a QKD simulation, if requested by the user. The desired authentication method needs to be specified in the request body of the HTTP request. Since, at the moment, the only supported method is SPHINCS+ for digital signature, the `distributeKeys` method just sends a pair of SPHINCS+ keys (a private and public one) to use for the authentication of the classical communication requested by a specific simulated protocol. If the topology modelled was already provided to the Simulator Manager module through the `netTopology` method described above, a pair of keys (a private and a public one) will be generated for each node of the network topology and the distribution will happen in the following way:

- the private and public keys generated for a specific node will be sent to such node through the Message Broker;
- the public key generated for a specific node will be also sent to all the network topology nodes connected to it through the Message Broker.

If the distribution was successful, a response with status code 200 will be returned. On the other hand, if the model topology hasn't been notified to the Simulator Manager module yet or any kind of error occurred during the generation and distribution of the keys, a response with status code 400 and a body containing the description of the error will be returned.

`begin` method allows to set up the simulation parameters for two specific nodes of the network topology modelled by the simulator. It is important to note that this method won't start any point-to-point QKD simulation it just sets up the configuration parameters for the single or set of simulations that will then be performed on the given network nodes through the appropriate link between them in the specified direction (source to receiver). The proper request body containing the simulation parameters is expected to be received. If the operations performed by this method all had a successful outcome, a response with status code 201 and body containing the ID of the configured connection will be returned. If, on the other hand, any error occurred during the execution of the method, a response with status code 400 and a body containing the description of the error will be returned

`startKeyExchange` is the method to be used to start a point-to-point QKD simulation between two specific nodes of the modelled network topology and through the appropriate link connecting them. This method expects as parameters in the request body some additional configuration parameters for the simulation such as the length of the key to generate, the protocol to simulate like BB84 for instance and finally, the ID of the opened connection through the `begin` method described above and the number of point-to-point simulations to perform.

If everything went according to plan during the execution of this method and the parameters received were properly formatted, a response with status code 201 and body containing the ID of the queue of the Message Broker on which we will be able to retrieve the results of the requested simulations for the given connection. Otherwise, if any error occurred, a response with status code 400 and a body containing the description of the error will be returned.

`end` method is used to close a connection to the simulator opened through the use of the `begin` method described above. This method expects as a parameter in the request body the ID of the opened connection to be closed.

If the specified parameter corresponds to a connection still open, such connection will be closed, deleted and a response with status code 200 and an empty body will be returned. If, on the other hand, the specified parameter didn't correspond to any connection Id or such connection was already closed and deleted previously, a response with status code 400 and body containing the description of the error will be returned.

6.4.2 Manager Interface

As we saw at the beginning of section 6.4, this interface is also implemented and exposed by the Simulator Manager module.

The scope of the Manager Interface is to provide an easy to use GUI to access simulator functionalities. This interface was realized by exploiting Jupyter Notebook and it allows the user to create many different notebooks, each having its specific function, such as:

- Creating the YAML file describing the network topology we want the simulator to model and needed by it to model it.
- Properly distributing the keys to the different network nodes of the topology modelled by the simulator. Such keys will be used to authenticate classical communications happening during the simulation of a protocol.
- Handling the opening and closing of connections between specific nodes of the network topology modelled by the simulator and performing the desired point-to-point QKD simulations between them and through the appropriate link.

The Manager Interface essentially allows an external user to have access to all of the different functionalities offered by the QKD Simulator without the hustle of having to use HTTP requests and properly configure their bodies to perform operations and use the simulator.

6.5 Workflow

6.5.1 Simulator And Topology Deployment

To start using the new version of the QKD Simulator, the first thing to do consists of deploying its different modules. To achieve such a purpose the Orchestrator component of the architecture can be exploited.

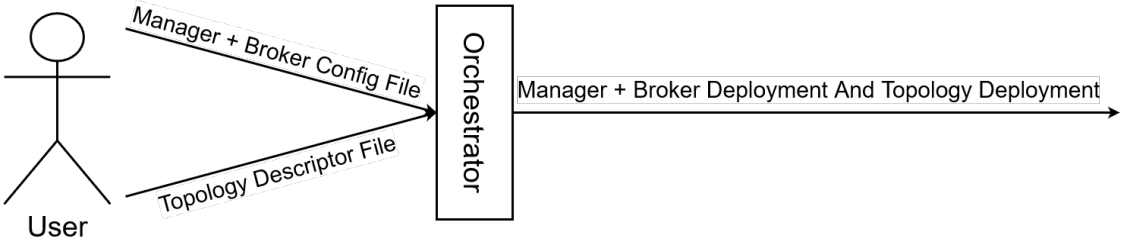


Figure 6.3: Simulato And Topology Deployment

As we can see from figure 6.3, to deploy the simulator and model a network topology to start QKD simulations on, the user will need to provide the Orchestrator component with two different custom files:

- one describing the configuration parameters to use during the deployment of the Simulator Manager and Message Broker modules;
- one illustrating the structure of the network topology the simulator will have to model.

As we said, these two files are custom, therefore their structure will be better analysed in the next chapter when discussing more in detail about the implementation of the new version of the QKD Simulator. However, it is important to note that the choice of dividing the deployment of the network topology to model from the deployment of the rest of the modules wasn't random.

In particular, the topology is assumed to be a resource a user will want to change frequently to perform QKD simulations on many different ones, therefore, its deployment has been split from the one of the other modules so that to swap the network topology modelled it won't be necessary to perform a redeployment of the whole QKD Simulator. The Simulator Manager and Message Broker modules can be left as they are while changing the modelled topology.

6.5.2 Init Phase

After having successfully deployed the Simulator Manager module, the Message Broker one and the network topology to model, the new version of the QKD Simulator will still need some operations to be carried out before the user will be able to launch one or more QKD simulations between two specific endpoints of the topology.

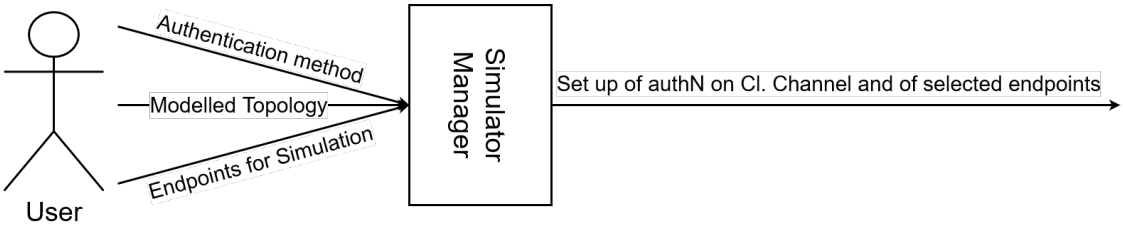


Figure 6.4: Simulator Init Phase

As we can see from figure 6.4, there are three different operations to be carried out:

1. notify the Simulator Manager module about the modelled network topology deployed;
2. notify the Simulator Manager module about any particular authentication method the user will want to use for the classical channel during QKD simulations;
3. choose the pair of directly linked endpoints of the topology to carry out QKD Simulations between.

As we now know, the network topology is considered to be a resource the user of the simulator will want to swap frequently to evaluate the performance of QKD protocols on different ones. Therefore, since in the new design of the QKD Simulator any interaction with it needs to pass through and is handled by the Simulator Manager, the first thing that needs to be done before being able to start any QKD simulations is to let that module know about the network topology that has been currently modelled and deployed.

Moreover, if the user will want to start any QKD simulation employing any specific kind of authentication method for the classical part of communication happening during the simulation, he will need to notify the Simulator Manager module about the specific authentication method desired so to provide the different endpoints of the modelled topology with the required parameters to apply that kind of authentication during the QKD simulation. This operation won't be required if the user will want to run just simulations without any authentication.

The last step needed before being able to start one or more QKD simulations is letting the Simulator Manager know about the two specific endpoints of the modelled topology the user will want to start simulations between.

6.5.3 Point-to-point QKD Simulation

Now that we have described all of the other operations needed for the new version of the QKD Simulator to be deployed and initialized correctly, we are finally ready to describe the most crucial and most complex one, the point-to-point QKD simulation of a protocol.

The scheme 6.5 will illustrate what is the simulation workflow of the BB84 protocol, which is at the moment the only one offered by the Simulator.

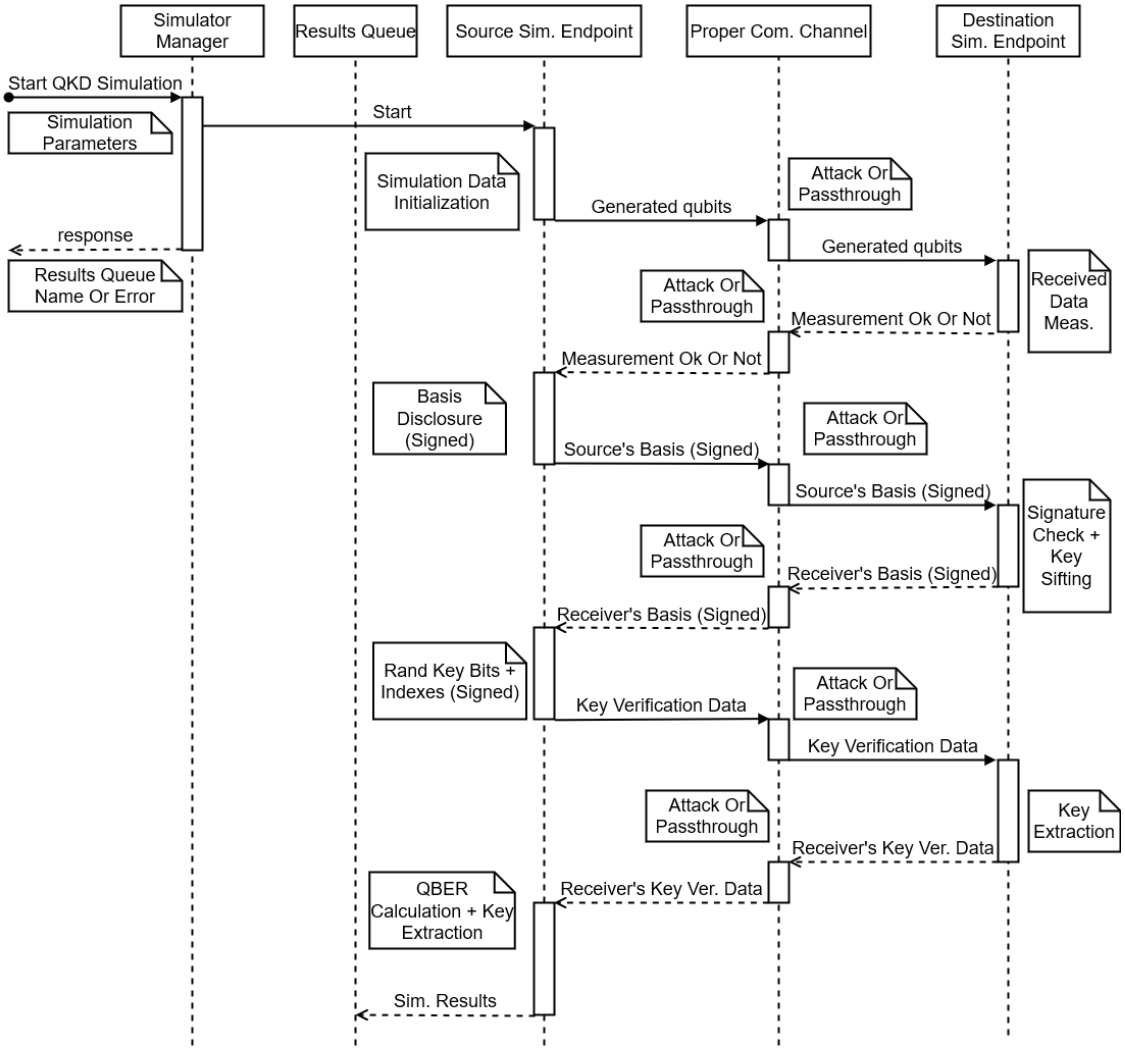


Figure 6.5: BB84 Point-to-Point Simulation Scheme

As figure 6.5 illustrates, when we want to start a point-to-point QKD simulation, we need to provide the simulator with the proper set of "Simulation parameters". These allow the Simulator to know which specific Simulation Endpoint Modules and Communication Channel module will be involved in the current simulation and what configuration parameters to use during it.

If the provided parameters are valid, the name of the message queue the Broker will put the results of the simulation or any error related to it on will be returned in the body of an HTTP response with status code 200.

On the other hand, if there is an issue with the provided parameters, a message describing the error will be returned in the body of an HTTP response with status code 400.

Assuming these initial steps are successful, we can now describe the workflow of a point-to-point simulation.

Figure 6.5 also illustrates all of the different steps of the BB84 protocol QKD simulation. The important thing to note is that the simulation core of the protocol hasn't changed when compared

to the previous version of the QKD Simulator. What differs is the way internal communication is handled, instead of using a TCP/IP network and rest API, the new version of the simulator leverages asynchronous messaging handled through the Message Broker module. This kind of communication allowed to improve the decoupling and scalability of the different modules of the simulator. Further details about the way the different BB84 simulation steps work will be found in the next chapter where the implementation of the new version of the QKD Simulator will be described.

6.6 Example of usage on a Distributed Infrastructure

In order to be used on a distributed infrastructure, the new version of the QKD Simulator can be used together with the Quantum Software Stack (QSS) already proposed by TORSEC research group of Politecnico di Torino[68]. Since the structure of the QSS isn't part of the focus of this work, we will simply describe the interaction of the new version of the simulator with it.

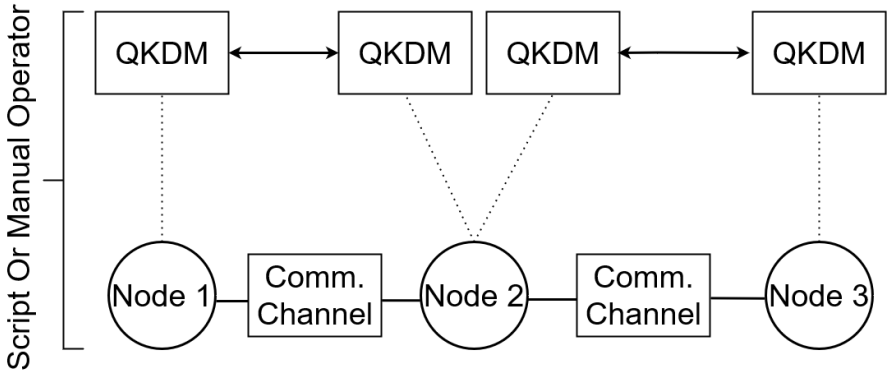


Figure 6.6: QSS and new QKD Simulator interaction

The interaction between the QSS and the new version of the QKD Simulator is shown in figure 6.6. In particular, a script or a manual operator will be in charge of deploying the simulator, the topology to model and give the different QKDSMs the needed configuration parameters to interact with the simulator. Each QKDM will need to know:

- the node it represents in the modelled topology;
- the destination QKDM it wants to communicate with;
- the node represented on the modelled topology by the destination QKDM
- the communication channel in the modelled topology that connects the nodes represented by itself and the destination QKDM

After that, the QKDM source of the communication will be able to directly use the REST APIs exposed by the Simulator Manager module of the simulator in order to start one or more point-to-point key exchanges. The QKDM source of the communication will also be in charge of notifying the destination QKDM about the name of the RabbitMQ from which it will be able to consume its copy of the exchanged key.

Chapter 7

QKD Simulator New Implementation

7.1 Overview

In the previous chapter, we went over the main design characteristics of the new version of the QKD Simulator.

All of the different technologies we chose to use for its implementation will be presented here and it will also be discussed more in detail the actual implementation of all of the different modules and other components needed for the Simulator to be easily up and running.

In particular, the technologies used for the implementation of the new version of the simulator were:

- Kubernetes;
- Qiskit Framework;
- RabbitMQ;
- Docker.

Given the main goals of the new version of the Simulator explained in section 6.1, Kubernetes and Docker were the technology used to implement the Simulator to easily allow its insertion inside the architecture of already deployed and distributed applications. In particular, Kubernetes is the orchestration technology that allows, as is, without any efforts, to easily deploy the new version of the QKD Simulator on multiple nodes and to scale the simulation ad libitum.

The Qiskit framework remains the basis of all of the QKD simulation related operations. Finally, RabbitMQ is the technology chosen for the implementation of the Message Broker module to handle the SImulator's internal communications.

7.2 Modules And Data Model Implementation

In section 6.2 we described the various modules the QKD Simulator is made of. Here, we will describe more in-depth their actual implementation and the technologies we chose for their realization.

In particular, as we said at the beginning of the chapter in section 7.1, we opted to rethink the whole Simulator architecture as part of a Kubernetes cluster. This choice was made to allow us to achieve the goals we had in mind when we decided to create a whole new version of the QKD Simulator.

The following figure 7.1 describes all of the various Kubernetes components characterizing the simulator inside of the cluster:

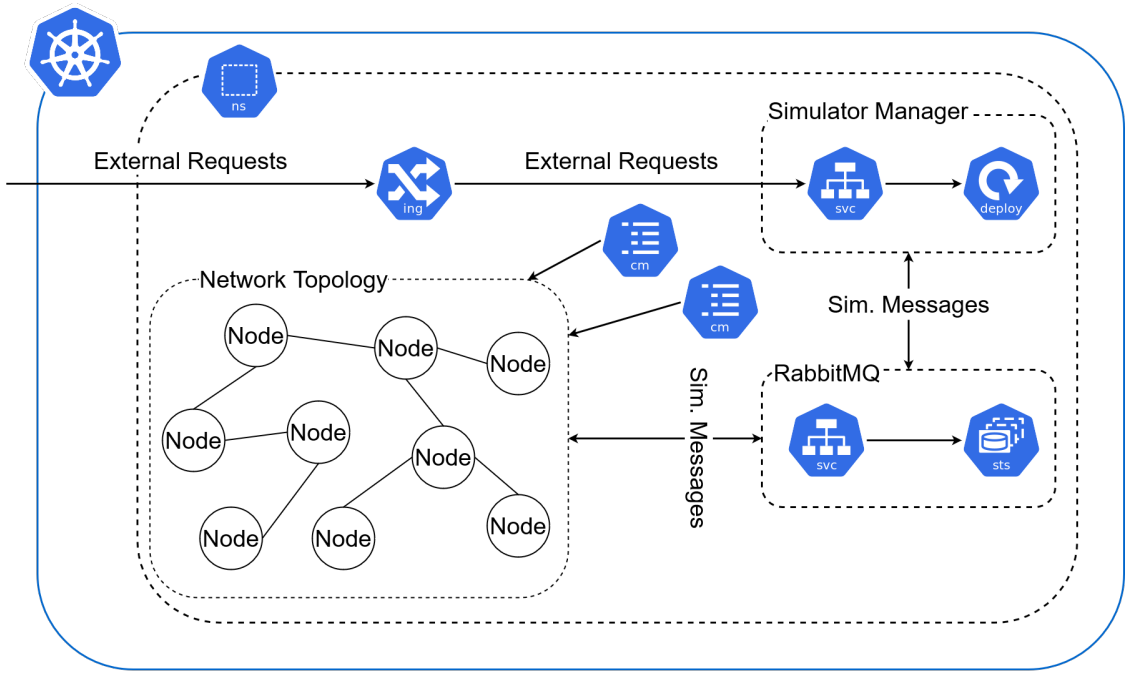


Figure 7.1: QKD Simulator Kubernetes Architecture

As we can see, figure 7.1 shows most of the Kubernetes components used to implement the new architecture of the Simulator.

In particular we can see what Kubernetes components were use dot implement two of its main modules:

- the *Simulator Manager*;
- the *Message Broker*.

The *Simulator Manager* was implemented using two different Kubernetes components:

- *a Deployment* to handle the creation of the Pod running a customer Docker Image implementing the module;
- *a Service* to allow the Pod to be able to communicate both with other Pods and the external world too.

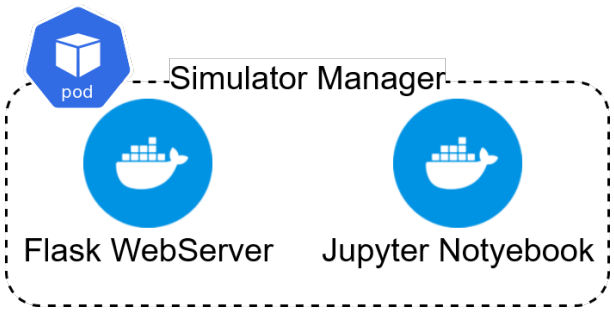


Figure 7.2: Simulator Manager Pod Internal Structure

As we saw in section 5.1.1 a Pod usually runs a single container inside of itself, but in the Simulator Manager case, as shown in figure 7.2, two different containers are running inside the Pod created thanks to the Deployment component.

From section 6.4 we know the SImulator Manager exposes two different interfaces:

- a Web Interface;
- a GUI created through Jupyter Notebook.

To implement those two different interfaces, we created two custom Docker Images: a Flask WebServer and a Jupyter Notebook one.

Each container will be listening on and exposing different ports to allow the Service Kubernetes component to properly identify and route the various requests it receives so they can be correctly dealt with. The Jupyter Notebook Docker Image corresponds to a full standard Image to deploy a Jupyter Notebook server but its configuration has undergone some adjustments.

In particular, authentication is performed through a password request instead of a token and the main directory for the notebooks has been set to a custom one that will contain different Python scripts allowing a User to properly interact with the simulator by just changing the values of some parameters. On the other hand, the Flask WebServer Docker Image is a fully custom one where we implemented the Web Interface described in section 6.4.1 through the use of REST APIs and Python Flask. Moreover, we have a different process listening on the RabbitMQ queue associated with the manager to properly log all of the operations, events and errors happening inside of the Simulator. Process management has been implemented through the use of the `multiprocessing` Python package.

The *Message Broker* module was implemented using different Kubernetes components compared to the Simulator Manager:

- *a StatefulSet* to handle the creation of the Pod running a customer Docker Image implementing the module;
- *a Service* to allow external access to the RabbitMQ Management Interface and allow connections from other modules of the simulator.

A StatefulSet component is required to implement and deploy RabbitMQ in Kubernetes. This component ensures that the RabbitMQ nodes are deployed in order, one at a time. This avoids running into a potential peer discovery race condition when deploying a multi-node RabbitMQ cluster. There are other, equally important reasons for using a Stateful Set instead of a Deployment such as:

- sticky identity;
- simple network identifiers;
- stable persistent storage;
- the ability to perform ordered rolling upgrades.

The Docker Image running inside of the Pod created by the StatefulSet component deployment is a standard RabbitMQ image with the management interface enabled and with the proper health checks have been set up to make sure RabbitMQ is always up and running before doing any operation on the Simulator.

As far as the implementation of the communication through RabbitMQ is implemented, there is a single queue per instance of Simulator's module. Each module instance will have a single connection to RabbitMQ associated with it and from this connection, it will leverage two different channels to implement the communication:

- a Sending Channel to send messages;
- a Receiving Channel to fetch messages from its queue.

The Exchange type used is the Direct type given the queue distribution we just went over. Moreover, it is important to note that any Python interaction with RabbitMQ has been implemented leveraging the Pika Python Library.

There are still a couple of Kubernetes components shown in figure 7.1 we haven't gone over yet. These components correspond to two different ConfigMaps:

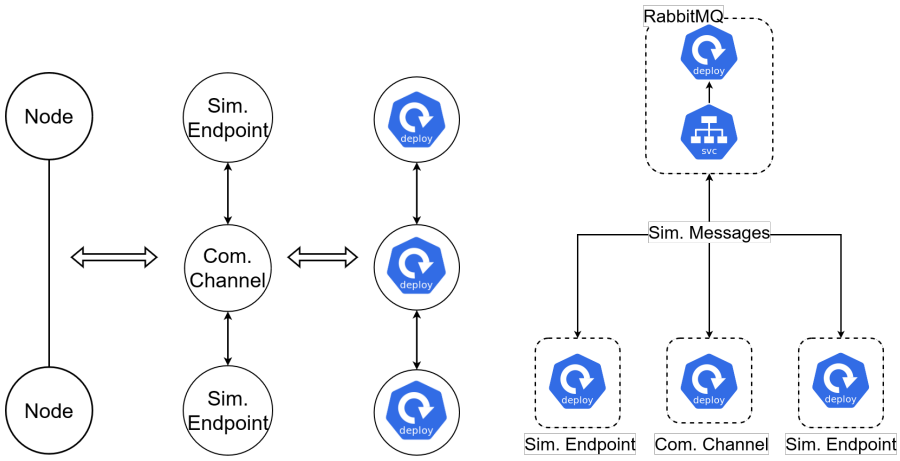
- *Simulation Endpoints ConfigMap*;
- *Communication Channels ConfigMap*.

As we know from section 5.1.1, ConfigMap components are used to provide configuration parameters to your application running on the cluster.

We use the two ConfigMaps to provide parameters like the RabbitMQ Service name and port to the two types of modules used to implement the modelling of a network topology. Other parameters provided through the ConfigMaps are whether to use a true quantum random number generator during QKD simulations and the amount of parallelism to employ for each module representing the modelled topology when consuming messages. Each of the modules will retrieve as environment variables the values of the different parameters inserted in the ConfigMaps. This operation is carried out in the *main* function. The two type of modules at issue are:

- *Simulation Endpoint*;
- *Communication Channel*.

Now, a more in depth description of their implementation will be given:



(a) Point-to-point Link Implementation

(b) Simulator Internal Communications Implementation

Figure 7.3: Network Topology Implementation

As we can see in figure 7.3a, both the above-mentioned modules have been deployed in Kubernetes thanks to the use of just a single component: a *Deployment*. This choice is related to the fact that, as we can see in figure 7.3b, all of the communications regarding the modules implementing the model of the network topology happen through the Message Broker, i.e. RabbitMQ. Therefore, none of these modules will need a Service component associated with them since each of them will always rely on the Message Broker to communicate. In conclusion, to send and receive messages, any of these modules will simply need to establish a connection to RabbitMQ and to do so they can leverage the Sevice component associated with it.

For both these modules, the Deployment will spawn a Pod in the Kubernetes cluster that will have just a single container running on the inside. In particular, the container will be running a Docker Image of a custom implementation of a Python RabbitMQ Consumer.

The implementation of the image differs according to the module being either a Simulation Endpoint or a Communication Channel, but its general structure remains the same:

```
class Module_Name():

        def type_switcher(self,argument):...

        def startListening(self):...

def RMQ_Consumer(rhost,rport,me,a_con,authN_data,l_acon,l_authN):

        #Set Up
        module_name = Module_Name()
        module_name.startListening()

        return

def main():
        logging.basicConfig(level=logging.INFO, format='%(asctime)s -
            %(levelname)s - %(message)s', handlers=[
        logging.FileHandler('module_name.log', mode="w"),
        logging.StreamHandler()
        ])

        #Fetch Environment variables
        p_num = int(os.environ.get("PNUM",3))
        ...
        rabbit_host = os.environ.get("RABBIT_HOST","rabbitmq")
        rabbit_port = os.environ.get("RABBIT_PORT",5672)

        logging.info("Starting the listening...")

        #Pool set up
        pool = multiprocessing.Pool(p_num)

        results = []

        #Start the consumers
        for i in range(p_num):
           r =
               pool.apply_async(RMQ_Consumer,args=(rabbit_host,rabbit_port,...))
           results.append(r)

        #Wait for processes to finish
        for r in results:
           r.wait()

        pool.close()
        pool.join()

        logging.info("Correctly quit the application")

if __name__ == "__main__":
        main()
```

Listing 7.1: General implementation structure of a topology component

As we can see from the code in listing 7.1, parallelism through the usage of a `multiprocessing.Pool` was implemented for each simulator module needed to model the network topology. In particular, depending on the amount of parallelism configured in the appropriate ConfigMap component and set as environment variable of the module during the deployment of the simulator to the Kubernetes cluster, a `multiprocessing.Pool` composed exactly of that number of processes will be created. Each process of the pool will run an instance of a RabbitMQ consumer running a custom logic to handle the particular messages of the simulator. As a result, the introduction of parallelism will allow having for each endpoint and communication channel of the topology to have multiple consumers ready to fetch and handle messages from the specific queue of the module they belong to. This allows to significantly improve the performance of the QKD Simulator both in terms of time and throughput when multiple QKD simulations involving a specific module of the topology are ongoing at the same time.

The overall structure of the RabbitMQ consumer run by each process is the same for both Simulation Endpoint and Communication Channel type modules. In particular, as illustrated in listing 7.1, in both cases, the *startListening* function will be called whenever the module receives a message from RabbitMQ and it will run *type_switcher* function on the Type field of the message.

What changes between the implementations of the Simulation Endpoint module and the Communication Channel one is the logic used to consume and manage each type of message of the QKD Simulation. Besides, it is worth noting that the implementation of the Simulation Endpoint kind of modules will also include the definition of a pair of data structures shared between the processes of the pool as well as their respective locks to manage concurrent access to them.

```
manager = multiprocessing.Manager()
active_connections = manager.dict()
l_actcon = manager.Lock()
authN_data = manager.dict()
l_authN = manager.Lock()
```

Listing 7.2: Shared data structures and locks for pool processes of a Sim. Endpoint

Listing 7.2 shows that the two shared data structures between the pool processes of a Simulation Endpoint module are two dictionaries used to respectively keep track of the different simulations the specific endpoint is currently involved in and save the parameters needed to apply a specific authentication method (SPHINCS+ is currently the only one supported) during a simulation. Each of these data structures has its associated lock, also shared between the different processes of the pool. The implementation of these variables was done through the use of a `multiprocessing.Manager`.

In section 6.3 we saw that the *Type* field of the message expresses the actual kind of the message and based on its value, the *type_switcher* function will delegate the handling of the message to the proper function. In the *main* function we can see how proper logging to a log file is set up before instantiating the *Module_Name* class and starting to listen for RabbitMQ messages on the proper queue.

The differences between the implementation of the two modules will be:

- the functions used to handle messages based on their Type field. Thes function will be the ones implementing the different Qiskit operations needed to simulate a specific QKD protocol;
- the values of the Type field of messages the module can handle and expects to receive.

In figure 7.1 there is still one Kubernetes component that hasn't been mentioned yet: **Ingress**. As we saw in section5.1.1, the Ingress component allows to open communication from external sources to our application and it makes the URL that will be used to access it more user friendly with a proper domain name. In particular, this component has been configured to route external requests to the proper interface exposed by the Simulator Manager module according to the provided endpoint and the domain name we chose to use is *qkdsim.com*. Moreover, it is important to note that the Ingress component is made out of:

- Ingress Controller;
- Ingress Rules.

The Ingress Controller is the first component that needs to be installed on the cluster for the rules specified in the Ingress Rule one to be respected. The combination of these two parts becomes the new entry point to all of the outside requests towards the specific domain or subdomain specified in the rules and, according to them, will route the requests to the specified internal Services of the appropriate components. This allows to keep all of the Services associated with the simulator's Kubernetes components internal ones, that is to say, Cluster-IP Services not reachable from the outside and to use a proper host domain instead of an IP address to reach the Simulator inside the cluster.

It's important to note that Kubernetes does offer a default Ingress Controller option that is installable on the cluster and this option is called Kubernetes Nginx Ingress Controller. There are also other options available, the only thing to keep in mind is that independently of the Ingress Controller implementation installed, it needs to be properly specified to the Ingress Rule component. Finally, the following table 7.1 illustrates how each field of a generic message handled by the Message Broker has been implemented:

Message Structure Implementation	
Kind	**Implementation**
Type	String
Source	String
Destination	String
Metadata	Dictionary
Body	Dictionary

Table 7.1

Table 7.1 illustrates the implementation of the data model used for the internal communications of the simulator and introduced in section 6.3. All of the different fields are grouped inside a Python List and serialized by using the Pickle Library before being sent.

7.3 Custom Operator Implementation

In the previous section, we described all of the different Kubernetes components of the QKD Simulator and we described their implementation more in detail. Now, we will analyze how a Kubernetes Custom Operator will allow to spawn and properly set up all of these components inside the cluster just thanks to the usage of two YAML Configuration files.

At this point, as we know from section 5.1.3, YAML files are the main tool to deploy and configure resources in a Kubernetes cluster. Moreover, in insection 5.1.4, a custom Kubernetes Operator is composed of a CRD (Custom Resource Definition) and a custom Kubernetes Controller associated with it. The custom Controller will intake the different parameters specified in the YAML Configuration file used to deploy a resource of the kind of the defined CRD and it will use them to interact with the Kubernetes cluster as we specified in the Controller implementation. For our Simulator, we technically created two different custom Kubernetes Operators:

- the QKDSimualtor Operator to handle the deployment of the SImulatorManager and Message Broker modules;
- the NetTopology Operator to manage the deployment of the different instances of the Simulation Ednpoitn and Communication Channel modules that are needed to implement the model of network topology.

The framework we decided to use to implement these custom Operators is *Kubebuilder* and it exploits the Go Language. Kubebuilder is a framework for building Kubernetes APIs using CRDs. Similar to web development frameworks such as Ruby on Rails and SpringBoot, Kubebuilder increases velocity and reduces the complexity managed by developers for rapidly building and publishing Kubernetes APIs in Go. It builds on top of the canonical techniques used to build the core Kubernetes APIs to provide simple abstractions that reduce boilerplate and toil.

Kubebuilder does not exist as an example to copy-paste but instead provides powerful libraries and tools to simplify building and publishing Kubernetes APIs from scratch, it is developed on top of the controller-runtime and controller-tools libraries and it will build and provide us with the scaffolding files to create a custom Kubernetes Operator, we will just need to edit the appropriate files to get the Operator working as we wish it to. The first thing kubebuilder needs is the editing of the file describing the CRD that will be associated with the custom Operator. In section 5.1.3 we went over how YAML Configuration files are characterized by three sections:

- the Metadata section;
- the Spec section;
- the Status section.

In Kubebuilder to create our CRD, we will need to edit the file specifying what kind of fields to expect in the last two sections of the YAML Configuration file representing our CRD, paying special attention to the specification one.

```
// QKDSimulatorSpec defines the desired state of QKDSimulator
type QKDSimulatorSpec struct {

        RabbitMQ_Host string `json:"rabbit-host,omitempty"`
        RabbitMQ_Port int32 `json:"rabbit-port,omitempty"`
        Manager_Host string `json:"manager-host,omitempty"`
        Manager_Rest_Port int32 `json:"manager-rest-port,omitempty"`
        Manager_Gui_Port int32 `json:"manager-gui-port,omitempty"`
        Trng int32 `json:"trng,omitempty"`
        Pnum int32 `json:"pnum,omitempty"`
}
```

Listing 7.3: QKDSimulatorSpec

As we can see from the Go code in listing 7.3, for the QKDSimulator Operator, the fields we expect to find in the CRD are:

- the host and port needed to set up the RabbitMQ Message Broker;
- the host and ports needed to set up the Simulator Manager Web and GUI interfaces;
- wether a true random number generator should be used for the QKD simulations or not when generating random qubits;
- the amount of parallelism each Simulation Endpoint and Communication Channel module representing the modelled topology will have.

```
// NetTopologySpec defines the desired state of QKDSimulator
type NetTopologySpec struct {

        Nodes []Node `json:"nodes"`
        Adjacency [][]Node `json:"adjacency"`

}
```

Listing 7.4: NetTopologySpec

On the other hand, as illustrated by listing 7.4, the NetTopology Operator expects to receive an array (a slice in Go Language) of Node structures describing the nodes of the network topology as well as a matrix (2-D slice in Go) of them implementing the adjacency list representation of the graph that the network topology we want to model can be reconducted to.

At this point, after having used the proper Kubebuilder to make the Kubernetes cluster include the CRDs we just defined, to deploy any of those, it will be sufficient to create a YAML Configuration File having as the value of the Kind field the name of the desired CRD and insert in the specification section of the file the desired values of the expected fields.

We still need, however, to let Kubernetes know what to do to properly deploy and manage such CRDs and this is what will be done next. The next step Kubebuilder requires is to edit the `Reconcile` function contained in the Go file representing the custom Controller that will be associated with our custom Operator. The Reconcile function is run by the Controller every time an event happens regarding the CRD it manages. This is the function to edit to specify how we want the controller to act and handle any event regarding the lifecycle of its associated CRD in the Kubernetes cluster.

```
func (r *CRDReconciler) Reconcile(ctx context.Context, req ctrl.Request)
    (ctrl.Result, error) {
        // Loggin Set up
        logger := log.FromContext(ctx)

        //Get data from the CRD
        var crd_data CRD

        if err := r.Get(ctx, req.NamespacedName, &crd_data); err != nil {
                logger.Error(err, "name")
                return ctrl.Result{}, client.IgnoreNotFound(err)
        }

        //Your Custom Deploying Logic

        return ctrl.Result{}, nil
}
```

Listing 7.5: Reconciler function structure

As we can see from listing 7.5, the first thing to do is use the Go Kubernetes API client to retrieve the values of the fields specified in the specification section of the CRD. The values of these fields will be specified in a YAML configuration file that will be fed to the cluster to deploy the CRD. At this point, using the retrieved fields we need to implement what we want Kubernetes to do when an event regarding the CRD associated with the Controller happens.

As far as our custom Operators are concerned, to avoid complicating the implementation too much, we opted to implement just the CREATE and DELETE operations support for the CRDs. In particular, the QKDSimulator Controller will be responsible for deploying:

- the Deployment and Service components implementing the Simulator Manager module;
- the StatefulSet and Service components implementing the Message Broker module;
- the Simulation Endpoints ConfigMap;
- the Communication Channels ConfigMap;
- the Ingress Rule component. The Ingress Controller component is assumed to be already installed on the Kubernetes cluster the QKD Simulator will be deployed to.

The NetTopology Controller, on the other hand, will be in charge of deploying:

- a Deployment component implementing the Simulation Endpoint module for each node of the modelled network topology;
- a Deployment component implementing the Communication Channel module for each link of the modelled network topology.

To handle the deployment of all of these various Kubernetes components, we created a separate Go file called *helpers.go* where we inserted the various helper functions to make the deployment of the components much easier.

```
func (r *CRDReconciler) desiredKubernetesComponent(crd CRDr, conf_param1,
    conf_param2) (KubernetesComponent, error) {...}
```

Listing 7.6: Generic structure of a helper function

The code in listing 7.6 represents the general structure of one of our helper functions. We will have a helper function per Kubernetes component type and each of these functions will instantiate in Go, an object corresponding to the desired component and, before returning it, it will properly configure it using the provided parameters. As a result, the actual deployment of the Kubernetes component done in the *Reconcile* function of the Controller becomes much easier and its genera structure can be summed up as follows:

```
kubernetes_component, err := r.helper_function(crd, conf_param1,
    conf_param2,...)

if err != nil {
        return ctrl.Result{}, err
}

applyOpts := []client.PatchOption{client.ForceOwnership,
    client.FieldOwner("CRD")}

err = r.Patch(ctx, &crd, client.Apply, applyOpts...)
if err != nil {
        return ctrl.Result{}, err
}
```

Listing 7.7: Generic worklof to deploy a component

As we can see from the code in listing 7.7, to deploy a Kubernetes component, it will be sufficient to:

1. instantiate a variable containing the specification that our CRD will have the ownership of the Kubernetes component we will deploy. This operation is needed so that when deleting our CRD from the Kubernetes cluster, all of the components its associated Controller deployed will be deleted too;
2. Use the *Patch* function of the Go Kubernetes client API to deploy the desired component to our cluster.

At this point, by issuing the "make" commands specified in the documentation of Kubebuilder, we will be able to have our two different custom Kubernetes Operators up and running on our cluster. Therefore, we will be able to simply feed the Kubernetes cluster with just two different YAML Configuration files for the deployment of our two different CRDs and, as a result, we will have an instance of the QKD Simulator fully ready to be used.

An example of the structure of these two YAML Configuration files can be the following:

```
apiVersion: qkdsim.s276624.qkdsim.dev/v1
kind: QKDSimulator
metadata:
  name: qkdsimulator-sample
spec:
  rabbit-host: rabbittomq
  rabbit-port: 5673
  manager-host: sim-manager
  manager-rest-port: 6969
  manager-gui-port: 6997
  trng: 0
  pnum: 3
```

Listing 7.8: QKDSimulator YAML Configuration File example

```
apiVersion: qkdsim.s276624.qkdsim.dev/v1
kind: NetTopology
metadata:
  name: nettopology-sample
spec:
  adjacency:
    - - id: b
      - id: c
    - - id: a
    - - id: a
  nodes:
    - id: a
    - id: b
    - id: c
```

Listing 7.9: NetTopoloy YAML Configuration File example

As we can see from the two snippets of code in listings 7.8 and 7.9, the structure of the QKDSimulator YAML file is pretty straightforward and requires just the specification of the different fields described previously.

On the other hand, the NetTopology one can seem more ambiguous but it just lists the values of two different fields: the node field and the adjacency one. In particular, in our example, we have a topology of three nodes, respectively "a", "b" and "c". Node "a" is connected to both nodes "b" and "c", while nodes "b" and "c" are just connected to node "a".

The example above highlights how we considered as undirected the graph representing a network topology, thus the repetitions in its adjacency list representation.

A more detailed description of the logic implemented by the two custom Kubernetes Operators can be found in Appendix B.

7.4 Workflow Implementation

7.4.1 Deployment and Init Phase

In section 6.5.1 we described how the deployment of the new version of the QKD Simulator is divided in two different phases:

1. deployment of the Simulator Manager and RabbitMQ modules;
2. deployment of the desired network topology to model.

As described in section 7.3, in order to fulfil these two steps, thanks to the implementation of two custom Kubernetes Operators, it will simply be sufficient to prepare two different YAML files following the structure shown respectively in listings 7.8 and 7.9 and feed them to the Kubernetes cluster through the appropriate `kubectl` command:

```
kubectl apply -f YAMLfilename.yaml
```

A more detailed description of how to set up and deploy an instance of the new version of the QKD Simulator can be found in Appendix A.

Moreover, in section 6.5.2, we described the different steps needed to be performed before actually being able to start a QKD simulation on the simulator. To be more specific, these steps were:

1. notify the Simulator Manager module about the modelled network topology deployed;
2. notify the Simulator Manager module about any particular authentication method the user will want to use for the classical channel during QKD simulations;
3. choose the pair of directly linked endpoints of the topology to carry out QKD Simulations between.

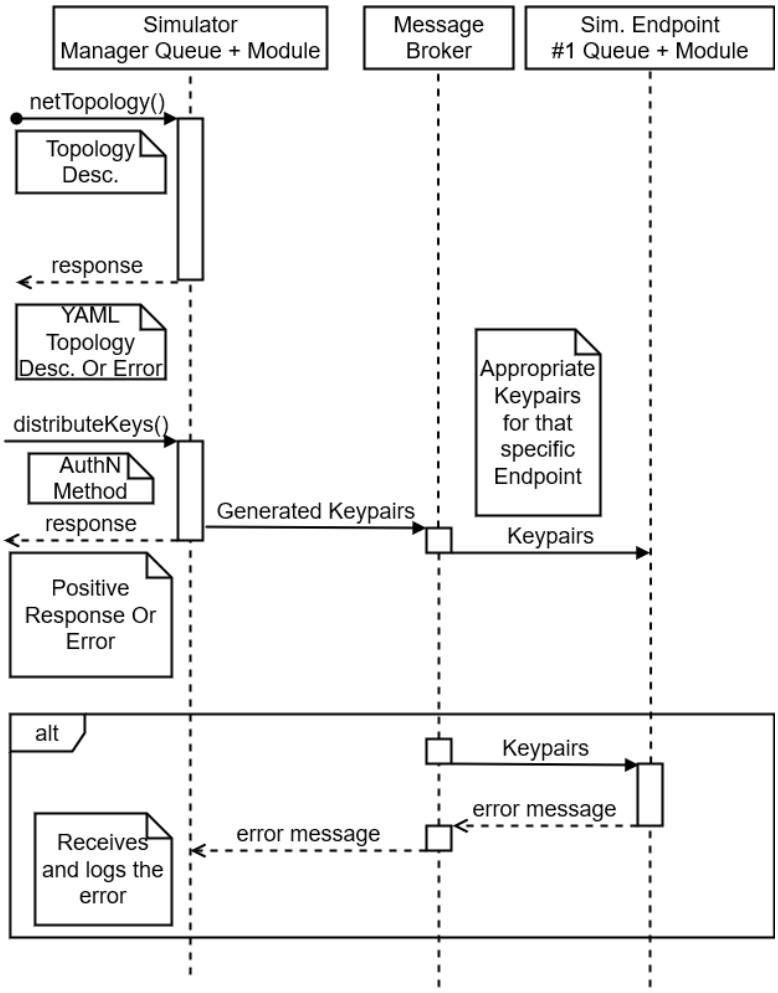


Figure 7.4: First 2 steps of Init phase

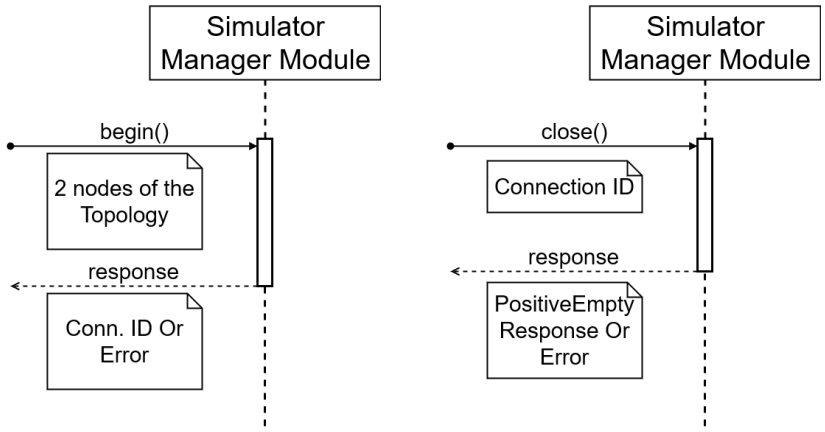
Other than through the GUI, the 2 steps mentioned can be carried out, as shown in figure 7.4, by using the Simulator's Web Interface through the `neTopology` and `distributeKeys` functions exposed as endpoints by the Flask WebSerevr container running inside the Simulator Manager module.

In particular, as far as the `getTopology` function is concerned, it will deserialize the data contained in the body of the receiver HTTP POST request and check that such data has been provided in the right format. Next, thanks to the use of the NetworkX Python Library we will turn the provided adjacency list representation of the topology into a `nx.Graph` object used to represent undirected graphs. Provided no errors occur, thanks to a dictionary and the PyYAML is a YAML parser and emitter for Python, the representation of the custom Kubernetes YAML file, describing the provided topology, will be returned as a response. This will allow the user to quickly check if the topology sent matches the one deployed or not. In case of errors, a 400 status code HTTP Response will be returned with the body containing a description of what happened.

On the other hand, in the `distributeKeys` function, according to the authentication method provided as a parameter in the body of the HTTP POST request, the appropriate Python library will be used to generate the keys to distribute. Afterwards, a loop is used to send the generated keys to the appropriate nodes of the topology. The keys will be sent through RabbitMQ messages with the Type field set to value "Keys". If the whole operation is terminated successfully, a positive 200 HTTP response will be returned. Otherwise, if any error occurred, a 400 HTTP Response will be returned and its body will contain a description of what happened.

Moreover, if any of the Simulation Endpoint modules encountered an error while receiving and managing the parameters contained in the messages of type "Keys", the error will be notified to the Simulator Manager through an appropriate message sent to its queue. The Simulator Manager when receiving this message will handle it by logging the issue.

In order to conclude the Init phase of the simulator, the last step to perform consists of choosing a couple of endpoints of the topology the user wants to carry out one or more QKD simulations between.



(a) Connection Opening Scheme (b) Connection Closing Scheme

Figure 7.5: Opening And Closing Connection Schemes

Other than through the GUI, this step can be fulfilled, as shown in figure 7.5, by using the Simulator's Web Interface through the `begin` and `close` functions exposed as endpoints by the Flask WebSerevr container running inside the Simulator Manager module.

As regards the `begin` function, after having deserialized the name of the two nodes of the topology the connection will be related to, if no errors occur, a unique ID associated with the new connection will be generated with the help of the UUID Python Library.

Next, the newly generated ID will be saved as the key part of a dictionary data structure entry and the corresponding value to such key will be the names of the 2 nodes received as parameters in the HTTP request. If everything went according to plan, a 200 status code HTTP response, containing the generated unique ID in its body, will be returned In case of errors, a 400 status code HTTP Response will be returned with the body containing a description of what happened.

On the other hand, the `close` function will deserialize the unique ID of the connection to close provided in the HTTP request body and if such ID has a match inside the dictionary

containing the currently open connections to the simulator, the key-value pair corresponding to such connection will be removed from the dictionary and a 200 status code HTTP response will be returned. In case the provided ID has no matches in the dictionary containing the currently open connections to the Simulator, a 400 status code HTTP Response will be returned with the body containing a description of the error.

It is important to note that `begin` represents the actual function implementing the last step of the Init phase, the `close` one is needed more to clean up the data structures when the user is satisfied with the QKD simulations run between two specific endpoints of the topology.

7.4.2 Point-to-point QKD Simulation

In this section, we will go over the implementation of a point-to-point QKD simulation of the BB84 protocol, given this is the only protocol supported by the Simulator at the moment.

It is important to note that, we aimed to make the structure of the steps chosen to implement the simulation of the BB84 protocol as general as possible for it to be easily reusable in the foreseeable future where the Simulator will be extended with the addition of more QKD protocols to simulate.

To make easier a thorough comprehension of the different steps behind the simulation, the scheme 7.6 will illustrate the different phases characterizing this operation.

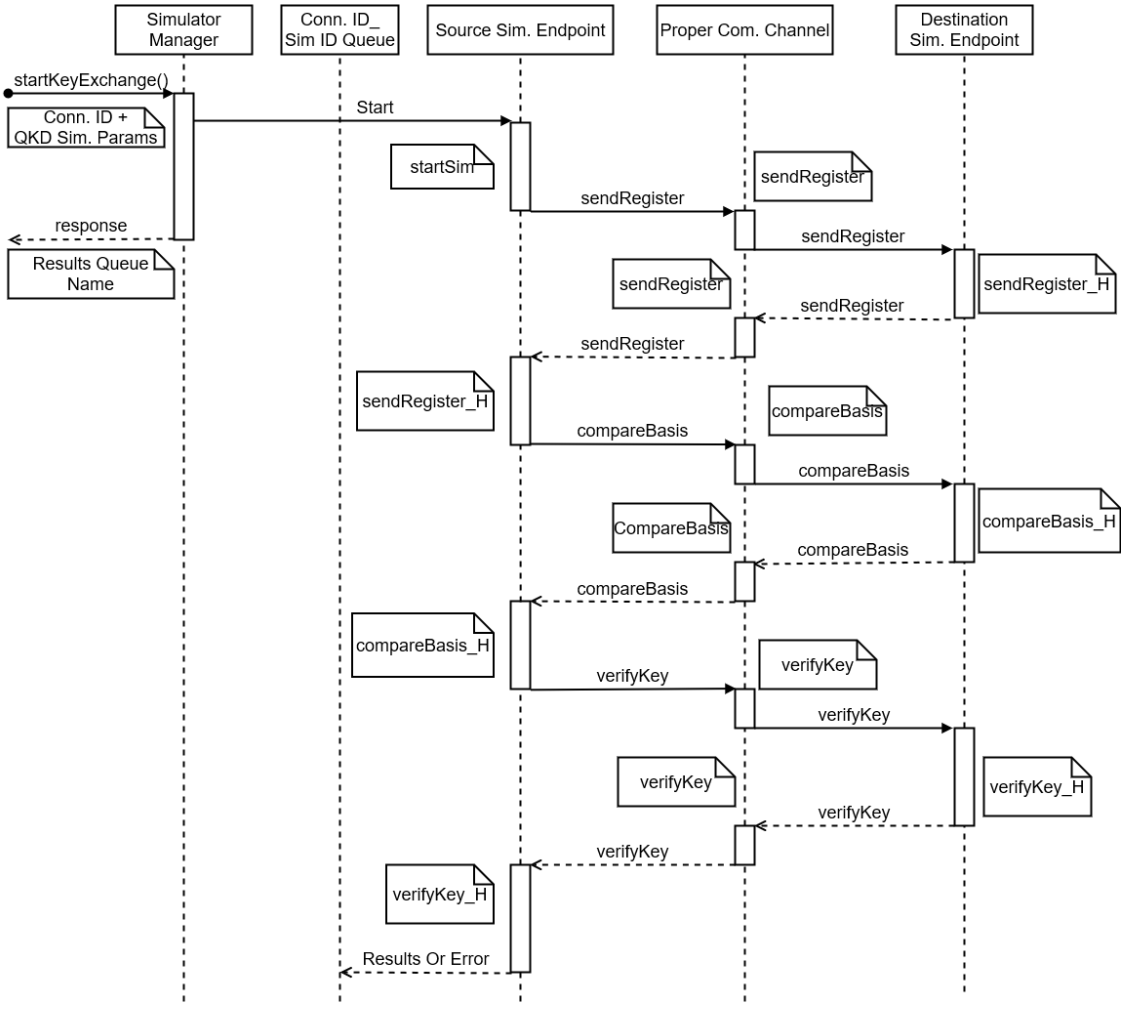


Figure 7.6: BB84 Point-to-Point Simulation Implementation Scheme

As figure 7.6 illustrates that, when we want to start a point-to-point QKD simulation, we need to provide the simulator with the ID of the open connection such simulation will be related to. This allows the Simulator to know which specific Simulation Endpoint Modules and Communication Channel module will be involved in the current simulation.

If the provided parameters are valid, the unique identifier of the message queue, RabbitMQ will put the results of the simulation or any error related to it on, will be returned in the body of an HTTP response with status code 200. On the other hand, if the provided connection ID doesn't have a match among the currently open connections to the Simulator, a message describing the error will be returned in the body of an HTTP response with status code 400.

Assuming these initial steps are successful, we can now describe the workflow of the implementation of a point-to-point simulation. Always keep in mind that the handling of the messages here is entirely done by RabbitMQ and that the Communication Channel module always acts as an intermediary in the flow of messages among source and receiver. In particular, during the communication, such module will either perform different attacks the BB84 protocol can be subject to or simply act as a passthrough. Besides, remember that all of the steps of the actual Quantum simulation are handled and implemented with the aid of the Qiskit framework.

After receiving a message with *type* field set to "Start", the source endpoint generates a bits string of the same length of the chunk by using a true random number generator. Next, by using a qiskit circuit, it encodes the bits string in a Statevector object by using a set of randomly chosen bases. The used bases are saved in a bases table variable for further processing. Finally, It serialises Statevector objects using the Pickle Python Library and sends them to the receiving endpoint using a message with type field set to "sendRegister".

It is important to note that all of the serialized objects to send will be inserted in the *body* field of the message. The receiver endpoint will choose a random sequence of basis it will then use to measure the received qubits. If no issues occurred, the receiver will send a message with Type field set to "sendRegister" and Body field set to an "OK" string back to the receiver.

The source endpoint checks the answer to the previous message. If it is "OK" it knows the receiver has randomly selected a set of bases and measured the received Statevector in those bases, else an error occurred and the function ends without returning a key. The source then signs the basis table formerly saved with its private key and sends it to the receiving endpoint through another message with type field set to "compareBasis".

After that, the receiver will compare the set of basis the source sent to its ones to sift the key: if a different basis is chosen for a particular qubit, the related bit is discarded. The receiver will then send a message containing its basis. This message will have the type field set to "compareBasis". As a consequence, such endpoint will also perform the steps characterizing the key-sifting process we just explained.

It is important to note that the contents of the messages with type field set to "compareBasis" is signed by using the authentication method chosen for the simulation and the keys related to it. If the signature is not valid the simulation ends and does not return a key, but instead, it will signal the error.

Now, the source randomly selects a set of bits along with the key that will be used to calculate the QBER value and save the indexes of these bits in a separated array. Then, it signs the selected bits and indexes array and sends them together with their signatures to the receiving endpoint by using a message with type field set to "verifyKey". It receives back the receiver's bits of the key at the same position as those sent together with a signature for those data.

Finally, the source compares the received bits with the one sent and checks the error rate as the number of equals bits divided by key length. Since this is just a simulation and channel errors are not possible this value will likely be equal to one (which means the QBER is 0%).

Hence if the value is different from one the key is discarded, otherwise the key is verified and successfully returned to high level module by sending an appropriate message with type field set to "Results" and the extracted key along with the QBER value in its body field. In case of errors, on the other hand, a message with type field set to "Error" and the description of the issue in its body field is returned on the queue instead.

The notes provided in the scheme illustrated by figure 7.6 shows the names of the functions used to handle the different steps of the simulation by properly managing a certain type of received message.

All of these functions will perform different operations according to the values of the *type* field of the received message and its flow direction. Such flow direction can be deduced from the values of the *destination*, *source* and *metadata* fields inside the message and the me.

A more detailed description of the logic implemented by each of the message handling functions can be found in Appendix B.

Chapter 8

Test and Validation

The proposed solution has been widely tested to identify and understand its limitations and come up with possible improvements.

All of the different tests were run on a system running Ubuntu 20.04 LTS with 16 GB of RAM, 256 GB of M.2 PCIe Gen3 NVME Solid State Drive and a CPU Ryzen 7 3700X. The used software components have the following versions:

- K3s version 1.21.5
- Go Programming Language version 1.16.8
- Python version 3.8.8 with the following modules installed:
  - Qiskit version 0.29.1
  - Flask version 1.1.1
  - PySPX version 0.4.0
  - Cryptography version 3.4.8
  - Pika version 1.2.0
  - NetworkX version 2.6.3
  - Matplotlib version 3.4.3
  - PyYAML version 5.3
  - Jupyter version 1.0.0

To perform the tests a new instance of the QKD Simulator was set up and deployed on the K3s Kubernetes cluster running on the system. Appendix A describes in detail how to configure and set up a completely new instance of the QKD Simulator on the Kubernetes cluster.

8.0.1 Tests

The tests run on the new version of the QKD Simulator were aimed at evaluating its performance during a point-to-point QKD simulation both in terms of:

- time;
- throughput (bit/s).

In particular, each test evaluated the performance variation for different values of a specific QKD Simulator and QKD simulation configuration parameter. To be more specific, the different parameters taken into consideration were:

- the *chunk size*, it represents the qubit length of the quantum register used by Qiskit during a QKD simulation;
- the *key length*, it represents the length of the key to generate at the end of a QKD simulation;
- the *key number*, it represents the amount of consecutive QKD simulation required between two specific endpoints of the modelled network topology;
- the *worker number*, it represents the amount of parallelism configured and to be used by each endpoint and communication channel of the modelled topology when consuming messages from their corresponding RabbitMQ queue.

Moreover, it is important to note that all of the tests that were run on the simulator were done with the BB84 protocol selected for simulation since it is currently the only one supported. Besides, the network topology modelled and deployed to the cluster was fixed too to prevent it from influencing the results obtained when varying the other configuration parameters listed above.

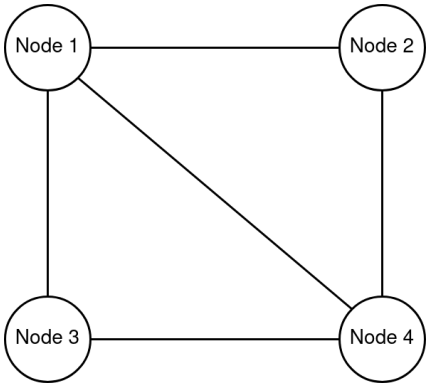


Figure 8.1: Network topology modelled and used during all the performed tests

Figure 8.1 represents the network topology modelled and used during each of the tests that were performed.

The first test to be run was aimed at identifying the best length of the quantum register used by Qiskit during a QKD simulation to minimize the exchange time. As far as the rest of the configuration parameters of the simulator are concerned, they were respectively set to: *key length* = 4096 bits, *key number* = 1 key and *worker number* = 1 worker.

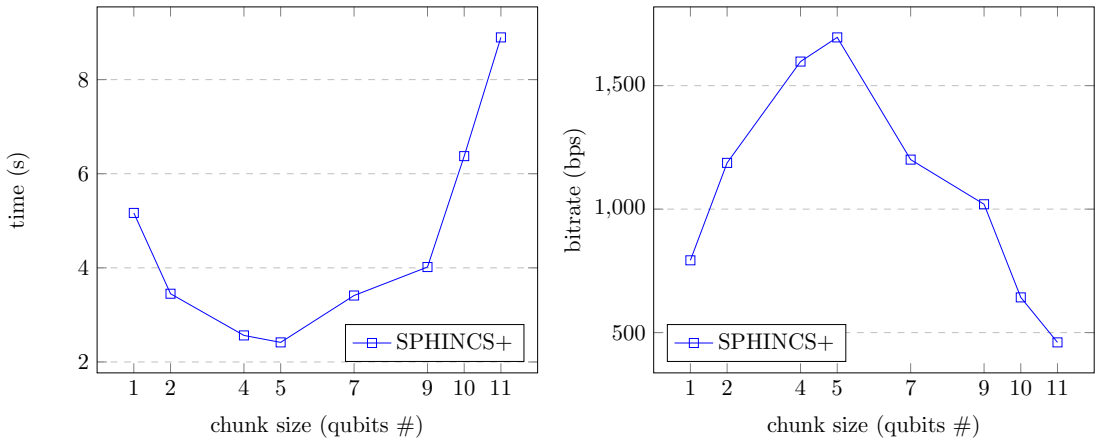


Figure 8.2: Performance in terms of time (on the left) and throughput (on the right) for a single BB84 4096 bits key exchange depending on the qubit register length. (*key length* = 4096 bits, *key number* = 1 key and *worker number* = 1 worker)

As we can see from figure 8.2 above, when varying the length of the quantum register to use during the QKD simulation, the performance changes with a peak when selecting a 5-qubit register. As a result, for the rest of the tests that were run on the simulator, the chunk size value chosen was indeed 5.

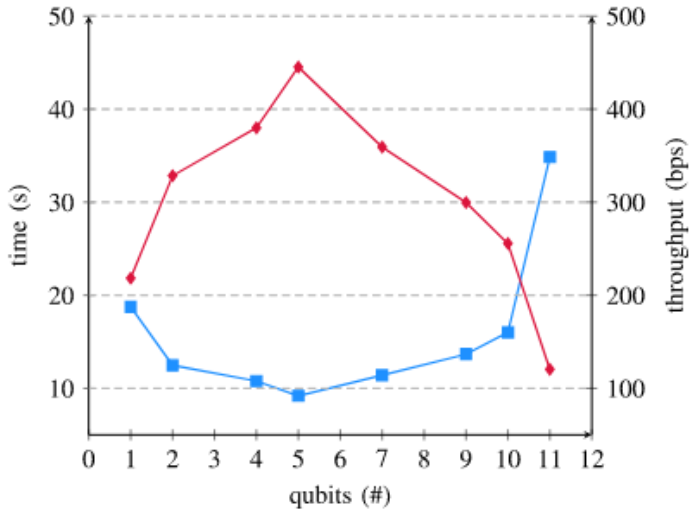


Figure 8.3: Time for a single BB84 4096 bits key exchange depending on the qubit register length for the older version of the QKD Simulator (*key length* = 4096 bits, *key number* = 1 key and *worker number* = 1 worker), source:[68]

It is important to note that as we can see from figure 8.3 above, the same kind of behaviour was encountered during testing of the previous version of the QKD Simulator as well. This result is coherent with what was expected since the Qiskit core to handle QKD simulation remained the same even in the new version of the simulator.

The second test to be run was aimed at evaluating the performance of the new version of the simulator when varying the length of the key to be generated during a single simulation. As far as the rest of the configuration parameters of the simulator are concerned, they were respectively set to: *chunk size* = 5, *key number* = 1 and *worker number* = 1.

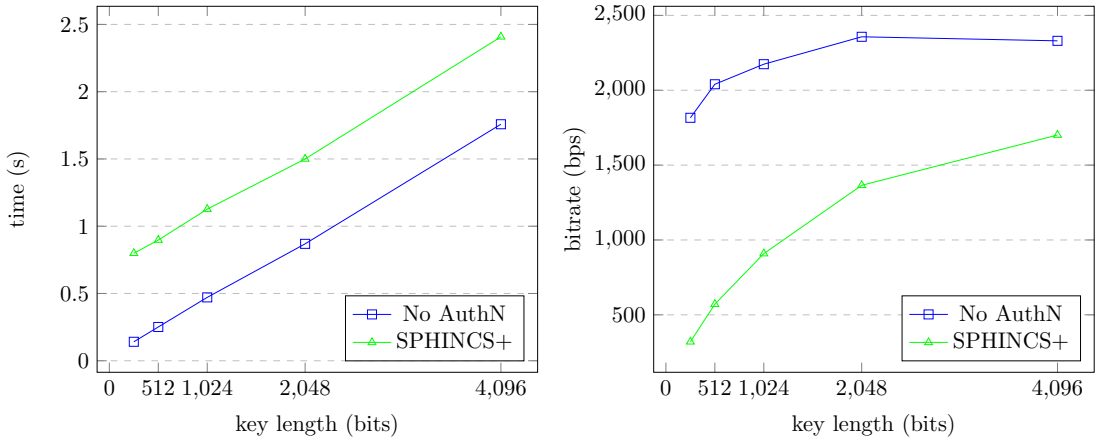
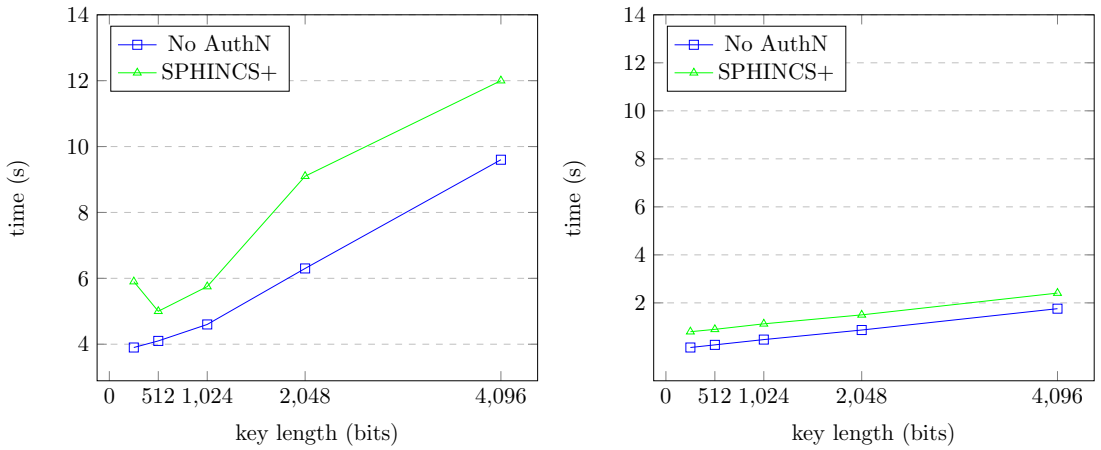


Figure 8.4: Performance in terms of time (on the left) and throughput (on the right) for a single BB84 key exchange depending on the key length.(*chunk size* = 5 qubits, *key number* = 1 key and *worker number* = 1 worker)

As we can see from the figure 8.4 above, the usage of an authN method like SPHINCS+ to perform digital signature and authenticate the classical channel part of the communication during

a QKD simulation significantly impacts the performance of the QKD Simulator both in terms of time and throughput. However, it is interesting to note how the gap in terms of throughput narrows itself the longer the requested key to be generated is. This is probably due to the fact that with the parallelism set to just 1 worker per entity of the modelled topology, a limit in terms of throughput is being reached between 2 and 2.5 Kbps.



(a) BB84 key exchange time depending on the key length for older version of the QKD Simulator, source:[68]

(b) BB84 key exchange time depending on key length for new version of the QKD Simulator

Figure 8.5: Time performance comparison between the old (on the left) and new version (on the right) of the QKD Simulator for a single BB84 key exchange depending on key length. (*chunk size* = 5 qubits, *key number* = 1 key and *worker number* = 1 worker)

As we can see from the graphs in figure 8.5, thanks to the implementation of the internal communications of the new version of the QKD Simulator through messages and a Broker like RabbtiMQ, when compared to its previous version (figure 8.5a), the new version of the QKD Simulator (figure 8.5b) has gained a significant amount of performance of the single simulation, even if the parallelism is set to only 1 worker, in terms of time and therefore of throughput too, The third test that was run aimed at evaluating the performance of the QKD Simulator when requesting a varying number of QKD simulations between two specific endpoints of the modelled topology all at once. As far as the rest of the configuration parameters of the simulator are concerned, they were respectively set to: *key length* = 4096 bits, *chunk size* = 5 qubits and *worker number* = 1 worker.

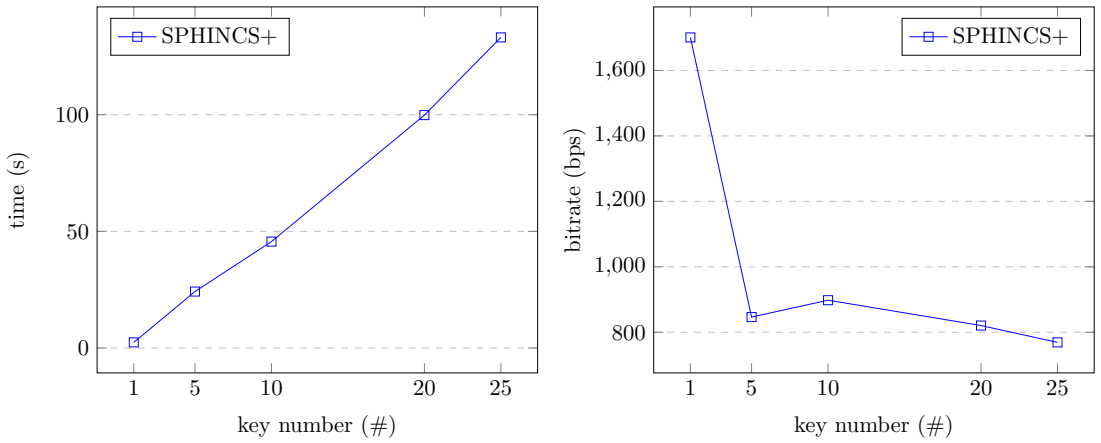


Figure 8.6: Performance in terms of time (on the left) and throughput (on the right) of new QKD Simulator depending on the number of keys requested all at once. (*chunk size* = 5 qubits, *key length* = 4096 bits and *worker number* = 1 worker)

As we figure 8.6 show, whenever more than one QKD simulation is requested to be executed between two specific endpoints of the modelled network topology, the performance of the new version of the QKD Simulator significantly drops both in terms of time and especially in terms of throughput. To mitigate this issue and improve the simulator's performance, parallelism, i.e. the worker number parameter, was introduced. The performance improvement introduced by parallelism is exactly what was tested next on the new version of the QKD Simulator. As far as the rest of the configuration parameters of the simulator are concerned, they were respectively set to:

- *key length* = 4096 bits;
- *chunk size* = 5 qubits;
- *key number* = 25 keys.

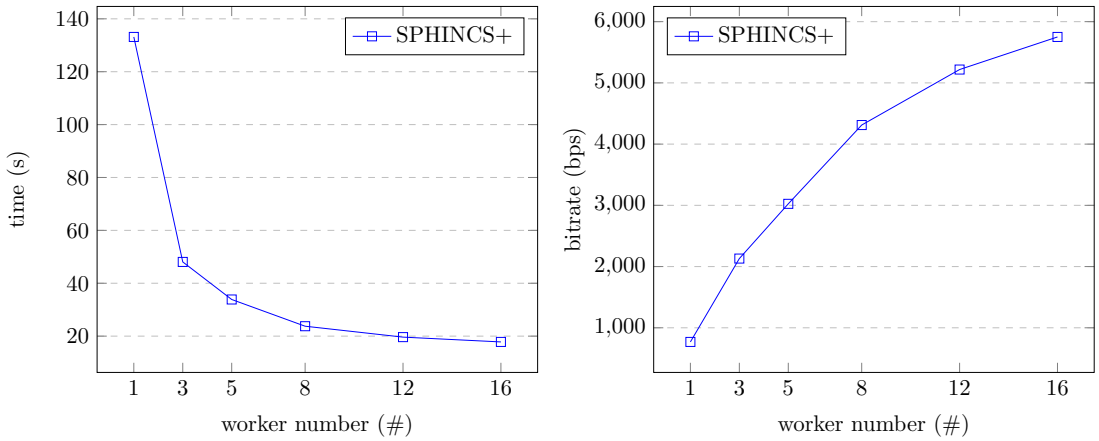


Figure 8.7: Performance in terms of time (on the left) and throughput (on the right) of new QKD Simulator depending on amount of parallelism configured. (*chunk size*=5 qubits, *key number*=25 keys, *key length*=4096 bits)

As we can see from figure 8.7 the introduction of parallelism, i.e. the increase in the number of workers for each endpoint and communication channel of the modelled network topology, significantly increased the performance of the QKD Simulator both in terms of time and throughput, especially whenever two or more QKD simulations between a specific pair of endpoints of the modelled topology are requested all at once. Moreover, it is important to note how the throughput curve illustrated in figure 8.7b tends to normalize between 5 Kbps and 6 Kbps. This is probably due to the fact that the maximum QKD simulation speed achievable through the Qiskit framework was reached.

Finally, a crash of either one of the two endpoints of the modelled topology has been verified to happen whenever any key exchange was requested for the BB84 protocol, on the usual topology, without using a true quantum random number generator and with the remaining configuration parameters set to the following values:

- *key length* = 4096 bits;
- *key number* $\geq$ 30 keys;
- *chunk size* = 5 qubits;
- *worker number* = 16 workers;
- *authentication* = SPHINCS+.

This is due to the fact that as the different QKD simulations were carried out, the RAM of the system filled up completely.

This issue is probably related to the way RabbitMQ handles message ordering on its queues. To be more specific, since RabbitMQ leverages a FIFO mechanism for managing the messages on a queue, whenever two or more QKD simulations between two endpoints are requested all at once, instead of progressively handling them by completing one before passing to the other, the different simulations will be handled altogether, in particular, the FIFO mechanism will force the simulator to finish handling a single simulation step for all of the requested ones before passing to the next one.

As a result, since the Simulator Endpoint modules of the QKD Simulator each maintain a data structure to keep track of the parameters for the different ongoing QKD simulations, their size increases and cannot be reduced until the handling of the last QKD simulation step was started to be managed for each of the requested simulations; it is during this final step that the data structure is freed from the parameters of the specific simulation.

This especially represents an issue whenever the new version of the QKD Simulator needs to be used to perform a virtually continuous ($key\ number \rightarrow \infty$) exchange of keys between two specific endpoints of the modelled topology. However, a first quick and easy way to overcome the issue would be to introduce a parametrized amount of time to wait in between smaller burst requests of QKD simulations to perform. Moreover, to fully exploit the configured parallelism, it would be advisable to set the $key\ number = worker\ number$ for each of these smaller burst requests. As a result, the problem can be solved but if not selected properly, the wait time value could negatively impact the QKD Simulator performance.

Chapter 9

Conclusions

The proposed work tried to provide a redesign and extension of the approach for Quantum Key Distribution Simulation proposed with the QSS QKD Simulator by the TORSEC research group of Politecnico di Torino.

In particular, the previous version of the simulator presented a series of limitations, such as:

- the possibility to perform point-to-point QKD simulations possible only between the same two endpoints;
- a limited scalability due to the use of a TCP/IP network and REST APIs to handle the internal communications of the simulator during simulations;
- a rather complex procedure for deployment on distributed infrastructures.

The new version of the QKD Simulator was exactly developed to address all of these limitations while also simplifying the architecture and improving performance. The core performing QKD Simulation operations has been kept the same, but thanks to the use of Kubernetes orchestration technology and the development of two different custom Operators for it, it is now possible to easily deploy the simulator by using just two YAML files. Kubernetes allows, as is, without any effort, to distribute our solution on multiple nodes and potentially scale it ad libitum.

Besides, one of the two custom Kubernetes Operators was explicitly developed to provide the new version of the simulator with a new functionality: the possibility of modelling and deploying an entire network topology. In particular, it will now be possible to perform point-to-point QKD simulations between any pair of modelled endpoints that are directly connected by a link.

The internal communications of the simulator during the QKD simulation are now handled by using asynchronous messaging implemented through RabbitMQ in order to improve scalability. Moreover, parallelism was implemented through Python multiprocessing when consuming and managing messages for each module of the simulator representing a node or link of the modelled topology. The number of concurrent workers to consume messages with can be easily set as a configuration parameter to be set when deploying the simulator.

Tests run on the new version of the simulator showed a flexible architecture and improved performance both in terms of time and throughput even when multiple QKD simulations are proceeding at the same time thanks to asynchronous messaging and parallelism. Some criticalities emerged in terms of the amount of resources consumed by the simulator when the number of concurrently ongoing QKD simulations on the modelled topology increases. This issue could be dealt with by limiting the number of requested QKD simulations that are allowed to proceed at the same time.

Future work on this subject may regard the further extension of the simulator functionalities by developing different back ends for simulating more QKD protocols or modelling different possible attacks and non-ideal scenarios of simulation with decoherence and errors on the quantum channel. Besides, a more direct evolution of the proposed work could also be the development of an improved way for the simulator to handle assigned resources in order to further improve its scalability while also maintaining acceptable performance.

Bibliography

[1] G. S. Vernam, "Cipher printing telegraph systems for secret wire and radio telegraphic communications", Transactions of the American Institute of Electrical Engineers, vol. XLV, January-December 1926, pp. 295–301, DOI 10.1109/T-AIEE.1926.5061224

[2] C. E. Shannon, "Communication theory of secrecy systems", The Bell System Technical Journal, vol. 28, October 1949, pp. 656–715, DOI 10.1002/j.1538-7305.1949.tb00928.x

[3] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring", Proceedings 35th Annual Symposium on Foundations of Computer Science, 20-22 Nov, 1994, pp. 124–134, DOI 10.1109/SFCS.1994.365700

[4] N. Gisin, G. Ribordy, W. Tittel, and H. Zbinden, "Quantum cryptography", Rev. Mod. Phys., vol. 74, March 2002, pp. 145–195, DOI 10.1103/RevModPhys.74.145

[5] V. Scarani, H. Bechmann-Pasquinucci, N. J. Cerf, M. Dušek, N. Lütkenhaus, and M. Peev, "The security of practical quantum key distribution", Rev. Mod. Phys., vol. 81, September 2009, pp. 1301–1350, DOI 10.1103/RevModPhys.81.1301

[6] E. Diamanti and A. Leverrier, "Distributing secret keys with quantum continuous variables: Principle, security and implementations", Entropy, vol. 17, August 2015, pp. 6072–6092, DOI 10.3390/e17096072

[7] E. Diamanti, H.-K. Lo, B. Qi, and Z. Yuan, "Practical challenges in quantum key distribution", npj Quantum Information, vol. 2, November 2016, p. 16025, DOI 10.1038/npjqi.2016.25

[8] D. Dieks, "Communication by epr devices", Physics Letters A, vol. 92, November 1982, pp. 271–272, DOI https://doi.org/10.1016/0375-9601(82)90084-6

[9] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned", Nature, vol. 299, October 1982, pp. 802–803, DOI 10.1038/299802a0

[10] P. A. M. Dirac, "A new notation for quantum mechanics", Mathematical Proceedings of the Cambridge Philosophical Society, vol. 35, October 1939, pp. 416–418, DOI 10.1017/S0305004100021162

[11] G. L. Khym and H. Yang, "Quantum entanglement does not violate the principle of special theory of relativity", Physics Essaysy, vol. 29, December 2016, pp. 553–554, DOI 10.4006/0836-1398-29.4.553

[12] J. S. Bell, "On the einstein podolsky rosen paradox", Physics Physique Fizika, vol. 1, November 1964, pp. 195–200, DOI 10.1103/PhysicsPhysiqueFizika.1.195

[13] M. Born, "Quantum mechanics of collision processes", Zeit. Physik, vol. 37, June 1926, pp. 863–867, DOI 10.1007/BF01397477

[14] H. E. Brandt, "Positive-operator and projection valued measurements in quantum key distribution", Journal of Modern Optics, vol. 54, February 2007, pp. 2357–2363, DOI 10.1080/09500340701639557

[15] M. Ben-Or, M. Horodecki, D. W. Leung, D. Mayers, and J. Oppenheim, "The universal composable security of quantum key distribution", Proceedings of the Second International Conference on Theory of Cryptography, Berlin, Heidelberg, February 10, 2005, pp. 386–406, DOI 10.1007/978-3-540-30576-7_21

[16] R. Renner and R. König, "Universally composable privacy amplification against quantum adversaries", Proceedings of the Second International Conference on Theory of Cryptography, Berlin, Heidelberg, February 10, 2005, pp. 407–425, DOI 10.1007/978-3-540-30576-7_22

[17] H.-K. Lo, X. Ma, and K. Chen, "Decoy state quantum key distribution", Phys. Rev. Lett., vol. 94, June 2005, p. 230504, DOI 10.1103/PhysRevLett.94.230504

[18] X. Ma and H.-K. Lo, "Quantum key distribution with triggering parametric down-conversion sources", New Journal of Physics, vol. 10, July 2008, p. 073018, DOI 10.1088/1367-2630/10/7/073018

[19] G. J. M. D.F. Walls, "Quantum optics", Springer, Berlin, Heidelberg, 2008

[20] F. Xu, X. Ma, Q. Zhang, H.-K. Lo, and J.-W. Pan, "Secure quantum key distribution with realistic devices", Rev. Mod. Phys., vol. 92, May 2020, p. 025002, DOI 10.1103/RevModPhys.92.025002

[21] G. Brassard and L. Salvail, "Secret-key reconciliation by public discussion", Advances in Cryptology — EUROCRYPT '93, Berlin, Heidelberg, May 23-27, 1993, pp. 410–423, DOI 10.1007/3-540-48285-7_35

[22] W. T. Buttler, S. K. Lamoreaux, J. R. Torgerson, G. H. Nickel, C. H. Donahue, and C. G. Peterson, "Fast, efficient error reconciliation for quantum cryptography", Phys. Rev. A, vol. 67, May 2003, p. 052303, DOI 10.1103/PhysRevA.67.052303

[23] C. Elliott, A. Colvin, D. Pearson, O. Pikalo, J. Schlafer, and H. Yeh, "Current status of the DARPA quantum network", Quantum Information and Computation III, May 25, 2005, pp. 138 – 149, DOI 10.1117/12.606489

[24] A. Nakassis, J. C. Bienfang, and C. J. Williams, "Expeditious reconciliation for practical quantum key distribution", Quantum Information and Computation II, August 24, 2004, pp. 28 – 35, DOI 10.1117/12.541698

[25] R. N. D.J.C. MacKay, "Near shannon limit performance of low density parity check codes", Electronics Letters, vol. 32, August 1996, pp. 1645–1646(1), DOI 10.1049/el:19961141

[26] D. MacKay, S. Wilson, and M. Davey, "Comparison of constructions of irregular gallager codes", IEEE Transactions on Communications, vol. 47, October 1999, pp. 1449–1454, DOI 10.1109/26.795809

[27] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes", IEEE Transactions on Information Theory, vol. 47, February 2001, pp. 619–637, DOI 10.1109/18.910578

[28] X.-Y. Hu, E. Eleftheriou, and D. Arnold, "Regular and irregular progressive edge-growth tanner graphs", IEEE Transactions on Information Theory, vol. 51, January 2005, pp. 386–398, DOI 10.1109/TIT.2004.839541

[29] M. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation", IEEE Transactions on Communications, vol. 47, May 1999, pp. 673–680, DOI 10.1109/26.768759

[30] J. Ha, J. Kim, D. Klinc, and S. McLaughlin, "Rate-compatible punctured low-density parity-check codes with short block lengths", IEEE Transactions on Information Theory, vol. 52, February 2006, pp. 728–738, DOI 10.1109/TIT.2005.862118

[31] T. B. Pedersen and M. Toyran, "High performance information reconciliation for qkd with cascade", 2013

[32] A. R. Dixon and H. Sato, "High speed and adaptable error correction for megabit/s rate quantum key distribution", Scientific Reports, vol. 4, December 2014, p. 7275, DOI 10.1038/srep07275

[33] G. Brassard, N. Lütkenhaus, T. Mor, and B. C. Sanders, "Limitations on practical quantum cryptography", Phys. Rev. Lett., vol. 85, August 2000, pp. 1330–1333, DOI 10.1103/PhysRevLett.85.1330

[34] C.-H. F. Fung, B. Qi, K. Tamaki, and H.-K. Lo, "Phase-remapping attack in practical quantum-key-distribution systems", Phys. Rev. A, vol. 75, March 2007, p. 032314, DOI 10.1103/PhysRevA.75.032314

[35] S.-H. Sun, M. Gao, M.-S. Jiang, C.-Y. Li, and L.-M. Liang, "Partially random phase attack to the practical two-way quantum-key-distribution system", Phys. Rev. A, vol. 85, March 2012, p. 032304, DOI 10.1103/PhysRevA.85.032304

[36] S.-H. Sun, F. Xu, M.-S. Jiang, X.-C. Ma, H.-K. Lo, and L.-M. Liang, "Effect of source tampering in the security of quantum cryptography", Phys. Rev. A, vol. 92, August 2015, p. 022304, DOI 10.1103/PhysRevA.92.022304

[37] Y.-L. Tang, H.-L. Yin, X. Ma, C.-H. F. Fung, Y. Liu, H.-L. Yong, T.-Y. Chen, C.-Z. Peng, Z.-B. Chen, and J.-W. Pan, "Source attack of decoy-state quantum key distribution using phase information", Phys. Rev. A, vol. 88, August 2013, p. 022308, DOI 10.1103/PhysRevA.88.022308

[38] W.-Y. Hwang, "Quantum key distribution with high loss: Toward global secure communication", Phys. Rev. Lett., vol. 91, August 2003, p. 057901, DOI 10.1103/PhysRevLett.91.057901

[39] X.-B. Wang, "Beating the photon-number-splitting attack in practical quantum cryptography", Phys. Rev. Lett., vol. 94, June 2005, p. 230503, DOI 10.1103/PhysRevLett.94.230503

[40] B. Qi, C.-H. F. Fung, H.-K. Lo, and X. Ma, "Time-shift attack in practical quantum cryptosystems", 2006

[41] S. Pirandola, U. L. Andersen, L. Banchi, M. Berta, D. Bunandar, R. Colbeck, D. Englund, T. Gehring, C. Lupo, C. Ottaviani, J. L. Pereira, M. Razavi, J. Shamsul Shaari, M. Tomamichel, V. C. Usenko, G. Vallone, P. Villoresi, and P. Wallden, "Advances in quantum cryptography", Advances in Optics and Photonics, vol. 12, December 2020, p. 1012, DOI 10.1364/aop.361502

[42] C. F. Fung, K. Tamaki, B. Qi, H. Lo, and X. Ma, "Security proof of quantum key distribution with detection efficiency mismatch", Quantum Inf. Comput., vol. 9, no. 1&2, 2009, pp. 131–165, DOI 10.26421/QIC9.1-2-8

[43] Y.-Y. Fei, X.-D. Meng, M. Gao, H. Wang, and Z. Ma, "Quantum man-in-the-middle attack on the calibration process of quantum key distribution", Scientific Reports, vol. 8, March 2018, p. 4283, DOI 10.1038/s41598-018-22700-3

[44] H.-K. Lo, M. Curty, and B. Qi, "Measurement-device-independent quantum key distribution", Phys. Rev. Lett., vol. 108, March 2012, p. 130503, DOI 10.1103/PhysRevLett.108.130503

[45] N. Lütkenhaus, "Estimates for practical quantum cryptography", Phys. Rev. A, vol. 59, May 1999, pp. 3301–3319, DOI 10.1103/PhysRevA.59.3301

[46] N. Gisin, S. Fasel, B. Kraus, H. Zbinden, and G. Ribordy, "Trojan-horse attacks on quantum-key-distribution systems", Phys. Rev. A, vol. 73, February 2006, p. 022320, DOI 10.1103/PhysRevA.73.022320

[47] C. Elliott, D. Pearson, and G. Troxel, "Quantum cryptography in practice", Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New York, NY, USA, August 25, 2003, pp. 227–238, DOI 10.1145/863955.863982

[48] U. Vazirani and T. Vidick, "Fully device independent quantum key distribution", Commun. ACM, vol. 62, April 2019, p. 133, DOI 10.1145/3310974

[49] R. Schwonnek, K. T. Goh, I. W. Primaatmaja, E. Y.-Z. Tan, R. Wolf, V. Scarani, and C. C.-W. Lim, "Device-independent quantum key distribution with random key basis", Nature Communications, vol. 12, May 2021, p. 2880, DOI 10.1038/s41467-021-23147-3

[50] H.-K. Lo, M. Curty, and B. Qi, "Measurement-device-independent quantum key distribution", Phys. Rev. Lett., vol. 108, March 2012, p. 130503, DOI 10.1103/PhysRevLett.108.130503

[51] C. H. Bennett and G. Brassard, "Quantum cryptography: Public key distribution and coin tossing", Theoretical Computer Science, vol. 560, May 2014, pp. 7–11, DOI 10.1016/j.tcs.2014.05.025. Theoretical Aspects of Quantum Cryptography - celebrating 30 years of BB84

[52] H. Bechmann-Pasquinucci and N. Gisin, "Incoherent and coherent eavesdropping in the six-state protocol of quantum cryptography", Phys. Rev. A, vol. 59, June 1999, pp. 4238–4248, DOI 10.1103/PhysRevA.59.4238

[53] H.-K. Lo, X. Ma, and K. Chen, "Decoy state quantum key distribution", Phys. Rev. Lett., vol. 94, June 2005, p. 230504, DOI 10.1103/PhysRevLett.94.230504

[54] X. Ma, B. Qi, Y. Zhao, and H.-K. Lo, "Practical decoy state for quantum key distribution", Phys. Rev. A, vol. 72, July 2005, p. 012326, DOI 10.1103/PhysRevA.72.012326

[55] V. Scarani, A. Acín, G. Ribordy, and N. Gisin, "Quantum cryptography protocols robust against photon number splitting attacks for weak laser pulse implementations", Phys. Rev. Lett., vol. 92, February 2004, p. 057901, DOI 10.1103/PhysRevLett.92.057901

[56] S. Ali, S. Mohammed, M. S. H. Chowdhury, and A. A. Hasan, "Practical sarg04 quantum key distribution", Optical and Quantum Electronics, vol. 44, September 2012, pp. 471–482, DOI 10.1007/s11082-012-9571-2

[57] C. H. Bennett, "Quantum cryptography using any two nonorthogonal states", Phys. Rev. Lett., vol. 68, May 1992, pp. 3121–3124, DOI 10.1103/PhysRevLett.68.3121

[58] A. Chefles, "Quantum state discrimination", Contemporary Physics, vol. 41, November 2000, pp. 401–424, DOI 10.1080/00107510010002599

[59] S. M. Barnett and S. Croke, "Quantum state discrimination", Adv. Opt. Photon., vol. 1, April 2009, pp. 238–278, DOI 10.1364/AOP.1.000238

[60] A. K. Ekert, "Quantum cryptography based on bell's theorem", Phys. Rev. Lett., vol. 67, August 1991, pp. 661–663, DOI 10.1103/PhysRevLett.67.661

[61] J. F. Clauser, M. A. Horne, A. Shimony, and R. A. Holt, "Proposed experiment to test local hidden-variable theories", Phys. Rev. Lett., vol. 23, October 1969, pp. 880–884, DOI 10.1103/PhysRevLett.23.880

[62] C. H. Bennett, G. Brassard, and N. D. Mermin, "Quantum cryptography without bell's theorem", Phys. Rev. Lett., vol. 68, February 1992, pp. 557–559, DOI 10.1103/PhysRevLett.68.557

[63] R. Satoh, M. Hajdusek, N. Benchasattabuse, S. Nagayama, K. Teramoto, T. Matsuo, S. A. Metwalli, T. Satoh, S. Suzuki, and R. V. Meter, "Quisp: a quantum internet simulation package", December 2021

[64] A. Varga, "Omnet++", Modeling and Tools for Network Simulation (J. G. Klaus Wehrle, Mesut Günes, ed.), pp. 35–39, Springer, Berlin, Heidelberg, 2010, DOI 10.1007/978-3-642-12331-3

[65] T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J. de Oliveira Filho, M. Papendrecht, J. Rabbie, F. Rozpedek, M. Skrzypczyk, L. Wubben, W. de Jong, D. Podareanu, A. Torres-Knoop, D. Elkouss, and S. Wehner, "Netsquid, a network simulator for quantum information using discrete events", Communications Physics, vol. 4, July 2021, DOI 10.1038/s42005-021-00647-8

[66] A. Dahlberg and S. Wehner, "Simulaqron-a simulator for developing quantum internet software", Quantum Science and Technology, vol. 4, September 2018, p. 015001, DOI 10.1088/2058-9565/aad56e

[67] J. Johansson, P. Nation, and F. Nori, "Qutip 2: A python framework for the dynamics of open quantum systems", Computer Physics Communications, vol. 184, April 2013, pp. 1234–1240, DOI 10.1016/j.cpc.2012.11.019

[68] I. Pedone, A. Atzeni, D. Canavese, and A. Lioy, "Toward a complete software stack to integrate quantum key distribution in a cloud environment", IEEE Access, vol. 9, 2021, pp. 115270–115291, DOI 10.1109/ACCESS.2021.3102313

[69] Qiskit, `https://www.qiskit.org/`

[70] Kubernetes, `https://www.kubernetes.io/`

[71] RabbitMQ, `https://www.rabbitmq.com/`

Appendix A

User's Manual

This appendix shows all the different steps needed to successfully install and run the proposed work.

A.1 Installing K3s

The first step to correctly run the QKD Simulator consists of installing any Kubernetes distribution. K3s[1] is the suggested distribution to install since it is very lightweight and versatile and it was also the one used during the development of the simulator. The easiest way to install K3s consists of issuing the following command inside of the Linux terminal:

```
curl -sfL https://get.k3s.io | sh -
```

It is possible to vary the installation configuration through the use of different environment variables listed on K3s documentation, but the default installation will suffice as well.

A.2 Installing Go

The next step to be able to have a working instance of the QKD Simulator consists of installing the Go[2] programming language. As listed by the documentation, in order to install Go, the first thing to do is download the archive containing its installer. Once downloaded, the archive will need to be extracted into `/usr/local` thus creating a tree in `/usr/local/go`. Next, Go will need to be added to the PATH environment variable and, to have a system-wide installation, it will be sufficient to add to `/etc/profile` file the following line:

```
export PATH=$PATH:/usr/local/go/bin
```

Now, the Go installation process is over, and to verify that everything was successful it is possible to check the installation by issuing the following command in the terminal:

```
go version
```

If everything went according to plan, the command should print out the installed version of Go.

[1] https://k3s.io/

[2] https://go.dev/

A.3 Simulator's Available APIs

The following table shows a list of all of the different REST APIs offered by the QKD Simulator in order to interact with it:

Web Interface		
Method Name	**URL**	**Access Method**
netTopology	http://{Simulator URL}/netTopology	POST
distributeKeys	http://{Simulator URL}/distributeKeys	POST
startKeyExchange	http://{Simulator URL}/startKeyExchange	POST
begin	http://{Simulator URL}/begin	POST
end	http://{Simulator URL}/end	POST

Table A.1

/netTopology is needed to notify the Manager of the QKD Simulator about the new network topology to be modelled so that it will correctly be able to interact with it. Furthermore, it also provides the user with the correct representation of the file to be used to deploy the specified topology on Kubernetes.

/distributeKeys is used to provide the different nodes of the modelled topology with the appropriate keys to use if the specified authentication method is selected for a QKD simulation.

/begin allows the user to choose two nodes of the topology he wants to start one or more QKD simulations between.

/startKeyExchange is needed to start one or more QKD simulations between two endpoints of the topology. Thanks to this API, the user can specify the configuration parameters to use and the number of simulations to carry out between the provided endpoints.

/end performs the appropriate cleanup operations once a user is done with the simulations run between two specific nodes of the topology.

All of these different REST APIs will be fundamental during the usage of the QKD Simulator.

A.4 Simulator Deployment

Now that all of the needed technologies to run the simulator have been successfully installed, it is time to deploy it before actually being able to use it.

The first thing to do is clone the GitHub Repository where the QKD Simulator project resides. This can easily be done by using git and issuing the following command in the terminal:

```
git clone https://github.com/ignaziopedone/QuaSi.git
```

As an alternative, it will be sufficient to download the ZIP file containing everything directly from https://github.com/ignaziopedone/QuaSi

Once downloaded the archive and extracted the QKD Simulator project, the next step is to set up and run the two different and custom Kubernetes Operators inside of the newly installed K3s Kubernetes cluster. These Operators will allow to fully deploy the QKD Simulator inside of the K3s cluster by using just two different custom YAML Configuration Files.

After having opened a terminal window inside the QKD Simulator project folder we will need to move inside the folder containing the custom Kubernetes Operators and run the following commands:

- `make manifests`;
- `make generate`;

- `make install`.

These commands are needed to let the K3s Kubernetes cluster know the Operators CRDs, i.e. structure of the custom YAML configuration files to be used for the deployment. Finally, to start the Operators it will be sufficient to build and run them through the following commands:

- `make build`;
- `make run`.

Now that the custom Kubernetes Operators are running, it is time to write the two custom YAML configuration files to be used for deployment. The first YAML file needed is the one that will allow the user to deploy all of the QKD Simulator's modules except the network topology to model. The following is an example of such YAML file:

```
apiVersion: qkdsim.s276624.qkdsim.dev/v1
kind: QKDSimulator
metadata:
       name: qkdsimulator-sample
spec:
       rabbit-host: rabbittmq
       rabbit-port: 5672
       manager-host: sim-manager
       manager-rest-port: 6969
       manager-gui-port: 6997
       trng: 0
       pnum: 8
```

Listing A.1: Example Of QKDSimulator Custom YAML File

The structure of the file needs to be as the one shown above, but the user can easily edit the configuration of the QKD Simulator by editing the values of the fields specified under the *spec* part of the file. The following table shows the different editable fields and what they represent:

QKD Simulator Configuration Fields	
Field Name	**Decsription**
rabbit-host	Name of the Kubernetes component running RabbitMQ
rabbit-port	Port Number for the AMQP Protocol exposed by the Kubernetes component running RabbitMQ
manager-host	Name of the Kubernetes component running the Simulator Manager
manager-rest-port	Port Number exposed by the Kubernetes component running the Simulator Manager to reach the Flask Web Server
manager-gui-port	Port number exposed by the Kubernetes component running the Simulator Manager to access Jupyter Notebook
trng	Boolean numerical parameter to choose to use a true quantum random number generator in the simulations or not
pnum	Numerical parameter expressing the amount of parallelism desired for each topology node and link

Table A.2: QKD Simulator Configuration Fields

Finally, one more field value of the file that the user can easily edit is the *name* under the *metadata* part of the file since it just represents the name the CRD symbolized by the file will have inside of the Kubernetes cluster.

Once the custom YAML file is ready, it will be sufficient to feed it to the K3s Kubernetes cluster through the following command:

```
k3s kubectl apply -f NameOfSimulatorCustomYamlFile.yaml
```

In response to this command, the custom Kubernetes Operators that were started before and are still running will react and spawn the requested QKDSimulator custom resource and all of the different modules of the QKD Simulator associated with it, i.e. all of the modules except the network topology to model. In particular, the Kubernetes component that will be spawned inside of the cluster as a result of the previous operation, are:

- Ingress Rule component for the simulator;
- ConfigMap component for the topology nodes;
- ConfigMap component for the topology links;
- Deployment component for the Simulator Manager module;
- StatefulSet component for the RabbitMQ module;
- Service component associated to the Simulator Manager Deployment;
- Service component associated to the RabbitMQ StatefulSet.

It will possible to check the status of the different components deployed by using the appropriate `kubectl` commands:

- `k3s kubectl get Deployment`;
- `k3s kubectl get StatefulSet`;
- `k3s kubectl get Service`;
- `k3s kubectl get Ingress`.

It is important to note that the Service component for RabbitMQ is of kind *LoadBalancer* thus making it an external service to which the Kubernetes cluster associates a public IP address as well. As a result, the management interface exposed by RabbitMQ can be easily accessed on its default port (15672) to check everything is behaving as expected. Besides, this also allows any entity external to the K3s Kubernetes cluster to directly interact with the RabbitMQ installation running inside of it through the AMQP protocol on the configured port in the custom YAML file.

A.5 Topology Deployment

Once sure that all of the previous resources were deployed correctly and are up and running, it is possible to work on the second custom YAML configuration file to deploy the desired network topology. The following is an example of such YAML file:

```
apiVersion: qkdsim.s276624.qkdsim.dev/v1
kind: NetTopology
metadata:
       name: nettopology-sample
spec:
       adjacency:
              - - id: node2
                - id: node3
                - id: node4
              - - id: node1
                - id: node4
              - - id: node1
                - id: node4
              - - id: node1
```

```
          - id: node2
          - id: node3
nodes:
        - id: node1
        - id: node2
        - id: node3
        - id: node4
```

Listing A.2: Example Of NetTopology Custom YAML File

The structure of the file needs to be as the one shown above, but the user can easily edit the network he wants the QKD Simulator to model by editing the values of the fields specified under the *spec* part of the file. The following table shows the different editable fields and what they represent:

NetTopology Representation	
Field Name	**Decsription**
adjacency	Adjacency list of the undirected Graph representation for the network topology to model
nodes	List of the nodes of the network topology to model

Table A.3: QKD Simulator Network Topology Description

One more field value of the file that the user can easily edit is the *name* under the *metadata* part of the file since it just represents the name the CRD symbolized by the file will have inside of the Kubernetes cluster.

Finally, it is important to note that the name of the nodes used inside the custom YAML configuration file can be changed to anything the user desires since they simply represent placeholders. Once deployed inside the cluster the name that each node of the topology will have is *endpoint-index_of_node_in_nodes_list*. The same will happen for the links of the modelled topology, each of them will have the following name: *ccindex1-index2* where the two indexes specified will be the ones of the nodes that specific link connects.

Once the custom YAML file is ready, it will be sufficient to feed it to the K3s Kubernetes cluster through the following command:

```
k3s kubectl apply -f NameOfTopologyCustomYamlFile.yaml
```

In response to this command, the custom Kubernetes Operators that were started before and are still running will react and spawn the requested NetTopology custom resource and all of the different modules of the described network topology associated with it. In particular, the Kubernetes components that will be spawned inside of the cluster as a result of the previous operation, are:

- A Deployment component for each node of the modelled topology;
- A Deployment component for each link of the modelled topology.

It will possible to check the status of the different components deployed by using the appropriate `kubectl` command:

```
k3s kubectl get Deployment
```

Furthermore, it will also be possible to check the successful deployment of the network topology onto the K3s Kubernetes cluster through the management interface exposed by RabbitMQ.

In order to access such interface, the first thing to do will be using the following command to check the public IP address the K3s Kubernetes cluster associated to the Service component for RabbitMQ:

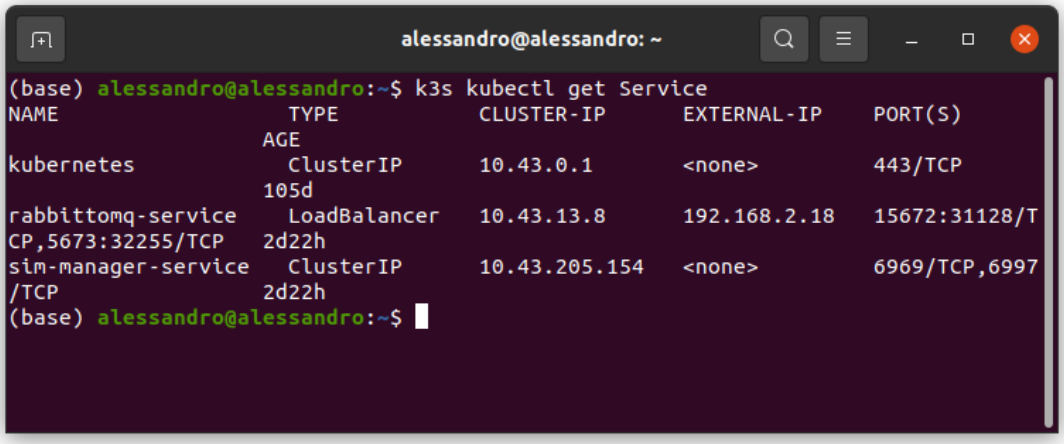


Figure A.1: Check External IP Associated to RabbitMQ Service Component

Next, it will be sufficient to connect to the external IP address on the default port RabbitMQ uses for exposing the management interface (15672) and use the default credentials to access:

- username: guest;
- password: guest.

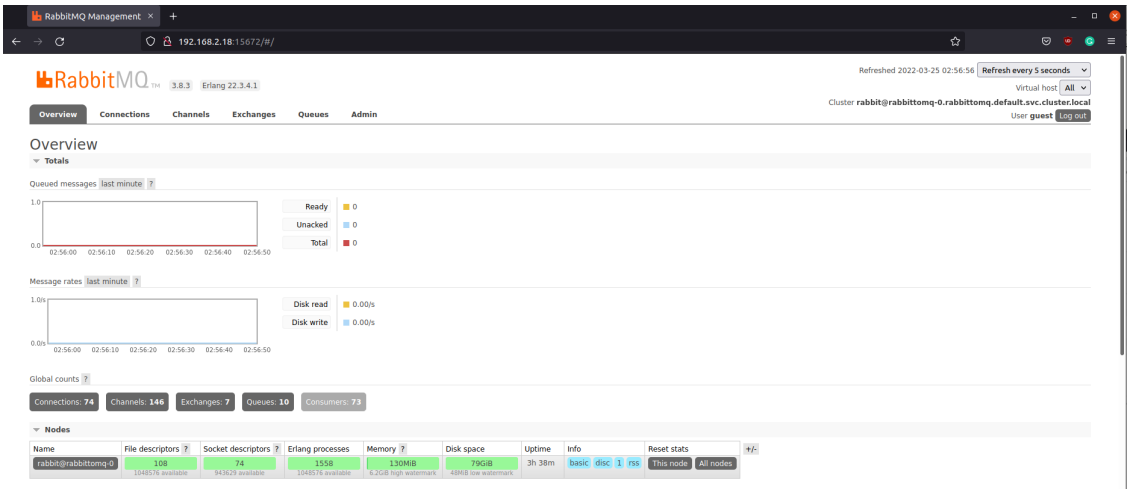


Figure A.2: RabbitMQ Management Interface

Thanks to this interface, the user will be able to verify the successful connection of every entity of the modelled network topology to RabbitMQ.

A.6 QKD Simulator Usage

A.6.1 Init Phase

Before actually being able to start one or more QKD Simulation, an initialization phase to properly set up the QKD Simulator has to be carried out.

The first thing to do consists of notifying the Simulator Manager about the network topology just modelled and deployed. To do so, it will be sufficient to leverage the appropriate REST API exposed by the Simulator Manager. In particular, the following POST HTTP request has to be sent:

```
http://qkdsim.com/netTopology
```

The appropriate representation of the modelled topology has to be sent as the body of the request. The following is an example of the JSON serialized representation of the modelled topology to be sent to the Simulator Manager:

```
{
        "directed": false,
        "multigraph": false,
        "graph": [],
        "nodes": [
                {"id": "node1"},
                {"id": "node2"},
                {"id": "node3"},
                {"id": "node4"}
        ],
        "adjacency": [
                [
                        {"id": "node2"},
                        {"id": "node3"},
                        {"id": "node4"}
                ],
                [
                        {"id": "node1"},
                        {"id": "node4"}
                ],
                [
                        {"id": "node1"},
                        {"id": "node4"}
                ],
                [
                        {"id": "node1"},
                        {"id": "node2"},
                        {"id": "node3"}
                ]
        ]
}
```

Listing A.3: Example of modelled network topology representation

It is important to note that such representation can easily be obtained by leveraging the *NetworkX* Python library as well:

```
import networkx
from networkx.readwrite import json_graph
import json

graph = networkx.Graph()
graph.add_node("node1")
graph.add_node("node2")
graph.add_node("node3")
graph.add_node("node4")
graph.add_edge("node1","node2")
graph.add_edge("node1","node3")
graph.add_edge("node1","node4")
```

```
graph.add_edge("node2","node4")
graph.add_edge("node3","node4")
topology_rep = json.dumps(json_graph.adjacency_data(graph))
```

Listing A.4: Example of modelled network topology through NetworkX Python Library

As a response to the request, the user will receive the JSON serialized representation of the custom Kubernetes YAML file to feed the cluster to model the specified topology. This happens because this REST API can also be used as a utility function before deploying the topology to be sure to obtain the correct syntax of the custom YAML file to feed the cluster

Once notified the Simulator Manager about the modelled network topology, the next step consists of properly setting up the nodes and links of the topology with the proper parameters in case the user wants to select a specific authentication method for the classical channel communication during a QKD simulation. To do so, it will be sufficient to use the appropriate REST API exposed by the Simulator Manager. In particular, the following POST HTTP request will need to be sent:

`http://qkdsim.com/distributeKeys`

The body of the request will contain the desired authentication method for which the user wants to configure the nodes and link of the topology.

`{ \authentication": \sphincs" }`

As a consequence of this request, the Simulator Manager will send the appropriate messages of type *Keys* containing the needed parameters for the selected authentication method to all of the different nodes and links of the modelled topology. Now, if during any QKD simulation the user will want to use such authentication method, the entities of the topology involved in it will have the appropriate parameters to apply such a method.

As a response to this request, the user will receive the outcome of the required operation.

It is important to note that the only currently supported authentication method is *SPHINCS+*, a post-quantum cryptography hash-based signature scheme.

Now it will be possible to carry out QKD Simulations on the network topology modelled by the simulator.

A.7 Carrying Out QKD Simulations

To carry out one or more QKD simulations on the modelled network topology, the first thing to do will be choosing two connected nodes to start one or more simulations between. This goal can be easily achieved by using the appropriate REST API exposed by the Simulator Manager and issuing the following POST HTTP request:

`http://qkdsim.com/begin`

The body of the request will need to contain the two endpoints of the modelled topology the user wants to carry out QKD simulations between:

`{ \SRC": \endpoint-1", \DST": \endpoint-3" }`

As a consequence of this request, the Simulator Manager will check if the two specified nodes of the topology are connected directly by a link and if this is the case will create and store a unique UUID associated with the pair of endpoints specified. The user will receive as a response to the request the outcome of the operation required.

Once successfully created a UUID associated with the specified pair of endpoints specified, it will be possible to start one or more QKD simulations between them to generate one or more keys. To do so, it will be sufficient to use the appropriate REST API exposed by the Simulator Manager and issue the following POST HTTP request:

`http://qkdsim.com/startKeyExchange`

The body of this request will need to contain all of the configuration parameters for the QKD simulation, the name of the endpoints to carry the simulation between and the one of the communication channel connecting them. The following is an example of body for such request:

```
{
        "protocol": "bb84",
        "key_length": 4096,
        "chunk_size": 5,
        "authentication": "sphincs",
        "com_channel": "cc1-3",
        "interceptAndResend": 0,
        "manInTheMiddle": 0,
        "source": "endpoint-1",
        "destination": "endpoint-3",
        "key_number": 25
}
```

Listing A.5: Example of body for the startKeyExchange REST API

It is important to note that the following parameters are mandatory:

- `protocol`;
- `key_length`;
- `source`;
- `destination`;
- `com_channel`;
- `key_number`.

On the other hand, a default value will be used for the remaining ones if omitted. In particular:

- `chunk_size` default value is 5;
- `authentication` default value is "none";
- `interceptAndResend` dafult value is 0 meaning false;
- `manInTheMiddle` default value is 0 meaning false.

As a response to this request, if everything was successful, the user will receive the UUID associated with the endpoints of the topology specified which will also be the name of the RabbitMQ queue the user will be able to find the results of the requested QKD simulations. On the other hand, if any errors occurred the response will be used to notify the user.

It is important to note that the results of the requested QKD simulations will be put on two different queues by RabbitMQ. The name of the queues will be:

- *"UUID"*, the one associated to the pair of endpoints the QKD simulations were carried between;
- *"UUID"_R*.

A copy of the result of each simulation will be put on both of the queues to easily allow two higher-level entities wanting to use the simulator to each consume a copy of the outcomes of the simulations.

Finally, once the user is satisfied with the amount of QKD simulations carried out between two nodes of the topology, it will be sufficient to use the appropriate REST API provided by the

Simulator Manager to clean up. In particular, the following POST HTTP request will need to be sent:

```
http://qkdsim.com/end
```

The body of this request will need to contain the UUID associated with the two specific topology nodes the user wants to stop carrying QKD simulations between: The following is an example of such a body:

```
{ \ID": \b5bbac53-dd63-4580-8a08-6e0243f95a0a" }
```

If the provided ID exists between the ones stored by the Simulator Manager, i.e. it was generated by a successful corresponding *begin* operation, the provided ID will be deleted and the user won't be able to carry out more QKD simulations between the corresponding nodes of the topology unless he performs a new *begin* operation. As a response, the user will receive the outcome of the requested operation.

A.8 Editing Configurations

A.8.1 Changing The Modelled Network Topology

To modify the modelled network topology, it will be sufficient to delete the corresponding custom resource deployed to the K3s Kubernetes cluster with the corresponding YAML file. In particular, the following command will need to be issued:

```
k3s kubectl delete NetTopology nettopologyCRDname
```

The *nettopologyCRDname* will correspond to the value of the *name* field under the *metadata* section of the network topology custom YAML configuration file used for its deployment.

Once removed the previous network topology, the user will just need to edit the previously used custom YAML file to model the new network topology he wants to model and simply feed it again to the K3s Kubernetes cluster by issuing the same command:

```
k3s kubectl apply -f NameOfTopologyCustomYamlFile.yaml
```

It is important to note that after the deployment of the new topology, before being able to carry out any QKD simulation again, he will need to repeat the steps of the *Init* phase explained in section A.6.1 of this appendix.

A.8.2 Changing Simulator Configuration

To edit the configuration parameters of the simulator, it will be sufficient to delete the corresponding CRD deployed to the K3s Kubernetes cluster by issuing the following command:

```
k3s kubectl delete QKDSimulator qkdsimulatorCRDname
```

The *qkdsimulatorCRDname* will match the value of the *name* field under the *metadata* section of the QKD simulator custom YAML configuration file used for its deployment.

It is important to note, that it is strongly suggested to delete the deployed network topology as well before changing the simulator's configuration to avoid any possible issues.

Once removed the corresponding custom resource, it will be sufficient to edit the previously used custom YAML file to re-deploy the simulator with the updated settings and feed it again to the cluster by using the same command:

```
k3s kubectl apply -f NameOfSimulatorCustomYamlFile.yaml
```

After the deployment of the simulator with the updated settings, the user will be able to repeat the deployment of the desired network topology and carry out all of the previously described operations needed to perform QKD simulations.

Appendix B

Developer's Manual

This appendix describes the implementation of the custom Kubernetes Operators needed for the deployment of the QKD Simulator to the Kubernetes cluster and it describes the implementation of its different modules.

B.1 Custom Kubernetes Operators Implementation

B.1.1 Kubebuilder

To implement the two custom Kubernetes Operators that allow the deployment of the simulator on the cluster by using just two custom YAML configuration files, the *Kubebuilder* Go framework has been used.

The first thing to do was follow the documentation[1] to properly install the framework and create the project for the custom Kubernetes Operators. When creating a new project, the Kubebuilder framework already provides its entire scaffolding needed to have a Kubernetes Operator up and running. The developer will just need to modify the appropriate files to implement his custom Custom Resource Definition (CRD) and Custom Controller Logic, i.e. the two parts composing a custom Kubernetes Operator.

In particular, after having cloned the GitHub repository[2] containing the implementation of the simulator, the Kubebuilder project implementing the two Custom Kubernetes Operators for deploying the QKD Simulator, is located inside the *KubernetesOperators* folder.

B.1.2 QKDSimulator CRD Implementation

The CRD of kind *QKDSimulator* was developed to allow the user to select the initial configuration parameters for the deployment of the simulator's modules except for the network topology to model. In particular, its implementation is contained inside the following file:

`KubernetesOperators/api/v1/qkdsimulator_types.go`

```
type QKDSimulatorSpec struct {
	// Important: Run "make" to regenerate code after modifying this file

	RabbitMQ_Host string `json:"rabbit-host,omitempty"`
	RabbitMQ_Port int32 `json:"rabbit-port,omitempty"`
```

[1] https://book.kubebuilder.io/quick-start.html

[2] https://github.com/ignaziopedone/QuaSi

```
        Manager_Host string `json:"manager-host,omitempty"`
        Manager_Rest_Port int32 `json:"manager-rest-port,omitempty"`
        Manager_Gui_Port int32 `json:"manager-gui-port,omitempty"`
        Trng int32 `json:"trng,omitempty"`
        Pnum int32 `json:"pnum,omitempty"`
}

// QKDSimulatorStatus defines the observed state of QKDSimulator
type QKDSimulatorStatus struct {
        Status string `json:"status,omitempty"`
}

//+kubebuilder:object:root=true
//+kubebuilder:subresource:status
//+kubebuilder:printcolumn:name="Status",type=string,JSONPath=`.status.status`

// QKDSimulator is the Schema for the qkdsimulators API
type QKDSimulator struct {
        metav1.TypeMeta `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`

        Spec QKDSimulatorSpec `json:"spec,omitempty"`
        Status QKDSimulatorStatus `json:"status,omitempty"`
}
```

Listing B.1: qkdsimulator_types.go (lines 26-60)

The `QKDSimulator struct` defines the structure of the custom YAML configuration file representing the CRD. In this case, the described structure matches the one of any Kubernetes YAML configuration file. In particular, the *Spec* part implemented by the `QKDSimulatorSpec struct` is the one expressing the desired state of the CRD inside of the Kubernetes cluster. Therefore, the `QKDSimulatorSpec struct` is where the different fields of the custom YAML configuration file to feed Kubernetes for deploying the CRD are defined.

As we can see, each of the fields of the QKDSimulator custom YAML configuration file has a corresponding variable defined inside of the `QKDSimulatorSpec struct` and there is a string expressed between backticks. For each variable of the struct, the string expresses the exact name of the corresponding field inside the *spec* section of the custom YAML configuration file and if `omitempty` is specified, this means that a specific field isn't mandatory inside of the YAML file and can also be omitted.

This is important because once built, Kubebuilder will also create the *json-schema* to use for validating the custom YAML file corresponding to the CRD. This schema will be provided to Kubernetes for validation when the project is run.

B.1.3 NetTopology CRD Implementation

The CRD of kind *NetTopology* was developed to allow the user to describe and deploy the desired network topology to the Kubernetes cluster. To be more specific, its implementation is contained inside the following file:

`KubernetesOperators/api/v1/nettopology_types.go`

```
//Custom Node struct
type Node struct {
    Id string `json:"id"`
}

type NetTopologySpec struct {
```

```
    // Important: Run "make" to regenerate code after modifying this file

    Nodes []Node `json:"nodes"`
    Adjacency [][]Node `json:"adjacency"`

}

type NetTopologyStatus struct {
    Status string `json:"status,omitempty"`
}

//+kubebuilder:object:root=true
//+kubebuilder:subresource:status
//+kubebuilder:printcolumn:name="Status",type=string,JSONPath=`.status.status`

type NetTopology struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec NetTopologySpec `json:"spec,omitempty"`
    Status NetTopologyStatus `json:"status,omitempty"`
}
```

Listing B.2: nettopology_types.go (lines 26-60)

The `NetTopology struct` defines the structure of the custom YAML configuration file representing the CRD. In this case, as well, the described structure matches the one of any Kubernetes YAML configuration file. In particular, the *Spec* part implemented by the `NetTopologySpec struct` is the one expressing the desired state of the CRD inside of the Kubernetes cluster. Therefore, the `NetTopologySpec struct` is where the different fields of the custom YAML configuration file to feed Kubernetes for deploying the CRD are defined.

As we can see, each of the fields of the NetTopology custom YAML configuration file has a corresponding variable defined inside of the `NetTopologySpec struct`. To be more specific, the `Node` custom Go structure represents the single node of the topology and inside of the `NetTopologySpec` only two specific fields have been inserted to fully describe the desired network topology to model:

- Nodes: an array (slice in Go) of type Node as the custom structure;
- Adjacency: a matrix (2D slice in Go) of type Node as the custom structure.

The `Nodes` field represents the list of nodes of the topology to model while the `Adjacency` variable is used to represent the links of the topology by expressing the adjacency list representation of the graph corresponding to the network topology desired.

For each field, there is a string expressed between backticks needed to express the exact name of the corresponding field inside the *spec* section of the custom YAML configuration file. It is important to note that this time `omitempty` isn't specified for any of the fields to symbolize the fact that they are mandatory since they are essential to express the structure of the desired topology to model. As a result, if any of these two fields is not specified inside the custom YAML file to feed the Kubernetes cluster for deploying the `NetTopology` CRD to the cluster, the validation of the file will fail and no topology will be deployed.

B.1.4 QKDSimulator Custom Controller Logic

As we know, a Kubernetes operator is composed of two different parts:

- a Custom Resource Definition (CRD);

- a Custom Controller to manage the lifecycle of the corresponding CRD inside the Kubernetes cluster.

As far as the QKDSimulator CRD is concerned, its corresponding Custom Controller can be found in the following file:

`KubernetesOperators/controllers/qkdsimulator_controller.go`

To be more specific, the logic of the Custom Controller is implemented inside the `Reconcile` function that the Kubernetes cluster will run every time the actual status of the corresponding CRD doesn't match the desired one expressed in the custom YAML configuration file.

```
func (r *QKDSimulatorReconciler) Reconcile(ctx context.Context, req
    ctrl.Request) (ctrl.Result, error) {
    logger := log.FromContext(ctx)

    // Logger Set Up
    logger.Info("Starting QKDSimulator Controller !!", "name",
        req.NamespacedName)

    //Get CRD data

    var qkdsimulator qkdsimv1.QKDSimulator

    if err := r.Get(ctx, req.NamespacedName, &qkdsimulator); err != nil {
        logger.Error(err, "name")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }
```

Listing B.3: qkdsimulator_controller.go (lines 52-65)

As we can see, the first thing to be done is to retrieve the data contained in the custom YAML configuration file representing the CRD when it is fed to the Kubernetes cluster and manage eventual errors during such operation. If everything was successful, the retrieved data needs to be used to spawn the Kubernetes components for all of the simulator's modules except the network topology.

```
rabbitmqstfs, err := r.desiredRabbitMQ(qkdsimulator,
    qkdsimulator.Spec.RabbitMQ_Host, qkdsimulator.Spec.RabbitMQ_Port)
if err != nil {
    return ctrl.Result{}, err
}

applyOpts := []client.PatchOption{client.ForceOwnership,
    client.FieldOwner("QKDSimulator")}
err = r.Patch(ctx, &rabbitmqstfs, client.Apply, applyOpts...)
if err != nil {
    return ctrl.Result{}, err
}
```

Listing B.4: qkdsimulator_controller.go (lines 69-80)

In the code listed above, we can see how the deployment of the StatefulSet Kubernetes component for RabbitMQ was implemented. To be more specific, an appropriate function is used to create a properly configured object representing the Kubernetes component to be deployed to the cluster. After that, the `Patch` function exposed by the Go client for the Kubernetes API is used to perform the actual deployment of the newly defined Kubernetes component to the cluster. Besides, thanks to this function the ownership of the newly created component is assigned to the CRD the lifecycle of which the Custom Controller manages. As a result, whenever the CRD is deleted from the Kubernetes cluster, all of the other components deployed in response to its

deployment are deleted as well since their ownership was assigned to the CRD by the Custom Controller. It is important to note that the other Kubernetes components the deployment of which is handled by the QKDSimulator Custom Controller are:

- a Service for RabbitMQ;
- a ConfigMap for the topology nodes configuration parameters;
- a ConfigMap for the topology links configuration parameters;
- a Deployment for the Simulator Manager;
- a Service for the Simulator Manager;
- an Ingress Rule for the Simulator Manager.

However, the implementation logic for their deployment essentially follows the same one used for the RabbitMQ StatefulSet except for some minor modifications.

B.1.5 NetTopology Custom Controller Logic

As far as the Custom Controller associated with the NetTopology CRD is concerned, the implementation of its custom logic can be found in the `Reconcile` function inside of the following file:

`KubernetesOperators/controllers/nettopology_controller.go`

```
//Get CRD data
var nettopology qkdsimv1.NetTopology

if err := r.Get(ctx, req.NamespacedName, &nettopology); err != nil {
    logger.Error(err, "Error fetching the resource")
    return ctrl.Result{}, client.IgnoreNotFound(err)
}
```

Listing B.5: nettopology_controller.go (lines 60-67)

As usual, the first thing to be done consists of retrieving the data contained in the custom YAML configuration file representing the CRD when it is fed to the Kubernetes cluster and managing eventual errors during such operation. If everything was successful, the retrieved data needs to be used to spawn the Kubernetes components for all of the simulator's modules except the network topology.

```
//Fetch any already existing topology and delete it before deploying the new
    one
var topologies qkdsimv1.NetTopologyList
if err := r.List(ctx, &topologies); err != nil {
    logger.Error(err, "Error is when listing")
    return ctrl.Result{}, client.IgnoreNotFound(err)
}

if len(topologies.Items) > 1 {
    for _, top := range topologies.Items {
        if nettopology.Name != top.Name {
            if err := r.Delete(ctx, &top); err != nil {
                return ctrl.Result{}, client.IgnoreNotFound(err)
            }
        }
    }
}
```

```
//Deploy newly provided network topology
//Configure Patch Options
applyOpts := []client.PatchOption{client.ForceOwnership,
    client.FieldOwner("NetTopology")}
//For each node i go and deploy an endpoint
for index := range nettopology.Spec.Nodes {
    endpdepl, err := r.desiredEndpointDeployment(nettopology, int32(index))
    if err != nil {
        return ctrl.Result{}, err
    }
    err = r.Patch(ctx, &endpdepl, client.Apply, applyOpts...)
    if err != nil {
        return ctrl.Result{}, err
    }
}

//Now we have to deploy the Toplogy's Communication Channels, one for each
    edge of the graph
for index, connections := range nettopology.Spec.Adjacency {
    for _, n := range connections {
        i := r.indexOf(n, nettopology.Spec.Nodes)
        if i > index {
            com_chandepl, err := r.desiredComChanDeployment(nettopology,
                int32(index), int32(i))
            if err != nil {
                return ctrl.Result{}, err
            }
            err = r.Patch(ctx, &com_chandepl, client.Apply, applyOpts...)
            if err != nil {
                return ctrl.Result{}, err
            }
        }
    }
}
```

Listing B.6: nettopology_controller.go (lines 73-122)

Since the network topology to model is assumed to be a resource that will be changed very often, in the logic of its Custom Controller a mechanism to delete any previously existing topology inside of the Kubernetes cluster before deploying the newly provided one is implemented.

Finally, by using the data describing the new topology to model fetched from the CRD at the beginning of the function, a Deployment Kubernetes component will be deployed for both each node of the described topology and each of its links as well. As far as the logic used to deploy these components, the same one described in the previous section of this appendix B.1.2 is used.

B.1.6 Helper Functions

Finally, it is important to note that all of the `desiredComponenttoDeploy` functions are custom functions the implementation of which can be found in the following file:

`KubernetesOperators/controllers/helpers.go`

```
func (r *QKDSimulatorReconciler) desiredConfigMap(qkds qkdsimv1.QKDSimulator,
    who_conf string, data map[string]string) (corev1.ConfigMap, error) {

    //Configure data map with parameters needed

    cmap := corev1.ConfigMap{
```

```
        TypeMeta: metav1.TypeMeta{APIVersion: "v1", Kind: "ConfigMap"},
        ObjectMeta: metav1.ObjectMeta{
            Name: who_conf + "-config",
            Namespace: "default",
        },
        Data: data,
    }

    if err := ctrl.SetControllerReference(&qkds, &cmap, r.Scheme); err != nil
        {
        return cmap, err
    }

    return cmap, nil
}
```

Listing B.7: helpers.go (lines 24-42)

As we can see, the function above has been created to be used whenever a new ConfigMap component needs to be configured and defined before deploying it to the Kubernetes cluster. The first thing the function does is create the appropriate Go object corresponding to the Kubernetes component in need to be deployed. Then, the newly created object is configured using the parameters provided to the function. In particular, it is important to note that the different attributes the object has, correspond exactly to the fields a regular YAML configuration file used to deploy the same Kubernetes component, would have. Finally, after associating the lifecycle of the modelled component to the one of the CRD provided in the parameters, the corresponding Go object is returned unless an error occurred.

It is important to note, that all of the remaining helper functions contained in the `helpers.go` file follow the exact same logic exposed by the previously described function. The major difference will just be the Go object representing the corresponding Kubernetes component to deploy to the cluster.

B.2 Simulator's Modules Implementation

Now that the description of the implementation of the custom Kubernetes Operators to deploy the different simulator modules has been completed, the current section will describe the implementation of the custom Docker Images implementing the different kinds of simulator modules. These will be run inside Docker containers by the simulator's Kubernetes components to implement the different modules.

B.2.1 Simulator Endpoint Implementation

As far as the implementation of the Simulator Endpoint module is concerned, this can be found in the following file:

`DockerImages/EndpointImage/Simulator.py`

```
def main():
    #Fetch Environment variables
    p_num = int(os.environ.get("PNUM",5))
    me = os.environ.get("ME","endpoint-default")
    rabbit_host = os.environ.get("RABBIT_HOST","rabbitmq")
    rabbit_port = os.environ.get("RABBIT_PORT",5672)

    #Pool set up
    pool = multiprocessing.Pool(p_num)
```

```
    #Shared Data Structures Set Up
    manager = multiprocessing.Manager()
    active_connections = manager.dict()
    l_actcon = manager.Lock()
    authN_data = manager.dict()
    l_authN = manager.Lock()

    results = []

    #Start the consumers
    for i in range(p_num):
        r = pool.apply_async(RMQ_Consumer,
                             args=(rabbit_host,rabbit_port,me,
                             active_connections,authN_data,l_actcon,l_authN))
        results.append(r)

    #Wait for processes to finish
    for r in results:
        r.wait()

    pool.close()
    pool.join()
```

Listing B.8: Simulator.py (lines 900-931)

As we can see, the first thing that has been done is fetching the configuration parameters for the topology endpoints. These parameters are the ones contained inside the ConfigMap Kubernetes component created for the nodes of the topology and they have been configured as environment variables during the deployment of the topology modules by the corresponding Custom Controller. After that, according to the configured value of parallelism, a `multiprocessing Pool` will be created and configured along with the shared data structures to keep track of the running QKD simulations involving the specific endpoint and the authentication parameters to use in case a specific authentication method is chosen to be used for the classical channel communication part of the simulation. Besides, the corresponding lock to handle concurrent accesses to the shared data structures are created as well.

Each process of the `multiprocessing Pool` will be running the `RMQ_Consumer` function.

```
#Wrapper Function For Set Up Of Each Process
def RMQ_Consumer(rhost,rport,me,a_con,authN_data,l_acon,l_authN):

    #Set Up
    simulator = Simulator(rhost,rport,me,a_con,authN_data,l_acon,l_authN)
    simulator.startListening()

    return
```

Listing B.9: Simulator.py (lines 891-898)

As we can see, the `RMQ_Consumer` function is just a wrapper, the real implementation of the Simulator Endpoint module is inside the `Simulator` class.

```
class Simulator():

active_connections = dict()
connection = None
rec_channel = None
send_channel = None
me = ""
```

```
authN_data = dict()
l_acon = None
l_authN = None


def __init__(self,rhost,rport,me,acon,authN_data,l_acon,l_authN):...

# RECEIVER SIMULATION FUNCTIONS

def getQuantumKey_R(self,qubits,key_length,alice_key,
                  alice_table,temp_alice_key):...

def compareBasis_R(self,table,signature,alice_key,
                  alice_table,temp_alice_key,metadata,destination):...

def verifyKey_R(self,bobKey,keySign,picked,pickedSign,
              alice_key,metadata,destination):...

# SOURCE SIMULATION FUNCTIONS

def generateQubits(self,chunk):...

def initSimParameters(self,params,key_length,chunk):...

def checkResponseAndSendBasis(self,response,params,auth):...

def compareBasisAndPrepareForKeyVer(self,msg,authenticationMethod,
                                  destination,params,key_length,chunk):...

def verifyAndSetSimResults(self,msg,authenticationMethod,
                         destination,params):...

# MESSAGE TYPE DISPATCHER

def type_switcher(self,argument):
    logging.info("Argument is " + str(argument))
    switcher = {
        "Keys": self.set_Sphincs_Keys,
        "Start": self.startSim,
        "sendRegister": self.sendRegister_H,
        "compareBasis": self.compareBasis_H,
        "verifyKey": self.verifyKey_H,
    }

    return switcher.get(argument,"Invalid Mssage Type Received")

# MESSAGE TYPE HANDLERS

# CLASSICAL CHANNEL AUTHENTICATION METHOD HANDLING FUNCTIONS
# SPHINCS+
def set_Sphincs_Keys(self,source,destination,metadata,msg):...

def startSim(self,source,destination,metadata,msg):...

def sendRegister_H(self,source,destination,metadata,msg):...

def compareBasis_H(self,source,destination,metadata,msg):...
```

```
def verifyKey_H(self,source,destination,metadata,msg):...

def startListening(self):...
```

Listing B.10: Simulator.py (Simulator Class)

As far as the class parameters are concerned, there are:

- `active_connections`, a dicitionary to keep track of the different QKD simulations the specific endpoint is involved in;
- `connection`, RabbitMQ connection variable;
- `send_channel` and `rec_channel`, RabbitMQ channels to send and consume messages from the appropriate queue;
- `me`, a variable to store the endpoint of the topology represented.
- `authN_data`, a dictionary to keep track of any eventually needed authentication parameters to use during simulations.
- `l_acon` and `l_authN`, two variables representing the locks for the corresponding shared data structures.

The `__init__` function of the class will be used to properly initialize the parameters of the class and create the connection and channels needed by every process of the pool to connect to RabbitMQ and start consuming messages on the corresponding endpoint's queue.

The `getQuantumKey_R`, `compareBasis_R` and `verifyKey_R` functions are the ones used to perform the appropriate BB84 simulation steps if the specific endpoint is acting as the receiver in the current point-to-point simulation. Depending on which step of the simulation has to be carried out, the appropriate function will be called by the message handling one.
Moreover, the `generateQubits`, `initSimParameters`, `checkResponseAndSendBasis`, `verifyAnd SetSimResults` and `compareBasisAndPrepareForKeyVer` functions are the ones used to perform the right BB84 simulation steps if the specific endpoint is acting as the receiver in the current point-to-point simulation. Depending on which step of the simulation has to be carried out, the appropriate function will be called.

The `type_switcher` function is used whenever a new message on the endpoint's queue is consumed to decide the appropriate handling function to call and manage the message accordingly. As we can see the different message handling functions for BB84 simulations are:

- `setSphincsKeys` function used to handle messages of type *Keys* save the keys to use for SPHINCS+ authentication during the simulation;
- `startSim` function used to handle messages of type *Start* to generate the initial parameters for the BB84 simulation if the endpoints is currently acting as source;
- `sendRegister_H` function used to handle messages of type *sendRegister* for BB84 simulation;
- `compareBasis_H` function used to handle messages of type *compareBasis* for BB84 simulation;
- `verifyKey_H` function used to handle messages of type *verifyKey* for BB84 simulation;

```
#Set up the channel to listen to message on queue
rec_ch = self.rec_channel

#Set up queue in case it doesn't exist
queueResult = rec_ch.queue_declare(queue=self.me,auto_delete=True)
rec_name = queueResult.method.queue
```

```
#Message Handler Function

def message_handler(ch,method,properties,body):

    #We assume the messages flowing on the queue will have the following
        predefined structure:
    # [type string, source string, destination string, metadata
        map[string][object], msg object]

    res = pickle.loads(body)

    type = res[0]
    source = res[1]
    destination = res[2]
    metadata = res[3]
    msg = res[4]

    #Now depending on the type, role fields of the message we will evaluate
        what to do:

    #Analyze messages of communication start from manager
    handler = self.type_switcher(type)
    handler(source,destination,metadata,msg)
    #Done processing message, acknowledge can be sent to RabbitMQ
    ch.basic_ack(delivery_tag=method.delivery_tag)


#Starting to consume messages
rec_ch.basic_consume(on_message_callback=message_handler, queue=rec_name)
rec_ch.start_consuming()

return
```

Listing B.11: Simulator.py (startListening function)

As we can see the `startListening` function is the one used to start consuming messages from the specific endpoint's queue. In particular, for handling each message received, the `message_handler` function will be called. For each message, this function will:

- deserialize it by using the *pickle* Python library;
- extract the different fields of the data model from the received message;
- evaluate the *type* of the message to retrieve the appropriate handling function to call;
- call the proper handling function;
- acknowledge the message to RabbitMQ after having managed it so that it will be erased from the endpoint's queue.

```
class BB84_Parameters:
    #Used both for receiver or source
    alice_table = np.array([])
    alice_key = ''
    #Used only for receiver
    temp_alice_key = ''
    #Used only for source
    picked = []
```

```
    verifyingKey = []
    start_sim = None
    start = time.time()
    includingVerifyLen = None
    qber = None
    return_flag = False

class Active_Connection:
    mess_id = None
    source = ""
    destination = ""
    params = None
```

Listing B.12: Simulator.py (BB84_Parameters and Active_Connections classes)

It is important to note that for each BB84 QKD simulation, an instance of the `BB84_Parameters` class will be created and used. Besides, the `Active_Connection` class is instantiated every time there is a new QKD simulation to keep track of. In particular the `params` variable will contain the instance of `BB84_Parameters` class for the specific simulation.

B.2.2 Communication Channel Implementation

As far as the implementation of the Communication Channel simulator's module is concerned, this can be found in the following file:

`DockerImages/CommunicationChannelImage/com_chan.py`

```
def RMQ_Consumer(rhost,rport,me,fendp,sendp):

    com_chan = Communication_Channel(rhost,rport,me,fendp,sendp)
    com_chan.startListening()

    return

def main():

        #Fetch Environment variables
        p_num = int(os.environ.get("PNUM",5))
        me = os.environ.get('ME',"endpoint-default")
        rabbit_host = os.environ.get("RABBIT_HOST","rabbitmq")
        rabbit_port = os.environ.get("RABBIT_PORT",5672)
        first_endp = os.environ.get("NODE0","endpoint-default")
        second_endp = os.environ.get("NODE1","endpoint-default")

        #Pool set up
        pool = multiprocessing.Pool(p_num)

        results = []

        #Start the consumers
        for i in range(p_num):
               r =
                   pool.apply_async(RMQ_Consumer,args=(rabbit_host,rabbit_port,
                          me,first_endp,second_endp))
               results.append(r)

        #Wait for processes to finish
        for r in results:
```

```
            r.wait()

        pool.close()
        pool.join()
```

Listing B.13: com_chan.py (lines 168-200)

As we can see, the initial steps in the implementation of the Communication Channel module remain the same as the ones used for the Simulator Endpoint module. The first step still consists of retrieving the various configuration parameters for the specific simulator's module. These parameters were configured as environment variables too but they were taken from the ConfigMap Kubernetes component deployed for the links of the network topology to model. After that, according to the amount of parallelism configured a `multiprocessing Pool` is created and each of its processes will run the usual `RMQ_Consumer` wrapper function to the real implementation of the Communication channel module that is contained in the `Communication_Channel` class.

It is important to note that, differently from the Simulator Endpoint module's implementation, the Communication Channel one doesn't need any shared data structure to be set up since this kind of module won't need to keep track of any parameters for a QKD simulation. To be more specific, unless a specific type of attack (interceptAndResend or manInTheMiddle) has been configured to be carried out, the Communication Channel will simply act as a pass-through of the simulation messages between the two endpoints it connects.

```
class Communication_Channel():

connection = None
rec_ch = None
send_ch = None
me = ""
first_endp = None
second_endp = None

def __init__(self,rhost,rport,me,fendp,sendp):...

def type_switcher(self,argument):
    switcher = {
        "sendRegister": self.sendRegister,
        "compareBasis": self.compareBasis,
        "verifyKey": self.verifyKey,
    }

    return switcher.get(argument,"Invalid Mssage Type Received")

def sendRegister(self,type,source,destination,metadata,msg):...

def compareBasis(self,type,source,destination,metadata,msg):...

def verifyKey(self,type,source,destination,metadata,msg):...

def startListening(self):...
```

Listing B.14: com_chan.py (Communication_Channel class)

As far as the class variables are concerned, there are:

- `connection`, a variable for the connection of the communication channel to RabbitMQ ;
- `rec_ch` and `send_ch`, variables to allow the simulator's module to send and properly consume messages from its corresponding queue;

- `me`, a variable to store which topology link the current module represents;
- `first_endp` and `second_endp`, variables to store which topology nodes the link the current module represents actually connects.

The `__init__` function of the class will be used to properly initialize the parameters of the class and create the connection and channels needed by every process of the pool to connect to RabbitMQ and start consuming messages on the corresponding communication channel's queue.

The `type_switcher` function is used whenever a new message on the channel's queue is consumed to decide the appropriate handling function to call and manage the message accordingly. As we can see the different message handling functions for BB84 simulations are:

- `sendRegister` function used to handle messages of type *sendRegister* for BB84 simulation;
- `compareBasis` function used to handle messages of type *compareBasis* for BB84 simulation;
- `verifyKey` function used to handle messages of type *verifyKey* for BB84 simulation;

```
def startListening(self):
    rec_ch = self.rec_ch
    queueResult = rec_ch.queue_declare(queue=self.me,auto_delete=True)
    rec_name = queueResult.method.queue

    logging.info("Connected to queue called " + rec_name)


    def message_handler(ch, method, properties, body):

        res = pickle.loads(body)

        #According to the type of message received, we will execute the
            appropriate function
        type = res[0]
        source = res[1]
        destination = res[2]
        metadata = res[3]
        msg = res[4]

        logging.info("Received message: Type-> " + str(type) + " S-> " +
            str(source) + " D-> " + str(destination))

        handler = self.type_switcher(type)
        handler(type,source,destination,metadata,msg)
        #Done procesisng message, acknowledge can be sent
        ch.basic_ack(delivery_tag=method.delivery_tag)

    logging.info('Starting to listen for messages on queue ' + str(rec_name))
    rec_ch.basic_consume(on_message_callback=message_handler, queue=rec_name)
        #Manual acknowledgement on by default
    rec_ch.start_consuming()
```

Listing B.15: com_chan.py (startListening function)

As we can see, the `startListening` function has the same structure as the one used for the Simulator Endpoint module implementation. The function is the one used to start consuming messages from the specific channel's queue. In particular, for handling each message received, the `message_handler` function will be called. For each message, this function will:

- deserialize it by using the *pickle* Python library;

- extract the different fields of the data model from the received message;
- evaluate the *type* of the message to retrieve the appropriate handling function to call;
- call the proper handling function;
- acknowledge the message to RabbitMQ after having managed it so that it will be erased from the endpoint's queue.

B.2.3 Simulator Manager Implementation

As far as the Simulator Manager module's implementation is concerned, it can be found inside of the following file:

`DockerImages/SimManagerImage/SimManager.py`

```
def main():
    serverPort = int(os.environ.get("PORT",6000))
    rabbit_host = os.environ.get("RABBIT_HOST","rabbitmq")
    rabbit_port = int(os.environ.get("RABBIT_PORT",5672))

    pool = multiprocessing.Pool(2)
    rest = pool.apply_async(run,args=(serverPort,rabbit_host,rabbit_port))
    rmqcons = pool.apply_async(logging_consumer,args=("manager_logger",
                               rabbit_host,rabbit_port))

    rmqcons.wait()
    rest.wait()

    pool.terminate()
    pool.close()
```

Listing B.16: SimManager.py (main function)

As we can see, the first thing to be done consists of retrieving the configuration parameters for the Simulator Manager module that have been set as environment variables when its corresponding Deployment Kubernetes component was deployed to the cluster by the QKDSimulator CRD Custom Controller. After that, a `multiprocessing Pool` of two different processes is created. The processes of the `Pool` will be respectively running:

- an instance of Flask Python WebServer in order to allow the module to expose the REST APIs needed to users to interact and use the simulator;
- a RabbitMQ consumer to centralize logging for all of the simulator's modules representing the network topology.

```
#Network Topology
networkTopology = nx.Graph()

connection = None
channel = None

#Connections To The Simulator
open_conns = dict()


def logging_consumer(logger_name,rabbit_host,rabbit_port):

def run(serverPort,rabbit_host,rabbit_port):
```

```
    global connection
    global channel
    params = pika.ConnectionParameters(host=rabbit_host,
                                port=rabbit_port, heartbeat= 0)
    connection = pika.BlockingConnection(params)
    channel = connection.channel()
    app.run(host='0.0.0.0', port=serverPort)

def check_Topology(topology):

def createAndDistributeKeys(topology):

def checkSim(topology,source,destination,com_chan):

def cleanRepeatedEdges(topology):

@app.route('/begin', methods=['POST'])
def begin():

@app.route('/end', methods=['POST'])
def end():

@app.route('/startKeyExchange', methods=['POST'])
def startSimulation():

@app.route('/netTopology', methods=['POST'])
def getAndDeployTopology():

@app.route('/distributeKeys', methods=['POST'])
def distributeKeys():
```

Listing B.17: SimManager.py (lines 19-325)

As far as the global variables are concerned, these will be needed by the process running the Flask WebServer:

- `networkTopology`, a variable to store the modelled network topology;
- `connection`, a variable to handle the connection to RabbitMQ;
- `channel`, a variable to handle sending messages using RabbitMQ;
- `open_conns`, a variable to keep track of the pairs of endpoints of the modelled topology to which it has been associated a UUID to perform one or more QKD simulations.

The `logging_consumer` function is the one run by the RabbitMQ consumer process used to centralize the logging. The `run` function is the one run by the other process of the pool to have a Flask WebServer up and running. After that, there is a series of utility functions:

- `check_Topology`, a function to check the modelled network topology has been provided following the expected syntax;
- `createAndDistributeKeys`, a function to generate and distribute the keys needed to use SPHINCS+ authentication during QKD simulations;
- `checkSim`, a function to check if the data provided to configure and start a QKD simulation has been provided as expected;
- `cleanRepeatedEdges`, a function to pass from a directed graph representation of the modelled topology to an undirected one.

Then there are the functions implementing the REST APIs exposed by the Simulator Manager and needed to interact and use the entire QKD Simulator:

- `begin`, a function implementing the REST API needed to select two nodes of the modelled topology to start QKD simulations between;
- `end`, a function implementing the REST API needed to unselect two nodes of the topology to perform QKD simulations between;
- `startSimulation`, a function implementing the REST API to start and perform one or more QKD simulations between two previously selected nodes of the modelled topology;
- `getAndDeployTopology`, a function implementing the REST API to notify the Simulator Manager module about the network topology currently modelled and deploy to the Kubernetes cluster.
- `distributeKeys`, a function implementing the REST API to enable using the preferred authentication method for the classical channel communication authentication during the various QKD simulations.

B.2.4 Jupyter Implementation

Finally, there is still one Docker Image left to describe: the Jupyter Notebook. This image is needed and run by the Simulator Manager module to expose a GUI through Jupyter Notebook where the user can develop his own custom scripts to interact with the QKD Simulator as he pleases. The implementation of such an image can be found inside the following folder:

`DockerImages/JupyterImage`

As far as this Docker Image is concerned, it consists of a standard Jupyter Notebook installation that is run using a custom configuration provided in the following file:

`DockerImages/JupyterImage/conf/jupyter.py`