Quantum Key Distribution in Kubernetes Clusters

future internet

MDPI

Article

Quantum Key Distribution in Kubernetes Clusters

Ignazio Pedone * and Antonio Lioy

Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy; antonio.lioy@polito.it

* Correspondence: ignazio.pedone@polito.it

Abstract: Quantum Key Distribution (QKD) represents a reasonable countermeasure to the advent of Quantum Computing and its impact on current public-key cryptography. So far, considerable efforts have been devoted to investigate possible application scenarios for QKD in several domains such as Cloud Computing and NFV. This paper extends a previous work whose main objective was to propose a new software stack, the Quantum Software Stack (QSS), to integrate QKD into software-defined infrastructures. The contribution of this paper is twofold: enhancing the previous work adding functionalities to the first version of the QSS, and presenting a practical integration of the QSS in Kubernetes, which is the de-facto standard for container orchestration.

Keywords: Quantum Key Distribution; Quantum Cryptography; software-defined infrastructures

check for updates

Citation: Pedone, I.; Lioy, A. Quantum Key Distribution in Kubernetes Clusters. *Future Internet* **2022**, *14*, 160. https://doi.org/10.3390/fi14060160

Academic Editors: Izzat Alsmadi, Samer Khamaiseh and Steven Furnell

Received: 5 April 2022
Accepted: 18 May 2022
Published: 25 May 2022

1. Introduction

Recent years have witnessed a growing concern regarding the impact of Quantum Computing on current asymmetric cryptography algorithms. This tendency is mainly due to the well-known Shor's Algorithm [1], proposed by Peter Shor in 1994, and its role in solving the integer factorisation problem in polynomial time on a quantum computer. For example, the widely used RSA cryptosystem relies on the assumption that prime factoring is hard to solve by current classical computers. As a result, the advent of quantum computing jeopardises all the protocols that involve RSA, as well as other fundamental security algorithms, such as the Diffie–Hellman key agreement. The remarkable advances achieved by quantum computing in recent years concern academia, industry, and government agencies, such as the National Institute of Standards and Technology (NIST). Accordingly, NIST launched a challenge to standardise new quantum-resistant classical cryptographic algorithms, reasserting the importance of the Post-Quantum Cryptography (PQC) research field [2]. Similarly, in Quantum Cryptography, the Quantum Key Distribution (QKD) technique allows solving the specific problem of quantum-resistant key exchange by leveraging quantum mechanical phenomena (more details about this technique in Section 2.1. QKD is promising because it does not rely on mathematical conjectures like classical algorithms but rather on well-known physical phenomena. One of the main issues of QKD is that it is tied to special-purpose devices, and they come with limitations, i.e., distance range limit and vulnerability to physical attacks. Besides, it is far from simple to integrate these systems in modern network infrastructures that heavily depend on virtualisation and require a high level of flexibility and scalability. The work behind this paper is a natural prosecution of the one presented in [3] which discusses a new Quantum Software Stack (QSS) that is able to act as a middleware between QKD devices and high-level security applications. A QSS manages all the complexity within a QKD network, providing a straightforward interface for the security applications. Starting from the work above, we enhanced some architectural and implementation aspects of the first QSS version. In particular, we introduced an asynchronous approach in developing the QSS components, the ability to support multi-hop (MH) and long-distance QKD exchanges, and routing capabilities within the QSS to select the best path for the exchange. Even

though those are essential and relevant contributions to the development of the QSS, this paper's major contribution lies in the integration of the QSS in a Kubernetes cluster. Kubernetes (which we examine in Section 2.2) is the de-facto standard for container orchestration and a platform widely used in many modern paradigms such as Cloud, Fog, and Edge Computing, Network Functions Virtualisation (NFV), and Internet-of-Things (IoT). Given these critical application scenarios, QKD integration into Kubernetes represents a consistent contribution on a practical level and paves the way for the adoption of QKD systems in multiple domains.

2. Background

In this section, we propose a brief digression on QKD, a presentation of Kubernetes [4] and its concept of *operator*, and an overview of the QSS and the related ETSI standards.

2.1. Quantum Key Distribution

QKD is a well-known technique of Quantum Cryptography and allows exchanging keys among parties by leveraging principles of quantum mechanics, such as entanglement and the no-cloning theorem [5]. Broadly speaking, QKD counts two main classes: Discrete Variable QKD (DV-QKD) and Continuous Variable QKD (CV-QKD). Differences depend on the physical aspects of the key exchange process, i.e., the method to encode quantum information.

Regardless of the specific class and protocol, QKD systems leverage physical devices such as photon sources and detectors and may use specific waveguides (e.g., optical fibre) to transmit quantum information. All those elements have intrinsic vulnerabilities and imperfections that may lead to attacks if not conveniently addressed by QKD protocols. From a high-level perspective, it is also worth noting that QKD schemes require two different channels for communication: a quantum channel and a classical authenticated channel. The first serves as a carrier for quantum information; the second bears additional classical information, which is generally required to carry out QKD protocols. Several of these protocols are available in the literature [5,6], e.g., BB84, B92, SARG04, E91, BBM92, and some of them are implemented in commercial devices, e.g., BB84, COW, T12 [7]. As a tangible example of a QKD system, ID Quantique [8], a renowned QKD devices manufacturer, recently presented the *Cerberis XG QKD System* [9] with a secret key rate of approximately 2 kbit/s and a maximum quantum channel length of 50 km. Moreover, Toshiba commercialises QKD devices [10] based on a modified version of BB84 (with decoy states and phase encoding) with a range of up to 120 km and an exchange rate of 300 kbit/s (at 10 dB loss). Those products provide a good outlook on commercial devices' state-of-the-art.

Despite the availability of these commercial devices, QKD represents a vibrant research field, where great effort is dedicated to enhancing current QKD systems in terms of security, distance range, and key exchange rate. Some recent proposals and experimental developments related to the so-called twin-field QKD (TF-QKD) protocol [11] are also worth mentioning. This protocol is based on a measurement-device-independent QKD (MDI-QKD) scheme [12]. The TF-QKD protocol is remarkable because, as the basic MDI-QKD, it has the advantage of removing all detector side-channels but additionally it proposes a novel approach to overcome the rate-distance limit of QKD. This method is relevant for developing future long-distance QKD exchange systems. This paper does not go into the details of QKD devices since we treat QKD as a "black box". We assume that Point-to-Point (PTP) key exchange between QKD devices is feasible leveraging commercial devices, and it comes with some limitations regarding endpoints distance and key rate. For the rest of the paper, we rely on the concept of the *QKD network*, a network where nodes leverage QKD devices capable of exchanging keys with their direct neighbours. In such a network, long-distance exchanges are possible by means of trusted repeaters (Section 3.2).

2.2. Kubernetes and Operators

Developing microservices instead of monolithic applications has become a widely adopted approach in distributed systems. Microservices leverage lightweight virtualisation technologies that run applications in sandboxes called containers. Kubernetes is the de-facto standard for container orchestration and allows the management of the whole life cycle of these objects.

Kubernetes sees everything related to an infrastructure as a resource: physical or virtual nodes, applications, exposed services, and even configurations. A set of physical or virtual nodes can be aggregated in clusters. Inside a cluster, nodes are labelled as masters or workers depending on their role: masters manage the cluster and retain sensitive information and configuration about the infrastructure, workers are deputed to execute the containerised applications and expose services. Here are definitions of some Kubernetes resources needed in the rest of this paper:

- *Pod:* is the smallest deployable object in Kubernetes, and it could be seen as a set of containers that share the same IP address. In order to run those containers, a Container Runtime is available on the node (e.g., Docker, containerd, cri-o);
- *Service:* is an abstract resource that represents a way to expose an application that could be composed of one or more pods, i.e., replicas of the same applications;
- *Deployment:* is an object that provides declarative updates for Pods. In practice, it allows managing changes to their status through the Deployment controller. It is possible to describe the desired number of replicas of a specific pod and the policies for the rollout or rollback of the application to previous versions;
- *Secret:* is an object that contains sensitive data such as passwords, tokens, or keys. Generally is used to avoid including sensitive information in an application or code;
- *Custom Resource (CR):* this is a way to extend the Kubernetes API by creating a new resource that allows storing objects of a certain type;
- *Custom Controller:* combined with the CR, it continuously monitors the cluster and based on the state of a CR, it applies changes to the cluster. In this sense, Kubernetes implements a declarative approach.

To summarise, Kubernetes allows the deployment of containerised applications over a distributed infrastructure, managing those resources and automatically scaling them when needed.

The relatively recent introduction in Kubernetes of the *Operator pattern* is also worth mentioning. As we already discussed, one of the principles of Kubernetes is the control loop where resources are declared, and controllers continuously monitor them taking actions if needed. The Operator pattern follows this logic by using CRs and controllers to monitor distributed applications and manage their life-cycle by instantiating, updating, and configuring them. The ultimate goal of an Operator in Kubernetes is to mimic the behaviour of a human operator who is in charge of managing resources in a Kubernetes cluster.

2.3. Quantum Software Stack and the ETSI Standards

The authors of [3] presented a first attempt in developing a software stack to enable cloud infrastructures, and more in general software-defined infrastructures, to use QKD. The above solution was called Quantum Software Stack (QSS) and released with an open-source license alongside that paper. QSS intends to fill the gap between QKD devices (that are often tailored to specific protocols and proprietary interfaces) and modern infrastructures, whose services need to be scaled, migrated, and reconfigured with great flexibility. QSS has also been designed to be compliant with the European Telecommunications Standards Institute (ETSI) standards on QKD [13].

As depicted in Figure 1, the first version of QSS was designed on four different layers. The lowest layer defines either QKD devices or QKD simulators useful for testing purposes. At this layer, QKD protocols take place with all their related requirements. For example, we generally need two communication channels for carrying on any protocols: a classical authenticated channel for sharing additional information during a key exchange

(e.g., chosen bases, privacy amplification or error correction info) and a quantum channel where quantum information flows (i.e., encoded in photons). The layer immediately above contains the QKD module (QKDM) which serves as a wrapper for the QKD device so that each device in the QKD network could expose the same interface to the upper layer. This interface is called the southbound interface. Inter QKDM communication within the network takes place through the sync interface. A layer above, we find the Quantum Key Server (QKS), which is the core of the QSS and manages all the information regarding the exchange, stores the generated keys, triggers the exchange process, and collects routing information to reach peers in the QKD network. The highest layer is the Secure Application Entity (SAE) one. This level contains high-level applications that require cryptographic keys through the QKS using the northbound interface.

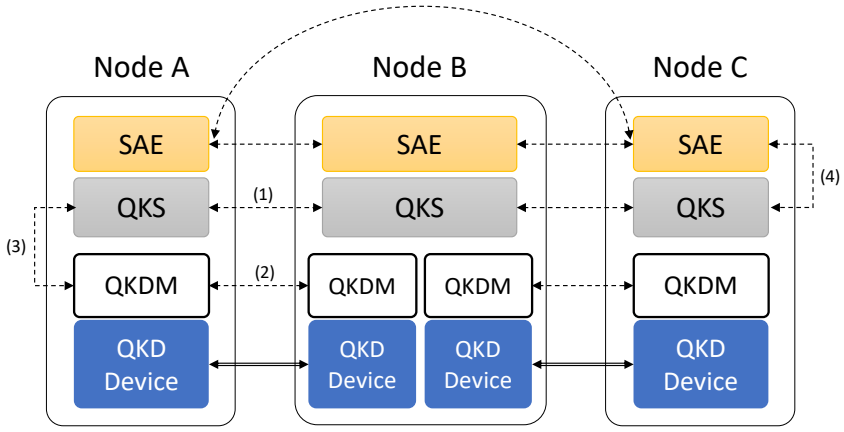

Figure 1. Quantum Software Stack (QSS) presented in [3]. Dashed lines represent TCP/IP secure connections. Thick, double lines indicate the combination of classical and quantum channel for the QKD exchange. The numbers enumerate the main interfaces: (1) External Interface; (2) Sync Interface; (3) Southbound Interface; and (4) Northbound Interface.

Figure 2 provides a more detailed view of the QKS. In Section 3, we discuss some of the architectural components, such as the Redis and Routing modules that are part of the new QSS version presented in this paper. The other architectural modules remained almost unchanged from the previous version but they consistently improved with new features and enhancements (Section 3). To summarize the primary logic behind the QSS, we could start from the QKDM perspective, where there is a continuous exchange of keys with one of its peers within the QKD network. We identify this flow of key material as *Key Stream (KS)*. The QKS controls the life-cycle management of a KS. Each QKS could have different QKDMs to allow the management of several devices and thus reach different destinations. This feature also allows making QKSs within a QKD network acting as trusted repeaters (feature introduced in Section 3.2). Besides, we could imagine mapping more QKDMs on the same physical device to sustain different flows when devices support multiple streams. As represented in Figure 1, we could have a PTP exchange among Node A and Node B, or we could reach Node C from Node A passing through Node B (trusted repeater). In the first version of QKS, this feature was not available (Section 3.2).

The QKS, in its original version, was mainly in charge of serving SAE key requests and providing QKDMs with information on where to store generated keys (e.g., ad-hoc Vault tokens). When an SAE wants to communicate securely with another one over the same QKD network, a *Key Reservation Process (KRP)* is required. In particular, when a source SAE (master or mSAE) contacts its QKS and requires key material with the `getKey` call, it specifies the number of keys, their length, and the destination SAE (slave SAE or sSAE). Then, the KRP takes place between source and destination QKSs (where SAEs are located). If the KRP succeeds, the source QKS returns the required keys and their IDs. mSAE at this point could share these IDs with the sSAE, allowing the latter to retrieve the duplicate keys

calling the `getKeyWithId` through the destination QKS. The KRP leverages the External Interface to accomplish this task.

When a QKDM registers to a QKS, a QKDM Registration Process (QRP) takes place with Keycloak (QSS uses Keycloak [14] as its Identity and Access Management service). The QRP can start only if the QKDM is available as a Keycloak user. Therefore, for authentication purposes, a registration of the QKDM to Keycloak is mandatory; only afterwards the QKDM can be configured by a system administrator and the QRP can start through the Southbound Interface. First, the QKDM sends a registration request to the QKS, providing information regarding reachable destinations and the underlying QKD device. Then, the QKS grants the QKDM limited access to the Vault instance and the DB. This process enables the QKDM to store keys in an isolated portion of the QKS resources.

Another critical aspect of the QSS is the link with the ETSI QKD specifications. The ETSI ISG (Industry Specification Group) on QKD covers different aspects related to the standardization of QKD, starting from the physical devices to the software interfaces that allow the communication of these devices with high-level security applications. The following specifications are relevant to the present paper:

- *ETSI GS QKD 004 V2.1.1* [15]: the specification "QKD; Application Interfaces" describes the interface to the physical device; in our case, it could be associated with the Southbound Interface;
- *ETSI GS QKD 014 V1.1.1* [16]: the specification "QKD; Protocol and data format of REST-based key delivery API" describes a high-level interface from the high-level application to a key manager. In our case, this corresponds to the Northbound Interface;
- *ETSI GS QKD 015 V1.1.1* [17]: the specification "QKD; Control Interface for Software-Defined Networks" is recent, and it is a first attempt to standardize a relatively recent trend in QKD device management, namely *Software-Defined Quantum Key Distribution* (SD-QKD). The concept is to leverage an SDN controller who knows all the topology of the QKD network and manages all the QKD nodes accordingly in a centralized way. This approach could be beneficial in scenarios where the QKD network is switch-based, meaning the quantum channels could be adjusted using optical switches, which the SDN controller itself would control.

3. Quantum Software Stack 2.0

As the first contribution of this paper, we introduce in this section a complete software refactoring of the original QSS with the addition of significant changes in both architecture and programming approaches. In Figure 2, we illustrate with dashed lines the components that have been either introduced or considerably modified from the earliest version. The rest of the components demanded minor changes.

Starting from code refactoring, QSS 2.0 introduces a paradigm shift from a synchronous approach based on multithreading to an asynchronous one based on multiprocessing (Section 3.1). Moreover, this new QSS version allows long-distance QKD exchanges through trusted repeaters, becoming suitable for large QKD network scenarios (Section 3.2). QSS 2.0 also includes a routing module within the QKS, making possible the discovery of the QKD network topology and finding the best path over trusted repeaters when a long-distance exchange occurs (Section 3.3).

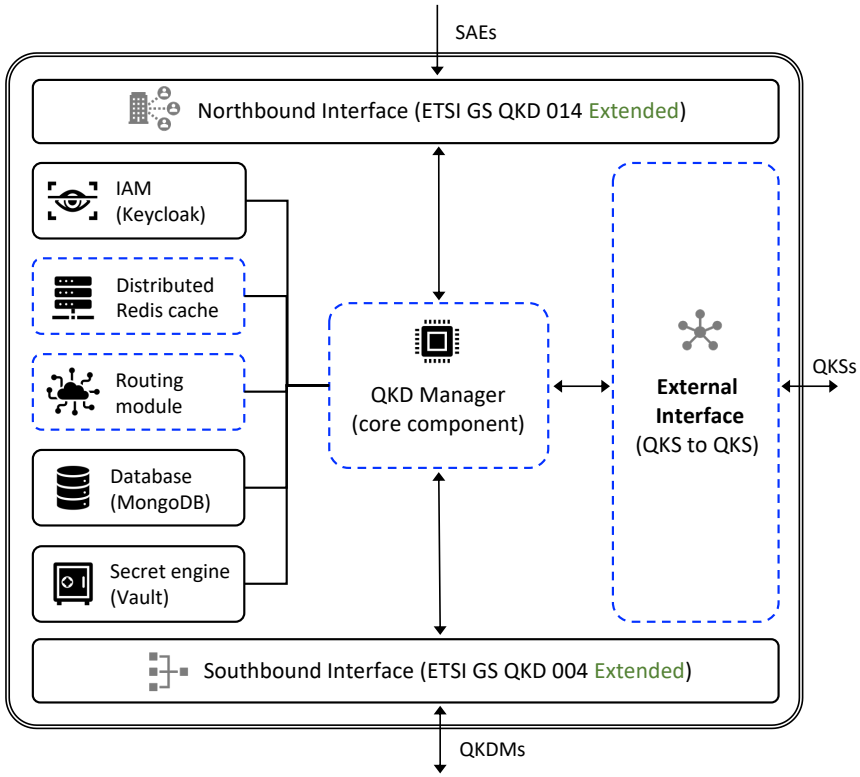

Figure 2. The architecture of the new Quantum Key Server (QKS) version.

3.1. Asynchronous Approach

The first version of the QKS used a multithreaded synchronous approach This choice was poor considering that the standard *CPython* interpreter limits the execution of multiple threads at a time by exploiting the *Global Interpreter Lock (GIL)* [18]. Besides, the QKD Manager, the core component within the QKS, is an I/O bound application, meaning that it spends a consistent amount of time waiting for results from other applications. Thus, the synchronous choice consistently slowed the whole system down. Because of this, we introduced a new version of the QKS entirely based on an asynchronous pattern and the `asyncio` Python library. In more detail, the QKD Manager exposes REST APIs leveraging the Quart [19] framework , which supports asyncio, and the Hypercorn [20]; asynchronous web server to serve incoming requests. Even the Routing module (Section 3.3), the available QKDMs, and the clients for MongoDB, Vault, and Redis now adopt an asynchronous approach and specific libraries. This new strategy and the adoption of Hypercorn allow scaling of the application both vertically and horizontally. Indeed, when the number of requests to the QKS grows, we could decide either to statically allocate additional workers on the Hypercorn side or dynamically allocate more replicas of the QKD Manager. Since we designed all the components according to the principles of the cloud-native applications, we could also deploy our QSS on Kubernetes, thus making the horizontal scaling process automated.

3.2. Trusted Repeaters

QKD commercial technologies are currently limited to PTP exchanges and cannot exceed a hundred kilometres of range. Because of this, repeaters are required to make possible communication over longer distance. As presented in [21], the lack of quantum repeaters based on entanglement swapping in commercial devices leads to the necessity of having trusted repeaters. These rely on the weak assumption that nodes within a QKD network are trusted, and thus they could be used as relays to transmit the key material over long distances.

The current version of the QKS introduces the possibility of using trusted repeaters and performing long-distance exchanges. All QKSs within a QKD network are provided with routing information coming from their routing modules Section 3.3 that keep this information updated according to the state of the network. When a long-distance exchange is required, i.e., an endpoint needs to pass through one or more QKSs to reach its destination, all the QKSs along the route are informed and reserve two keys to allow the flow of key material (during the KRP). We adopted the "store and forward" strategy as described in [22], where data are encrypted and decrypted on each relay. Clearly, this strategy has its flaws, though it is out of the scope of this paper to find an optimal algorithm for performing exchange in the case of trusted repeaters. However, due to the modular software architecture of our solution, it is possible to rely upon the KRP over the trusted repeaters and develop a custom exchange strategy. It is worth mentioning that, from a QKS user perspective, the presence of trusted repeaters is entirely transparent as they could send a request as in the PTP case, only indicating the destination SAE.

3.3. Routing

We introduced the *Routing Module* (RM) to allow proper traffic steering within a QKD network and connect two arbitrary endpoints (SAEs) supporting long-distance QKD exchanges. From a high-level perspective, each QKS has its own RM, which shares routing information with its peers in the network, enabling the QKS to establish whether an SAE is reachable and the best path to arrive at it.

The RM leverages a modified version of the *Open Shortest Path First (OSPF)* protocol [23] to select the best path, an approach similar to the one adopted by DARPA and SECOQC in [24,25]. Information such as SAEs reachable from a specific QKS and recently opened KSs are transmitted, sending *Link State Advertisements (LSAs)*. The sending of an LSA is triggered on specific events: registration or removal of a new SAE, opening a new KS, and expiration of the internal RM timer. This timer is required to tackle situations where the signalling mechanism fails, i.e., when a node is no longer reachable from the network and routing tables still need to be updated. The RM stores the collected information in a Redis data structure, as we depicted in Table 1.

Table 1. Description of the routing table entries in Redis.

Name	Type	Description
`SAE_ID`	String	name used to identify the destination SAE
`next_hop`	String	ID of the next QKS in the path to the destination SAE
`dest`	String	ID of the QKS of the destination SAE
`cost`	Integer	the cost $C(t)$ associated to the path from the current SAE to the destination
`length`	Integer	number of nodes in the path to reach the destination

The communication among RMs takes place by leveraging a TCP connection and custom packets whose structure is shown in Table 2. There exist two packet types so far: the S packet, which carries information regarding the SAE on a specific QKS and is typically sent after an SAE registration, and the packet K, which holds information on the QKS neighbours and the costs to reach them.

The routing protocol computes the routing tables using the Dijkstra algorithm, and the cost function $C(t)$ is defined by Equation (1), where $\mathbb{P}$ is the set containing all the links of a chosen path.

$$C(t) = \sum_i K_i(t) \qquad \forall i \in \mathbb{P}. \tag{1}$$

$K_i(t)$ is the key load and is defined by the Equation (2). It represents the cost of a single link based on the available keys on that link at the time t and $t-1$. The coefficient c_0 represents the cost when no key is available. This cost is mitigated by the second term in which K represents the total number of keys that the buffer could contain and $k_i(t)$ is

the number of the currently available keys at the time t. The more $k_i(t)$ grows, the more this term decreases the key load. The third term models the tendency of $k_i(t)$ to increase or decrease over time. Since the only available information as a metric is related to the available keys and no Quality-of-Service (QoS) parameter (e.g., error rate, jitter) is known, this is an attempt to model other cost contributions such as congestion on a specific link. The weights w_1 and w_2 allow the balance of the contribution of each term.

$$K_i(t) = c_0 - w_1 \cdot \frac{k_i(t)}{K} - w_2 \cdot \frac{k_i(t) - k_i(t-1)}{K}. \quad (2)$$

The RM allows the management of the routing within a QKD network, making each QSS independent from the others. Other strategies, like the ones based on SDN [17], offer advantages such as the centralised management of the whole network by knowing a priori its topology. They could also consistently reduce the number of packets exchanged among RMs, thus minimising the requirement for additional key material (all routing packets need authentication). On the bright side, with our decentralised solution, no central entity (e.g. SDN controller) has to have, even if partial, access to the QSSs. Undoubtedly, the SDN approach could be integrated easily into our solution by modifying the routing components and yet profiting from the key management process.

Table 2. Routing LSA packet structure.

Field Name	Description
`version`	protocol version
`type`	type of information carried, `S` for SAEs and `K` for QKDM links
`source`	QKS source ID, address and port
`routing`	source routing module address and port
`forwarder`	ID of the last QKS which forwarded the packet
`neighbors`	list of connected SAEs or active QKDM links, based on the `type` field
`timestamp`	packet creation time
`authentication`	data for authentication and integrity checks

4. Integration in a Kubernetes Cluster

This section presents the core of this work: the integration of the QSS in Kubernetes. Considering the impact of microservices over modern paradigms such as Cloud, Edge and Fog computing, and Kubernetes as the de facto standard for the orchestration of these microservices, it is pivotal to integrate quantum-safe cryptography with this technology. In particular, the QSS already takes care of decoupling the physical technologies needed for the QKD and the requirements of the high-level security applications. The final piece of the puzzle is to integrate the QSS into a real orchestration platform so that the applications running in pods can leverage QKD keys without changing how they interact with classical secrets provided by the platform.

In a Kubernetes cluster, the control plane, i.e., master nodes, retains all the sensitive information about the applications. It also provides the high availability of those data leveraging technologies such as *etcd* [26]. Since each Kubernetes cluster holds its own control plane and secrets management system, a reasonable scenario can involve two or more distributed clusters sharing cryptographic keys with QKD. Secrets in Kubernetes are unique resources that provide cryptographic keys, credentials and other sensitive data to applications. Our idea of integration lies in delivering a mechanism that allows a QKD key to be requested through the Kubernetes APIs from a specific application. Then this resource can be stored within the application namespace as a secret. Once available as a secret, the application can use it as a classical key.

To add the functionalities of a QSS in a Kubernetes cluster, as we describe in Section 4.1, we used the operator pattern (Section 2.2). A set of Kubernetes clusters could be imagined as nodes of a QKD network, and each one integrates a QSS as an operator. This strategy

allows applications to treat the QSS as a native resource that could exploit its functionalities without significant changes.

4.1. Quantum Software Stack Operator

As mentioned in Section 2.2, an operator is a software extension to Kubernetes, which leverages CRs and controllers following the control loop pattern. Therefore, we could create additional CRs to map out SAEs and key requests in Kubernetes. Then, using custom controllers, we could add a logic behind events of the life-cycle related to those resources (e.g., creation, removal, update). We defined two CRs, `Sae` and `KeyRequest`, and developed the *QSS Operator* to operate them. All the components of the classical QSS, such as QKS and QKDM, are available as deployments applied to the Kubernetes cluster.

In Figure 3, a PTP exchange between two clusters is depicted and described in Section 4.2. `Sae`, when applied, creates a new logical SAE in the scope of the cluster, registers this SAE to Keycloak, and makes sure that this is available at the application level of the QKD network. `KeyRequest` instead starts the KRP at the QKS level depending on the number of keys requested and their length.

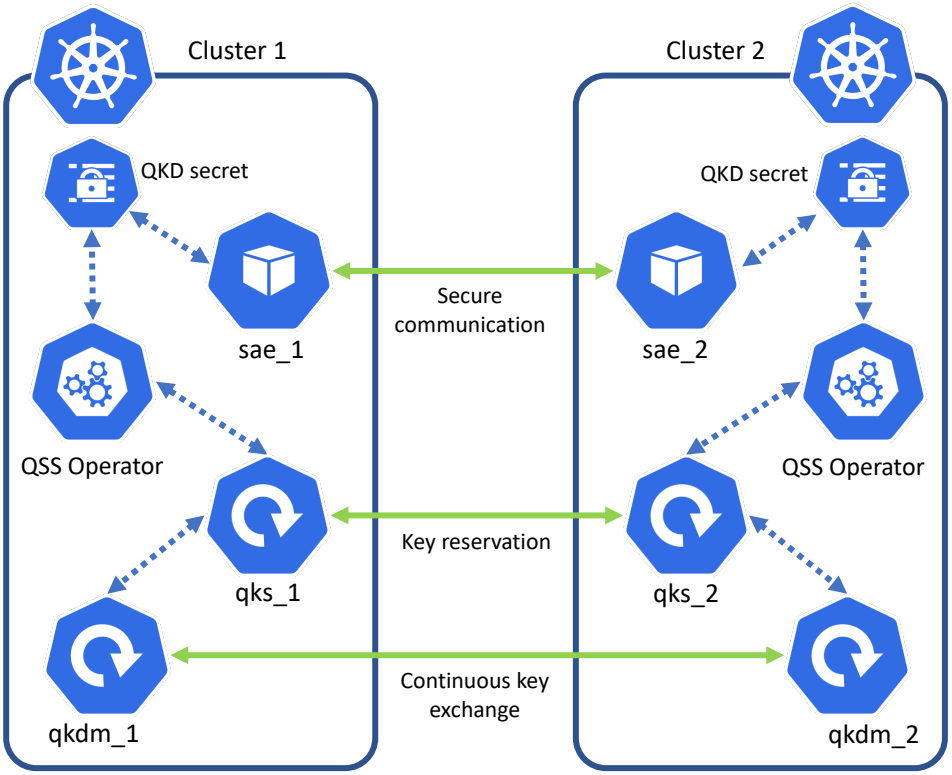

Figure 3. PTP exchange among clusters.

From a cluster perspective, the final goal is to have a QKD secret available for an application as a result of the declaration of the `KeyRequest` resource. In Figures 4 and 5, we reported examples of `Sae` and `KeyRequest` descriptors for creating the those resources.

```
apiVersion: "qks.controller/v1"
kind: Sae
metadata:
name: sae1
spec:
id: sae_1
registration_auto: true
```

Figure 4. `Sae` resource YAML descriptor.

```
apiVersion: "qks.controller/v1"
kind: KeyRequest
metadata:
name: request1
spec:
number: 1
size: 128
master_SAE_ID: sae_1
slave_SAE_ID: sae_2
```

Figure 5. `KeyRequest` resource YAML descriptor.

4.2. Resource Creation and Key Exchange

In this section, we analyse the details of the resource creation and the overall logic behind PTP and MH exchanges. There are some assumptions behind the possibility of registering SAEs and performing key requests. First, we need to have a QKD network and two or more Kubernetes clusters, as depicted in Figure 3. Second, at least one QKDM has to be deployed on each cluster as well as the QKS deployment. Third, we need to install and configure the QSS operator on each cluster to work with the Kubernetes API instead of the QKS interface. The last assumption is that there is a continuous exchange among the QKDMs involved, and so all the registration processes and the KS creation have been handled as described in Sections 2.3 and 3. Regardless of the type of exchange (PTP or MH), we need all the SAEs involved registered within the network. This operation is feasible according to the workflow depicted in Figure 6. An SAE admin, who is in charge of managing the SAE resource in Kubernetes and requiring QKD keys, can create a `Sae` resource. A custom operator can also handle this action without human intervention. After the resource creation, the QSS operator recognises the new `Sae` resource and starts the registration of this resource to Keycloak. Again we have two options: register the SAE to Keycloak in advance (manual registration) or let the QSS operator handle this part (automatic registration). This choice can be selected in the descriptor in Figure 4. Once an SAE can be authenticated through Keycloak, the QSS Operator registers it to the QKS, and if the automatic registration process is enabled, it retrieves the credentials to access the QKS by creating a specific Kubernetes secret. Notice that since the scope of the operator is global within the cluster, an SAE could be in a namespace different from the QKS one.

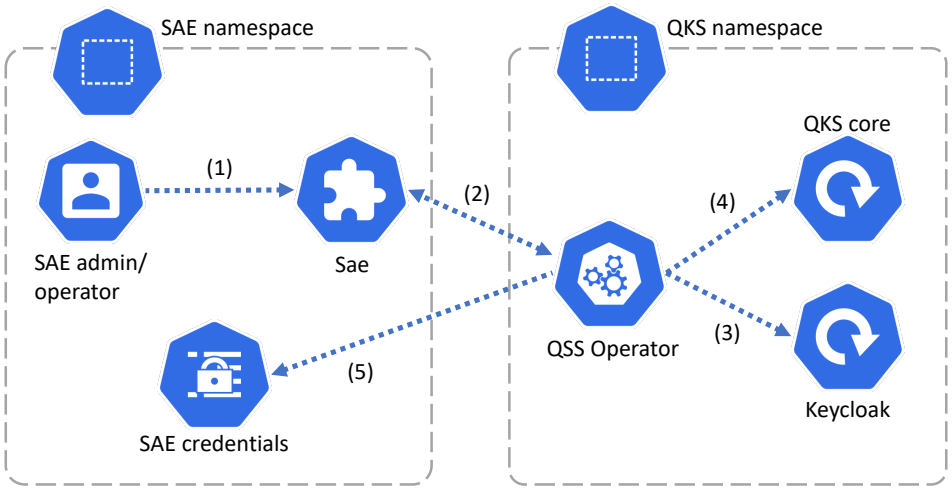

Figure 6. SAE registration process. SAE admin/operator creates a resource of type Sae (1). QSS operator monitors resource creation (2). When it finds the new Sae it starts the registration to Keycloak (3) and the QKS core (4). At the end of the process it retrieves the SAE credentials (5).

Once two registered SAEs want to exchange a key, a resource of type `KeyRequest` needs to be created. In particular, the SAE admin/operator again is in charge of applying

the resource (Figure 5), specifying as parameters the key length, the number of keys required, mSAE, and sSAE. The workflow (depicted in Figure 7) is similar to the previous case of SAE registration: the QSS operator detects the creation of the `KeyRequest` resource, authenticates the SAE through Keycloak, requests the key material to the QKS, which starts the KRP and retrieves the key material to the QSS operator, and stores the final keys as a Kubernetes secret.

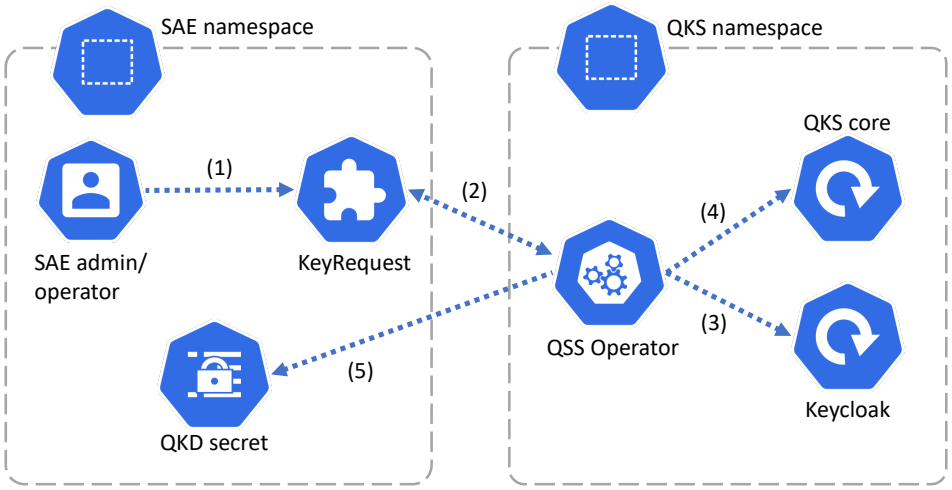

Figure 7. Key request process. SAE admin/operator creates a resource of type KeyRequest (1). QSS operator monitors resource creation (2). When it finds the new KeyRequest it starts the authentication through Keycloak (3) and forward the request to the QKS (4). At the end of the process it retrieves the QKD secret (5).

Looking at Figure 3, it is worth mentioning that, given an SAE within Cluster 1 as the initiator (or mSAE) of the request, the responder SAE (or sSAE) within Cluster 2 has to perform another `KeyRequest` indicating the IDs of the keys that it wants to retrieve. That information could be exchanged on an authenticated channel, and also this process could be automated using a custom operator in Kubernetes. After this second request, all the SAEs involved have access to a Kubernetes secret representing the key material generated through the QKD.

The operator also supports MH exchanges, whose scheme is similar to the one depicted in Figure 1 where it is mentioned the possibility to exchange keys between Node A and Node C, using Node B as a Trusted Repeater. This is compliant with what we presented in Section 3.2; thus, given a QKD network where clusters are the nodes, we still can perform exchanges traversing those clusters and leveraging QSS operators.

4.3. Applications to Suitable Use Cases

Multiple classical applications can enhance their security by leveraging QKD [27]. For example, in a multiuser smartphone network with a star-type architecture, a central session initiation protocol (SIP) server may share a key with each of its clients. In this case, QKD can supply these keys to symmetric cryptosystems such as the Advanced Encryption Standard (AES) that are adopted for encrypting the communication. Another interesting idea could be to use QKD systems based on free-space optics technologies to secure drone communications [28]. Additionally, QKD might be adopted in combination with other security protocols such as Internet Protocol Security (IPsec) and Transport Layer Security (TLS) to provide reliable key material. This approach can be implemented by adapting the latter protocols to use QKD keys through specific APIs. In various cases, it adopts particular variants or extensions of these protocols (e.g., TLS-PSK).

From a modern network standpoint, there are a plethora of cloud-native applications that could either directly or indirectly benefit from our solution. Indeed, security applications running on a cluster, such as VPN endpoints, may directly require QKD keys from

the Kubernetes Operator and use them to configure a VPN tunnel among two endpoints. Furthermore, some applications may adopt special microservices called *service mesh* (e.g., Istio [29] and delegate them security aspects such as the management of TLS connections. These objects can act as proxies and serve the upstream application, unaware of the TLS connection. Clearly, these microservices could also benefit from our solution to improve the TLS protocol security by using QKD.

Starting from this wide range of possible applications, a timely and engaging example are the blockchain technologies. Today, blockchains serve different scenarios ranging from cryptocurrencies to supply chain management. Due to the sensitive data stored in a blockchain and the quantum threat affecting its technologies, it is reasonable to focus on building quantum-resistant blockchain networks. Several studies analyse the security of blockchain technologies, the impact of the quantum advent, and possible mitigation both coming from Post-Quantum and Quantum Cryptography [30–32]. Similar to the previous scenarios, QKD can enhance the security of blockchain technologies. A more practical example could be given by the Hyperledger project [33] and, in particular, by the Hyperledger Fabric framework. The nodes within this permissioned blockchain network extensively use the TLS protocol with algorithms such as RSA and ECDSA to communicate with each other. The role of QKD, in this case, is to build a quantum-resistant communication protocol between nodes and components of the framework, either based on an enhanced version of TLS or a new protocol. Regardless of the specific adoption as the final protocol, our solution could provide a consistent mechanism to integrate the QKD in such a scenario. The Hyperledger Fabric framework can be easily deployed as nodes within multiple Kubernetes clusters, thus requiring no changes to our current model.

Finally, it is also worth mentioning that the QSS Operator has been designed to be flexible and to adapt to multiple contexts and scenarios. In particular, the QSS operators may serve as a central entity in a large cluster composed of hundreds or thousands of nodes as well as it could fit in scenarios with fixed resources such as in edge computing and IoT systems. Moreover, a tool such as K3s [34] can deploy an optimised Kubernetes cluster on a single node with limited resources. These are the minimum requirements to run our software and, clearly, the resources can be scaled as needed.

5. Related Work

Many works have been published on different aspects of the QKD integration in specific environments or regarding the management of QKD networks. For the scope of this paper, it is interesting to briefly discuss some papers that propose routing protocols for QKD and other attempts to integrate QKD in specific domains. The authors of [24] propose a routing protocol based on a custom version of OSPFv2. In this case, the cost function heavily relies on the key material available on each QKD link. This avoids having paths without the minimum number of keys required, but it does not consider congestion or other network peculiarities. The authors of [22] also proposed a modified version of OSPFv2, though they trace the traffic load on the various links and the cost function depends on that. Another work, discussed in [21,35] and regarding the Chinese HCW QKD network, proposes the use of OSPF in combination with a *Quantum Key Reservation Approach (QKRA)* based on the IntServ model. The OSPF protocol finds the shortest path between source and destination, and then the source sends a request of key reservation to all the nodes within the path. Even in this case, they assure that the key material is sufficient for the exchange, but the path may not be optimal. They also pointed out in [21,36] that even an extended version of OSFPv2, including QoS constraints, could not be optimal for QKD networks. In general, as stated in [37], we must also evaluate the impact of a public authenticated channel on the QKD performances. This assertion is true also for the routing protocol. For example, suppose we use QKD generated keys for the authentication and the integrity of the public channel. In that case, we should also optimise their consumption to be efficient regarding the number of packets exchanged for routing purposes.

Other works target QKD integration in various domains. It is worth mentioning the work of [38] related to the SD-QKD and introduced in Section 2.3. The authors, in that case, proposed a new programmable software network architecture based on SDN to integrate QKD in network operator infrastructures. A similar work was proposed in [39] regarding the specific domain of NFV. In this case, the idea was to manage QKD systems using an SDN controller which instructs optical switches on how to forward the traffic of the quantum channel. In this case, the NFV orchestrator was in charge of both managing the SDN controller and requesting key material to the key server. In contrast with the previous ones, our work tries to generalise the problem of the integration of QKD in software-defined infrastructures by proposing a software stack that could be easily integrated into wide adopted technologies.

6. Test and Validation

In order to validate our solution, we estimated the throughput and the exchange time in both PTP and MH case scenarios. We also evaluated the routing algorithm besides the QSS infrastructure. For the tests regarding the key exchange, we used a scenario with three Kubernetes clusters deployed on three different physical nodes with the following characteristics: CPU Intel Core i5-5300U 2.30 GHz (marketed by Intel Corporation, Santa Clara, California, U.S.), 16 GB of RAM DDR3 1600 MHz (marketed by Kingston Technology Corporation, Fountain Valley, California, U.S.), Ubuntu 20.04.3 LTS, K3s version 1.22.5 and containerd v1.5.8 as the container runtime. We used a single virtual machine to test the routing algorithm where the QKD network nodes were simulated as Docker containers. The spawned VM had the following characteristics: 16 vCPUs, 40 GB of RAM, Ubuntu 18.04.5 LTS, and Docker CE version 20.10.5.

6.1. Key Exchange Time and Throughput

We started the tests with some assumptions regarding the exchange rate at the QKDM/QKD device levels. For these tests, we used a special-purpose QKDM which allows exchanging the key at a specific rate between QKD nodes. This is useful because we could change this rate and analyse the system's behaviour in different conditions. These tests, focusing on the QSS at QKS and QSS operator level, are entirely agnostic to the underlying devices. Eventually, as future work, they could be enhanced using a QKD simulator as in [3]. Since we could set the low-level exchange rate between devices arbitrarily (to values far beyond the capability of the systems—up to 2 Mbit/s) we started attesting the key exchange capacity of the QKDMs. We immediately noticed that the bottleneck of this process is the Vault. Indeed, given the Vault key size configuration at a value of 128 bits and the fact that to store a key we need on average 5.6 ms, we could exchange a maximum number of 178 keys/s from a QKDM to another one, that is a throughput of 22.8 kbit/s. Even if this value is compliant with the current QKD devices, clearly it has to be enhanced in order to support future and more advanced systems. To enhance the performance, we could act on two aspects: replicating the Vault instance, since it is running in a pod, or expanding the size of the keys. The first approach has certain limits since Vault is a resource-demanding application. The second approach is feasible and straightforward and could be simply be adopted by changing some parameters in our solution. We decided to adopt this key size because we wanted to avoid wasting key material, so we chose the minimum reasonable amount of bits to represent one key and compose longer keys. Summarising, the primary assumption for all the following tests is that the maximum low-level throughput is of 22.8 kbit/s and keys exchanged among QKDMs have a length of 128 bits.

In our first test (Figure 8), we collected data regarding the execution time related to the application of a `KeyRequest` type resource. This takes into account the time for the QKS to perform the getKey operation, which is the greediest in terms of time requested, and all other operations performed by the QSS operator that are labelled as "QSS operator time" (Figure 7). Varying the number of keys requested and the length of those keys, we

observe an expected growth in terms of time required for the QKS to perform the `getKey`, while the time of the operator remains constant. Even if all the required keys are already available in Vault, we need to consider that the `getKey` operation has to perform the KRP according to the number of keys requested and then extract those keys from Vault, whose time depends on both the number and length of the required keys. We covered the part of the optimisation regarding the length of keys in Vault, yet there is the necessity to tackle the multiple requests factor. In order to optimise that aspect, we already adopted the asynchronous approach (Section 3.1); indeed, the performance of the system, even considering the QKDM bottleneck, the fact that we used only one worker in Hypercorn for the tests, and only one replica of the QKS in Kubernetes, are already satisfying, i.e., for 100 keys of length 1024 the throughput is approximately 56 kbit/s. Since our system is modular and scalable, we could adjust the parameters above (e.g., Hypercorn workers, QKS replicas) according to the coming evolution of QKD devices and reach throughput far beyond the one reported in this paper.

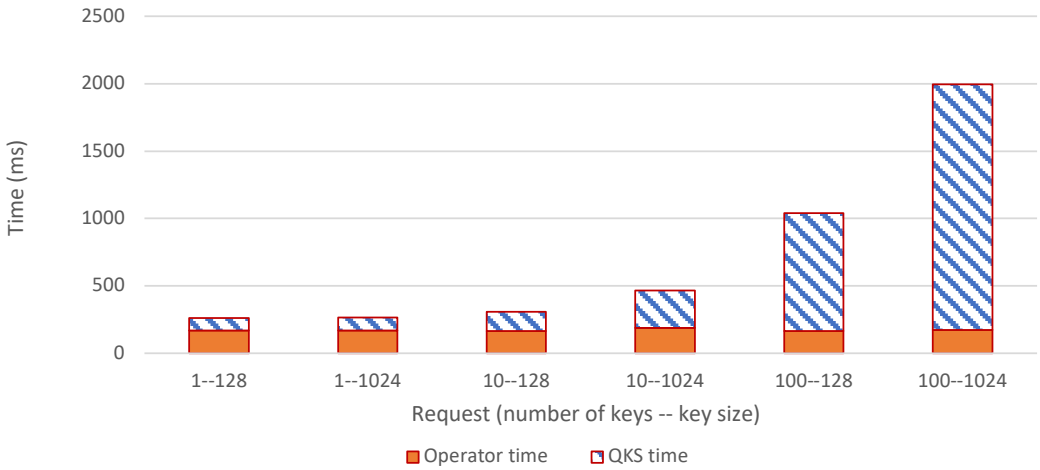

Figure 8. Execution time to handle a `KeyRequest` resource type.

The second test, as depicted in Figure 9, shows the success rate of the `KeyRequest` depending on the underlying exchange rate. In order to perform the tests, we set first the throughput of the special purpose QKDM to 2 kbit/s, then to 1 kbit/s and varying the rate of the requests we tested the resilience of our solution. As we expected, we see in Figure 9 that using a 2 kbit/s low-level exchange and requesting 1 kbit/s of key material leads to 100% success in key exchanges. Maintaining this exchange rate and changing the low-level exchange rate to 1 kbit/s already leads to a success rate lower than 100%. Indeed, this happens with a certain probability because of some delay in producing the required keys. From an operator standpoint, this event results in a failure in retrieving the QKD secret, which could be solved by essentially forwarding the same request again. With the same method, we increased the requests per second, and according to what we expected, the success rate decreased accordingly. The bottom line is that when we vary the exchange and request rate, the system is resilient even if it cannot serve all requests. Besides, it notifies the upper level of the failure, letting the high-level application take consequent actions. When normal conditions are established again, the success rate returns to 100%.

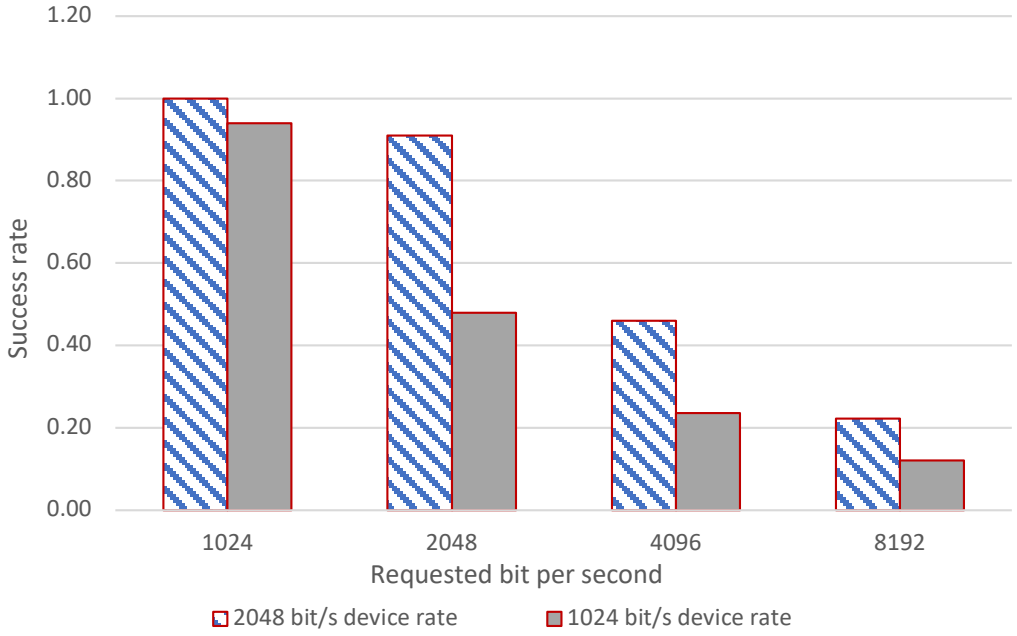

Figure 9. Success rate of the getKey requests to the QKS varying the underlying exchange rate.

As we did for the PTP exchange in the first test, in the third one, we collected data on the execution time for an MH exchange. In Figure 10, the results of this test are reported and they show how the `getKey` and `getKeyWithId` execution time varies depending on the number of hops. We performed these tests in a configuration with three Kubernetes clusters. As we also expected, the results show that the operator time does not change during the various experiments. Even the `getKeyWithId` time does not change because this operation is the same performed by the sSAE, and it does not depend on the number of hop between source and destination. The only time that change is the `getKey` time which increases linearly according to the number of hops. This is due to the chain of key reservations that must be built along with each link of the path.

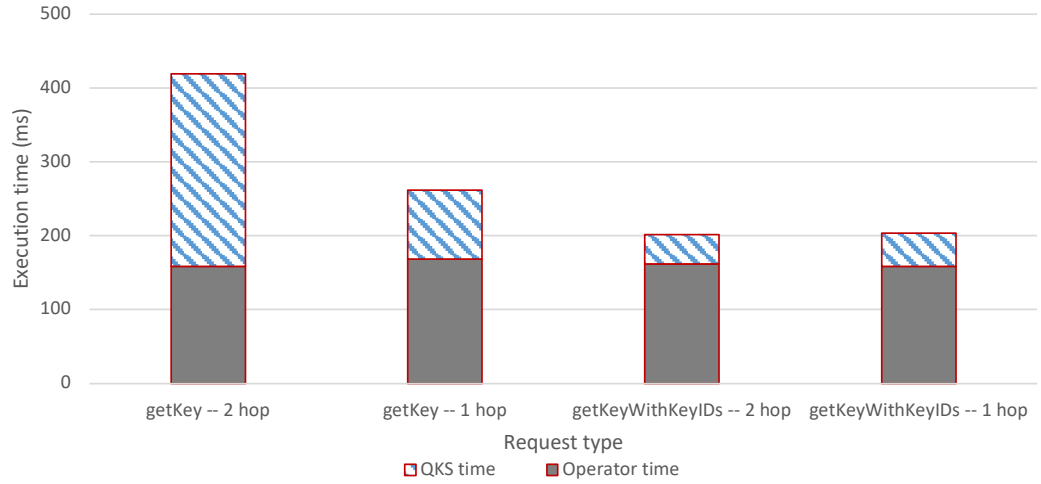

Figure 10. Execution time of the getKey and getKeyWithId functions for Point-to-Point and Multi-hop exchanges.

6.2. Routing

The routing protocol with the underlying algorithm, described in Section 3.3, has been tested as a separate component. This strategy allows us to evaluate the routing protocol in large QKD networks. In order to test its behaviour, we used the virtual machine described at the beginning of Section 6. We did not deploy whole QSSs in this case, but we ran only

the routing module inside a Docker container. Therefore, the QKD network becomes a set of Docker containers running on a specific host and communicating over a virtual TCP/IP network. We simulated a scenario in which events such as SAEs and QKDMs registrations and removals were randomly triggered every 10 s. These events continuously initiated the update process to send LSA packets over the network and update neighbours. The first results proved that in our module, each node sees the same network topology for every topology represented by a connected graph and can reach every other network node.

Moreover, Figure 11 shows the convergence time of the algorithm depending on the number of nodes within the network. We tested it for a network of up to 50 nodes where this time is close to 50 ms. We performed the test on a virtual network, suggesting that in a real scenario, with physical links, the convergence time will grow according to the delay introduced by the network to transmit the packets. In that case, it is essential to update the route invalidation parameter to allow convergence. The computational complexity, according to Dijkstra, is $\mathcal{O}(|E| + |V| \log |V|)$ where $|E|$ is the number of links and $|V|$ is the number of nodes. This may affect the future large QKD networks and require support for different autonomous systems [23].

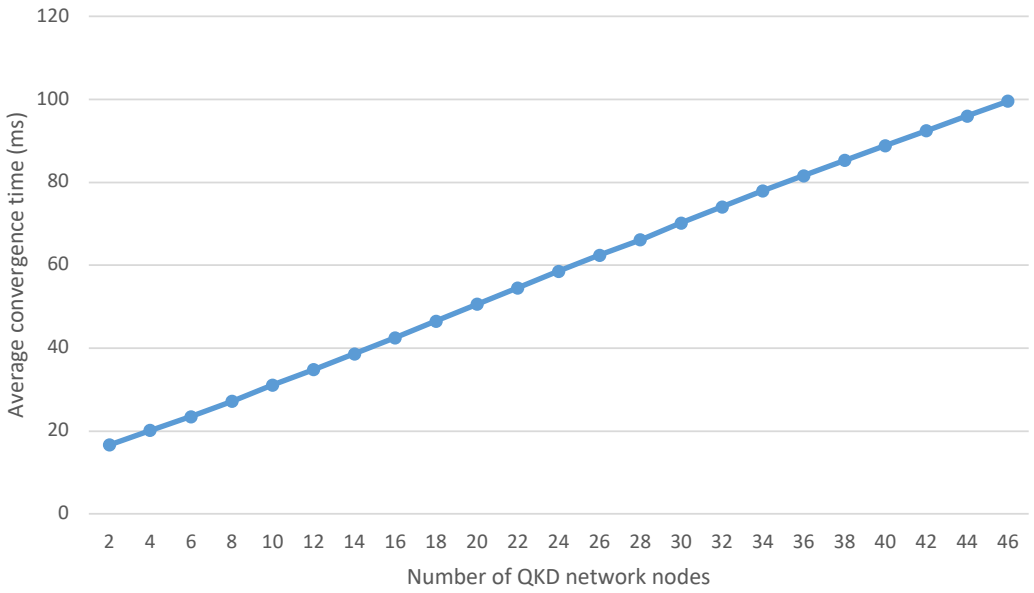

Figure 11. Average convergence time of the routing algorithm depending on the number of nodes in the QKD network.

7. Conclusions

This paper, starting from the work in [3], proposes a new version of QSS that presents multiple enhancements: an increase in performance due to an asynchronous and multiprocess approach, the compliance with multi-hop long-distance QKD using trusted repeaters, and a routing module to select the best path for an MH exchange. Moreover, the solution has been integrated into Kubernetes as an operator, allowing the containerized application to leverage keys exchanged with QKD. This approach is modular, scalable, and almost entirely agnostic to high-level security applications. The source code is publicly available on GitHub [40,41]. Some existing limitations, such as the maximum throughput in key exchange among QKDMs and possible straightforward remediations, have been discussed. Even if this solution is still at its early stage, it is promising and as a future work it could be interesting to test it in actual use case scenarios.

Two of the most relevant outcomes of this work are the fact that this is an entirely open-source vendor-agnostic framework, the first trying to simplify the integration of QKD in Kubernetes, and the possible applications to interesting scenarios such as the blockchain technologies. In future work, it would be interesting to test practical blockchain applications using the QSS operator and a multi-cluster Kubernetes architecture. In addition, targeting

large and complex QKD networks is still a challenging task. In this regard, it would be intriguing to introduce SDN technology to simplify the management of the QKD network apparatuses offloading the routing module. Another relevant aspect for the future of QKD networks will impact the choice of relays: overcoming trusted repeaters in favour of quantum-based relays. Finally, multiple aspects regarding the security of the QSS software itself running on a cluster shall be considered in the future: being compliant with the paradigms of Confidential Computing as well as adopting consistent software protection techniques.

Author Contributions: Conceptualization, I.P. and A.L.; methodology, I.P.; software, I.P.; validation, I.P.; formal analysis, I.P.; investigation, I.P.; resources, I.P. and A.L.; data curation, I.P.; writing—original draft preparation, I.P. and A.L; writing—review and editing, I.P. and A.L.; visualization, I.P. and A.L.; supervision, A.L.; funding acquisition, A.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the European Union's Horizon 2020 Project "CyberSec4Europe" under Grant 830929.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We acknowledge the contribution of Lorenzo Pintore to the implementation of the solution.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AES	Advanced Encryption Standard
API	Application Programming Interface
CR	Custom Resource
CV-QKD	Continuous-Variable QKD
DV-QKD	Discrete-Variable QKD
ECDSA	Elliptic Curve Digital Signature Algorithm
ETSI	European Telecommunications Standards Institute
ETSI ISG	ETSI Industry Specification Group
KRP	Key Reservation Process
KS	Key Stream
GIL	Global Interpreter Lock
IoT	Internet of Things
IPsec	Internet Protocol Security
MDI-QKD	measurement-device-independent QKD
MH	Multi-Hop
mSAE	master SAE
LSA	Link State Advertisement
NFV	Network Functions Virtualisation
NIST	National Institute of Standards and Technology
OSPF	Open Shortest Path First
PQC	Post-Quantum Cryptography
PTP	Point-to-Point
QKD	Quantum Key Distribution
QKDM	QKD Module
QKRA	Quantum Key Reservation Approach
QKS	Quantum Key Server
QoS	Quality of Service

QRP	QKDM Registration Process
QSS	Quantum Software Stack
RM	Routing Module
RSA	Rivest Shamir Adleman
SAE	Secure Application Entity
SD-QKD	Software-Defined QKD
SDN	Software-Defined Networking
SIP	Session Intiation Protocol
sSAE	slave SAE
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLS-PSK	TLS pre-shared key ciphersuites
TF-QKD	twin-field QKD
VPN	Virtual Private Network
YAML	YAML Ain't a Markup Language

References

1. Shor, P.W. Algorithms for quantum computation: Discrete logarithms and factoring. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA, 20–22 November 1994; pp. 124–134. [CrossRef]
2. Chen, L.; Chen, L.; Jordan, S.; Liu, Y.K.; Moody, D.; Peralta, R.; Perlner, R.; Smith-Tone, D. *Report on Post-Quantum Cryptography*; US Department of Commerce, National Institute of Standards and Technology: Washington, DC, USA, 2016; Volume 12. [CrossRef]
3. Pedone, I.; Atzeni, A.; Canavese, D.; Lioy, A. Toward a Complete Software Stack to Integrate Quantum Key Distribution in a Cloud Environment. *IEEE Access* **2021**, *9*, 115270–115291. [CrossRef]
4. Kubernetes Project. Available online: https://kubernetes.io (accessed on 7 April 2022).
5. Xu, F.; Ma, X.; Zhang, Q.; Lo, H.K.; Pan, J.W. Secure quantum key distribution with realistic devices. *Rev. Mod. Phys.* **2020**, *92*, 025002. [CrossRef]
6. Pirandola, S.; Andersen, U.L.; Banchi, L.; Berta, M.; Bunandar, D.; Colbeck, R.; Englund, D.; Gehring, T.; Lupo, C.; Ottaviani, C.; et al. Advances in quantum cryptography. *Adv. Opt. Photonics* **2020**, *12*, 1012–1236. [CrossRef]
7. Lucamarini, M.; Patel, K.; Dynes, J.; Fröhlich, B.; Sharpe, A.; Dixon, A.; Yuan, Z.; Penty, R.; Shields, A. Efficient decoy-state quantum key distribution with quantified security. *Opt. Express* **2013**, *21*, 24550–24565. [CrossRef] [PubMed]
8. ID Quantique Manufacturer. Available online: https://www.idquantique.com (accessed on 7 April 2022).
9. Cerberis XG QKD System. Available online: https://www.idquantique.com/quantum-safe-security/products/cerberis-xg-qkd-system/ (accessed on 7 April 2022).
10. Toshiba QKD systems. Available online: https://www.toshiba.co.jp/qkd/en/products.htm (accessed on 7 April 2022).
11. Lucamarini, M.; Yuan, Z.L.; Dynes, J.F.; Shields, A.J. Overcoming the rate-distance limit of quantum key distribution without quantum repeaters. *Nature* **2018**, *557*, 400–403. [CrossRef] [PubMed]
12. Liu, Y.; Chen, T.Y.; Wang, L.J.; Liang, H.; Shentu, G.L.; Wang, J.; Cui, K.; Yin, H.L.; Liu, N.L.; Li, L.; et al. Experimental measurement-device-independent quantum key distribution. *Phys. Rev. Lett.* **2013**, *111*, 130502. [CrossRef] [PubMed]
13. ETSI QKD Standards. Available online: https://www.etsi.org/committee/qkd (accessed on 7 April 2022).
14. Keycloak Solution. Available online: https://www.keycloak.org (accessed on 7 April 2022).
15. *Quantum Key Distribution (QKD): Application Interface*; Technical Report; European Telecommunications Standards Institute (ETSI): Nice, France, 2020.
16. *Quantum Key Distribution (QKD): Protocol and Data Format of REST-Based Key Delivery API*; Technical Report; European Telecommunications Standards Institute (ETSI): Nice, France, 2019.
17. *Quantum Key Distribution (QKD): Control Interface for Software Defined Networks*; Technical report; European Telecommunications Standards Institute (ETSI): Nice, France, 2021.
18. Global Interpreter Lock Documentation. Available online: https://wiki.python.org/moin/GlobalInterpreterLock (accessed on 7 April 2022).
19. Quart Project. Available online: https://gitlab.com/pgjones/quart (accessed on 7 April 2022).
20. Hypercorn Project. Available online: https://gitlab.com/pgjones/hypercorn (accessed on 7 April 2022).
21. Mehic, M.; Niemiec, M.; Rass, S.; Ma, J.; Peev, M.; Aguado, A.; Martin, V.; Schauer, S.; Poppe, A.; Pacher, C.; et al. Quantum key distribution: A networking perspective. *ACM Comput. Surv.* **2020**, *53*, 1–41. [CrossRef]
22. Peev, M.; Pacher, C.; Alléaume, R.; Barreiro, C.; Bouda, J.; Boxleitner, W.; Debuisschert, T.; Diamanti, E.; Dianati, M.; Dynes, J.; et al. The SECOQC quantum key distribution network in Vienna. *New J. Phys.* **2009**, *11*, 075001. [CrossRef]
23. Moy, J. *OSPF Version 2*; RFC 2328, IETF; Ascend Communications, Inc.: Alameda, CA, USA, 1998. [CrossRef]
24. Elliott, C.; Colvin, A.; Pearson, D.; Pikalo, O.; Schlafer, J.; Yeh, H. Current status of the DARPA quantum network. In Proceedings of the Quantum Information and computation III. International Society for Optics and Photonics, Orlando, FL, USA, 25 May 2005; Volume 5815, pp. 138–149. [CrossRef]

25. Maurhart, O. QKD networks based on Q3P. In *Applied Quantum Cryptography*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 151–171. [CrossRef]
26. Etcd Project. Available online: https://etcd.io (accessed on 7 April 2022).
27. Sasaki, M. Quantum Key Distribution and Its Applications. *IEEE Secur. Priv.* **2018**, *16*, 42–48. [CrossRef]
28. Conrad, A.; Isaac, S.; Cochran, R.; Sanchez-Rosales, D.; Wilens, B.; Gutha, A.; Rezaei, T.; Gauthier, D.J.; Kwiat, P. Drone-based quantum key distribution: QKD. In *Proceedings of the Free-Space Laser Communications XXXIII*; International Society for Optics and Photonics: Bellingham, WA, USA, 2021; Volume 11678, p. 116780X. [CrossRef]
29. Istio Project. Available online: https://istio.io (accessed on 7 April 2022).
30. Zhang, P.; Wang, L.; Wang, W.; Fu, K.; Wang, J. A blockchain system based on quantum-resistant digital signature. *Secur. Commun. Netw.* **2021**, *2021*. [CrossRef]
31. Ikeda, K. qBitcoin: A peer-to-peer quantum cash system. In *Proceedings of the Science and Information Conference*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 763–771. [CrossRef]
32. Allende, M.; León, D.L.; Cerón, S.; Leal, A.; Pareja, A.; Da Silva, M.; Pardo, A.; Jones, D.; Worrall, D.; Merriman, B.; et al. Quantum-resistance in blockchain networks. *arXiv* **2021**, arXiv:2106.06640. https://doi.org/10.48550/arXiv.2106.06640.
33. Hyperledger Project. Available online: https://www.hyperledger.org (accessed on 7 April 2022).
34. K3s Project. Available online: https://k3s.io (accessed on 7 April 2022).
35. Cheng, X.; Sun, Y.; Ji, Y. A QoS-supported scheme for quantum key distribution. In Proceedings of the 2011 International Conference on Advanced Intelligence and Awareness Internet (AIAI 2011), Shenzhen, China, 28–30 October 2011; pp. 220–224. [CrossRef]
36. Apostolopoulos, G.; Guerin, R.; Kamat, S. Implementation and performance measurements of QoS routing extensions to OSPF. In Proceedings of the IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320), New York, NY, USA, 21–25 March 1999; Volume 2, pp. 680–688. [CrossRef]
37. Mehic, M.; Maurhart, O.; Rass, S.; Komosny, D.; Rezac, F.; Voznak, M. Analysis of the public channel of quantum key distribution link. *IEEE J. Quantum Electron.* **2017**, *53*, 17140300. [CrossRef]
38. Aguado, A.; Lopez, V.; Lopez, D.; Peev, M.; Poppe, A.; Pastor, A.; Folgueira, J.; Martin, V. The engineering of software-defined quantum key distribution networks. *IEEE Commun. Mag.* **2019**, *57*, 20–26. [CrossRef]
39. Aguado, A.; Hugues-Salas, E.; Haigh, P.A.; Marhuenda, J.; Price, A.B.; Sibson, P.; Kennard, J.E.; Erven, C.; Rarity, J.G.; Thompson, M.G.; et al. Secure NFV orchestration over an SDN-controlled optical network with time-shared quantum key distribution resources. *J. Light. Technol.* **2017**, *35*, 1357–1362. [CrossRef]
40. QSS Source Code. Available online: https://github.com/ignaziopedone/qkd-keyserver (accessed on 7 April 2022).
41. QKDM Source Code. Available online: https://github.com/ignaziopedone/qkd-module (accessed on 7 April 2022).