POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Quantum Key Distribution in softwarised infrastructures

Supervisors
prof. Antonio Lioy
dott. Ignazio Pedone

Candidate
Lorenzo Pintore

December 2021

A mio padre

Contents

Chapter 1

Introduction

Quantum computing (QC) is one of the main concerns of information security in recent years. With its different computational logic to classical computers, it carries the capability of breaking the most used cryptographic algorithms exposing the entire IT infrastructure to threats.

Asymmetric cryptography (also known as public-key cryptography) is one of the building blocks of our systems: it is widely used for encryption, digital signature and key agreement algorithms (e.g. RSA [1], DSA [2] and ECDH [3]) and these algorithms are embedded in the most widely used protocols in internet communication (as TLS [4]). These cryptosystems rely on the assumption that some problems, such as prime factorization and discrete logarithms problems, are complex to solve in a reasonable amount of time using conventional computing: they were known to be secure thanks to this "computational" security and the lack of an efficient known algorithm able to break them. Quantum computing leverages quantum physics, providing a completely different environment and as consequence new algorithms capable of solving hard problems in polynomial time (e.g. Shor's algorithm for prime factorization [5]). So far there are not quantum computers powerful enough to be a real threat for those cryptosystems but research is proceeding faster and faster with a lot of contributors such as Google[1] and IBM[2], pushing up the state of the art. Discussions on breaking time for some algorithms have already been performed [6], reaching a result of 8 hours to factor 2048 bit RSA. Alternative solutions to classical cryptosystems are needed to overcome this threat.

So far two different approaches to reach *"quantum-safe"* protocols have been proposed: Post-quantum Cryptography (PQC) and Quantum Cryptography. The PQC idea is to use classical cryptographic techniques and to develop new quantum-resistant algorithms to replace weak ones. The main advantage of this solution is that it is compatible with the infrastructures already in place that will only require an update for the supported algorithms without affecting the upper structure. On the other hand, its biggest disadvantage is being only computationally secure: it still relies on the assumption that attacker's resources are limited, thus these systems will be secure only until new algorithms able to break them will be found. Several PQC algorithms have been proposed both for key exchange and for digital signature functionalities (e.g. NTRU[3] and SPHINCS+[4]); and some of them have already been implemented successfully in some existing protocols (as Google did with TLS [5]). Quantum cryptography instead takes advantage of quantum mechanics laws to build a cryptosystem that doesn't rely on assumption about attacker's resources to establish its security. Quantum Key Distribution (QKD), which is the most promising application, allows to securely share secret keys among peers. It prevents interception of messages to

[1]https://research.google/research-areas/quantum-computing/

[2]https://www.research.ibm.com/quantum-computing/

[3]https://ntru.org/

[4]https://sphincs.org/

[5]https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html

any eavesdropper and is proven information-theoretical secure [7]: it provides long term security despite the advances in classical or quantum computing. Even if experts account QC to become a real threat in a few decades [6], migrating from current cryptosystems to quantum-safe ones is neither a fast nor easy task: procedures to develop new standards have already started both for PQC algorithms from the National Institute of Standards and Technology (NIST) [7] and for QKD devices and APIs from European Telecommunications Standards Institute [8] (ETSI).

Cloud computing is a paradigm that is growing extremely fast and its adoption is spreading in several fields of the IT world: a lot of old infrastructures are currently migrating to this new paradigm that can ensure scalability and can dynamically adapt to the user needs, reducing the costs and increasing the flexibility. It shifts the infrastructure from private data centers to shared ones, often sparse over different physical nodes and even in different locations. The security aspect of the communications among instances is an emerging concern that requires policies and protocols to protect them.

This work proposes a QKD software stack for key management able to deliver keys from the physical devices to high-level applications. It is compliant with the ETSI standard in terms of interfaces and communication protocols, guaranteeing wide compatibility with devices. It has been designed embracing the microservices pattern to work in a cloud-native environment, to prove that QKD can be integrated into already deployed scenarios. Despite a first solution has already been proposed by Pedone *et al.* [8], this new version overcomes its issues about speed and reliability, introducing a new asynchronous programming pattern and faster technologies. Moreover, it now supports long-distance key exchanges through intermediate nodes (e.g. *trusted repeaters*) in a hop-by-hop manner. Routing functionalities have been implemented adapting communication routing techniques to the QKD networks. Another relevant contribution is the integration of the stack in a Kubernetes cluster. Kubernetes is nowadays one of the most used orchestrators for containerized environments, providing the possibility to integrate the software into other applications. The *operator* pattern has been used to simplify the stack integration in the cluster automating the key retrieval process. Tests performed showed that the stack adds an overhead compatible with the current communication protocols, that it can support current QKD device rates without issues. The QKS proved to be able to work consistently in networks composed of dozens of nodes.

[6]https://globalriskinstitute.org/publications/quantum-threat-timeline-report-2020/

[7]https://csrc.nist.gov/projects/post-quantum-cryptography

[8]https://www.etsi.org/committee/qkd

Chapter 2

Quantum Key Distribution

Quantum Key Distribution is the most relevant application of QC in information security. It allows two peers to exchange keys securely, replacing the need for a public key cryptography infrastructure. Qubits are sent over a quantum channel from a peer to another, carrying information about each bit of the key in their polarization state: at the end of the channel they are measured and the information is decoded. QKD takes advantage of quantum mechanics laws to guarantee that the presence of an eavesdropper during the key exchange will be spotted thanks to the no-cloning principle and the Heisenberg uncertainty principle. Heisenberg principle ensures that any qubit can't have its unknown quantum state measured without forcing it to collapse to a certain state: a third party can't look at a qubit without modifying it. The no-cloning principle [9] ensures that it is impossible to make a perfect copy of a qubit in an unknown quantum state: there is not any possibility for an eavesdropper to store qubits for following attempts to gain information without being spotted. These two principles combined allows two peers to securely send qubits in clear without the need to encrypt them in any way. As Shannon's theory [10] said a theoretical secure communication is possible if and only if a one-time pad cryptosystem is used and the key has the same length as the message, each key is used only once and the key is random. QKD meets the first two requirements and easy to build quantum random number generators [11] allows meeting the third one. With this principle in mind is clear that QKD key exchange security completely depends on the implementation of quantum devices and the security of the upper application only depends on how the key is used.

2.1 Channels and devices

As already said QKD protocols require different channels to establish secret keys: both quantum channels and classical ones. Classical channels are required to transmit information on how to measure qubits after their transmission and to check about interference presence. They don't require encryption of transmitted information but they require authentication to avoid man-in-the-middle attacks from an eavesdropper. Here is where PQC needs to be integrated with QKD: standard non-quantum resistant authentication techniques would rapidly become unusable and algorithms able to provide secure authentication are required.

Quantum channels are used to transmit qubits, encoded as photons, between peers. Transmission can be performed both through optical fiber cables both through free space (as between satellites), with different performances in bitrate, maximum distance and error rate. The way light is treated and qubit encoded leads to two different approaches: Discrete Variable QKD (DV-QKD) if each qubit is encoded into a single photon emitted with very weak laser pulses, granting higher transmission rate; Continuous Variable QKD if qubits are encoded into the amplitude or the phase of light waves [12].

In DV-QKD the information carried by each photon is often described as its polarization over one of two orthonormal bases. Quantum channels of this type are composed of a laser emitter to produce photons, an encoder device to polarize qubits over a basis, a decoder at the other end to perform the same process and a photon detector that transforms polarization data into electrical signals. Decoding qubits means measuring their polarization against a basis. If the chosen basis for the measurement is not the one the photon is polarized over the detection results will be completely random. This principle is what allows to spot the presence of any intruder: if he tries to measure qubits randomly guessing the correct basis it will introduce a perturbation in the system. QRNG are used to randomly select bases for measurement. A simple QKD system is described in figure 2.1: it is possible to observe at the sender side both the photon emitter (the signal source) and the encoder, both connected to the QRNG and at the receiver the decoder connected to two detectors (one for each basis). At both ends, the control electronics are responsible for managing different protocol phases. Observe that also the classical channel can be implemented with optical fiber.

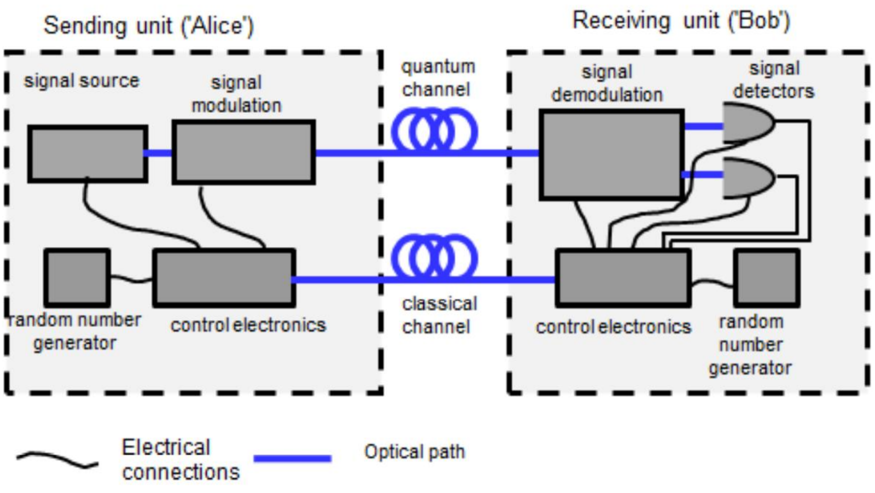


Figure 2.1. Simple QKD system (source: ETSI GR QKD 003).

2.2 Errors and privacy amplification

Even if theoretically QKD can provide perfect security [7] its real devices are not safe from errors: polarization defects, decoding errors, loss due to distance and interferences. Peers cannot understand if errors during transmission are due to this type of physical problem or generated by a third party intruder who wants to steal information. Quantum Bit Error Rate (QBER) is the parameter used to track these errors, whatever their cause is. If QBER exceeds a certain threshold (11% is the value often used) the exchange is considered insecure and peers can decide to discard the key and repeat the process. If the error rate is below that an error correction procedure is applied to reach a common set of bits. To reduce any information gained by the eavesdropper privacy amplification techniques should be implemented after the qubit exchange phase: starting from the received set of bits hashing functions are used by both peers to produce a more robust key (even if shorter) and without any correlation with the original one [13].

2.3 QKD protocols

Since the publication of the first QKD protocol in 1948 a lot of others have been proposed. QKD protocols can be divided into two families: *"prepare and measure"* and *"entanglement based"*. In the former category, photons are encoded by the sender in given states and the receiver should measure them with the same basis to obtain the correct information. In the latter two photons are put into an entanglement relationship before transmission: thanks to this relationship photons are bound to each other and any measurement on one particle will affect the other despite the distance.

Table 2.1 report a list of currently developed QKD protocols [14]:

Protocol name	**Type**	**Family**
BB84	DV	prepare and measure
E91	DV	entanglement
B92	DV	prepare and measure
BBM92	DV	prepare and measure
SSP	DV	prepare and measure
SARG04	DV	prepare and measure
T12	DV	prepare and measure
MSZ96	CV	prepare and measure
COW	CV	prepare and measure
DPS	DV	prepare and measure
KMB09	DV	prepare and measure
AE17	DV	entanglement

Table 2.1. Current QKD protocols. Source: [14]

2.3.1 BB84

BB84 is the first developed QKD protocol, proposed by Charles H. Bennett and Gilles Brassard in 1984 [15]. It is the simplest protocol from which many others in the previous list derive. It is a discrete variable *prepare and measure* protocol and exploits photons polarization to encode information and the no-cloning theorem to ensure security.

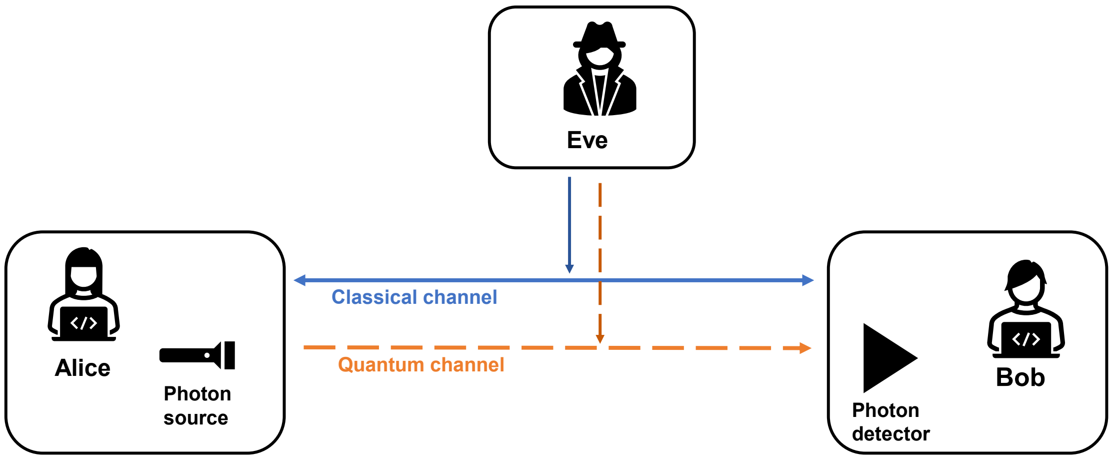


Figure 2.2. BB84 schema.

This section describes how the BB84 protocol work. Two peers, the sender (Alice) and the receiver (Bob), wants to exchange a secret key, an eavesdropper (Eve) tries to steal information. Eve has access both to the quantum channel both to the authenticated (and not encrypted) classical channel 2.2. The protocol requires a set of two orthonormal bases for polarization and an encoding rule: often the chosen basis are the rectilinear and the diagonal one, and the four polarization states are 0°, 45°, 90° and 135° to the horizontal axis, but any other set of basis and encoding is possible.

In the first phase, Alice generates a random key of the desired length and encodes each bit over one randomly chosen basis; then she sends qubits over the channel. After receiving a qubit Bob immediately measures it over one randomly chosen basis and repeat the process for each one, due to the technical difficulties of reliably storing quantum information, then Bob communicates the

Basis / bits	0	1
$+$	$\rightarrow$	$\uparrow$
$\times$	$\searrow$	$\nearrow$

Table 2.2. Encoding schema

set of used bases over the classical authenticated channel. The next phase is the sifting procedure: chosen bases are compared bit by bit and if they used the same basis that bit is kept, otherwise if Bob chose a different basis the obtained result is random and therefore that bit must be discarded. The remaining bits are known as *"sifted key"* An example of these phases is reported in table 2.3.

Alice's key	1	1	0	0	0	1	0	0
Alice's bases	$\times$	$+$	$+$	$\times$	$+$	$\times$	$\times$	$+$
photons	$\nearrow$	$\uparrow$	$\rightarrow$	$\searrow$	$\rightarrow$	$\nearrow$	$\searrow$	$\rightarrow$
Bob's bases	$+$	$+$	$\times$	$\times$	$+$	$\times$	$+$	$\times$
Bob's measurement	$\rightarrow$	$\uparrow$	$\nearrow$	$\searrow$	$\rightarrow$	$\nearrow$	$\uparrow$	$\searrow$
Bases match	✗	✓	✗	✓	✓	✓	✗	✗
classical channel sifting procedure								
Sifted key	-	1	-	0	0	1	-	-

Table 2.3. BB84 exchange and sifting procedure

Sifted keys are then randomly sampled and compared over the classical channel to compute the QBER: differences in samples are due to transmission errors or introduced by Eve's actions. If QBER is higher than the threshold the key is considered not sufficiently secure and therefore discarded and the protocol restarts, otherwise they continue with the error correction process to obtain two equal keys and apply privacy amplification techniques.

If Eve tries to intercept and resend photons, measuring them over a random guessed basis, she modifies them. If we consider only bits where Alice and Bob used the same basis, Eve will choose the correct one only half of the times, passing them to Bob who now has a 50% chance of getting the wrong result even if he used the same basis as Alice. This 25% QBER introduced if Eve intercepts every photon is easily detectable in the sampling phase, leading to a restart in the process. Due to randomly chosen bases errors should be evenly distributed over a sufficiently long key: thanks to this fact also sampling results should match the QBER of the entire key. An example with Eve is reported in table 2.4.

Differences in the base choice reduce the number of usable bits: when the key length is established the number of bits to send should consider this reduction factor.

2.4 QKD criticalities and attacks

Quantum key distribution has the capability of providing theoretically proven security [7] but its implementations are not free from defects and flaws that can lead to attacks and have issues with real/world uses. Some of these attacks and criticalities are described in this section.

2.4.1 Classical channel

As seen in previous sections all currently developed QKD protocols require both a quantum and a classical channel and the latter must be an authenticated one. Authentication ensures that the two peers who are performing the exchange can be sure that messages come from the other peer: if this doesn't happen it's easy for an eavesdropper who can read the channel to perform a man-in-the-middle (MITM) attack, replacing data with other without being noticed. In a MITM attack, in a scenario where Alice and Bob want to securely exchange a key, if the eavesdropper

Alice's key	1	1	0	0	0	1	0	0
Alice's bases	×	+	+	×	+	×	×	+
photons	↗	↑	→	↘	→	↗	↘	→
Eve's bases	+	×	+	×	×	+	×	+
Eve's measurement	↑	↗	→	↘	↗	↑	↘	→
Bob's bases	+	+	×	×	+	×	+	×
Bob's measurement	↑	→	↗	↘	↑	↗	→	↘
Bases match	✗	✓	✗	✓	✓	✓	✗	✗
classical channel sifting procedure								
Bob's sifted key	-	0	-	0	1	1	-	-
Error	-	✓	-	✗	✓	✗	-	-

Table 2.4. BB84 exchange and sifting procedure with eavesdropper.

Eve pretends to be Bob, without an authentication schema Alice can not distinguish between her and Bob and the same happens from Bob's point of view. When Alice sends a message to Bob she encrypts it with her key, which is a completely valid key but in reality, shared with Eve (key_{AE}), Eve can decrypt it, read it and then forward it to Bob encrypting the message with Bob's key (key_{BE}) without being noticed not by Alice not by Bob who thinks they are communicating securely.

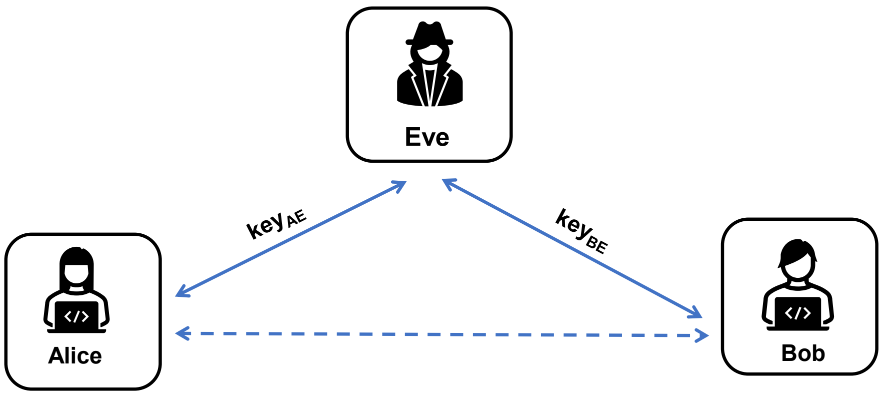


Figure 2.3. Man-in-the-middle attack schema.

In addition to authentication also integrity is required: peers need to know that transmitted data have not been modified, otherwise Eve can modify messages even if she can not decrypt them, altering the communication. Common authentication schemas (which also grant integrity) relies on public-key cryptosystems, with the signer that uses its private key to perform the signature and allowing the receiver to check it with the public one. Their problematics in a quantum environment have already been discussed: they can't provide security anymore and therefore it's required to substitute them with PQC algorithms that are quantum-safe. It is important to say that authentication needs to be verified only during the protocol execution: if the security gets broken after protocol completion it is not an issue because the eavesdropper can't steal any information about the key.

2.4.2 Quantum attacks

Any real device is not a perfect implementation of the theoretical one, therefore its behaviour can divert from what is expected: this can lead to errors which can reduce protocols efficiency but can also expose to attacks. A lot of them have been studied, exploiting defects in multiple components. Some of them exploit photons detectors, which have a narrow window of detection to intercept photons forwarding them later breaking the required synchronization between emitter and detector (time-shift attack), others exploit defects in photos emitter which sometimes produce more than one photon allowing the eavesdropper to intercept and measure one of them without modifying the other (photon-number-splitting attack) and others exploit the required calibration procedure of the quantum channel (channel calibration attack) [13]. A list of currently known attacks is reported in table 2.5

Attack name	**Target**	**Year**
Photon-number-splitting	Source	2000
Detector fluorescence	Detection	2001
Faked-state	Detection	2005
Trojan horse	Source and Detection	2006
Time shift	Detection	2007
Time side-channel	Detection	2007
Phase remapping	Source	2010
Detector blinding	Detection	2010
Detector control	Detection	2011
Faraday mirror	Source	2011
Wavelength	Detection	2011
Dead-time	Detection	2011
Channel calibration	Detection	2011
Intensity	Source	2012
Phase information	Source	2012
Memory attacks	Detection	2013
Local oscillator	Detection	2013
Laser damage	Detection	2014
Laser seeding	Source	2015
Spatial mismatch	Detection	2015
Detector saturation	Detection	2016
Covert channels	Detection	2017
Pattern effect	Source	2018
Polarization shift	Detection	2019

Table 2.5. Know quantum hacking attacks. Source: [13]

2.4.3 Real implementation criticalities

Today biggest limitations of QKD implementation are not related to security problems but more to the technical difficulties that it introduced. Even if some companies as MagiQ[1], ID Quantique[2], Toshiba[3] and many others have already produced some functioning QKD systems over quantum fiber for the market they are not cheap and not highly available. Even if the research progresses year by year quantum channels enforce limitations in the distance between peers: optical fiber links can reach a maximum length of about 100 km while free-space channels require a visible light path and good atmospheric conditions. Even with short channels and reduced losses the average key exchange rate over a link is not very high, reaching a few hundreds of kbit/s [16] only in the most recent implementations. Moreover, QKD protocols only allow a point-to-point (PTP): this is a big limitation in a real environment where PTP links are not sufficient and are also unfeasible to build between each couple of peers who wants to securely communicate.

Due to these reasons, quantum key distribution networks, which allows communication over multiple peers and over longer distances are required: their implementation, defects and currently active projects will be described in the next chapter.

[1] https://www.magiqtech.com/

[2] https://www.idquantique.com/

[3] https://www.toshiba.co.jp/qkd/en/

Chapter 3

QKD networks and initiatives

QKD networks allow to overcome the limitation imposed by single point-to-point QKD links and to extend the communication range by building an infrastructure composed of several interconnected nodes and multiple links among them. This chapter analyzes their implementation, then it describes some real use cases that can take advantage of them and in the end discusses currently deployed networks.

3.1 QKD network implementation

QKD networks are composed not only of QKD devices and quantum channels but also more general components to solve other functions are required. They can be described as a composition of several layers [17], where each one has its own specific set of responsibilities:

- the *Quantum Layer*, build from QKD devices and quantum channels, responsible only for key exchange;
- the *Management Layer*, build from nodes (classical computer considered secure and trusted) and classical communication channels, to solve tasks as routing, ensuring quality of service (QoS) and managing key storage;
- the *Communication Layer*, where exchanged keys are used by applications to perform encryption and exchange data securely.

It is important to say that to use resources in the most efficient way quantum devices have to perform key exchanges in a no-stop manner: delegating nodes to securely store key material and avoiding waste of time allows to mitigate limits in key-exchange rate.

3.1.1 QKD Network types

As it happens in classic networks with data, qubits need to travel over several links before reaching their destination, in a hop-by-hop manner. There are two ways to implement this process: *optical switches* and *trusted repeaters*. In switched networks nodes are interconnected via a completely optical network with some optical switches that allow creating new optical paths establishing required point-to-point QKD connections between each peers couple. To allow links to be established between each couple of nodes the latter can act both as the sender and as the receiver in the QKD protocol. Even if this solution allows to solve the problem of single PTP links, adding optical switches causes losses in signals, intensifies distance limitations and reduces network range [18]. Moreover, these devices can only manage fiber connections and there are not switches able to commute from free-space signals to fiber ones, therefore it is not possible to combine these two technologies.

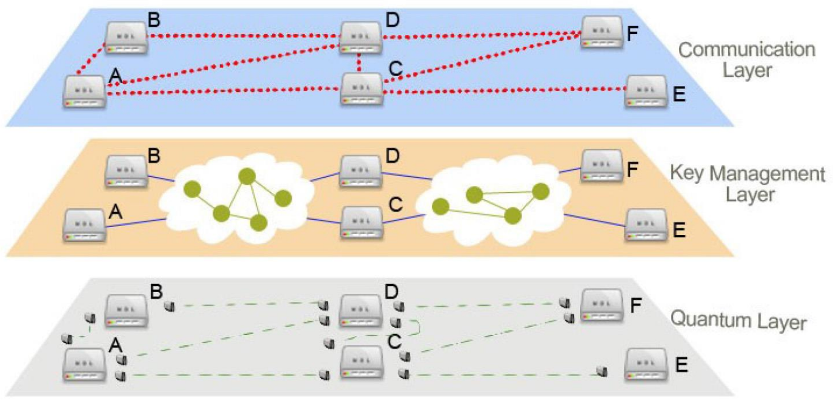


Figure 3.1. QKD network layers. Source: [17]

In trusted repeater networks communication are established between an optically connected couple of peers, and data are forwarded hop-by-hop to the destination by nodes themselves. This solution does not have any limitation in distance or implementation technology because qubits are measured and then re-emitted at each intermediate node. The main drawback is that the security of the whole exchange depends on the security of each traversed node: in theory nodes are considered secure, but this is not guaranteed in real implementations therefore if an eavesdropper can break into a node, which is a classical computer, he can gain full access to key information and can modify routing behaviour controlling routing messages [19].

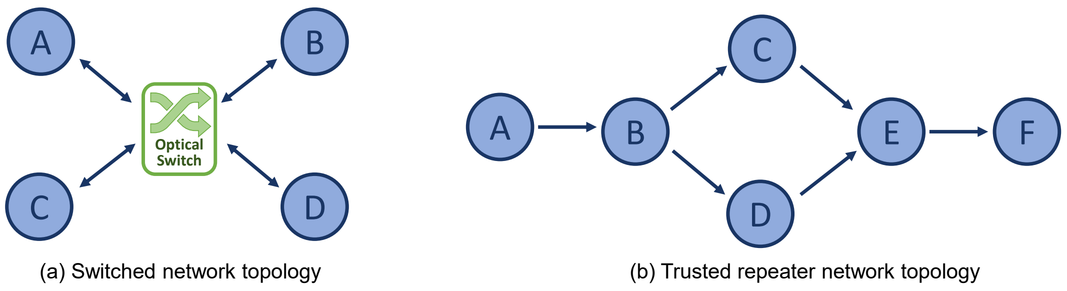

A
B
C
D
Optical Switch
(a) Switched network topology
A
B
C
D
E
F
(b) Trusted repeater network topology

Figure 3.2. Network types

To overcome the weakness of both these categories quantum repeaters are required. They are quantum devices that work by exploiting photons entanglement and entanglement swapping to allow communication over multiple links [20], forwarding qubits without causing signal deterioration but also without security issues of trusted repeaters, Although several theoretical research works have been proposed, the real implementation of these devices is still unreachable with current technologies mainly because they require quantum memories and the fidelity of the entanglement state decrease with the distance [21].

Currently developed QKD networks are mainly built with trusted repeaters [22] [23], accepting their security limitations, but some experimental switched networks have been proposed too [24].

3.1.2 Routing inside QKD networks

As said in the previous section there are multiple strategies to build the QKD network quantum layer but in all of them the management layer is required to perform routing and to create links between peers not directly connected. In the management layer information are exchanged over a classical network managed by one or more Internet service providers (ISPs). The management layer network is therefore an overlay network build on top of the classical one, which is independent of the ISP and that use the latter to encapsulate its packets inside standard internet packets. This allows nodes in the management layer to execute their routing algorithm, analyzing network parameters that may differ from the classical ones. Packets are exchanged on the overlay network performing routing without even considering routing domains and ISP paths, but in a hop-by-hop manner among QKD network nodes, which receive and forward packets to other nodes. This allows to compute paths based on the quantum layer information and to provide the required QoS, even choosing paths that are not optimal for the ISP point of view or performing multipath routing to increase security.

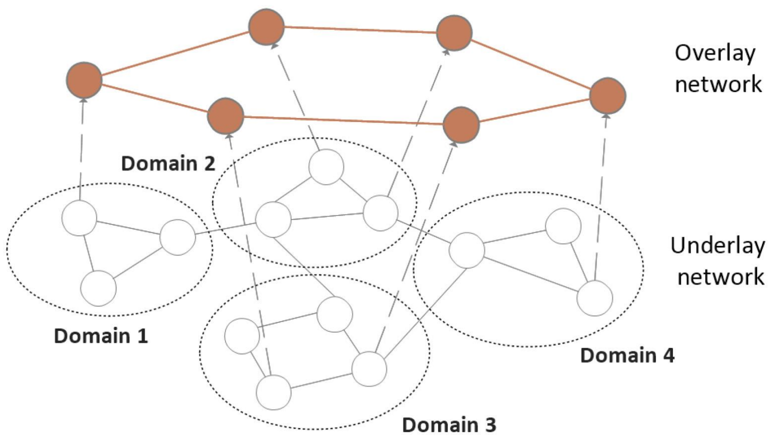


Figure 3.3. An overlay network build of top of the classical one. Source: [17]

To perform routing over multiple nodes the first developed QKD networks choose to use the Open Shortest Path First (OSPF) routing protocol [25] already developed and largely used in classical networks, based on the link-state algorithm. In OSPF each node has its routing tables and database about the other nodes and links and the metric of each one; information is exchanged with link state advertisement (LSA) packets sent by each node to its neighbours and then forwarded hop-by-hop to the whole network; each node computes its tables executing Dijkstra's algorithm. It allows usage of different metrics to compute the best path, and different modified versions of this algorithm working with different metrics and cost functions have been developed to specifically work inside QKD networks. In the DARPA QKD network [22], researchers developed an OSPFv2 personalized version [26] that uses as metric the amount of key material stored and available at each node:

$$m = \begin{cases} 100 + \frac{1000}{q-t}, & q > t, \\ \infty, & q \leq t. \end{cases}$$

where m is the link cost, q is the amount of key material expected to be available over that QKD link and t is a threshold for the minimum amount of key material to keep at each link. This metric allows to avoid paths that can be shorter in terms of hop but where there are no sufficient keys to satisfy the request, but it only takes into account QKD links status and not the management channel one.

Also in SECOQC a modified version of OSPFv2 that supports multiple routing paths between nodes is used, computing a routing table for each quantum link it has and merging them [23]. The cost $L_i(t)$ for the link i at time t is computed as follows:

$$L_i(t) = (1 - \frac{1}{w}) \cdot L_i(t-1) + \frac{1}{w} \cdot l_i(t)$$

where $l_i(t)$ is the instant load of the link i and w is a constant. The instant load $l_i(t)$ is the number of transmitted bits in the previous unit time.

In the Chinese HCW network [27] a different approach was proposed, where the OSPF algorithm is used in combination with a reservation protocol called *Quantum Key Reservation Approach*: the best path is computed with a hop count metric with OSPF, then all nodes in the path are reached by a reservation request from the starting node to reserve keys for that multi-link communication. This technique avoids any error due to insufficient key material available on the path but often chooses a path that is sub-optimal because hop count alone is not sufficient to guarantee QoS. In general, each routing algorithm that wants to reach the best performances in a QKD network should take into account both the quantum channel and the management channel state [28] and should limit the number of exchanged packets to reduce key consumption due to management purposes (even if packets are not encrypted, they require authentication and integrity checks to avoid man-in-the-middle attacks)

3.1.3 Key forwarding in multi-link scenarios

As seen in previous sections trusted repeaters networks require a hop-by-hop key forwarding strategy to build a multi-link secure path between two not directly connected nodes. Several strategies have been adopted in different networks, here the two main ideas are presented.

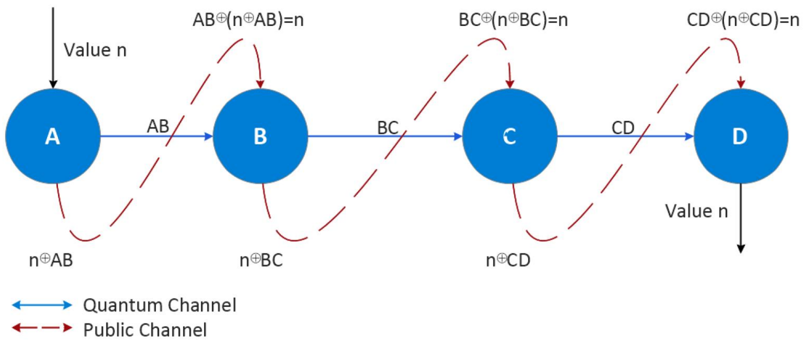


Figure 3.4. “Store and forward” key forwarding technique. Source: [17]

The first approach is the simplest one, known as *“store and forward”* and it has been adopted in the SECOQC network [23]. In a multi-link path the first node generates the key for the end peer and forwards it encrypted with a shared key to the next hop, the latter receives the message, decrypts it and then re-encrypts the received secret with a key shared with its next hop; the key proceeds in this way until the destination. An example of this technique with an exchange of the key n from node A to node D is reported in figure 3.4

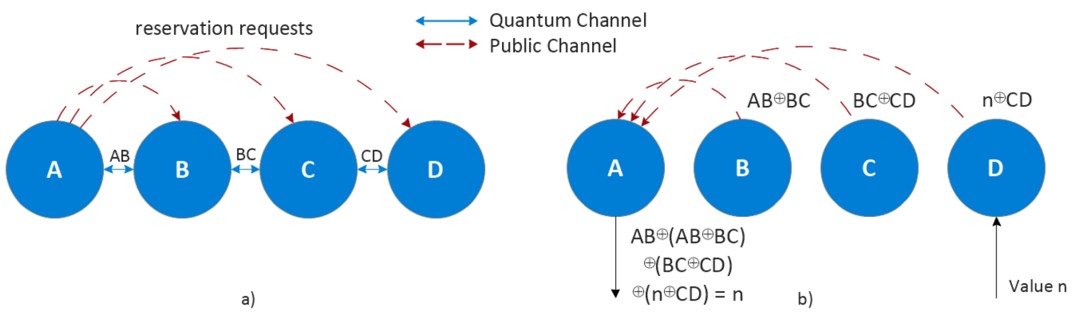


Figure 3.5. BBN’s Key repeater protocol. Source: [17]

The second one is called *“BBN’s Key Repeater Protocol”* and it has been used in the DARPA network [26] When a multi-link connection is required the source node execute the routing protocol and defines the connection path, then it reaches all other nodes and asks them to reserve a key with their predecessor in the path. The last block sends to the first one the secret key XORed with the key established with its predecessor while each intermediate node sends to the first the XOR results of the key established with its successor and predecessor. To retrieve the original key the source node performs a XOR operation among all received blocks. An example of this protocol is reported in figure 3.5 with an exchange of key n from node A to node D: figure 3.5(a) shows the reservation phase, figure 3.5(b) shows the key retrieving procedure.

Despite their differences each of these approaches suffers from the security issue of trusted repeaters networks: each node has to be secure and trusted to ensure the key security.

3.1.4 Software defined networking for QKD

The adoption of QKD on a large scale has in the requirement of quantum devices and dedicated quantum channels one of its biggest limitations, both in terms of costs and complexity: it is unfeasible to completely modify the current network. Some recent research works explored the possibility of integrating QKD inside a Software Defined Networking (SDN) pattern [29] to overcome these difficulties. SDN technologies allow the integration inside the existing network of new types of communications, leading to an efficient configuration of existing devices and flexible management of the network itself, avoiding the need for new dedicated channels. The main idea behind the SDN pattern is to decouple the data plane of the network from the control plane, moving management responsibilities from decentralized and multiple nodes to a centralized controller (the SDN controller). The network can be seen as a composition of three layers (as in figure 3.6), that communicates through well-defined interfaces (e.g. *OpenFlow*[1]):

- the *application plane* that abstract the underlying layer to users and allows them to require network resources (and, in the case of QKD networks, secret keys);
- the *control plane* where a centralized SDN controller knows the whole network status, accepts requests from the upper layer and dynamically configures network devices and set up paths to fulfil them;
- the *data plane*, where devices forward packets following rules received by the control plane. In the case of QKD networks here quantum devices exchanges qubits to generate secret keys.

This allows QKD devices to be connected to the standard telecommunication network over optical fiber exploiting wavelength division multiplexing (WDM) techniques [31] and not to a dedicated quantum network and allows users to take advantage of QKD security even without having physical access to a quantum device, via standardized APIs. Embedding QKD inside the telecommunication network has some advantages not only for users but even from the point of view of network security: any communication over the data plane or the control plane can be secured thanks to QKD derived keys [32].

3.1.5 Integration in the network stack

Quantum key distribution by itself is a key distribution protocol that has to be used with other cryptosystems or embedded in other protocols to be useful. It can be integrated into multiple levels of the network stack, as it happens for other primitives because it does not enforce any requirements on the key usage. Following the OSI network model [33], it is possible to integrate QKD in the Data Link layer (L2), in the Network layer (L3), in the Transport layer (L4) and in the Application layer (L7):

[1] https://opennetworking.org/sdn-resources/customer-case-studies/openflow/

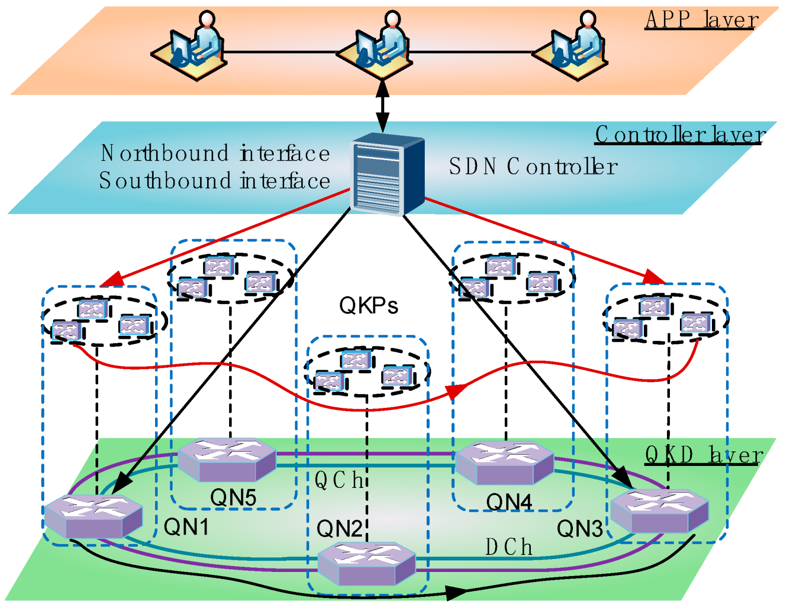


Figure 3.6. A QKD network enabled by SDN. Source: [30]

- at L2 it can be integrated into the Encryption Control Protocol (ECP) [34] to provide security for frames exchanged with the Point to Point Protocol (PPP), or it can provide keys for the IEEE 802.1 MACsec [35] protocol, both widely used in LANs;
- at L3 it can provide pre-shared secret keys for the Internet Key Exchange (IKE) protocol [36] which is used to set up security associations for Internet Protocol Security (IPSec) as shown in [22]
- at L4 security is provided by the Transport Layer Security (TLS) protocol [4]; QKD can be integrated into the handshake phase to substitute asymmetrical algorithms with pre-shared keys to be used as session keys (as shown in [37]);
- at L7 QKD can be used to establish shared keys among applications that use them for their specific protocols both for encryption and for authentication.

3.1.6 QKD trusted repeater network vulnerabilities

Trusted repeater networks are the most adopted network technology, but they are not free from defects. As already said the entire network security is based on the security of every single node, therefore they are called "trusted": if a node is compromised, all the traffic coming from and traversing that node is no more secure. Nodes are classical computers that can communicate with all other network nodes to exchange routing and control information and with the quantum devices to retrieve exchanged keys. As with any other computer they can suffer from several vulnerabilities, know and yet unknown: it is essential to protect them from attacks coming from the network and keep them physically inaccessible to unauthorized people. To limit risks caused by the hijacking of a node, multipath routing for keys is required: if the key does not travel over a single path, but over several and disjoint paths, an attacker must take control over multiple nodes to obtain key information [19]. In addition to nodes security, a QKD network is subject to links attacks both to the quantum channel and the classical one, as described in section 2.4. *Denial of service* (DOS) attacks on links can threaten the network security too: saturating the bandwidth of a link cause the traffic to be redirected over other links, therefore attackers can force some paths overloading specific links. A robust network should also provide advanced routing functions that can exclude compromised nodes and links from the path, redirecting the traffic over other trusted nodes. The network topology highly influences network security: the presence of redundant disjoint paths must be kept into consideration during the design phase as much as other parameters [38].

3.2 QKD network projects

In recent years a lot of effort was put into QKD research, leading to the development of the first QKD networks both for experimental purposes and for real use. In this section, the main networks are described, while a summary of the obtained results is reported in table 3.1.

3.2.1 DARPA QKD Network

The Defense Advanced Research Projects Agency (DARPA) QKD Network was the first ever built QKD network in the world, started with collaboration among BBN Technologies and Harvard and Boston Universities in December 2002 [22]. It started with four nodes (two transmitters and two receivers) interconnected with a passive optical switch that allows point-to-point communication, then three free-space links by different producers have been added and connected to the main network with other QKD links (figure 3.7). With its hybrid solution between trusted repeaters and optical switches, the DARPA QKD network demonstrated the advantages and disadvantages of both solutions. Because this was the first-ever developed QKD network there were no available schemas or protocols for routing and key agreement, therefore BBN proposed its protocol. Due to the slow key generation rate, it aimed to reduce as much as possible the number of exchanged messages to avoid more slowdowns over the channel. QKD generated keys were used to secure communication over channels, firstly as pre-shared keys for IKE protocol with an IPSec tunnel due to very slow key exchange rate, later for an OTP schema when better and faster QKD devices and channels were available [26]. To overcome this slow key rate a *key reservoir* approach was proposed: available keys were put in this storage and picked by nodes when required. Only network nodes have access to keys: DARPA network did not provide any key access directly to final users, but only to network nodes. The network never exceeds a key generation rate of 400 bit/s [22], but despite its limitation, the DARPA network is a milestone in QKD development: it laid the foundation for the other networks produced in the following years, both in terms of infrastructure and protocols. It was shut down in 2006.

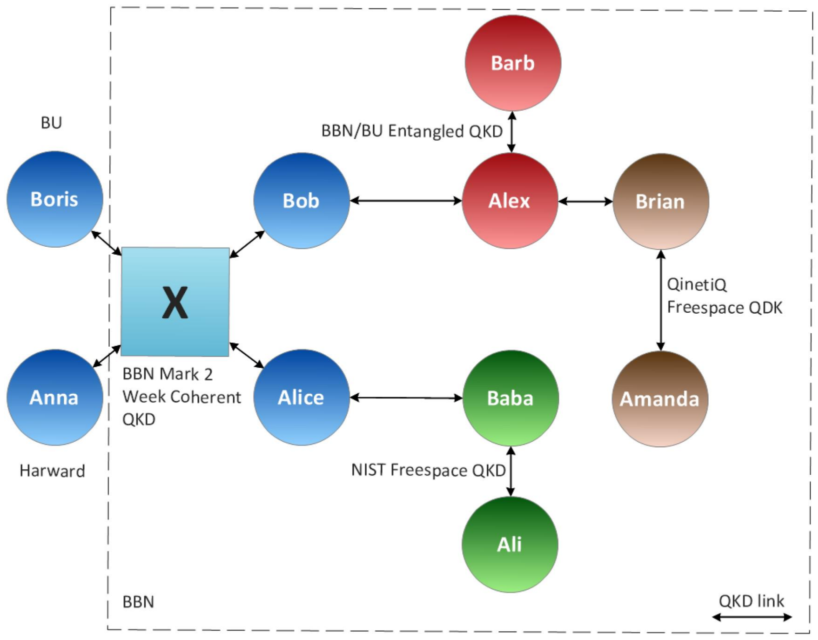


Figure 3.7. DARPA network topology at its last stage. Source: [17]

3.2.2 SECOQC QKD Network

The Secure Communication based on Quantum Cryptography (SECOQC) Network was a QKD Network promoted by the European Commission to push forward research on quantum cryptography: 13 countries took part in the project, with several industrial partners [23]. When the project started, in 2004, some results from the DARPA network were available and have been taken as starting point. A six nodes network, the *"Quantum Backbone"* was built in Vienna, with seven fiber channel links that extend for several kilometers and one short free space channel (figure 3.8). Thanks to its multiple partners, different technologies and different QKD protocols have been used and tested in different links. Thanks to DARPA results here only a trusted repeater structure was used, ditching optical switches, and a similar *key reservoir* approach was adopted for key management. Differently from the DARPA idea, here nodes were seen as access-point for the end-users, the effort was put into the development of a more complex and efficient routing algorithm (section 3.1.2) and more software modules to manage the network. Thanks to the advances achieved in the key exchange rate and in the stability of the network, which reached a maximum of 3.1 kbit/s, in October 2008 a public demonstration of the network capabilities was given: an encrypted telephone call and a video conference both with symmetric encryption and with OTP encryption have been shown [23]. SECOQC network also demonstrates, with its longest link of 82 km, that long-distance QKD communication is possible. The project was shut down in 2010.

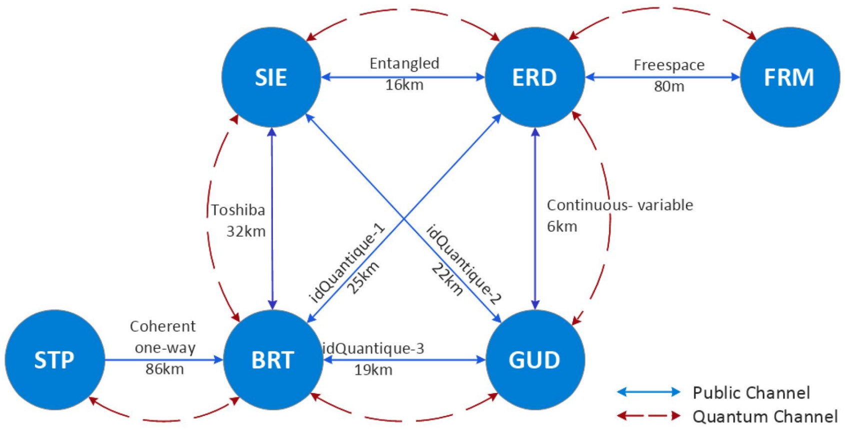


Figure 3.8. SECOQC network topology. Source: [17]

3.2.3 Tokyo UQCC QKD Network

In 2010 the building of a new QKD network was started, under the *"Updating Quantum Cryptography and Communications"* (UQCC) project, joining efforts of Japan and European Union researchers. The network consists of six nodes that are part of the *"Japan Giga Bit Network 2 plus"* (JGN2plus) in Tokyo, connected by six QKD links built by different partners [39]. The main difference with the previously built network was that the Tokyo network uses a centralized Key Management Server (KMS) to manage keys, while on each node a Key Management Agent was installed that respond to it, shifting toward an SDN based approach (section 3.1.4). The links supported incredibly high speeds compared to the previous results, reaching a key rate of 304 kbit/s. Public demonstrations of the network capabilities have been performed, showing the KMS capability to detect eavesdroppers on links based on statistical data received by nodes agents.

3.2.4 China Wide Area QKD Network

In recent years China pushed forward in the development of QKD networks not only for research projects but also for real use. Several metropolitan QKD networks have been built, and they

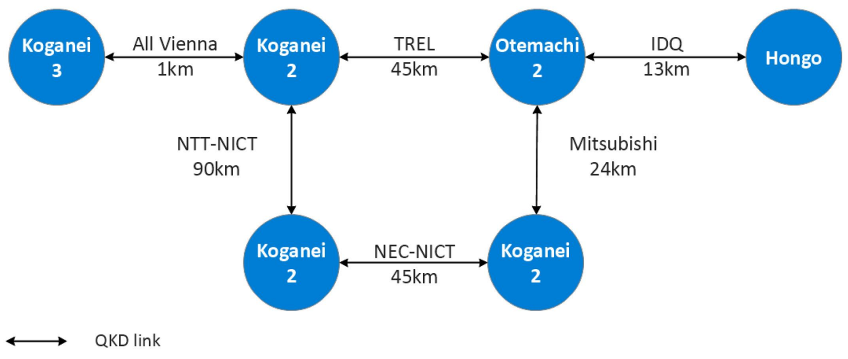


Figure 3.9. Tokyo UQCC network topology. Source: [17]

have been connected by a large QKD backbone network allowing secure communication over high distances [40]. The Beijing-Shanghai Backbone QKD Network is the longest QKD network in the world, with a 2000 km span and connects multiple metropolitan QKD networks. It is composed of 32 nodes connected by fiber links, of which 6 are access points and the other are only intermediate nodes necessary to perform communication over that distance (figure 3.10). The network is supported by two to eight links between each couple of nodes for redundancy reasons, for a total of 135 QKD links, combined to at most 4 links over the same fiber cable with WDM techniques [41]. In 2018 it was broadened and connected to the Wuhan Metropolitan Area QKD network and there are plans to connect it to other smaller networks over the country.

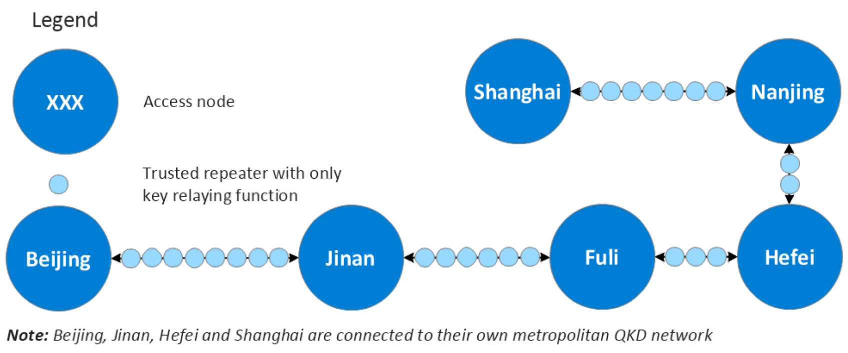

Legend
XXX
Access node
Trusted repeater with only key relaying function
Shanghai
Nanjing
Beijing
Jinan
Fuli
Hefei
Note: Beijing, Jinan, Hefei and Shanghai are connected to their own metropolitan QKD network

Figure 3.10. Beijing-Shanghai backbone network topology. Source: [17]

Remarkable results in terms of node and access point numbers have been achieved in metropolitan QKD networks. In Jinan, the government private QKD network counts 23 end-user nodes, 8 trusted repeaters and a central management node for a total of 32, and 33 links [42]. In Wuhan, the network is composed of a central node, 10 trusted repeaters and 60 end-user nodes, with 74 QKD links and a central ring topology have been built to improve disaster tolerance. Their topology is reported in figure 3.11

3.2.5 Recent initiatives and projects

In addition to the projects presented in the sections above, there are a big number of other initiatives that are currently working and researching about QKD; here some of them are reported.

Regarding free space QKD communication, the most remarkable project is the Chinese *Quantum Experiments at Space Scale* project: a satellite named *Micius* equipped with QKD devices was sent in space in 2016 [43]. It was used both as a trusted repeater, allowing communication between labs in Beijing and Vienna in 2017, and as a source of entanglement photons for ground stations in Delingha and Nanshan that uses them as input for entanglement-based QKD protocols.

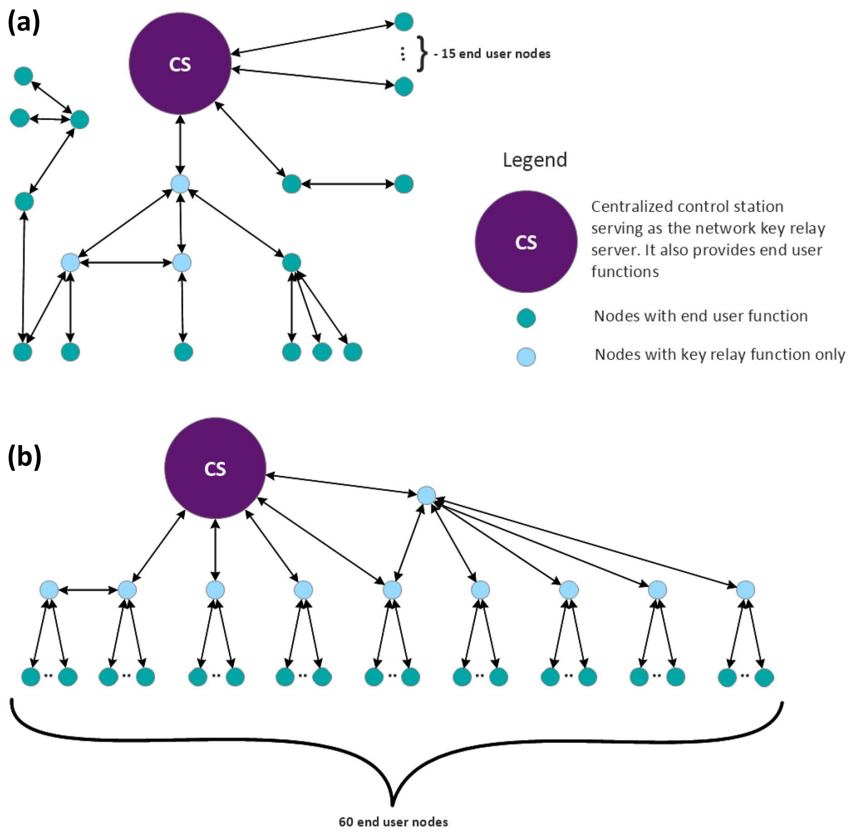

(a)
CS
- 15 end user nodes
Legend
CS
Centralized control station serving as the network key relay server. It also provides end user functions
Nodes with end user function
Nodes with key relay function only
(b)
CS
60 end user nodes

Figure 3.11. Jinan (a) and Wuhan (b) metropolitan area network topologies. Source: [17]

	DARPA	SECOQC	Tokyo UQCC	Beijing-Shanghai	Jinan
Year	2002-2006	2004-2008	2010	2014-2017	2017
Country	USA	Austria	Japan	China	China
Nodes	10	6	6	32	71
Network type	Switched & trusted repeater	Trusted repeater	Trusted repeater	Trusted repeater	Trusted repeater
Max key rate	400 bit/s	3.1 kbit/s over 33 km	304 kbit/s over 45 km	250 kbit/s over 43 km	141 kbit/s over 16.5 km
Most used protocols	BB84	BB84, SARG04, COW	BB84, BBM92	Decoy state BB84	Decoy state BB84

Table 3.1. QKD networks summary. Source: [17]

In Europe quantum computing research is led by the Quantum Flagship initiative[2]: born in 2016, funded by the European Commission, it is a large project which goal is to promote research works to consolidate European knowledge in this area and to kick-start the industry in quantum technologies. About QKD the *OPEN QKD* European consortium[3] has the goal to allow cooperation among the academic world and the industry, uniting 38 partners from 13 countries, to deploy open testbed sites for QKD, to increase awareness and to develop an ecosystem for

[2]https://qt.eu/

[3]https://openqkd.eu/

quantum communication devices and technologies.

In the US market, the company *QuantumXChange*[4] sells a software solution for enterprise customers ready for use that offers a key-delivery architecture that combines QKD over a QKD trusted repeater network and PQC algorithms to securely exchange keys over the internet. IDQuantique, a leader in the QKD devices market, recently partnered with Fortinet to commercialize a quantum-safe VPN solution that provides L3 encryption with QKD-generated keys based on the Fortinet FortiGate firewall and the IDQ Cerberis device[5].

In Italy, a recent agreement between the *TOP-IX* consortium, *Italtel* and *CSI Piemonte*[6] made it possible to test the Italtel *Quantum Secure Network* in the TOP-IX network to provide QKD functionalities over a fiber link.

In recent years telecommunication companies show their interest in the QKD technology and started some trials about QKD integration in their networks, like *Verizon* [7] and *SK Telecom*[8]

[4] https://quantumxc.com/

[5] https://www.idquantique.com/id-quantique-partners-with-fortinet-to-commercialize-a-quantum-safe-vpn-solution/

[6] https://www.italtel.com/quantum-technology-makes-data-and-services-safer/

[7] https://www.verizon.com/about/news/verizon-achieves-milestone-future-proofing-data-hackers

[8] https://www.idquantique.com/idq-sk-telecom-nokia-secure-optical-transport-system-using-qkd/

Chapter 4

QKD standards

In recent years different entities started working on the standardization process of QKD. Being QKD a multidisciplinary technology, it requires standardization not only at the software and at the protocol level but also for devices and their components and on how software and devices should communicate among them. Standards should guarantee interoperability between products of different companies and help in reducing bugs and defects in implementations. Here the standards ETSI and ITU are working on are presented.

4.1 ETSI standard

ETSI is a European standards organization[1] that produce standards in the fields of telecommunications networks and services. Since 2010 ETSI Industry Specification Group (ISG) is working in the standardization process of QKD, proposing several *Group Specification* (GS) documents not only about software and devices but trying to provide a complete view on the QKD ecosystem. Here the document list with a summary of their content is reported:

- *Use Cases* (ETSI GS QKD 002) [44] provides an overview of possible application scenarios in which QKD systems can be used as building blocks for ICT systems and the mechanisms to drive development towards a security certification of QKD systems;
- *Components and Internal Interfaces* (ETSI GS QKD 003) [45] describes properties and details for quantum devices required in a QKD system;
- *Application Interface* (ETSI GS QKD 004) [46] specifies an Application Program Interface (API) between a QKD Module called "key manager" that is in charge of securely managing keys produced by QKD devices and delivering them to applications;
- *Security Proofs* (ETSI GS QKD 005) [47] explains the QKD security claim and specify the role of QKD devices and their implementation defects in the security of a QKD system;
- *Vocabulary* (ETSI GS QKD 007) [48] collects QKD related definitions and abbreviations used in the other documents;
- *QKDM Security Specification* (ETSI GS QKD 008) [49] describes the specifications for a QKDM to protect it from unauthorized physical access and known attacks and the reaction processes required in case of penetration;
- *Component characterization: characterizing optical components for QKD systems* (ETSI GS QKD 011) [50] gives the specification for the optical components required in a QKD system, with examples of specific tests to perform to check their configuration;

[1] https://www.etsi.org

- *Device and Communication Channel Parameters for QKD Deployment* (ETSI GS QKD 012) [51] describes the main communication resources involved in a QKD system and the possible architectures that can be chosen for the deployment over an optical fiber network infrastructure;
- *Protocol and data format of REST-based key delivery API* (ETSI GS QKD 014) [52] specifies a communication protocol and its data format for a QKD network to allow applications to communicate via REST API to the network node;
- *Control Interface for Software Defined Networks* (ETSI GS QKD 015) [53] describes the management interfaces for the integration of QKD in the network with SDN, modelling the communication between the QKD node and the SDN controller.

The standardization process is not completed yet, therefore some of these documents are updated with new versions when changes are required.

4.1.1 Use cases

The ETSI GS QKD 002 document [44] describes some common use cases for QKD, the most significant ones are reported here below.

In a PTP scenario, QKD can be used in combination with a link encryptor, which is a system able to encrypt and decrypt all the traffic exchanged between two nodes at the data link layer (L2), using QKD keys and acting transparently for the upper layers. The easiest use case is the one described in figure 4.1: a company owns a primary site where business operations happen, a backup site where data are saved periodically and a private point-to-point classical and quantum link. Backup data, that can be used to recover data due to a disaster at the main site, are confidential therefore they must be encrypted when they are outside the corporate network perimeter. With keys obtained by the QKD system, link encryptors can encrypt the traffic with a symmetric algorithm, freeing the application from this task. Due to the large amount of data to be transferred an OTP schema can become unfeasible and symmetric key algorithms are more appropriate, key renewal policies can be set to match the required security level until they remain below the QKD link generation rate.

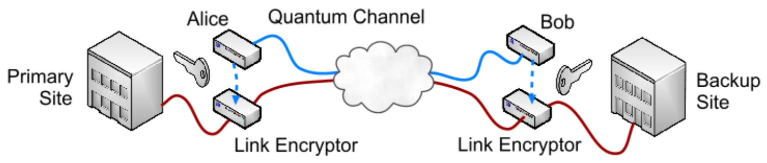


Figure 4.1. QKD link encryptor for secure backup procedure. Source: [44]

This use case can be extended to several nodes and data centres, connected by several encrypted links and multiple QKD channels, over a larger area (figure 4.2(a)). Link encryptors can be an alternative solution to L3 or L4 Virtual Private Networks (VPNs) where available thanks to the fact that they are transparent to the application and does not require any change at upper levels. When distances among nodes are too high to support single QKD links, a private QKD network can be built (figure 4.2(b)). In this case, link encryptors can be substituted by higher-layer security protocols, allowing a single encryption operation for the whole packet life instead of hop-by-hop management, using the key received from the QKD exchange over the network.

QKD is not limited to optical fiber transmission but it can be used also through free-space connections from the ground to satellites to cover long haul connections where fiber links are unfeasible. A low-orbiting satellite that passes over two very distant sites can exchange a key with the first one (figure 4.3(a)), then later exchange a key with the second one (4.3(b)) and

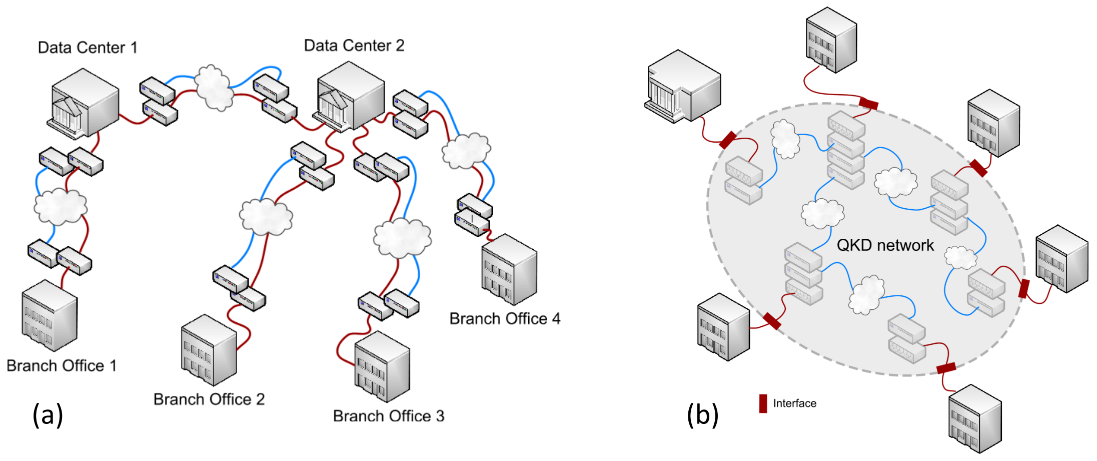


Figure 4.2. Enterprise Metropolitan area network using QKD link encryptors (a) and Wide area network using a QKD network (b). Source: [44]

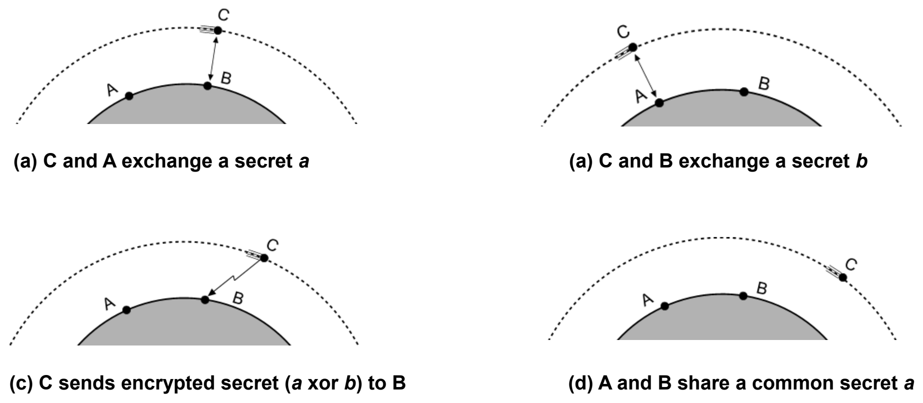

C
A
B
(a) C and A exchange a secret *a*
(a) C and B exchange a secret *b*
(c) C sends encrypted secret (*a* xor *b*) to B
(d) A and B share a common secret *a*

Figure 4.3. Key exchange with the use of a low-orbiting satellite. Source: [44]

send to it the first key encrypted with an OTP schema (4.3(c)), allowing the two sites to share a secret key without using the ground infrastructure.

QKD can be integrated into the existing optical network, it can provide secure keys between the ISP and the end-users. Common telecommunication networks are Passive Optical Networks (PONs) that connects one Optical Line Terminal (OLT) installed at the ISP facility with several Optical Network Units (ONUs) near end users. Information is broadcasted from the OLT to users over the same optical fibre cable, therefore each ONU can see the whole traffic while upstream traffic is transmitted with a WDM schema. Encryption is required to protect downstream traffic from the eavesdropping of other ONUs and QKD can be employed in the substitution of standard asymmetric-key techniques using the same optical path used for standard data. The OLT can send quantum information encoded in single photons over the optical fibre, knowing that each photon can reach one and only one ONU thanks to the no-cloning principle, and these received qubits can be used to set up a secret key. Even if the OLT must provide keys for multiple end-users the short distance in PON structures allows a high key rate. With current technology, where detectors are more expensive than photon sources, the opposite scenario where ONUs are equipped with photons emitter and the OLT can detect qubits is also viable. An example of this architecture is shown in figure 4.4

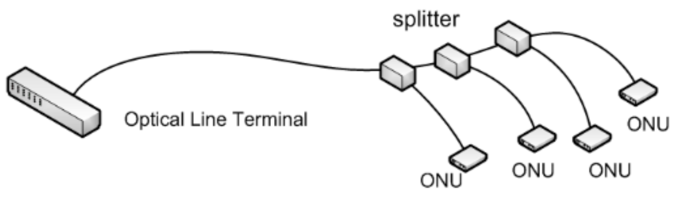


Figure 4.4. PON architecture. Source: [44]

4.1.2 Application Interface

The ETSI GS QKD 004 document [46] describes the APIs between the *QKD Key Manager* and applications. The Key Manager is a software component that manages the secure keys produced by a QKD protocol and delivers them in identical pairs to an application at the upper level through these APIs. The Key Manager is part of a *QKDM* (QKDM), composed by itself and the quantum module which is responsible for the QKD protocol execution. A simple architectural schema is presented in figure 4.5, where two applications in two different sites can communicate using keys produced by a QKD protocol received from the Key Manager.

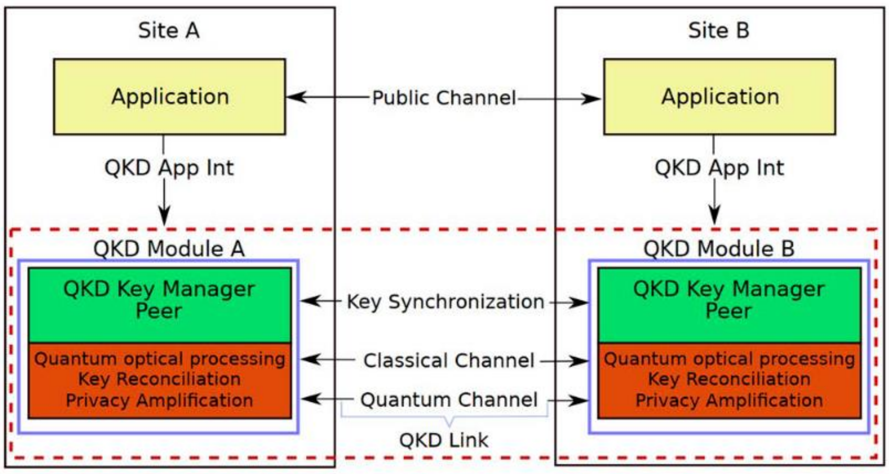


Figure 4.5. QKD Application Interface and peer relationships. Source: [46]

The interface is composed of three functions, `OPEN_CONNECT`, `GET_KEY` and `CLOSE`, which syntax is the following:

```
fun OPEN_CONNECT (source, destination, QOS, Key_stream_ID) -> (QOS,
    Key_stream_ID, status)
fun GET_KEY (Key_stream_ID, index, Metadata) -> (index, Key_buffer,
    Metadata, status)
fun CLOSE (Key_stream_ID) -> (status)
```

To provide high compatibility between products of different companies, the standard proposes a core limited in functionalities that can be expanded by each vendor according to their needs.

`OPEN_CONNECT` reserves an association for a set of future keys at both ends of the QKD link, identified by the `Key_stream_ID` parameter. `Key_stream_ID` is a unique identifier for the association, whose type is an UUIDv4[2], that must be derived independently from the key material, shared by both Key Manager peers. The function must be invoked in both peers to create a valid

[2]https://tools.ietf.org/html/rfc4122

association, with the same Key_stream_ID. If the function is invoked with the Key_stream_ID set to NULL a new identifier is generated by the peer itself. Applications are responsible for sharing the id with their peer. A required QoS level can be specified in the request, on return, the applied QoS is the best that the system can provide. The source and the destination parameters are URIs that identify the peer applications. Meanings for the return parameter status are reported in table 4.1 and are shared with the other functions in the interface. More information about parameters can be found in the original document [46].

GET_KEY returns a key that belongs to the specified Key_stream_ID, together with its position index. The function can be called with a specified index to retrieve a specific key, as should be done by the peer application to retrieve the same key as the first one. Metadata parameter carries more information about the key, such as the number of traversed trusted nodes or the age of the key.

CLOSE terminates the association established for the received Key_stream_ID, no further keys will be exchanged for this association but the already present key can still be retrieved. It should be called at both peers.

Each couple of applications can manage several and independent associations with different Key_stream_IDs. Thanks to the high flexibility of this API it is possible to implement an upper level of API where a centralized Key Server can control multiple Key Manager, building an architecture where the application can contact a single manager for the entire node and not one for each link, as described in ETSI GS QKD 014 (section 4.1.3).

The document does not specify how two Key Manager peers should communicate to exchange Key_stream_IDs or other information, nor how internal operations are accomplished. Implementation choices like the programming language or the interface architectural style are left to the developer.

Value	**Meaning**
0	Successfull
1	Successful connection, but peer not connected
2	GET_KEY failed because insufficient key available
3	GET_KEY failed because peer application is not yet connected
4	No QKD connection available
5	OPEN_CONNECT failed because the Key_stream_ID is already in use
6	TIMEOUT_ERROR. The call failed because the specified TIMEOUT
7	OPEN_CONNECT failed because requested QoS settings could not be met, counter proposal included in return
8	GET_KEY failed because metadata field size insufficient. Returned Metadata_size value holds minimum needed size of metadata

Table 4.1. Status parameter meaning. Source: [46]

A sequence diagram of the workflow is reported in figure 4.6, where two apps (APPA and APPB) want to retrieve a shared key retrieved from QKDA and QKDB. APPA starts calling OPEN_CONNECT() with the source and destination application as parameters and NULL as Key_stream_ID (KSID). The generated KSID is sent from the QKDA peer to QKDB and APPA communicates its KSID to APPB over the public channel (functions NewAPP() and Send_KSID() are not described in the ETSI document). To complete the association APPB invokes OPEN_CONNECT() with the received KSID. Both APPA and APPB can retrieve keys calling GET_KEY() function until they need keys. The association is closed calling the CLOSE() function both in QKDA and QKDB, even at different moments.

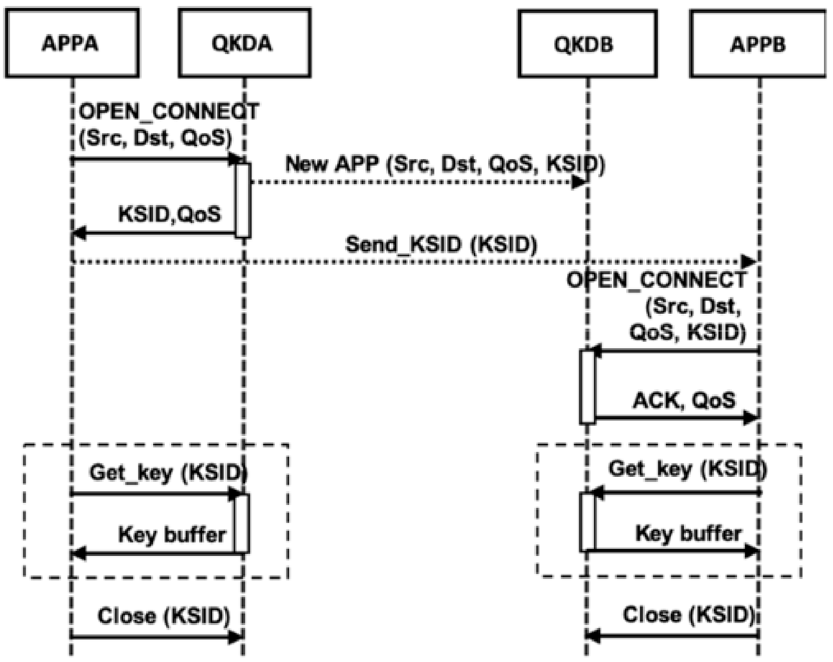


Figure 4.6. Workflow of two applications to require a key. Source: [46]

4.1.3 REST-based key delivery API

The ETSI GS QKD 014 document [52] describes the REST-based API required in a trusted node to deliver keys to applications in a QKD network. This API allows communication between a *Key Management Entity* (KME) and a *Secure Application Entity* (SAE): the former is the system in charge of managing keys in the network synchronizing with other KMEs, the latter is the entity that requires keys to the KME. The KME retrieves from one or more *QKD Entity* (QKDE) which are the QKDM described in ETSI GS QKD 004 document (section 4.1.2). When two SAEs wants to share a key the one that starts the communication is called *master SAE* and the peer is called *slave SAE*. Each SAE and each KME has an identifier that is unique not only at the node level but in the entire QKD network.

This API consists of three functions compliant with the REST architecture, that must be accessed by SAEs through specific URLs (table 4.2), using the HTTPS protocol[3]. Data are sent and received in the JSON format[4], the structure of the JSON object for each request and response is described in figure 4.7. Some data are optional and some of them are not currently used in the standard, but they are there for future use. The standard put some requirements on the security of the involved entities: mutual authentication between SAE and KME is required, each request must use the HTTPS protocol (at least with TLS in version 1.2), and each entity is considered secure and the node securely operated.

`Get status` is the method that returns information about the keys that can be requested by the *master SAE* for the *slave SAE* specified in the URL, in terms of number and characteristics. The access URL contains all the information needed for the request because `GET` requests do not have a body, while return information is contained in a JSON object described in figure 4.7. The `KME_hostname` URL parameter is the hostname or the IP address of the KME, while the `slave_SAE_ID` is the URL encoded identifier of the peer SAE.

`Get key` returns a key container JSON object that contains keys with their respective identifier that the *master SAE* shares with the *slave SAE* and, if requested, additional information on the

[3]https://tools.ietf.org/html/rfc7230

[4]https://tools.ietf.org/html/rfc8259

Method name	URL	Access Method
Get status	https://{`KME_hostname`}/api/v1/keys/{`slave_SAE_ID`}/status	GET
Get key	https://{`KME_hostname`}/api/v1/keys/{`slave_SAE_ID`}/enc_keys	POST (or GET)
Get key with key IDs	https://{`KME_hostname`}/api/v1/keys/{`master_SAE_ID`}/dec_keys	POST (or GET)

Table 4.2. REST-based interface methods.

returned keys. This request can be performed with the `GET` method only if there are no additional parameters besides the key number and size, which can be in this case embedded in the URL as a query parameter. If supported by the implementation this method allows retrieving keys that can be shared with more than one peer with a key multicast operation.

`Get key with key IDs` allows the peer SAE (*slave SAE*) to retrieve the key delivered to the *master SAE* specifying the IDs received from it. This request can be performed with the GET method only if one single key is requested, in that case the key identifier is embedded in the URL as a query parameter. With this method, the *slave SAE* does not need to specify the key size or any other information besides the identifier, because they have already been specified in the `Get key` request from the *master SAE*. The return JSON object is shared with the `Get key` method. The `master_SAE_ID` URL parameter is the URL encoded identifier of the master SAE.

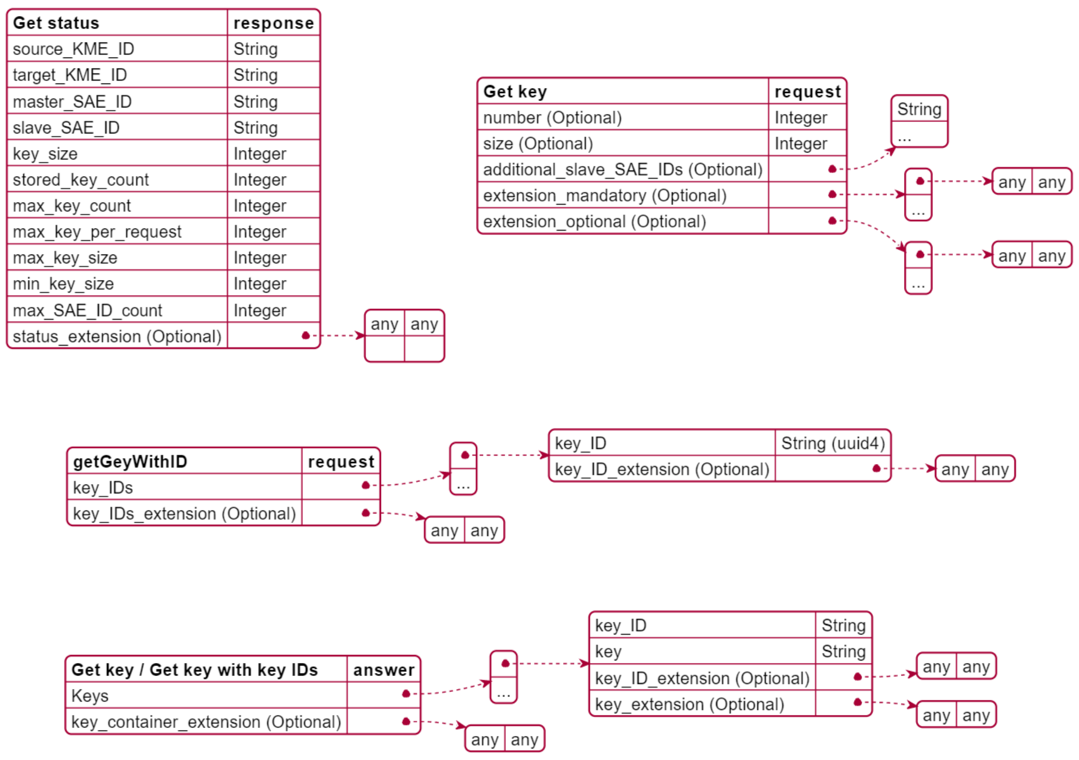

Get status | response
source_KME_ID | String
target_KME_ID | String
master_SAE_ID | String
slave_SAE_ID | String
key_size | Integer
stored_key_count | Integer
max_key_count | Integer
max_key_per_request | Integer
max_key_size | Integer
min_key_size | Integer
max_SAE_ID_count | Integer
status_extension (Optional)
any | any
Get key | request
number (Optional) | Integer
size (Optional) | Integer
additional_slave_SAE_IDs (Optional)
extension_mandatory (Optional)
extension_optional (Optional)
String
...
any | any
any | any
getGeyWithID | request
key_IDs
key_IDs_extension (Optional)
key_ID | String (uuid4)
key_ID_extension (Optional)
any | any
any | any
Get key / Get key with key IDs | answer
Keys
key_container_extension (Optional)
key_ID | String
key | String
key_ID_extension (Optional)
key_extension (Optional)
any | any
any | any
any | any

Figure 4.7. Interface methods JSON objects.

As for the *Application Interface* described in the previous section, this interface is kept very minimal, both as methods both as information carried in JSON objects and allows the developer to improve it as required. Neither implementation details nor restrictions on the programming language to be used are given by the standard.

A schema of the interface workflow is reported in figure 4.8, where `SAE A` and `SAE B` want to retrieve a shared key retrieved from `KME A` and `KME B`. `SAE A`, which is the master SAE in this example, calls `Get key` and retrieve keys with their identifier, then it sends those ids to `SAE B` that can retrieve the same keys calling `Get key with key IDs` specifying the ids received from its peer. The standard does not describe how the two KMEs should exchange information about the received request or the connected SAEs, nor how the SAEs should exchange the key identifiers.

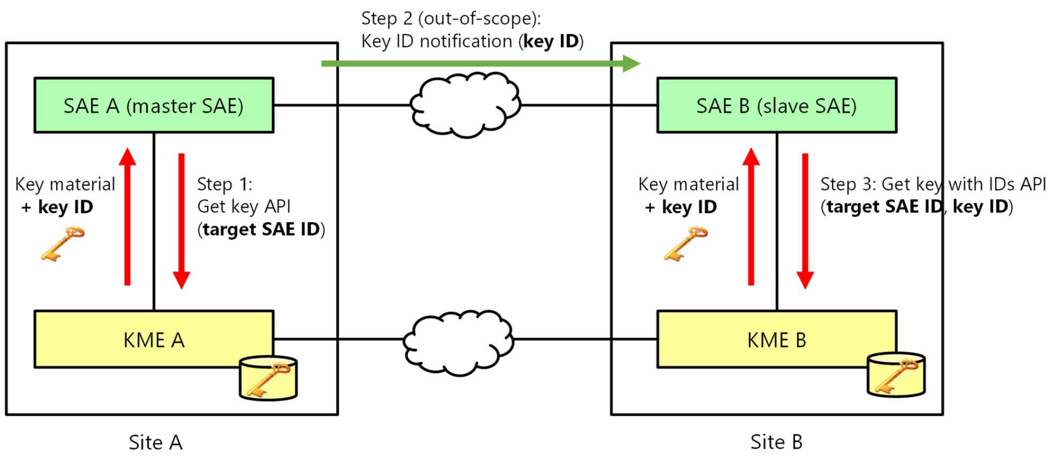


Figure 4.8. Example of the interface workflow. Source: [52]

4.2 ITU standard

The International Telecommunication Union (ITU) is the United Nations specialized agency for ICTs. In 2019 the ITU begin the QKD standardization process[5], publishing the first overview document. Documents from Y.3800 to Y.3999 are reserved for QKD network topics, and the already available ones are summarized here:

- *Overview on networks supporting quantum key distribution* (Y.3800) [54] gives an overview on QKD networks focusing on technologies, network capabilities and conceptual network structure;
- *Functional requirements for quantum key distribution networks* (Y.3801) [55] describes requirements for quantum, key management and control layers in QKD networks;
- *Quantum key distribution networks - Functional architecture* (Y.3802) [56] specifies the functions that are required inside a QKD network, by each layer and each device;
- *Quantum key distribution networks - Key management* (Y.3803) [57] describes the key format and how keys should be managed inside nodes regarding their life cycle and their forwarding over nodes;
- *Quantum key distribution networks - Control and management* (Y.3804) [58] specifies functions and procedures required in the upper layer of the network for the management and the orchestration as routing and QoS.

[5]https://news.itu.int/new-itu-standard-networks-support-quantum-safe-encryption-authentication/

Chapter 5

Cloud-native applications

In the last decades, the market shifted from an on-premise infrastructure, where each company has its servers in private data centers, to cloud-based infrastructure as a Service (IaaS) architecture where shared data centers host several applications from different customers on virtual machines. Using cloud-hosted servers allows companies to get computing resources at request, without the need of managing the physical hardware and the overall infrastructure, enabling easy scalability for applications. With the continuous growth of applications, both in complexity and in scalability requirements, it was clear that the common "monolithic" architecture was no more viable and alternative solutions have been proposed, leading to the development of lightweight virtualization solutions as containers and microservices architectural pattern.

5.1 Containers

Containers are a form of lightweight virtualization that allows overcoming some of the main drawbacks of virtual machines (VMs), leading to faster and easier management of virtualized instances. In an environment where several processes run on the same host, strong isolation between them is required, ensuring that data can not be shared and that a vulnerability in one service is not able to affect the others or the host itself. VMs are in theory a viable solution, but they are slow to start, require a lot of resources and provide features like hardware virtualization or the possibility to run different operating systems (OSs) that is often not required. Containers implement strong process isolation in the OS kernel without much overhead, leveraging features like *cgroups* and *namespaces* that are already built in the Linux kernel (which is the standard OS for server environment). The formers allow to limit resource consumption of each process in terms of memory and CPU usage, the latter provides isolation primitives to define what each container can see about the host and other containers in terms of network, processes and file system. This OS-level virtualization does not require a hypervisor on top of the host OS kernel, but it is the kernel itself that provide structural support for the containers: they share the same kernel of the host and can build their service (with its libraries) on top of it and therefore require fewer resources. Figure 5.1 describes layers both in containers and in VM architectures.

The first basic project to implement containers was *Linux Containers*[1], based on the standard Linux kernel, then other solutions like *Docker*[2] and *rkt*[3]. Docker is by far the most used container environment. It is mainly focused on containers as applications, providing an environment where it is easy to deploy them in a lightweight and self-contained form, solving the problem of dependencies: Docker containers can run anywhere there is the Docker engine installed, with the same exact behaviour, because they are not dependent from the underlying OS version or its installed

[1] https://linuxcontainers.org/

[2] https://www.docker.com/

[3] https://cloud.redhat.com/learn/topics/rkt

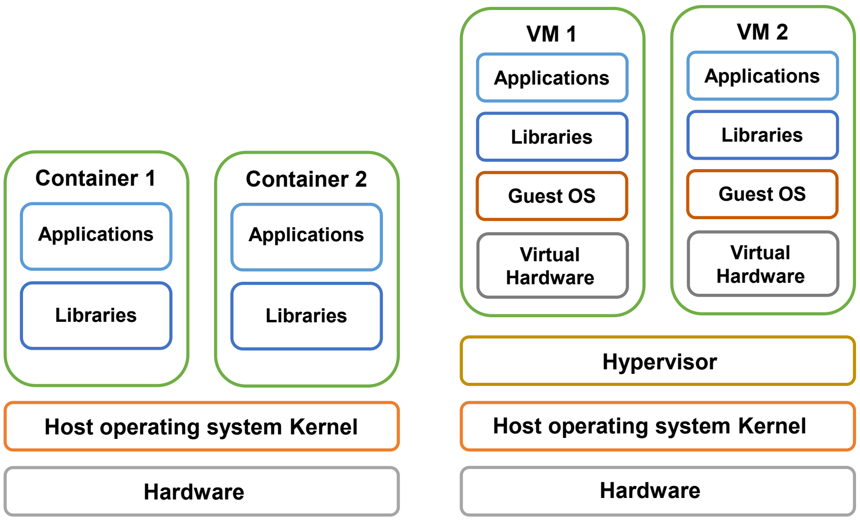


Figure 5.1. Containers architecture compared to VMs.

packages. Through *images*, Docker allows the developer to easily build an application from a sequence of commands written in a file (the *Dockerfile*) and containers can be deployed from their images in multiple copies and different places. The Docker platform also provides simplified management of the network requirements, allowing both to connect containers over a virtual network and to reach them from the host OS via port forwarding with few settings. Despite their lightness, spawning multiple several containers can require a lot of storage space, because its container has its own file system. To solve this issue Docker adopts a layered file-system structure: the final file-system is composed of a set of read-only layers that can be shared with other containers (like for libraries and static data) and a single private read-write layer that contains private data and modified files. There is also the possibility to map the host file-system to container one, with *volumes*, allowing an easy way to share data. More information on how to use, deploy and manage Docker containers can be found at the Docker documentation website[4]. Nowadays containers are always more often used to build portable applications, especially in the cloud environment due to their lightness and their ease of use.

```
# syntax=docker/dockerfile:1
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

Figure 5.2. Example of a Dockerfile to build an image for a simple Flask webserver

[4] https://docs.docker.com/

5.1.1 Containers security

Despite their speed and agility, containers involve some security problems that in VMs can be avoided: sharing the kernel with the host means that there is one less protection layer with respect to VMs architecture [59]. Both the other containers and the host require protection from a potential malicious container, but attacks to the host kernel can lead to major damages because they can harm the whole system. This is the reason for resource limits and isolation requirements: the former is essential to limit damages due to Distributed Denial of Service (DDoS) attacks that a container can try to perform exploiting all resources, the isolation properties are required both between containers to prevent the leak of sensitive information from the file system, from network scanning or resource usage analysis but also between a container and the host to prevent unauthorized access or, in the worst-case scenario, privilege escalation and container escape attacks that allow the container application to get control over the whole system. Keeping resources and privileges of a container to the minimum necessary is a primary requirement in a container architecture. To reach the security goal the security of the container itself is an extremely important factor: container images that come from untrusted sources should not be used because they can contain malicious code or backdoors and even trusted images should be checked against known vulnerabilities to reduce both the attack surface from the outside of the system and attack risks from other containers running on the same hosts.

A survey of 2017 showed that 90% of the images hosted on Dockerhub (which is the main repository for Docker container images) suffers from high severity issues [60], and despite the security improvement that can be achieved, side channels attacks remain an unsolved problem in host security: hardware vulnerabilities can lead to undetected information leaking. To reduce security risks in containers environment several guidelines have been proposed, like the ones from NIST [61].

5.2 The microservice pattern

The main idea of the microservices pattern is to decompose an application into multiple and smaller loosely coupled components, each one with a single basic function, called “microservice”, that together compose the application. This pattern provides a lot of benefits, leading to an application better suited for working in a cloud environment. Each component can use the more appropriate programming language for its tasks, it is small and with a single job, therefore it is less affected by bugs and it is easy to maintain and develop. Inside the application environment, each microservice can be deployed in multiple instances, removing the single point of failure caused by a crash that can harm the entire app; this also allows the system to be scaled up or down in case of traffic variations, simply spawning more copies of the same service, even over different machines and balancing the traffic among the copies. This scaling can be performed independently for each component, reducing resource waste. Thanks to lightweight virtualization technologies often each microservice is allocated on a different container because they allow fast and easy spawning and restarting, without introducing too much overhead on the hosts, and they are by default isolated from the host. Having multiple copies of the same service allows to upgrade it to a new version and test it seamlessly without blocking the production environment, slowly migrating from the previous version to the new one and redirecting traffic when needed, allowing a faster reaction to requirements changes.

Working in a microservices environment require some significant changes in the application architecture: the interaction between services has to be performed over the network because a service can not access data or functions of another one. The communication should be performed over well defined interfaces both in a synchronous way, via REST APIs over HTTP protocol or gRPC[5], or asynchronously via messages queues likes Apache Kafka[6] or RabbitMQ[7], where a

[5]https://grpc.io/

[6]https://kafka.apache.org/

[7]https://www.rabbitmq.com/

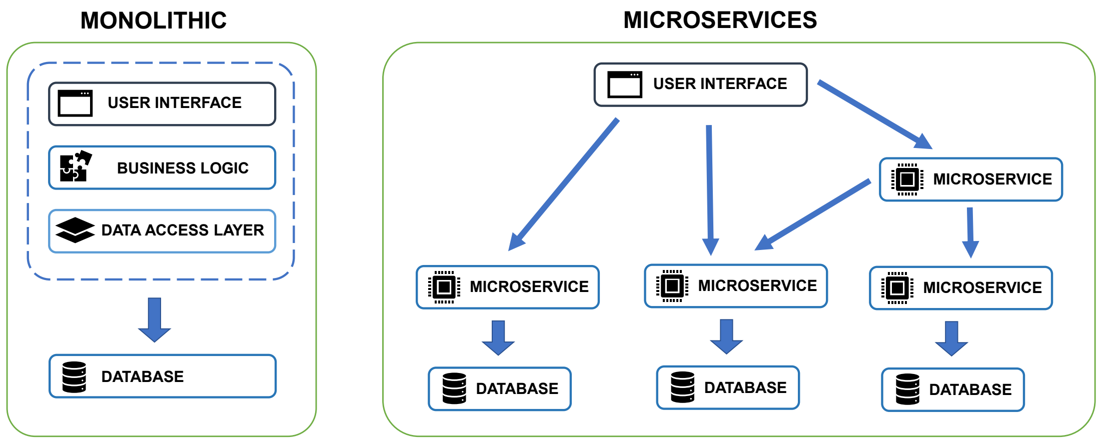


Figure 5.3. Monolithic and microservices architecture compared

service posts a message over a topic the receiver previously subscribed to and that can receive and react to that message. Due to the replacement of containers and their scalability, the application should run only stateless processes, which does not share anything with other instances and all information required must be persisted in a database outside of the container, allowing subsequent requests by the same user to be served by different instances of the service in a transparent way.

Despite its benefits, the microservice pattern is not free from defects. Having an application over multiple containers require careful management of the network, not only to guarantee communication among services even in case of crashes but also to ensure the necessary security requirements. Security should be an important phase of the design process: having multiple components that must be isolated and can access only information belonging to them require customized security policies inside the data center, while access from the outside world require a dedicated service that manages authentication and authorization process of requests. Despite having simpler components drastically simplify the testing and debugging process of every single function, which can be performed independently from the other services and that does not suffer from the entangled nature of the modules of a monolithic application, testing a microservices application at the system level require interaction between components over the network, therefore it is more complex to track and identify the source of a bug.

The management at run time of such a complex architecture is not a feasible task for humans, therefore ad-hoc software platforms called *"orchestrators"* that can automatically manage both the containers and the network configuration, dealing with failures and scaling, are used for this purpose. Some of the most used solutions are Kubernetes[8], Docker Swarm[9] and Mesos[10]

5.2.1 The twelve-factor application

In 2011, Adam Wiggins with others Heroku developers proposed a set of twelve principles[11] that describe how to build cloud-native "Software as a Service" application based on microservices, easy to scale, reliable and maintainable. The proposed principles are here described.

Codebase the code and the required files for the deployment is stored in a single repository under a version control system and only this repository is used for deployments;

[8] https://kubernetes.io/

[9] https://docs.Docker.com/engine/swarm/

[10] http://mesos.apache.org/

[11] https://12factor.net/

Dependencies all external dependencies are explicitly declared in a file, not stored in the codebase and without implicit reliance;

Config the application is agnostic to the environment where it is executed, therefore configuration data is injected as environment variables and it is not written in the code files;

Backing services external systems as databases or message brokers are treated as attached resources reached through the network and managed independently from the application;

Build, release, run there is a strict separation between build, release and run stages and each stage is independently replicable;

Processes the app is executed as one or more stateless processes and no session data is kept inside the instance execution environment;

Port binding the service is exposed to the outside via port binding on a well-defined port, it is not dependent on an external webserver;

Concurrency services are separated by purpose and each service can be scaled up or down independently from the others when required, allowing horizontal scaling;

Disposability the application can start fastly and can stop gracefully, without affecting the user and keeping a coherent state when shut down;

Dev/Prod parity the gap between development and production environment is kept as minimal as possible therefore it is easy to move a version of the application from the former to the latter environment;

Logs logs data generated by services are print on the standard output as a stream and are not written in a file, the execution environment collects and aggregates them and another service can perform processing;

Admin processes admin and management tasks are treated as part of the application, therefore they are kept in the same codebase and are executed as automated one-off processes when required.

5.3 Kubernetes

Kubernetes is an open-source orchestrator to automatically deploy, scale and manage containers. It was designed by Google, derived from its Borg project, in 2015; it reached version 1.22 and the release of version 1.23 is expected for December 2021[12] and since version. Since its first release, it has been managed by the Cloud Native Computing Foundation[13] (CNCF) which Google is part of. It is currently one of the most used orchestrators, several customized versions have been developed both for private users and for the enterprise world (e.g. Red Had OpenShift[14]) and Kubernetes clusters are offered in the cloud by all the main cloud provides. The following sections describe Kubernetes architecture components and working logic [62].

5.3.1 Kubernetes architecture and components

A kubernetes installation is called *cluster*. Each cluster is composed of a set of machines called *nodes* and a *control plane*. Nodes are the base components of the cluster, they run containerized applications and report their status to the control plane, the latter manages worker nodes in the cluster and make decisions reacting to events. Kubernetes supports several container runtimes,

[12]https://www.kubernetes.dev/resources/release/

[13]https://www.cncf.io/

[14]https://www.redhat.com/en/technologies/cloud-computing/openshift

not only Docker but also any other that supports the Kubernetes Container Runtime Interface (CRI) such as *cri-o*[15]. The cluster behaviour is managed through a set of control loop processes: each controller watches the state of a resource and performs action trying to shift the current state toward the desired state for that object. Figure 5.4 describes the main components of a Kubernetes cluster:

Controller manager is the control plane components the runs the controller processes. All controllers are compiled into a single binary file that runs in this component;

Cloud controller manager is the control plane component that manages cloud-specific control logic, linking the cluster to the cloud provider's API. This component is not present in clusters that do not belong to any providers. Its work logic is the same as the controller manager, but watching different resources;

API server is the front-end component that exposes the API: it receives REST requests, manages them and provides replies. It can scale horizontally, deploying multiple instances of this component;

Etcd is a distributed key-value store used as backing store for cluster data and to share data among cluster members;

Scheduler is the control plane component that watches for new pods (a set of containers) and assigns them to a node to run on based on several parameters like resource requirement, policies and hardware constraints;

Kubelet is an agent that runs on each node in the cluster, it checks the status of containers running on the node and reports problems to the API server;

Kube-proxy is a network proxy that runs on each node, managing network rules on its node to enable the required communication.

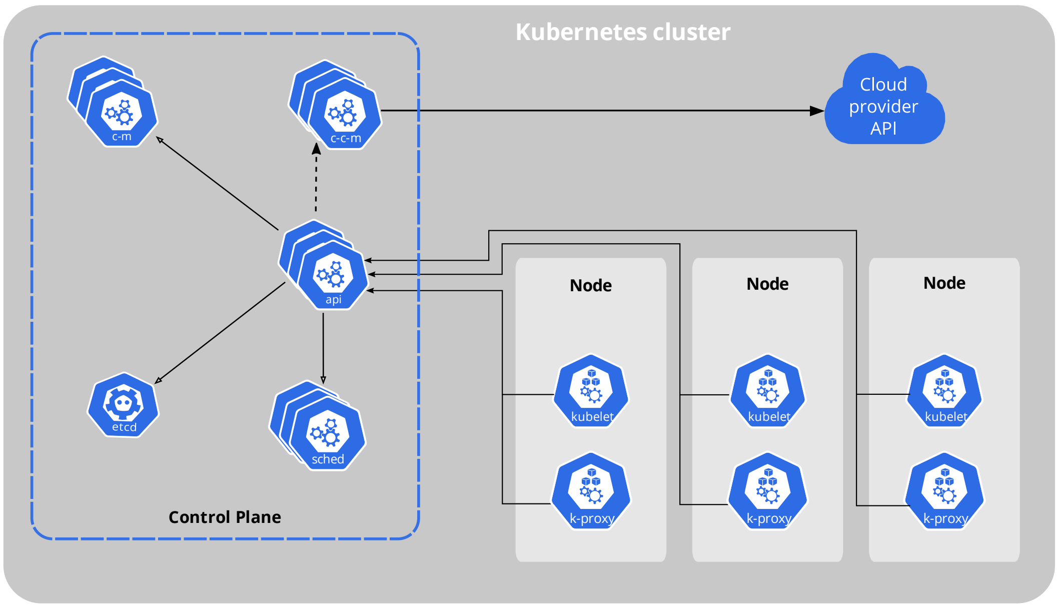


Figure 5.4. A Kubernetes cluster with its main components. Source: Kubernetes documentation

All the configuration elements in a Kubernetes cluster are called *resources*. Each resource, despite its function, is described and deployed in the same way, via its `.yaml` configuration file,

[15]https://cri-o.io/

which contains the `apiVersion` and the `kind` fields that uniquely identify the object, the `spec` field that describes the expected object state and the `metadata` field that contains other additional information. Each resource can be reached via its corresponding endpoints in the Kubernetes API, which allows retrieval of related information. Figure 5.5 reports an example Kubernetes cluster schema with some resources allocated that can help in visualizing how they are related to each other. Here the main pre-defined resources are described:

Namespace is the "logical cluster" resource that allows introducing scope for objects names and limits access and visibility of components inside the same physical cluster;

Pod is the minimum and basic element in Kubernetes. It is composed of one or more highly coupled containers and volumes, it will be spawned and executed on a single node. Additional containers over the main one are called sidecars and should be placed in the same pod only if they extend and enhance the main one. Containers in the same pod share the same network stack and can reach each other on `localhost` address. Pod supports *init containers*, specialized containers that run in the pod before the main one, for setup purposes;

Replica Set is the resource that specifies the static number of replica required to run at the same time for a specified pod;

Deployment is the object that allows managing pods version and their replicas at a higher level of abstraction;

Horizontal Pod Autoscaler is the object that allows defining scaling rules for another object (like a `deployment`) based on metrics like resource usage or custom application metrics and performs automatic scaling;

Service is the object that exposes a single stable access point for an application that may run on multiple pods, decoupling the exposition of a service from its execution units. The service can be of type `ClusterIP` (reachable only from within the cluster, over a cluster-internal IP), `NodePort` (reachable also from the outside on a static port), `LoadBalancer` (reachable from the outside using a cloud load balancer) and `ExternalName` to map the service to a DNS record;

Ingress is the object that exposes the cluster to the outside and manages external access via routing and filtering rules;

Config Map is a set of key-value pairs that can be injected into pods as files or environment variables;

Secret is an object used to inject into pods secret information (e.g. passwords), that is stored encrypted in `etcd` storage;

Network Policy is the object that implements firewall policies across pods to limit access among them. Filtering rules can be specified with IP address, pod name or namespace;

Resource Quota is the resource that allows to limits the amount of objects that can be created in a namespace, defining limits is computing and memory resources;

Limit Range is the resource that defines resource limits for each pod or each container in a pod.

5.3.2 Custom resources and operators

Kubernetes allows extending its standard behaviour via the definition of *custom resources*: resources that are not available by default in the cluster and that provide customized communication with objects of some kind. They are defined as a set of available HTTP methods with a corresponding path that may go over the standard communication pattern of REST APIs. In this way, the cluster admin can easily implement communication patterns with any kind of object that can consume requests and react to API calls. Custom resources are defined via objects of *CustomResourceDefinition* (CRD) kind, which are handled by the built-in API server that is able

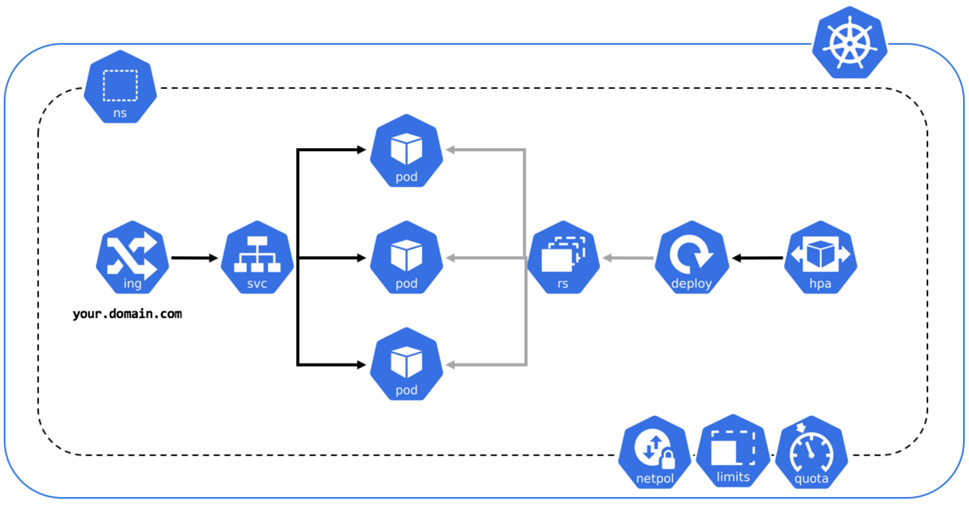


Figure 5.5. An example of a Kubernetes cluster with some allocated resources. Source: GitHub

to understand and serve custom requests, freeing the developer from writing its own API server. If the functionalities offered by the build-in API server are not sufficient for a specific purpose, Kubernetes can be also extended with a custom API server that serves those API calls with custom logic.

To exploit the full capability of custom resources they are often used in the *Operator* pattern. Operators are applications designed to watch and react to specific CRD manipulations, following the standard "control loop" behaviour of the control plane, used to automate some actions on the cluster such as the deployment of custom resources or periodical backups of application data. The easiest way to build an operator is to add a custom *Controller* that runs on the cluster, associated with a CRD. Despite controllers can be implemented in any programming language, because indeed they are simple API consumers, most of the code can be generated with some tools, because is often similar between several operators, simplifying a lot the development process.

More information on these concepts is available in the Kubernetes official documentation, in the "Extending Kubernetes" section[16].

5.3.3 Security of Kubernetes clusters

Kubernetes takes full advantage of all security advantages derived from the microservices pattern: pods guarantee, by default, isolation among different application components from the OS point of view and built-in resources like limit ranges and resource quota can limit malicious behaviour in resource consumption. Security should also be addressed at the cluster level, not only at the container level [63]: a cluster is way more complex than a single containerized application and can expose a higher attack surface, that should be carefully analyzed. Regarding security from inside the cluster, namespaces and network policies resources can provide limitations on the components that can be seen and reached, moreover, Kubernetes supports encryption through TLS protocol for communication between the control plane and worker nodes agents to keep it private and authenticated.

A big threat for any cluster comes from the outside: exposing not only services but also the cluster administration API to the outside require them to be carefully protected through strong authentication and authorization mechanisms. Kubernetes provides a small number of built-in mechanisms for authentication, only suitable for not production environment or very basic configurations as basic HTTP authentication or x509 certificates, and more secure options like

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

OpenID Connect token received by external identity providers or any other protocol that can be integrated via an authenticating proxy or the authentication webhook feature[17] such as LDAP, SAML or Kerberos. Authorization can be managed either with *role-based access control* (RBAC) or with *attribute-based access control* (ABAC) mechanism, in both cases defining policies for each role or attribute as cluster resources.

5.4 Beyond cloud-computing

The research about virtualization technologies and the growth in devices computing power lead to the development of new technologies in the cloud world that exploit the knowledge acquired with microservices applications and container management to introduce cloud-computing concepts in several fields.

In the network world, the main advance is the *Network Function Virtualization* paradigm [64]. It proposes to take advantage of the virtualization capabilities of modern computers to run the required network functions not on specialized hardware (NATs, routers, firewalls) but on its virtualized copies. In this way, ISPs can drastically reduce devices and operation costs, using common off the shelf hardware, and improving scalability because they can remotely deploy new virtual devices when required in a fast and easy way and a more reliable manner.

The spread of cheap but powerful hardware and the ubiquity of internet connection leads to the development of more and more connected objects: the *"Internet of Things"* (IoT) [65]. These IoT devices can be deployed in several everyday life objects, such as smart speakers, cars, home appliances, thermostats or medical equipment: they obtain data to share among them or with a cloud server that processes them to provide useful information to users. This paradigm is strictly related to the idea of fog and edge computing [66]: these paradigms extend the concept of cloud computing, moving part of the application from the cloud node to the edges of the network. Devices can obtain data from other sensors and devices and pre-process and store them before transmitting them to the data center. This is crucial both for high traffic applications, which can reduce the amount of data transmitted over the network by aggregation and filtering processes and for applications that are highly sensitive to latency, which can have a faster response from nearer nodes. Both fog and edge computing follow the same principle of moving computation at network edges and the two words are often erroneously interchanged but fog computing differs for a hierarchical and seamless structure that spans from the edges to the cloud and that is not limited to a single level of external nodes.

All these paradigms can be referenced as *"Softwarized infrastructures"*: infrastructures that strongly relies on virtualization paradigms and an internet connection to solve complex tasks in a scalable, efficient and reliable way.

[17] https://kubernetes.io/docs/reference/access-authn-authz/authentication/

Chapter 6

Quantum Key Server

The Quantum Key Server (QKS) is an implementation of a node in a QKD trusted repeater network, compliant with the ETSI GS QKD 014 standard [52] described in section 4.1.3, whose main purpose is to securely distribute QKD-derived keys from the underlying layer to authorized applications even in multi-hop scenarios typical of QKD networks.

A single QKS can be connected to multiple QKDMs (section 4.1.2) and can be reached by several authorized SAEs. Figure 6.1 describes the software stack architecture of a network node, composed of a single QKS, its attached QKDMs and the SAEs registered to it. This allows SAEs to only communicate with their QKS, asking it keys and information despite the location of their peer SAE, without any knowledge about the network or the QKDMs implementation and without managing the synchronization aspects of the exchange.

Pedone *et al.* in 2021 proposed a first implementation of the stack, developing the QKS, the QKDM and a simulator for QKD protocols to provide a complete working solution of a QKD network node. This chapter analyzes the QKS design and its architectural components, describes the interfaces, presents the QKDM and the simulator implementation and analyze the criticalities of this first version, while chapter 7 propose an updated version that tries to overcome these limitations.

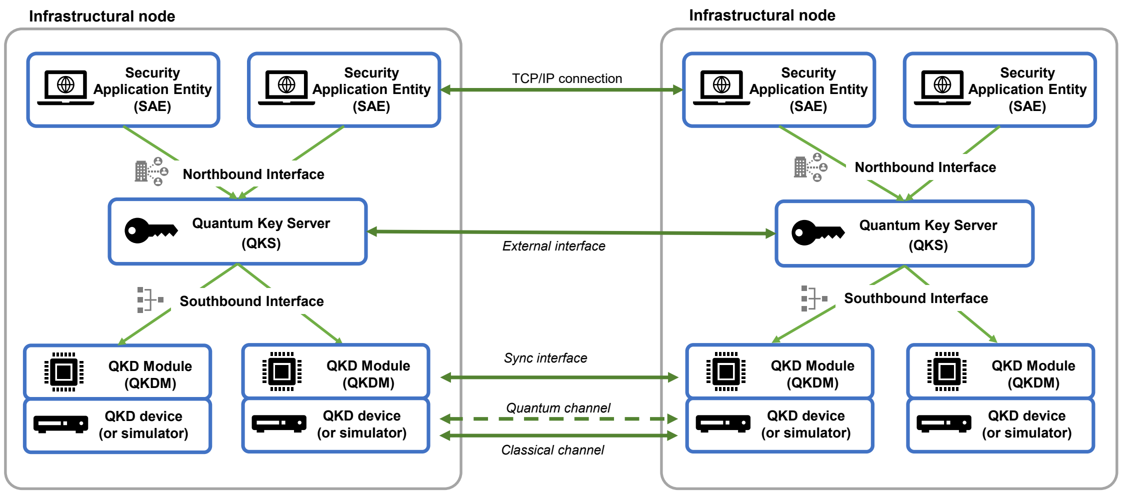


Figure 6.1. QKD software stack architecture

6.1 Design and architecture

The QKS has been designed trying to follow the microservices pattern presented in section 5.2: for this reason, different modules interact only over the network and that can run on different hosts. The ETSI does not provide any details on how the QKS should be implemented, neither regarding the language nor the architecture, therefore this solution is designed trying to keep each component as less coupled as possible, allowing it to be substituted with any other component from different manufacturers. The following paragraphs present the components of the QKS, while figure 6.2 describes how they are connected and which interfaces are used to communicate.

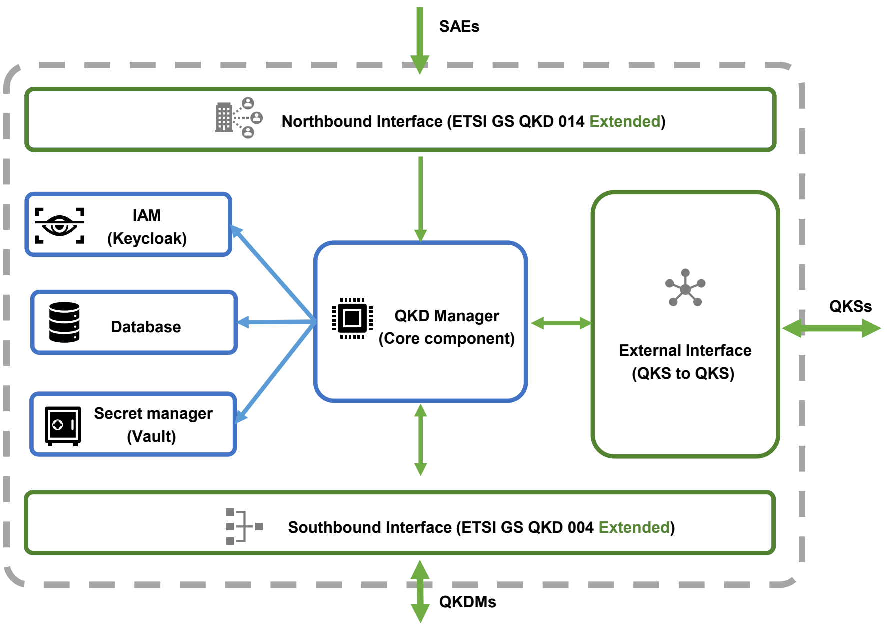

SAEs
Northbound Interface (ETSI GS QKD 014 Extended)
IAM (Keycloak)
Database
Secret manager (Vault)
QKD Manager (Core component)
External Interface (QKS to QKS)
QKSs
Southbound Interface (ETSI GS QKD 004 Extended)
QKDMs

Figure 6.2. QKS components and interfaces

QKD manager (core)

The QKD manager is the core component of the system. It is the one that implements ETSI API to provide keys to SAEs (ETSI GS QKD 014), that communicates with QKDMs to ask them keys (over interface presented in ETSI GS QKD 004 described in section 4.1.2) and communicates with other nodes over an “External Interface” to manage active key exchanges. It is designed to be scaled up and down when needed, working in a completely stateless way: no information is saved in memory, everything is stored in the database. It can aggregate different fixed-size keys received from QKDMs into a single key of any arbitrary length, as requested by SAEs.

Identity and Access Manager

An identity and access manager (IAM) component is required to guarantee authentication and authorization for SAEs and QKDMs. Regarding SAEs this is essential to guarantee that the SAEs that try to access the QKS have the right to do it and to ensure that some operations are performed only by system administrators; while regarding QKDMs authentication is required to guarantee that the connected QKDM is a trusted one. When an entity performs a request to the QKS, both over the Northbound and the Southbound interfaces, it is redirected to the IAM server through and the request is handled by the QKS only if this operation is completed successfully.

Database

The database component is used to store information about the other QKS in the network, the QKDM registered to the node and the active key exchanges. There is only one DBMS for each node, that contains information for the core component and all the attached QKDMs With a stateless workflow, all information must be saved in the database and the latter acts as the synchronization platform, therefore queries are performed to guarantee integrity even with multiple instances of the QKD manager.

Secret manager

Keys are exchanged by underlying components and managed by QKDMs. These data require secure storage, because they should not be saved in clear and accessible by everyone, therefore this component has been introduced. There is a single secret manager in the node, that stores keys belonging to different QKDMs. The QKS does not use directly Vault because it receives keys from QKDMs but must manage their access and the secret engines life cycle.

6.1.1 Key Streams management

This implementation of the QKS diverges from the idea proposed by ETSI about the management of SAEs and key streams. A key stream is a key exchange process between two connected QKDMs; that in the ETSI standard is associated with a SAE pair, that are the only actors allowed to access its keys. In this solution each couple of SAEs has its private key stream, leading to an exponential growth in the number of streams with respect to the number of SAEs in the network, the QKS is a middle layer that only allows SAEs to reach different QKDMs in a centralized way. The solution adopted in this work follows a more flexible approach, where the QKS is the only one in charge of managing the QKDMs and their streams, regardless of the binding between SAEs. A single key stream is created between each QKDM pair and the keys are continuously exchanged until the storage capacity is saturated, even if no SAEs required keys on that link: this allows to efficiently use the QKD link, avoiding dead times and growing up storage to deal with high loads in the network, and simplify the management of QKDMs that only has to work on a single stream.

6.2 Interfaces

The QKS communicates with other actors with three different interfaces: a *Northbound* interface used to receive requests by SAEs, a *Southbound* interface to talk to QKDMs and an *External* interface to communicate with other QKS in the network. Each interface is accessible through a web server on the core component, which is the only component directly accessible from the outside.

Northbound interface is the interface that maps the ETSI REST-based API [52] described in section 4.1.3 and extend it with new functionalities to support not only the key delivering process but also the management of QKDMs and key streams and the registration of the SAEs, directly through the QKS that acts as the only component to interact with both for SAEs and administrators.

Southbound interface is an extension of the ETSI Application Interfaces described in the ETSI GS QKD 004 [46]. While the main functions are exposed by the QKDM with a web server, which is controlled by the QKS itself, this interface extension is required to support the registration process.

External interface is the interface in charge of supporting the communication between QKS nodes in the network. The ETSI standard does not describe it in detail but give some hints on the required functions: it is used to manage the key streams life cycle and the synchronization of the keys.

6.3 QKD Module

The QKDM is an implementation of the QKD Key Manager described in the ETSI GS QKD 004 standard [46] that manages the QKD devices and provides keys to applications or upper-level entities like QKSs.

It is composed of only one module, the *Key Manager* which handles requests from the upper layer through the *Southbound Interface* and synchronizes itself with the peer QKDM through the *Sync Interface*, both managed through a web server. The QKDM controls the QKD device and uses it to continuously perform key exchanges, saving the received key in the secret manager, through an interface class. The device is the component that performs the key exchange, which must implement the `QKDcore` interface to deliver keys of predefined length to the QKDM. It can be both a physical device or, as in this implementation, a software simulator.

Besides the core module, the QKDM requires two additional components: a database and a Vault secret manager instance. It can receive their access data from the QKS it is attached to, or directly in the configuration file if it operates without an upper layer. Figure 6.3 describes the components and interfaces of a QKDM.

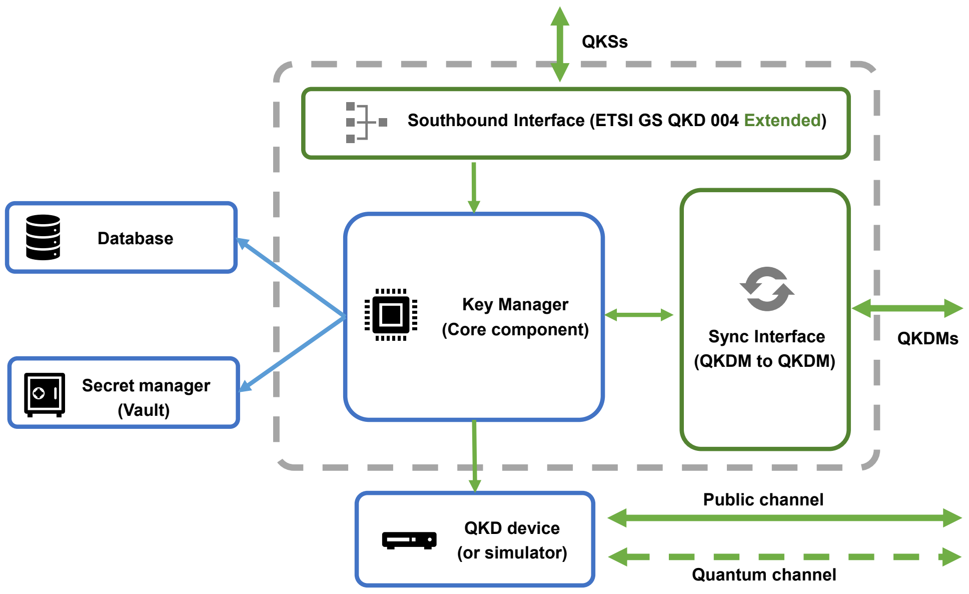


Figure 6.3. QKDM architecture and interfaces

The QKDM supports the *Southbound* application interface proposed in the ETSI GS QKD 004, extending it with new functionalities and implementing a new interface not described by ETSI, called *Sync Interface*, to communicate with the peer QKDM for synchronization purposes. The QKDM has been drawn as a simple component that only transfers keys from the physical device to the upper layer, that can be operated both alone both in combination with the QKS, therefore it does not implement any networking functionalities, that are instead managed by the QKS, and the extension of the ETSI interface only helps in the synchronization tasks but does not introduce any new functionality. Because the key manager is associated with a physical device (or its simulator) in a one-to-one relationship it has not been developed as a component able to scale up or down responding to traffic loads.

6.4 QKD Simulator

The simulator is a module developed to complete the stack that composes a QKD node in a network, allowing a complete testing process on the QKS and QKDM functionalities even without real QKD devices. It is composed of several elements:

QKD node is the core component that models the QKD device. It contains the software to simulate a QKD exchange and to manipulate qubits

Quantum channel and Eve is a QKD node that reproduces an eavesdropper behaviour over the quantum channel

Entanglement pairs generator is a component used to generate and provide to the nodes pairs of entangled qubits

QKD simulator manager is a central management unit that collects data and provides an interface for system management.

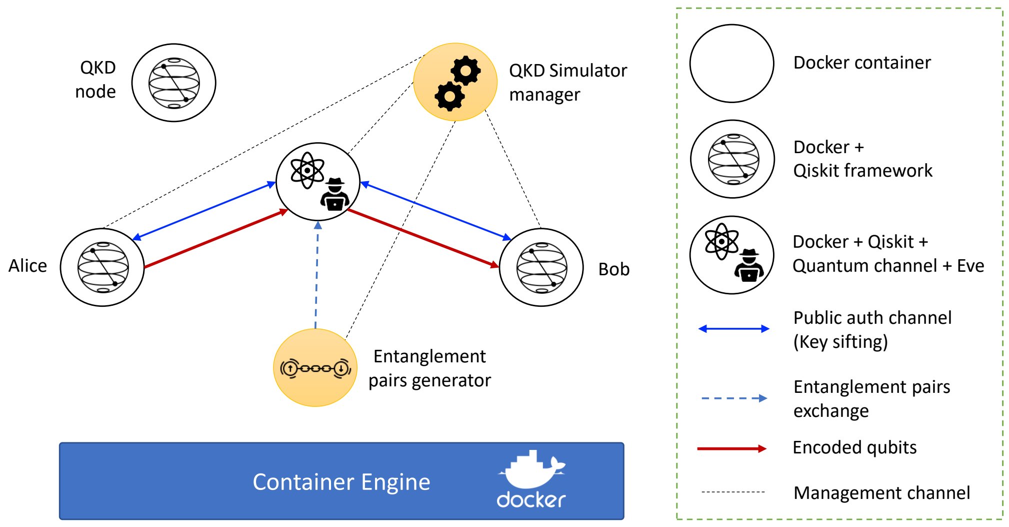

QKD node
QKD Simulator manager
Alice
Bob
Entanglement pairs generator
Container Engine
docker
Docker container
Docker + Qiskit framework
Docker + Qiskit + Quantum channel + Eve
Public auth channel (Key sifting)
Entanglement pairs exchange
Encoded qubits
Management channel

Figure 6.4. QKD simulator architecture. Source: [8]

Quantum phenomena and qubit manipulation are performed using the Qiskit Python framework, which allows simulating quantum operations and measurements through the description of quantum circuits and algorithms, treating sets of qubits as Python objects. that can be serialized to send them over the network. To reproduce a malicious actor that intercepts communications on the quantum channel a container that acts as an eavesdropper has been introduced: because it is impossible to mimic quantum channel properties on a TCP/IP communication, packets are diverted to Eve that can deserialize and manipulate them as common objects.

Currently, the simulator supports BB84 and E91 protocols, but thanks to its modularity it can be extended to other overriding methods of an abstract class that is used to manage the life cycle of key exchanges. This class is the one imported into the QKDM QKD core component to perform key exchanges. Devices are deployed as Docker containers, that communicates over HTTP requests handled with a web server: the basic setup is composed of two devices (Alice and Bob) but an eavesdropper entity can be deployed in a new container (Eve) and in the case of entanglement-based protocols the relative component can be added in the same way. The high-level architecture of the simulator is represented in figure 6.4

The public classical channel required by each protocol is implemented as a standard TCP/IP channel and two different strategies have been proposed to provide authentication: SPHINCS+ and AES with Galois/Counter Mode (AES-GCM) algorithms. More information about the implementation details and the obtained results can be found in the reference paper [8].

6.5 Criticalities

This implementation, despite being fully functional and providing a first solution in the deployment of QKD in a softwarised infrastructure presents some major criticalities. In terms of functionalities, it does not support multi-hop key exchanges setting a not negligible limitation in the usability, especially in a scenario where QKD is being gradually adopted and therefore a large number of QKD devices is not available from the early stages. Moreover, it does not support networking functionalities and the SAEs location is discovered through a polling process (a QKS ask all the other QKS in the network if the requested SAE is registered on their node) that generates a lot of network traffic and a slow convergence time.

Regarding performances this version does not reach a sufficient speed to be used in a real-world scenario with several SAEs and nodes: on average it requires a couple of seconds to share a key of proper length between two peers. This is caused by the not optimal management of the interactions between components, which are synchronous and therefore block the system while waiting for a response, and the usage of locks to prevent race conditions among QKS replicas.

From the security point of view the presence of a single database and a single secret engine shared by all components, that does not provide any form of isolation and assigns to each element the same credentials and the same authorizations is a major vulnerability that should be addressed: if a QKDM gets compromised an attacker can see all the keys stored in the secret engine and all the data in the database and can perform administration task tanks to the credential shared with the QKS received in the module registration process. Some other relevant security issues have already been identified in the reference paper [8], regarding the communication between components and network nodes Communications that travel outside the infrastructure borders (horizontal communication) require both authentication and encryption: the former because peers must be sure that their peer is trusted, the latter because even if no keys are exchanged in clear over these interfaces they can leak data about the network topology and its status. Currently, the QKS-to-QKS messages are not protected and, even if it is extremely easy to set up a channel secured with TLS, it is essential to consider that in a quantum scenario using algorithms that are not quantum-safe does not provide real security, therefore the only viable options are the one that relies on post-quantum cryptography or that directly uses keys obtained by QKD devices. Despite the ETSI standard proposes TLS 1.2 as the minimum requirement for communications between components on the same logical network node (vertical communication), a similar approach to the one proposed for horizontal communication should be considered, also because keys are exchanged over the Southbound and the Northbound interfaces and in a cloud environment there is no guarantee that components are deployed on the same physical machine.

Chapter 7

Quantum Key Server 2.0

The Quantum Key Server 2.0 is the core element of this work. This new version aims to overcome the issues of the first version in terms of speed with the use of new components and a change in the programming pattern and implements support for long-distance key exchange thanks to the development of a new routing module. This chapter describes the enhancements, focuses on interfaces and implementation details and shows a standard workflow for the application.

7.1 Overview

In the early adoption stages of QKD, networks will have small amounts of QKD devices, hence not all nodes will be able to share keys with PTP exchanges. Moreover, in real scenarios, PTP exchanges are limited by a distance of hundreds of kilometres: the development of multi-hop key exchanges procedure allows distant peers to share keys broadening the usability of the system. The new routing module build paths to reach other peers and managing dynamic scenarios where QKDMs and SAEs are registered and removed frequently, enabling the support for long distance exchanges in a large network. The workflow of this exchange process is described in section 7.7, while a detailed description of the routing module can be found in section 7.5

The first QKS version does not ensure a response time that is compatible with current protocols and does not provide a key retrieval rate that reaches the one provided by physical devices. These issues are caused by synchronous calls interacting with other modules and a not efficient management of the concurrency, thus a new asynchronous approach has been used to overcome them and a new strategy has been used to guarantee data consistency among multiple QKD manager instances without using any lock that can keep the whole system in a waiting state. Detailed information on the asynchronous behaviour is shown in section 7.4.1 Regarding secure access, the database and the secret engine management have been improved allocating a private database and a secret engine for each QKDM during the registration phase: in this way, each QKDM can only access its information and its keys and is completely isolated from the others.

7.2 Architecture

This version keeps a similar architecture to the first one presented in the previous chapter because it demonstrated high flexibility and has been designed to be integrated into a cloud-native scenario. Two new components have been added to improve the networking functionalities, keeping the separation of duties of the microservices pattern: the routing module and a distributed cache. This section shows the chosen products, while figure 7.1 present the QKS architectural schema. The architecture is not bound to specific products, hence components can be replaced with others that accomplish the same tasks. Each component has a corresponding Docker image that is used for the deployment.

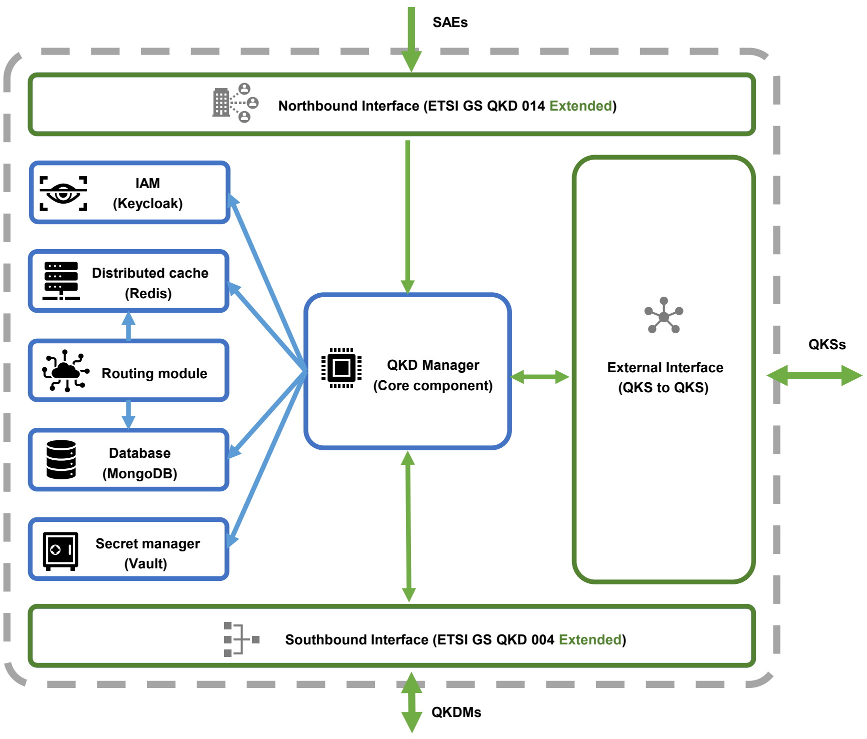

SAEs
Northbound Interface (ETSI GS QKD 014 Extended)
IAM (Keycloak)
Distributed cache (Redis)
Routing module
Database (MongoDB)
Secret manager (Vault)
QKD Manager (Core component)
External Interface (QKS to QKS)
QKSs
Southbound Interface (ETSI GS QKD 004 Extended)
QKDMs

Figure 7.1. QKS 2.0 components and interfaces

QKD manager (core)

The QKD manager extends the one in the previous version with support to multi-hop key streams and exploiting the asynchronous pattern. While the core functionalities for key retrieval have been kept unchanged, changes have been performed in the code to produce a faster and more reliable API server that can handles REST calls received over all three interfaces. In the case of long-distance peers, it performs multi-hop exchanges generating a key that is forwarded encrypted with QKD keys in a hop-by-hop manner to the destination. It is responsible for the creation of QKDM databases and secret engines, and their deletion when the modules are unregistered from the node.

Identity and Access Manager

An identity and access manager (IAM) component is required to guarantee authentication and authorization for SAEs and QKDMs. *Keycloak*[1] is an open-source IAM that can simplify the management of both authentication and authorization policies in an environment with a high number of actors. When an entity performs a request to the QKS core, either over the Northbound or the Southbound interfaces, it must be authenticated with the OpenID Connect[2] protocol. Each SAE, administrator and QKDM has a corresponding *user* and the QKS is described as a *client*. Before performing requests actors should require an access token logging into Keycloak with their credentials. The core components receive the token embedded in the request headers and forwards it to Keycloak which performs validation. The username and the roles of the sender are used to determine if the request is valid or not. Administrators must register SAEs and QKDMs before initializing them, providing them credentials and configuration data needed to require their token.

[1]https://www.keycloak.org/

[2]https://openid.net/connect/

Database

MongoDB[3] has been chosen for this module because it is a fast and distributed database with native support to replicas, hence it can easily scale up when needed. A NoSQL document-based database is a crucial choice for this application: each action performed by the QKD manager require a lot of information, hence saving them in a single document and not sparse over multiple tables allows to reduce query complexity and optimize the transfer rate reducing the number of transactions. Moreover it simplifies the data management at application level because MongoDB objects can be easily manipulated in Python with dictionaries. A detailed description of the MongoDB collections and how data are organized can be found in section 7.4.2.

Secret manager

Vault[4] is an open-source secret manager developed and distributed by Hashicorp that allows to securely store structured data, allowing data to keep the same structure used for MongoDB, accessible via REST API. The QKS uses Vault to temporarily store secret keys obtained through multi-hop exchanges that are not managed by QKDMs, over than managing the secret engines life cycle when QKDMs are registered or removed. Each QKDM uses it private secret engine to store data retrieved from the corresponding QKD device. A detailed description of the data organization can be found in section 7.4.3.

Routing module

The routing module is the component that manages the information about the network topology and the status of each QKD link. This component sends and receives information over TCP connections from its neighbours in the network through a customized routing protocol based on OSPF [25]. The module updates the network topology based on data received from its peers and on messages from the core component over Redis topics, then it pushes routing tables to the distributed cache, allowing fast access for the QKD manager. This module is not designed as a scalable component: it is not a stateless application, the network topology is stored in the main memory and the received packets are not saved anywhere, therefore multiple instances can not share the same data guaranteeing consistency. Despite this pattern does not respect the guidelines for cloud-native applications, it drastically reduces the complexity of the module, removing the need for a distributed graph structure, without a high impact on performances: with the current size of QKD networks of hundreds of nodes and tens of neighbours at most, a single instance can easily manage the routing work by itself, as demonstrated in the tests 9. This module is described in detail in section 3.1.2

Cache and message queue provider

The core component requires fast access to routing tables generated by the routing module, while the latter requires updates from the former when change happen in the registered SAEs or QKDMs. *Redis*[5] is an open-source in-memory data store that is used both as distributed cache and as a message broker. It has been chosen because it is faster than a common database, which helps in the management of routing tables that are updated and accessed frequently and because it supports message queues and hence avoids the need for another component, keeping communication between the core and the routing component in a single element. The routing module writes routing tables on Redis cache, while each QKD manager instance publishes updates on the message queue on a shared topic, exchanging information asynchronously. This pattern ensures data consistency in presence of multiple instances of the core component and avoids any

[3]`https://www.mongodb.com/`

[4]`https://www.vaultproject.io/`

[5]`https://redis.io/`

time loss due to unavailability of the peer if a crash occurs. Detailed information on Redis can be found in section 7.4.4

7.3 Interfaces

The three interfaces presented in the previous chapter are kept in this QKS version, but they have been extended to support new functionalities or to simplify the management of the system for the administrators. This section describes in detail the methods of each interface and their enhancements to the ETSI standard.

7.3.1 Northbound interface

The Northbound interface has been extended to support multi-hop key streams and the related key exchanges, that is performed by the QKS, and provide better management of QKDMs regarding QKDMs key streams avoiding the need for administrators to directly interact with the underlying modules. All the methods in this interface require authentication: they must be called by a user with `sae` or `admin` role, that must authenticate himself against Keycloak, embedding the returned token in the `Authorization` header. The functions are described here below, while table 7.1 summarizes them and their access URL.

getStatus is a function of the ETSI standard. It is used to retrieve information about the available keys that can be shared with the SAE specified in the `slave_SAE_ID` path parameter.

getKey is the ETSI function called by the first SAE to retrieve keys of any length and their corresponding identifier. If there are not enough keys all the available ones are returned, trying to satisfy the request as close as possible. In the case of long-distance exchange, this function generates the key and forwards it to the next-hop through the `forwardData` function of the external interface, matching the size and number in the request. During the execution of this method, a `reserveKeys` request is sent to the peer QKS and keys are returned only if it is successful. This version supports only one extension, the `require_direct` that can force the QKS to return nothing if there is not a direct QKD link. The response body always contains a field to specify if the returned keys are retrieved from a direct or a multi-hop stream. Keys are returned encoded in base64 [67].

getKeyWithKeyIDs is the ETSI function called by the peer SAE to retrieve keys with the identifiers received from the first SAE. Because SAEs does not know stream details, this method can retrieve keys from different streams. It returns all the available keys among the requested ones: if some are not available the methods successfully returns only the available ones, matching the `getKey` behaviour. Keys are returned encoded in base64.

getQKDMs returns the list of the QKDMs connected to this QKS, along with their information such as the protocol, the standard key size and the peer QKDM. It is not part of the ETSI standard and can be called only by admins.

registerSAE is the method used by SAEs to register themselves to the server. It saves the SAE identifier and sends it to the routing module through Redis so that the SAE can be discovered in the other network nodes. The SAE identifier must be unique not only in the node scope but in the entire network. It is not part of the ETSI standard. If called by a SAE it can register only itself, an administrator instead can register any SAE.

unregisterSAE is the method opposite to the previous one: it is used to unregister a SAE from the QKS, deleting its information and notifying the routing module about that. After the unsubscribe process the SAE can be registered to any other QKS in the network. If called by a SAE it can unregister only itself, an administrator instead can operate on any SAE.

startQKDMStream is the method called to start a new direct stream between a pair of QKDMs. This QKS implementation only supports one stream between each QKDM pair, therefore this function fails if a stream is already present. The QKS invokes `open_connect` on the QKDM and then informs the destination QKS calling the `createStream` function of the external interface, therefore the synchronization is managed automatically by the QKS and it is not required to call this function on the peer. When the stream has been created successfully the information is pushed to Redis and the routing module shares it with the other nodes. It is not part of the ETSI standard and can be called only by administrators.

deleteQKDMStream is the method opposite to the previous one. It is used to delete the key stream between a pair of QKDMs. It should be called before the unsubscribe process of the connected QKDM, to safely remove it. The QKS invokes the `close` method on the QKDM and informs the destination QKS calling the `closeStream` function of the external interface, therefore the method should be called on one QKS only. The information is pushed to Redis so that can be shared through the network and the link will be considered inactive until a new stream is opened. It is not part of the ETSI standard and can be called only by administrators.

registerQKS is a method used to register a QKS in the network. This process is required only once when a new QKS is connected to the network for the first time, and it should be registered to the peer QKS it will be connected to with a QKD link: its information will be spread by the routing modules reaching all the other nodes after the first stream has been created, but a QKS will refuse a connection with a QKDM that allows reaching an unknown QKS. It is not part of the ETSI standard and can be called only by administrators.

Method name	**URL**	**Access Method**
getStatus	/api/v1/keys/{`slave_SAE_ID`}/status	GET
getKey	/api/v1/keys/{`slave_SAE_ID`}/enc_keys	POST
getKeyWithKeyIDs	/api/v1/keys/{`master_SAE_ID`}/dec_keys	POST
getQKDMs	/api/v1/qkdms	GET
registerSAE	/api/v1/saes	POST
unregisterSAE	/api/v1/saes/{`SAE_ID`}	DELETE
startQKDMStream	/api/v1/qkdms/{`QKDM_ID`}/streams	POST
deleteQKDMStream	/api/v1/qkdms/{`QKDM_ID`}/streams	DELETE
registerQKS	/api/v1/qks	POST

Table 7.1. QKS Northbound interface methods summary.

7.3.2 Southbound interface

The Southbound interface does not provide any new functionalities but allows delivering private access data for each QKDM to access their database and secret engine. The two functions are described here below, while table 7.1 summarizes them and their access URL. Both methods require authentication: they must be called by a user with `qkdm` or `admin` role, that must authenticate himself against Keycloak, embedding the returned token in the `Authorization` header.

registerQKDM is called by the QKDM that want to start the registration process. It is the function that manages the registration by saving QKDM information received in the request body and that creates the new database and secret storage accessible only to the subject QKDM, returning them in the response body. The registration process does not create any stream by default. If this method is not called by an admin the module identifier must match the username retrieved from the authentication token.

unregisterQKDM is the function that performs the opposite job with respect to the preceding one: it deletes the QKDM database and secret engine, if and only if there are no active key streams for that module. This function is not called by the QKDM itself but by an administrator.

Method name	**URL**	**Access Method**
registerQKDM	/api/v1/qkdms	POST
unregisterQKDM	/api/v1/qkdms/{`qkdm_ID`}	DELETE

Table 7.2. QKS Southbound interface methods summary.

7.3.3 External interface

The External interface is the interface in charge of supporting the communication between QKS nodes in the network. The ETSI standard does not describe it but give some hints on the main functions: it is used to manage the synchronization of streams and extend the previous version with support to multi-hop key streams required when two nodes are not directly connected by a QKD link. Despite there were faster alternatives, such as messages exchanged over TCP sockets, the REST interface has been kept to guarantee consistency among every interface exposed by the QKS and because it does not introduce relevant overheads in the requests execution time.

createStream is used to notify the destination QKS about the creation of a new key stream, both for streams over single and multiple QKD links. Tt calls the `open_connect` function on the underlying QKDM. The creation of a new stream is signalled to the routing module through Redis.

closeStream is used to notify the closure of a key stream. The QKS invokes the `close` method on the QKDM and notifies the routing module through Redis.

reserveKeys is the function used to synchronize the key retrieving procedure. It notifies the destination that the master one has delivered a set of keys to a SAE, and their identifiers are sent in the request body: in this way, the peer QKS can reserve those keys for the right SAE and will not allow any other application to retrieve them.

forwardData is the function that allows sending encrypted data in a hop-by-hop manner. Data received through this function are decrypted and, if the destination is not the current QKS, encrypted again with a new key and sent to the next hop in the path. If an error occurs in the forwarding chain the error is notified back in the response body. It is used to forward keys for multi-hop exchanges to the end peer.

Method name	**URL**	**Access Method**
createStream	/api/v1/streams	POST
closeStream	/api/v1/streams/{`key_stream_ID`}	DELETE
reserveKeys	/api/v1/keys/{`master_SAE_ID`}/reserve	POST
forwardData	/api/v1/forward	POST

Table 7.3. QKS External interface methods summary.

7.4 Components details

The QKS has been designed as a cloud-native application, following the microservices pattern, to create a scalable and maintainable software stack. Each component is developed as a separate service, hence it interacts with the others only through the network, can run on a different physical host and can be substituted with a new version (or a new product, in the case of components by external vendors) without affecting the system. To reach this goal each component corresponds to a separate Docker container, either built from a custom image or the official image provided by the component developer, and all the interactions are carried over the network. The QKS code is available on GitHub [6], where also a Docker compose file can be found to simplify the deployment. This section describes why and how the asynchronous pattern has been used and the details of the components chosen.

7.4.1 Asynchronous pattern

The first QKS version has some limitations in terms of parallelism that slow it down remarkably: the software has been developed in Python 3 following a multithreaded synchronous approach that does not efficiently exploit the CPU resources. Multithreading in Python is limited by the *Global Interpreter Lock*[7] (GIL), a mutex lock managed by the interpreter itself that blocks concurrent access to the same object preventing multiple threads from executing Python bytecodes at once. Despite it ensures the absence of race conditions, it forbids the program to run on multiple processor cores. While the use of other interpreters that have removed the GIL, is indeed possible, the standard *CPython* interpreter has been preferred also in this new version due to its compatibility with tools and libraries and its average speed. The asynchronous pattern has been introduced since Python 3.4 to provide an event loop and non-blocking version of I/O functions: tasks are assigned to the event loop, that pauses them until they are not completed, and only at that moment they are resumed and the computation can continue. Because the event loop knows which tasks are ready to be resumed it can avoid time wasted on operations in a waiting state. Tasks are not spawned in a new thread, but a fixed number of threads is allocated at the event loop creation and tasks are assigned to a thread when the previous one is completed, to reduce the time spent for the context switching procedure to swap out the currently active thread.

The two components that have been coded in this project are the routing module and the QKD manager. Both have been developed in Python, version 3.9. The QKD manager exposes the REST APIs through the Quart[8] framework which supports `asyncio` standard library over the Hypercorn[9] asynchronous web server to handle incoming requests, while the routing modules directly utilizes asynchronous sockets for TCP connections. Each operation that can be parallelized is executed in a task and pushed in the main event loop; the interactions with MongoDB, Vault and Redis and the requests to QKDMs and other QKSs are performed through asynchronous drivers and libraries. To allow multiple replicas of the QKD manager working in parallel the locking access to the database of the first version has been removed, ensuring consistency through checks inside queries and not by blocking the other instances.

7.4.2 Data model

This section wants to describe how data are organized and managed inside the QKS application and in the MongoDB database. Each QKS and QKDM has a separate database that is used only by itself, even if they are hosted on the same MongoDB instance for performance and memory

[6]`https://github.com/ignaziopedone/qkd-keyserver/tree/async`

[7]`https://wiki.python.org/moin/GlobalInterpreterLock`

[8]`https://gitlab.com/pgjones/quart`

[9]`https://gitlab.com/pgjones/hypercorn`

reasons. The QKS creates the database for the QKDMs during the registration phase, along with the users and roles required to access them privately.

The data structure inside the database takes full advantage of the No-SQL document-based architecture of MongoDB: in such architecture tables are replaced with collections and data are stored in JSON documents that do not have a fixed structure, but can be organized in any way. Each object is identified by its `_id` field, which must be unique in the collection. Documents are retrieved and inserted through asynchronous drivers that provide a non-blocking behaviour and are managed in Python as dictionaries.

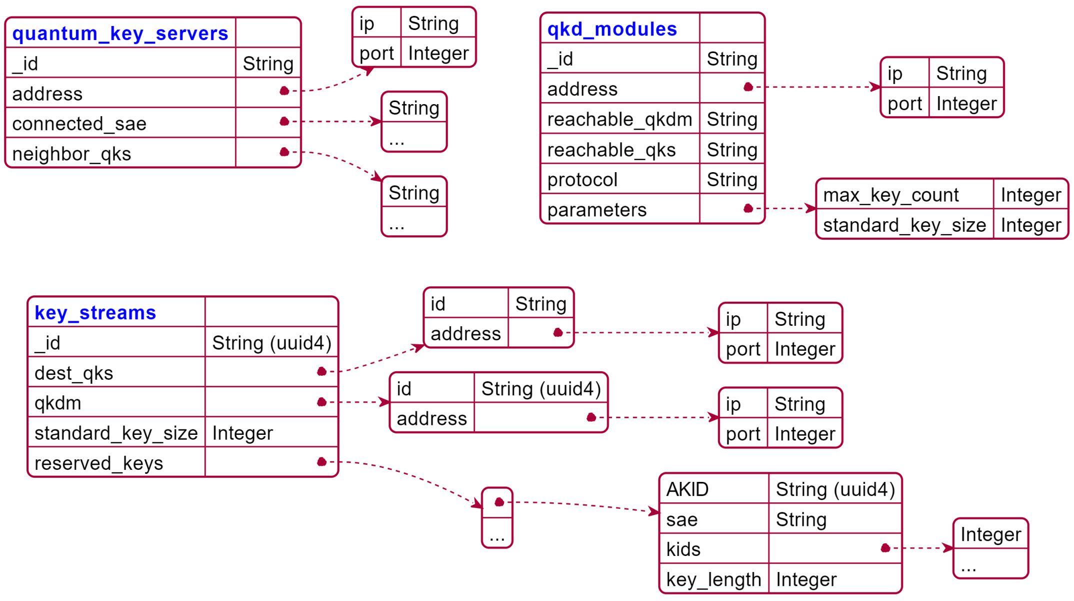


Figure 7.2. Structure of the collections in the QKS database

The QKS database has 3 collections:

quantum_key_servers contains information about the other QKS in the network, how to reach them, which SAEs are connected and which other QKS they can reach. SAEs and links information are not managed by the core component but are used by the routing module as a backup store to recreate the network graph when the system is initialized.

qkd_modules contains information about the QKDMs connected to the QKS: their name, their address, which QKDM they can reach, the QKD protocol used and some QoS parameters. Information in this collection is inserted during the registration phase and are used for the key stream creation process.

key_streams describes active key streams between the QKS and a peer which are connected through a QKDM pair. Taking advantage of the free structure of MongoDB objects, data about the QKDM and the destination QKS are replicated here to avoid expensive join operations between collections: the core component can retrieve the `key_stream` object to obtain all the information needed with a single lightweight query. To synchronize requests between the two QKSs, keys that have been retrieved in one peer are marked as reserved in the other: their identifiers and the SAE they belong to are saved in this collection, allowing the core component to check for their presence before interacting with the QKMD or Vault.

Figure 7.2 shows the document structure of the QKS collections, while tables 7.4, 7.5 and 7.6 describe the fields of the corresponding collection.

Name	Type	Description
`_id`	String	name of the QKS described by the object
`address`	Object	address of the QKS, contains both the IP and the port
`connected_sae`	Array	list of names (strings) of SAEs registered to the QKS
`neighbor_qks`	Array	list of names (strings) of QKSs connected with this node though QKDM pairs

Table 7.4. Description of the `quantum_key_servers` collection in the QKS database

Name	Type	Description
`_id`	String	name of the QKDM described by the object
`address`	Object	address of the QKDM, contains both the IP and the port
`reachable_qks`	String	name of the QKS which the other peer is connected to
`reachable_qkdm`	String	name of the QKDM this one is connected to
`protocol`	String	name of the QKD protocol used by the module
`parameters`	Object	set of parameters for the QKDM, they can be of any type

Table 7.5. Description of the `qkd_modules` collection in the QKS database

7.4.3 Secret engine

Vault is used as a secret engine both for the QKS and the QKDMs: keys retrieved by QKDMs from QKD devices or by QKSs from multi-hop exchanges are securely saved into it in an encrypted form. It supports several types of secret engines that allows storing different types of data, according to the scenario; the QKS and the QKDM use the standard *Key/Value* secret engines which support key-value objects. The interaction with Vault is performed in an asynchronous and non-blocking way through a REST interface exposed by the Vault server. Information are organized in a folder structure and they are accessed through a path specified in the HTTP request. When an object is accessed it is not possible to retrieve a subset of its field, but only the complete retrieval is supported, hence each key is saved as a separate object.

Keys can be accessed through the path

```
/{secret_engine}/{key_stream_ID}/{key_ID}
```

where:

secret_engine is the name of the secret engine used, which corresponds to the name of the user (the QKDM or the QKS)

key_stream_ID is the identifier of the stream that the key belongs to when the requests come from QKDMs while QKSs must use the string "*indirect*" for multi-hop exchanges

key_ID is the identifier of the key

Keys are returned as a key-value pairs where the former is the `key_ID` converted to string and the latter is the base64 encoded key. Keys and values in the *Key/Value* secret engine have to be strings, hence base64 representation is required to store the set of bytes of each secret key.

7.4.4 Redis

Redis is an in-memory fast database used for the communication between the routing and the core module. It allows the former to push the routing tables that can be accessed by the core

Name	**Type**	**Description**
`_id`	String	identifier of the key stream described by the object
`dest_qks`	Object	object containing name and address of the QKS the stream connects to
`qkdm`	Object	object containing name and address of the QKDM which manages the stream
`standard_key_size`	Integer	standard size for the key retrieved by the stream, in bits
`reserved_keys`	Array	array of objects that describes the reserved keys with their identifier, their size and the SAE they are reserved for

Table 7.6. Description of the `key_streams` collection in the QKS database

component when it must check for known SAEs and the path, working as a distributed cache, faster than a database. Redis supports message queues, thus it is used to send notifications about new SAEs and new QKD links attached to the corresponding QKS through its *PubSub* module. Such centralized management of these communications is essential when the core component runs in several replicas, removing the complexity of sharing routing tables among them all. These communications are asynchronous and do not require an immediate interaction from the receiver, therefore pushing routing tables or sending messages neither lead the system to a waiting state nor require any synchronization between the core and the routing module.

Routing tables are saved as *HashSets*: a set of key-value pairs identified by a primary key used to retrieve them. As for MongoDB and Vault, they are managed in Python as dictionaries. One entry in the routing tables is generated by the routing algorithm for each SAE and pushed to Redis, deriving the information from the network graph. The `SAE_ID` is used as the primary key of the objects; the structure of each object is described in table 7.7.

Name	**Type**	**Description**
`SAE_ID`	String	name of the SAE, used as a key to retrieve the table
`next_hop`	String	ID of the QKS which is the next node in the path to reach the SAE
`dest`	String	ID of the QKS in which the SAE is registered
`cost`	Integer	cost of the entire path from the current node to the destination
`length`	Integer	number of nodes in the path to reach the destination

Table 7.7. Description of the routing table entries in Redis

Three different message topics are used for the communication from the core to the routing module:

sae for information related to SAEs registration or removal

qks for the registration of a new QKS in the network

link for the creation or closure of a key stream between a QKDM pair

A dedicated task in the routing algorithm waits for messages over these topics and reacts when one is received. Data stored into Redis are ephemeral: they are not saved on persistent storage, therefore they are deleted when the component is restarted. This behaviour is possible because routing tables are recomputed frequently and the information only depends on messages received through the network. Data related to SAEs and QKDMs attached to the QKS the module is running on are persisted in the database. More details on how messages are formatted to describe events and on the used libraries can be found in appendix B.

7.5 Routing module

The routing module is the component in charge of receiving information from other nodes, notifying them about new SAEs and active QKDMs on its QKS and computing the routing tables for its node. It is an essential component of this QKS version because it guarantees interaction in a truster repeater network and allows QKSs to know how to reach their peers. This section describes the routing algorithm used, the format of the packets exchanged, the function used to compute the costs of each path and the structure of the routing tables.

7.5.1 Routing algorithm

The routing module adopts an approach similar to the one chosen in the DARPA [26] and the SECOQC [68] networks: it uses a protocol similar to OSPF [25], in a simplified form that does not support routing over different Autonomous Systems (AS) nor QoS based on physical properties of the link (e.g. error rate or jitter), based on the Link State algorithm. A distributed solution like this one has the advantages of avoiding a single point of failure and letting each node be free of taking its decisions, but concerning a centralized solution like SDNs (described in section 3.1.4) it is slower in reacting to changes in the topology and has not the possibility to take global decision to ensure a higher QoS. A routing module sends Link State Advertisement (LSA) packets each time there a timer expires or after any change in the connected SAEs or the active QKDMs. Each module has a task waiting for packets on a network socket, which receivers and decode data from. Packets are sent only to neighbour QKSs, which are the QKSs reachable with a direct QKD link over a pair of QKDMs; when a node receives packets it forwards the packets to all its neighbours except the sender one: in this way packets reach each node in a hop-by-hop way. Each packet contains a `timestamp` parameter that contains the exact creation time of that packet, useful to block packets that for any reason reach their destination not in order or too late: if a new packet carrying the same information about link status or the connected SAEs has already been received the old information is discarded and not forwarded. Each link or SAE is marked with its receiving time and if no updates are received in the last three timer periods the information is discarded because too old: this ensures that the network graph at each node converges to the current topology and thus that broken links, due to a crash or some networking issues, are not used until they do no notify their presence again. Five events can trigger the LSA packet to be sent: the registration and the removal of a SAE, the opening of a new key stream and the closure of it and the expiration of a timer. Despite the first four events can in theory be enough to share all the changes in the topology, the timer is required in static situations when there are no updates and a mechanism to refresh the routes and to notice when a QKS (or its routing module) is no more active is required. At the timer expiration, two packets are sent, one for SAEs and one for the QKD links. By default, the timer is set to 10 second, but it can be changed adapting to channels characteristics and the QoS required.

Figure 7.3 is an example of two possible packets: packet `pA` is created after the registration of `SAE A1` and carries information about SAEs connected to `QKS A`, packet `pB` is created due to the opening of a key stream between a pair of QKDMs at `QKS B` and `QKS C` and carries information about the link and their costs. When `SAE A1` is registered the corresponding routing module receives a notification on the Redis topic and triggers the packet creation procedure: the list of all SAEs connected to the source node is encoded in a packet of type `S`, the timestamp is added and the packet `pA` is sent to each QKS which has a direct QKD connection with the source (`QKS B` and `QKS C` in the figure). When the peers receive the packet they decode it, look at the timestamp and check if they have received newer updates or not. If the packet is not discarded it is forwarded to the neighbours except for the one which has sent it (in the example `QKS B` forwards packet `pA` to `QKS C` and `QKS D`), then the other information is processed: if there are new SAEs or known SAEs are not found in the packet the network graph is updated and the routing tables are renewed. QKSs that receives the same packet more than once (like `QKS C` that receives `pA` from `QKS A` and `QKS B`) will discard the following looking at the timestamp. When a key stream is opened or closed between two QKDMs the routing modules which are listening for any event get triggered and a packet of type `K` is generated at both ends and sent to their neighbors. The cost of each QKD link is computed and appended to the corresponding QKS ID in the packet. The packet is

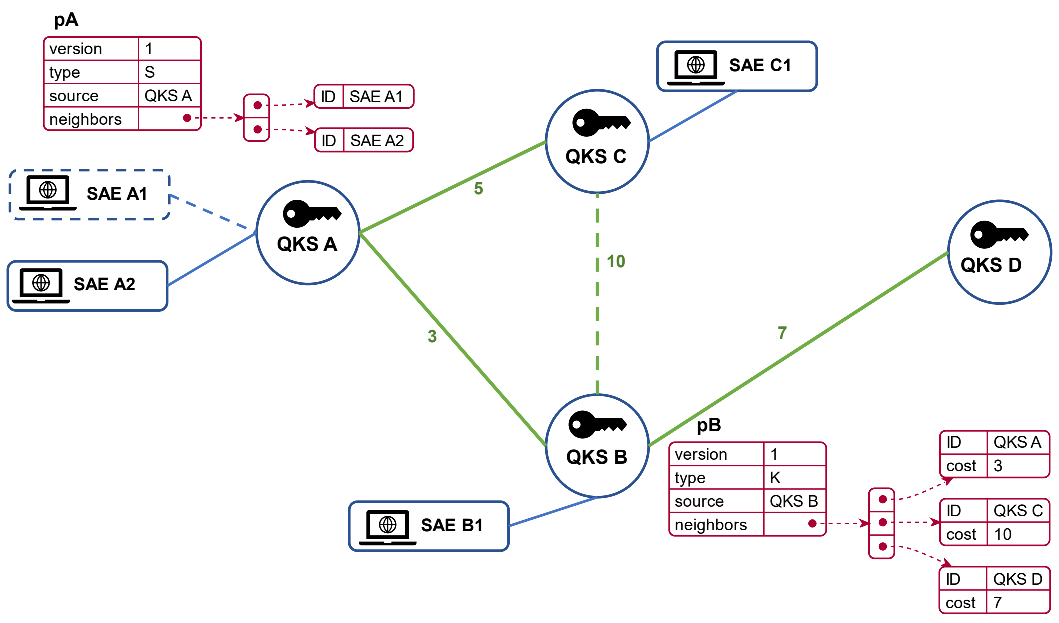


Figure 7.3. Routing algorithm example: packets created after the connection of `SAE A1` (pA) and the QKD link between `QKS B` and `QKS C` (pB).

forwarded and processed following the same logic of the `S` packets, but the update of each link in the graph is performed also for changes in the cost and not only based on the presence or absence of the information. QKSs at both ends send the link information because the graph can be disconnected and thus this is required to ensure all nodes are reached, the timestamp is used to ensure that the information is processed only once even if two packets from two different sources are received (in the example `QKS A` will receive the same information about link B-C both from `QKS B` and `QKS C` and both packets are processed because they come from different sources, but the information on the same link does not trigger the graph update twice). Algorithm 1 describes the logic of the module main functionalities.

7.5.2 Routing packet

Packets can be sent over the network both as serialized JSON objects or as a set of fields encoded into a byte array, encapsulated into TCP packets. Two versions have been developed to adapt to the different systems the QKS is deployed on: the JSON one guarantees large interoperability among programming languages and simple management in the code, allowing developers to extend the routing functionalities more easily, while the raw byte-array can be managed also by simpler devices that work at a lower level. The module uses Python *asyncio* standard library to send and receive them, working in an asynchronous and thus more efficient way than standard blocking sockets. Figure 7.4 shows the packet structure, both in the JSON and in the raw encoded version, and the fields are here described:

version is an integer value that describes the version of the algorithm used, to support future updates in the structure. By default, it is set to 1.

type is a character that indicates which information the packet carries. The two possible values are `S` for SAEs and `K` for QKD links. In this way, all packets exchanged by the routing module follow the same structure, independently from their data.

source is an object that contains the identifier of the source QKS, its address and the port it exposes. This is required because when a node discovers a new QKS it has to save in the database the information on how to reach it, allowing the core component to interact with it. In the raw encoded version, the field is split into three separate components:

```
Def receiver:
    data = socket.read()
    packet = decode(data)
    if packet.timestamp > old_timestamp then
        old_timestamp = packet.timestamp
        forward(packet)
        network_graph.update(packet)
        newtwork_graph.compute_paths()
    end

Def sender(packet):
    packet = new Packet(type, neighbors, link_costs)
    for neighbors do
        socket.send(packet)
    end

Def forward(packet_type):
    for neighbors do
        if neighbor != packet.forwarder then
            socket.send(packet)
        end
    end

Def wait_for_changes:
    while true do
        event = wait_event()
        network_graph.update(event.type, event.subject)
        newtwork_graph.compute_paths()
        sender(event.type)
    end

Def timer(time):
    while true do
        wait(time)
        network_graph.compute_paths()
        sender(S)
        sender(K)
    end
```

Algorithm 1: Routing algorithm pseudocode.

routing contains the address and the exposed port of the routing modules of the source QKS. This is required because the routing module exposes a different port and can be deployed on a different node with respect to the core component. In the raw encoded version, the field is split into two components:

forwarder contains the ID of the last QKS which forwarded the packet, it is used to avoid sending the packet to nodes that already received it.

nAdj is an integer value used only in the raw encoded version, which specifies the number of items in the neighbors list.

neighbors is an array that contains the identifiers of the SAEs connected to the sender QKS in the case of a `S` packet and, in the case of a `K` packet, the other QKSs reachable from the source through QKDM links and their costs. The cost field is not present in the case of an `S` packet in both packet versions.

timestamp contains the timestamp at which the packet has been created, encoded as a string.

authentication contains the information used by the routing module to validate the identity of the sender and the integrity of the packet information. In the JSON version bytes are encoded in a base64 string because raw bytes are not supported in JSON.

Despite the presence of the `authentication` field, this version currently does not implement an authentication mechanism, but it can be extended easily without changes to the packet structure. Authentication is required in a real scenario because routing modules have to be sure about the identity of the packet sender and that data has not been modified, to avoid building a wrong network graph and thus choosing wrong paths. All routing modules in the network must use the same packet version to work properly, the coexistence of both versions is currently not supported. To ensure that all data have been correctly received before decoding a packet, its structure is always preceded by its size in bytes: if the amount of received data does not match the size the packet is immediately discarded. Table 7.8 summarizes the packets fields.

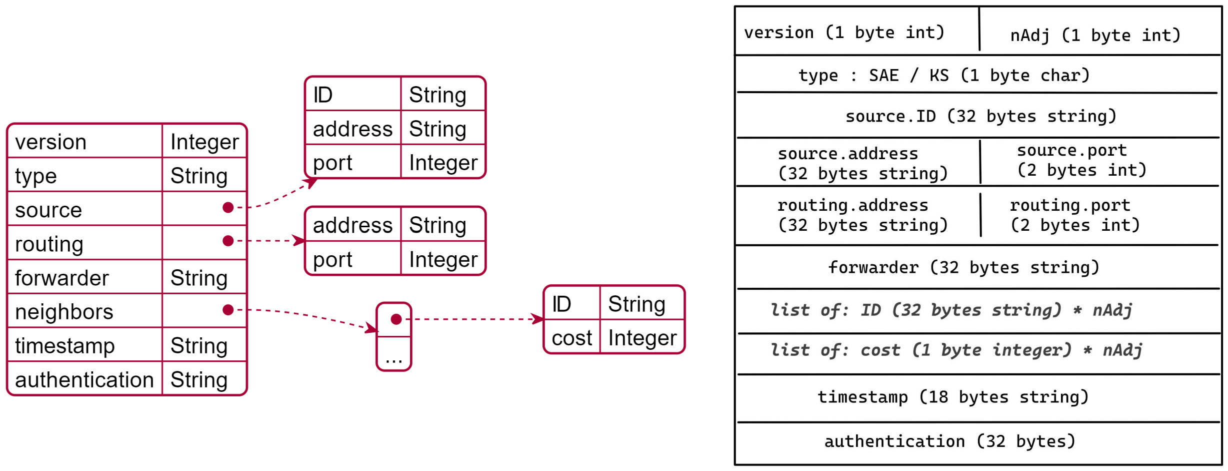

version Integer
type String
source
routing
forwarder String
neighbors
timestamp String
authentication String
ID String
address String
port Integer
address String
port Integer
...
ID String
cost Integer
version (1 byte int)
nAdj (1 byte int)
type : SAE / KS (1 byte char)
source.ID (32 bytes string)
source.address (32 bytes string)
source.port (2 bytes int)
routing.address (32 bytes string)
routing.port (2 bytes int)
forwarder (32 bytes string)
list of: ID (32 bytes string) * nAdj
list of: cost (1 byte integer) * nAdj
timestamp (18 bytes string)
authentication (32 bytes)

Figure 7.4. QKD LSA packet structure in JSON (left) and raw encoded (right) versions

Field name	**Description**
`version`	version of the packet. 1 by default
`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

Table 7.8. QKD LSA packet summary.

7.5.3 Cost function

To compute the best path from a point to another is necessary to define a cost function for each link, which allows choosing the better option between two alternatives in a deterministic way. The cost of the path is a crucial aspect of the routing algorithm and for the QKD network: it drives the path choice and thus all the multi-hop exchanges between distant QKSs. The chosen path to a specific destination changes accordingly to the amount of key material available in each node over the path, allowing QKSs to choose always the path with the highest success rate for a key request. Weighing the cost on the variation of available key material is a strategy that allows adapting in advance to QKD links which are overloaded by too many exchanges. The cost of each QKD link is computed with the following formula:

$$cost_i(t) = a_0 + a_1 \frac{av_k(t)}{av_{tot}} + a_2 \frac{av_k(t) - av_k(t-1)}{av_{tot}} \tag{7.1}$$

where $cost_i(t)$ is the cost for the link i at time t, a_0, a_1 and a_2 are fixed coefficients defined in the configuration, av_{tot} is the maximum number of keys that can be stored over the link i, $av_k(t)$ and $av_k(t-1)$ are respectively the number of available keys on the link at time t and the one obtained in the previous computation. The number of available keys on each link is obtained by the routing module querying the corresponding QKDM. The parameter a_0 is a positive integer number that defines the maximum cost for the link (by default is 100), a_1 and a_2 are negative integer numbers (-50 and -25 by default) that weigh respectively the influence of the currently available keys and the variation of available keys with respect to the preceding computation on the link const.

The routing modules computes the routing tables executing the Dijkstra algorithm on the graph that represents the network topology, obtaining the least costly path to reach each destination. Dijkstra algorithms guarantee to find the cheapest path to reach each node from a specified starting point, if and only if there are no negative cost links; for this reason, the cost of a QKDM link is limited to 0. The cost of the path is computed as the sum of the costs of each link it is composed of:

$$cost_p(t) = \sum_i cost_i(t) \quad \forall i \in path \tag{7.2}$$

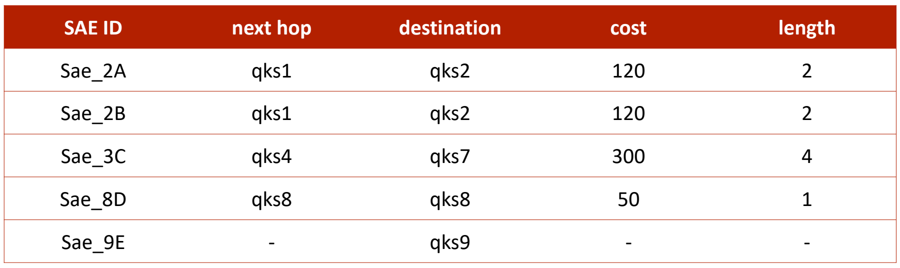
where $cost_p(t)$ is the cost for the path p at time t while $cost_i(t)$ is the cost of the link computed as in equation 7.1. In this implementation, a linear equation has been chosen to compute the cost but other alternatives can be easily implemented, considering not only the available keys but also other parameters like the load on the public channel [28] or response time of the QKDMs which may help in describing the load and thus react in advance. The current QKS core version does not support multi-path routing and keys for a long-distance exchange are forwarded over a single path, therefore the Dijkstra algorithm is configured to search for the cheapest path only but in future versions it can be adapted to compute several paths ordered by cost allowing the core component to use different paths for different exchanges based on some QoS parameter or to split each key reducing the risk of attackers.

7.5.4 Routing tables

Routing tables are objects that describe the next hop in the path to reach each destination. They are derived from the graph after executing the Dijkstra algorithm. To keep them up to date they are recomputed after receiving any packet that causes a change in the network graph, both in terms of nodes or in links costs. The routing modules pushes them to Redis, allowing the core component to fastly access them. Their fields and detailed information about the interaction with Redis can be found in section 7.4.4. Figure 7.5 shows an example of the information stored in the routing tables: an entry is generated for each SAE, even if several SAE are registered on the same QKS and thus they share the same path. This allows the core component to perform fast research of a SAE without checking in the database the QKS where it is registered. If a destination is known but currently unreachable the routing table is marked with an empty next-hop and the core component can notify the calling SAE about this issue. In Redis tables are saved as hash sets, hence there is not a real tabular structure and the figure is used only as an example.

7.6 QKD Module 2.0

The QKDM architecture has not been updated with respect to the previous version. The main changes refer to the asynchronous pattern: the QKDM exploits it to improve its performances,



SAE ID	next hop	destination	cost	length
Sae_2A	qks1	qks2	120	2
Sae_2B	qks1	qks2	120	2
Sae_3C	qks4	qks7	300	4
Sae_8D	qks8	qks8	50	1
Sae_9E	-	qks9	-	-

Figure 7.5. Example of the entries of a routing table

which is even more relevant since the QKDM is not able to scale horizontally in case of high loads. As for the QKS, the interfaces are implemented with the Quart framework and exploit the Hypercorn web server. The simulator code has been adapted to work with the asynchronous code too so the QKDM can manage it without spawning new threads, but relying completely on Python tasks. It can be deployed in a Docker container, that contains both the Key Manager and the QKD simulator. The code is available on GitHub[10].

7.6.1 Interfaces

The Southbound and the Sync interfaces of the QKDM do not provide any new functionality, trying to keep the QKDM as simple as possible and leaving all the complexity of managing the network or the multi-hop exchanges to the QKS. Some methods have been slightly modified to improve performances but the overall structure has not been changed, therefore the methods here described are very similar to the ones proposed previously. Tables 7.9 and 7.10 summarize the API of those two interfaces, while the methods of both of them are explained in detail in the following paragraphs.

Southbound Interface

open_connect is the function that maps the ETSI `OPEN_CONNECT` method. It allows creating a *key stream* between the two peers and by consequence in this implementation between two QKS. If it is called by the first QKS it generates the `Key_stream_ID`, communicates it to the peer with `open_stream` and returns the identifier to the QKS. If it is called by the second QKS it checks the presence of the requested identifier and if successful it signals with `exchange` to the first peer that the key exchange can begin. This version does not support QoS parameters.

get_key is the function that maps the ETSI `GET_KEY` method. It receives a `key_stream_ID` and a list of `index` that are used to find and return the required key, if indexes are not specified it does not return any key. It does not provide any synchronization mechanism: it is up to the QKS to signal to its peers the retrieved keys, the QKDM is not aware of the keys already retrieved in its peer. This implementation does not support any metadata.

close is the function that maps the ETSI `CLOSE` method. It closes the *key stream* whose identifier is received as a parameter, closing the exchange task that interacts with the QKD device (or simulator) and removing all information and keys related to that stream. With this function the stream is closed unilaterally, it does not communicate anything to its peer because this task is managed by the QKS that has to call the `close` on the other end.

get_key_id allows retrieving available key identifiers for a key stream. It is possible to retrieve the whole list, a list of specified length or just the count of the available indexes based on the value of the `count` URL parameter. It is not a function described in the ETSI standard.

[10] `https://github.com/ignaziopedone/qkd-module/tree/async`

check_key_id receives a list of indexes and allows to check if they correspond to available keys for the specified key stream. It is not a function described in the ETSI standard, is used from the QKS for synchronization purposes.

attach_to_server is a function used by an administrator to trigger the registration process to the specified QKS. The QKS returns access data for the database and the secret storage and therefore this function is mandatory to be called for the module to work if it has not been provided with access data through configuration files.

Sync Interface

open_stream is used to notify the other peer about the creation of a new key stream and the chosen `key_stream_ID`. It is called by the first peer during the execution of the `open_connect`.

exchange is used to communicate to the other peer that the key stream creation process has been completed successfully and that the key exchange can begin. It is used by the second peer during the execution of the `open_connect`.

Method name	URL	Access Method
open_connect	/api/v1/qkdm/actions/open_connect	POST
get_key	/api/v1/qkdm/actions/get_key	POST
close	/api/v1/qkdm/actions/close	POST
get_key_id	/api/v1/qkdm/actions/get_ID/{`key_stream_ID`}?count={`count`}	GET
check_key_id	/api/v1/qkdm/actions/check_ID	POST
attach_to_server	/api/v1/qkdm/actions/attach	POST

Table 7.9. QKDM Southbound interface methods summary.

Method name	URL	Access Method
open_stream	/api/v1/qkdm/actions/open_stream	POST
exchange	/api/v1/qkdm/actions/exchange	POST

Table 7.10. Sync interface methods summary.

7.6.2 Database description

As for the QKS, also the QKDM adopt MongoDB as the database in this version. It provides the same advantages as for the QKS and allows to have a single database management system (DBMS) in the entire stack. The QKDM database has only one collection, `key_streams`, which describes the active key streams and keeps track of the available keys stored in Vault. Figure 7.6 shows the document structure of the collection, while table 7.11 describes its fields. Keys IDs are saved in the database because it allows faster access than Vault and supports more efficient queries, allowing the QKDM to perform checks without interacting with the secret engine. Thanks to the document-based structure of MongoDB all the information related to each key stream can be saved in the same collection, avoiding join operations.

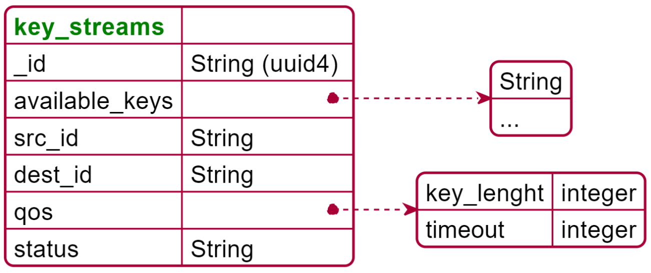


Figure 7.6. Collection structure in the QKDM database

Name	Type	Description
`_id`	String	identifier of the key stream described by the object
`available_keys`	Array	array containing the identifiers (in string form) of the keys available in Vault for this stream
`src_id`	String	name of actor who started the key stream; if the QKDM is connected to a QKS this field correspond to its name
`dest_id`	String	name of destination of the stream, at the other end; if the QKDM is connected to a QKS this field correspond to the QKS the peer module is connected to
`qos`	Object	object containing QoS parameters; they can be of any type
`status`	String	Status of the stream, it can be set to `waiting` or `exchanging`

Table 7.11. Description of the `key_streams` collection in the QKS database

7.7 Workflow

This section wants to describe in detail how the different component interacts with each other in a fully configured network. Figure 7.7 describes the standard workflow for two nodes directly connected with a QKD link, from the initialization phase to the key request, while figure 7.8 describes how a long-distance exchange is performed. In both cases, each element in the system is assumed correctly configured and functioning, all the requests from SAEs and QKDMs to the QKS are authenticated through Keycloak and possible network errors or component failures are not considered here.

PTP exchange

Consider two nodes, Alice's node and the Bob one. In each node, there are deployed a QKS, a QKDM and a SAE, which is already successfully registered to the QKS. In both of them, the management operations are carried on by an administrator, not shown in the figure to not thicken the diagram. It is possible to define three different phases in the workflow: the module registration (a), the key stream creation (b) and the key request (c).

The registration phase is started by an administrator that calls the `attach_to_server` function on the QKDM (1.1 and 1.4), sending in the request body the information on how to reach the QKS. The QKDM perform the registration with the `registerQKDM` on the relative QKS sending its information (1.2 and 1.5); the procedure ends with the response from the QKS (1.3 and 1.6) that contains the access data to Vault and MongoDB. After the registration, the QKDMs are connected but they are inactive: the creation of a key stream is not performed automatically but requires an admin to start it, after ensuring that the modules in both nodes have been correctly registered.

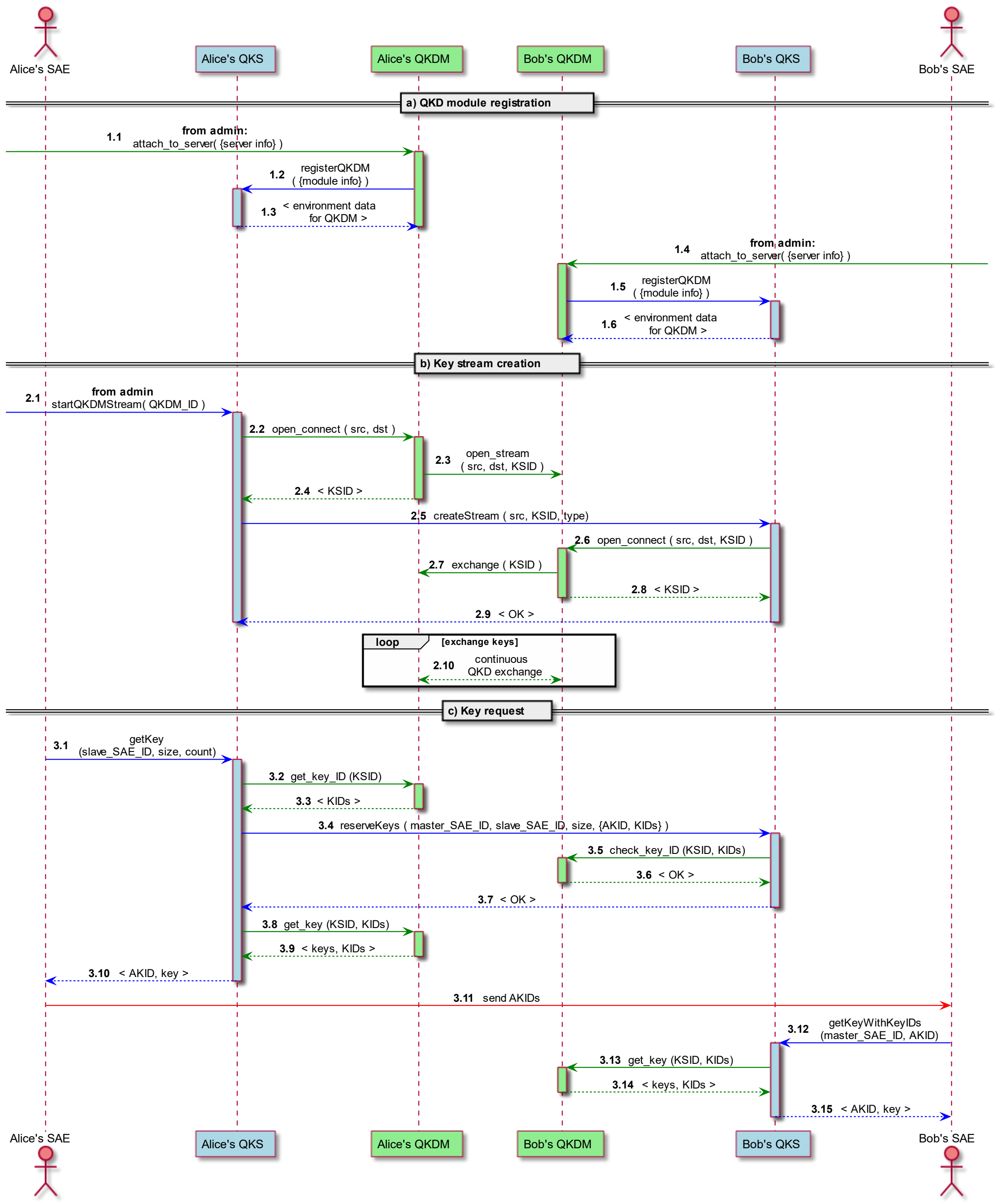

Alice's SAE
Alice's QKS
Alice's QKDM
Bob's QKDM
Bob's QKS
Bob's SAE
a) QKD module registration
1.1 from admin: attach_to_server({server info})
1.2 registerQKDM ({module info})
1.3 < environment data for QKDM >
1.4 from admin: attach_to_server({server info})
1.5 registerQKDM ({module info})
1.6 < environment data for QKDM >
b) Key stream creation
2.1 from admin startQKDMStream(QKDM_ID)
2.2 open_connect (src, dst)
2.3 open_stream (src, dst, KSID)
2.4 < KSID >
2.5 createStream (src, KSID, type)
2.6 open_connect (src, dst, KSID)
2.7 exchange (KSID)
2.8 < KSID >
2.9 < OK >
loop
[exchange keys]
2.10 continuous QKD exchange
c) Key request
3.1 getKey (slave_SAE_ID, size, count)
3.2 get_key_ID (KSID)
3.3 < KIDs >
3.4 reserveKeys (master_SAE_ID, slave_SAE_ID, size, {AKID, KIDs})
3.5 check_key_ID (KSID, KIDs)
3.6 < OK >
3.7 < OK >
3.8 get_key (KSID, KIDs)
3.9 < keys, KIDs >
3.10 < AKID, key >
3.11 send AKIDs
3.12 getKeyWithKeyIDs (master_SAE_ID, AKID)
3.13 get_key (KSID, KIDs)
3.14 < keys, KIDs >
3.15 < AKID, key >
Alice's SAE
Alice's QKS
Alice's QKDM
Bob's QKDM
Bob's QKS
Bob's SAE

Figure 7.7. Application workflow for QKDM registration (a), key stream creation (b) and key request (c).

The key stream creation requires an administrator to start it only on one node: the QKS performs the required synchronization by itself. The process is started on Alice's nodes, when an admin calls the `startQKDMStream` on Alice's QKS, specifying the target QKDM (2.1). Alice's QKS receives the request and calls `open_connect` on the QKDM (2.2), without a key stream identifier (KSID): Alice's QKDM receives the request, generates a KSID and call `create_stream` on Bob's QKDM to synchronize the procedure (2.3). When Alice's QKDM returns the KSID (2.4, the QKS sends it to its peer through `createStream` (2.5), with `direct` as type parameter. Bob's QKS uses the received KSID to call `open_connect`(2.6) on its QKDM, which in this way completes the key stream creation with `exchange` on Alice's QKDM (2.7). At this point the two QKDMs continuously exchange QKD keys (2.10) through their QKD devices (or simulators), until the storage buffer is full, to optimize resource usage.

To require a key shared with Bob's SAE, Alice's SAE calls `getKey` (3.1) to her QKS, with the size, the number of keys and the destination in the request body. Alice's SAE is the one that starts the request, therefore she is the *master SAE*, while Bob plays the *slave SAE* role. Alice's QKS ask its QKDM for the list of available KSIDs for the key stream in use (3.2), selects the required number among the available ones and notifies them to Bob's QKS calling `reserveKeys` (3.4). Because the size of the keys in the QKDM can differ from the one required by the SAE, an Aggregate Key Identifier (AKID) that maps all the needed KIDs for each key is generated by the master QKS: these are the only identifiers returned to SAEs, the QKDM KIDs are managed internally by QKSs. Bob's QKS checks if the received KIDs are available in its QKDM with `check_key_ID` (3.5) and, in case of a positive answer, it marks the AKIDs and their KIDs as reserved in its database. After a correct return status, Alice's QKS can retrieve the KIDs from its QKDM with `get_key`(3.8), aggregate them to build keys of the desired length and return both the AKIDs and the keys to Alice's SAE (3.10) Alice's SAE can now share her AKIDs with Bob's SAE (how this is performed is out of the scope of this application) which can retrieve the same key calling `getKeyWithKeyIDs` to his QKS. Bob's QKS asks its QKDM for the required keys, aggregates them and, in the end, returns them to Bob's SAE.

Long distance exchange

Consider a network with three nodes (Alice, Bob and Carol) and two QKD links, one between Alice and Carol and the other between Carol and Bob. Each node has its QKS and a connected SAE, Bob and Alice have a single QKDM while Carol has two of them, one connected to each peer. The QKDMs, which are not represented in the diagram to keep it more readable, are considered already registered to their respective QKS and a key stream is supposed active on both of the pairs.

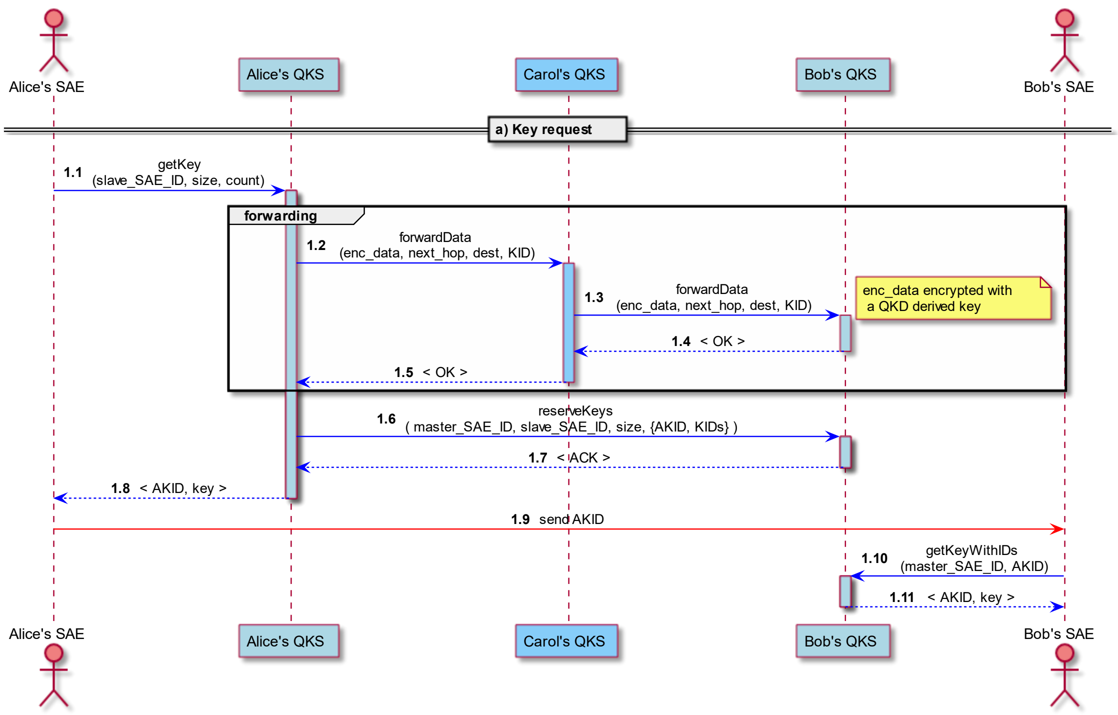

Alice's SAE
Alice's QKS
Carol's QKS
Bob's QKS
Bob's SAE
a) Key request
1.1 getKey (slave_SAE_ID, size, count)
forwarding
1.2 forwardData (enc_data, next_hop, dest, KID)
1.3 forwardData (enc_data, next_hop, dest, KID)
enc_data encrypted with a QKD derived key
1.4 < OK >
1.5 < OK >
1.6 reserveKeys (master_SAE_ID, slave_SAE_ID, size, {AKID, KIDs})
1.7 < ACK >
1.8 < AKID, key >
1.9 send AKID
1.10 getKeyWithIDs (master_SAE_ID, AKID)
1.11 < AKID, key >
Alice's SAE
Alice's QKS
Carol's QKS
Bob's QKS
Bob's SAE

Figure 7.8. Application workflow for an indirect key stream creation (a) and a key request over it (b).

The key request starts in the same way as in the PTP scenario: from the SAE point of view, there is no difference between the two situations. After the request from the master SAE, Alice's QKS looks at the routing tables and understands that the slave SAE is not reachable with a direct link, thus a multi-hop exchange is required. It generates new random keys matching the request size and quantity, saves them into Vault and sends them to the next hop in the path through `forwardData` (1.2). It retrieves a QKD derived key from its QKDM, acting like a SAE in the workflow described in the previous section, uses this key to encrypt the data with a symmetric encryption algorithm (e.g. AES256 [69]) obtaining the `enc_data` and sends them to Carol specifying which is the destination of the forwarding process and the `KID` to decrypt them. Carol receives this information, retrieve the key shared with Alice, decrypts the data and performs the same sequence to forward the data again (1.3). This operation is repeated until the data reach their destination, where Bob's QKS decrypts them and securely stores the received keys into Vault. Note that in this implementation if a hop in the forwarding process fails the error is propagated back to the source QKS, that waits for a completion acknowledgement; but other asynchronous approaches focused on speed are possible. The keys are then reserved calling `reserveKeys` as in the previous section (1.6) and in case of a positive acknowledgement they are retrieved from Vault and delivered to Alice's SAE (1.8) As in the direct stream scenario Alice's SAE has to send the AKIDs to Bob's SAE, which can retrieve them calling `getKeyWithKeyIDs` (1.10) on its QKS.

In this implementation the keys generated for multi-hop streams have the correct key length, hence the aggregation procedure for longer keys is unnecessary. This implementation of the long-distance exchanges suffers from the security issues typical of trusted repeaters QKD networks (section 3.1.6), therefore the exchange can be considered secure if and only if all the nodes in the path can be trusted.

7.8 Future works

Despite this QKS implementation tries to solve some of the defects of the first version as well as provide the support for a complex structure QKD network, it is not ready for a real scenario yet and some issues should be addressed in future versions. Two main sets of issues can be pointed out: security ones related to authentication and encryption of messages and ones related to trusted repeaters.

The security aspects of communication inside and outside the node have not been addressed with version 2.0 and they remain an issue as described in the previous chapter: external communication must be secured and a careful analysis should be performed also for communication among components in the same node. The routing module requires authentication in its communication with peer components, while currently these messages are exchanged in clear and without authentication: this is a major issue because an intruder can easily manipulate the messages and therefore the network topology that nodes see, hijacking the paths and redirecting the traffic to specific nodes. Standard security protocols such as TLS 1.3 are not sufficient in a quantum world hence quantum-safe solutions based on QKD derived keys or post-quantum cryptography techniques must be integrated into the QKS.

The security aspects of a trusted repeater network can not be avoided until a reliable implementation of quantum repeaters is available thus the security of the network is limited by the security of each node. For this reason, improvements related to the key management and the security of the long-distance exchange procedure can be considered. A multi-path approach for the key forwarding can be implemented, splitting the encrypted data over multiple peers: this ensures that if an attacker gains access to data in a single node he can not reconstruct the entire key, because all the pieces are required. Regarding key management, both the QKS and the QKDM should carefully consider how keys are saved in memory because currently no security measures are taken, hence security memory regions (e.g. Intel SGX platform[11]) can be a viable solution to provide secure access to keys.

[11]https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/software-guard-extensions.html

Chapter 8

QKS integration in a Kubernetes cluster

With the growth of cloud computing and the shifting of standard applications to cloud-native ones, datacenters and orchestrators to manage them acquires more and more importance, and with them their security aspects. In a future where quantum computing spreads, quantum-safe cryptography must be available for cloud-native applications, and in scenarios where QKD has been adopted its derived keys must be brought to applications without physical access to a private QKD device that is not compatible with the microservices pattern where applications are split into several independent components and where several applications can be deployed in the same datacenter. For this reason, this work proposes an implementation of the QKS described in the previous chapter inside a Kubernetes cluster, to allow QKD secured communication between applications in two or more different clusters: each cluster can be described as a node of a QKD network while applications running inside it are SAEs. The cluster contains a QKS instance with all its modules and a specialized controller component that manages the interaction between SAEs and the QKS in an automated way that does not require cluster administrator intervention. Each SAE can require keys through Kubernetes resources, without the need to know QKS API's, and can receive keys asynchronously through cluster secrets. Figure 8.1 shows how two QKS stacks can be deployed in Kubernetes with the corresponding operators, allowing SAEs to securely share keys through them.

Section 8.1 describes how the QKS has been deployed inside the cluster and how the components have been configured while section 8.2 describes how the key request process has been handled with an operator.

8.1 Cluster configuration

The QKS has been deployed in a Kubernetes cluster exploiting its capabilities of providing redundancy and scalability through its built-in resources. Figure 8.2 describe the cluster configuration with a QKS deployed in it. The cluster is divided into different namespaces: the QKS run in its namespace, while each SAEs and each QKDM have their own. This not only simplifies the resources management for the administrators providing logical separation but also ensures resource disjunction, avoiding components to access data, configuration files and secrets not belonging to them. This is crucial in the QKD scenario where secret keys are constantly exchanged and stored in memory and their security is a fundamental requirement. Each QKS architectural component described in the previous chapter is here composed of different resources: a service to allow it to be reachable and a set of pods running the code. The service is the resources that provide access to the components from the cluster or from the outside: components that should be reached only from the inside of the cluster (MongoDB, Vault, Keycloak and Redis) has a service of type `ClusterIP` while the QKS core and the routing module has a service of type `LoadBalancer` allowing them being reached from the outside to communicate with their peers in the QKD network. The QKS

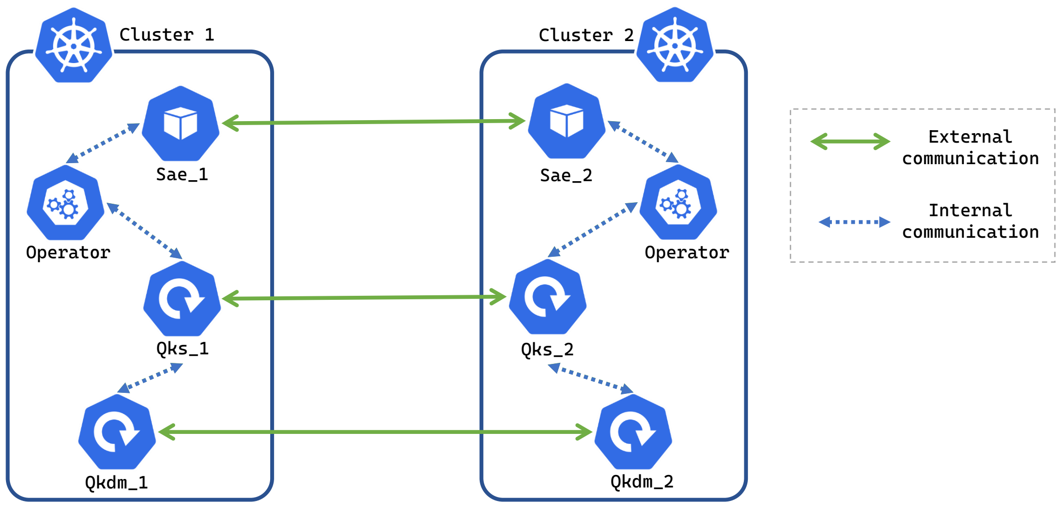


Figure 8.1. Configuration and communications of two QKS stacks deployed in different Kubernetes clusters

core, the routing module, and Redis are stateless applications and therefore they are deployed as `Deployment` resources: in this way, their pods are automatically restarted in case of failures and there is no need to manage pods internal data because the information is saved in the database. Stateful components like MongoDB, Vault and Keycloak are deployed with `StatefulSet` resources and they have a `PersistentVolumeClaims` that allows mapping a `PersistentVolume` resources to provide a not-ephemeral place where data can be stored: in this way even after a crash the new pods don't lose saved data. Both `Deployments` and `StatefulSets` allows the administrator to define the number of replicas wanted, that should be defined based on the components loads. The number of replicas can be dynamic based on some metrics if an `HorizontalPodAutoscaler` resources are configured to manage the corresponding component. Configuration data are injected into pods through `ConfigMaps` and `Secrets` resources that should be properly configured by the cluster administrators. They can be consumed by the pod both as environment variables both as configuration files, based on the application requirement. The QKDMs are deployed in separate namespaces. The QKDM is a stateless application and therefore it is described with a `Deployment` resources but the number of replicas is set to one because it has not been developed to be scaled up. Kubernetes ensures that a pod is always running without spawning others until the first one can not perform its job anymore. It has a corresponding `Service` resource of type `LoadBalancer` to let it be able to communicate with its peer module to perform QKD exchanges that exposes two ports: one for the QKD manager and one for the QKD device. Its configuration data are injected through a `ConfigMap` that belongs to its namespace and is not accessible by other components. It reaches the database and the secret storage through their services, which by default are not bound to a specific namespace and allows their corresponding pods to be reached from all over the cluster. SAEs are here described as simple pods for simplicity in their private namespace but they can be any type of application with any type of resources and components. The process in which they can register themselves to the QKS and can require QKD keys are described in the next section.

8.2 Custom resources and Operator

To support a complete integration of the QKS inside Kubernetes, a set of Custom Resources has been defined, and the Operator required to manage them has been developed. They free the administrator from directly interacting with the QKS core and Keycloak, allowing it to deploy a QKS instance inside a cluster without knowing the APIs details of the server. Two custom resources has been defined: the `Sae` and the `KeyRequest`. The former is required to describe a

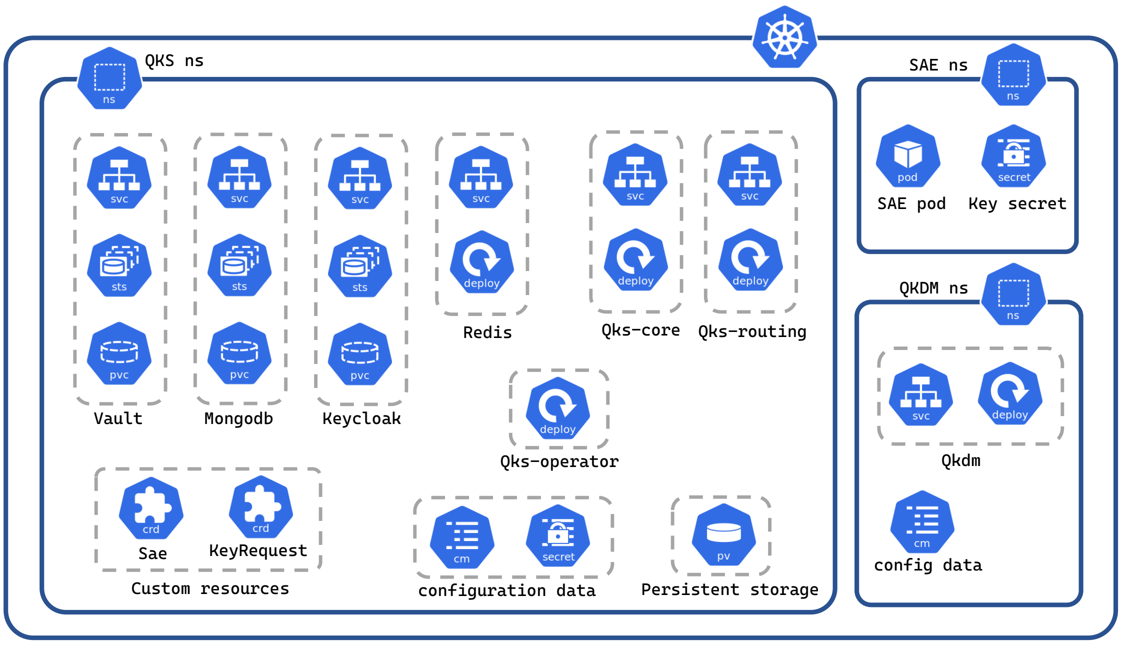


Figure 8.2. QKS deployment in a Kubernetes cluster

new Sae that should be registered to the QKS in the cluster. Each SAE is described by their `name`, and an additional property called `registration_auto` which specify if the registration process to Keycloak should be carried on by the operator or manually by the cluster administrator.

```yaml
apiVersion: "qks.controller/v1"
kind: Sae
metadata:
  name: saeA01
spec:
  id: sae_A01
  registration_auto: true
```

Figure 8.3. Example of a Sae resource

When a Sae resource is created in the cluster the operator is triggered: it retrieves the object (figure 8.4.1), it perform the registration of the Sae as a user in Keycloak setting the `name` property as username and with a random password (8.4.2). The Sae is then registered to the QKS with the `registerSAE` method of the QKS northbound interface (8.4.3). If the process is completed successfully, Keycloak credentials are stored in a secret in the SAE's namespace (8.4.4). When a Sae resource is deleted the operator reacts unregistering it from the QKS with the `unregisterSAE` method. The resource can be extended in the future with additional properties that can trigger other behaviours in the operator, such as the automated deployment of an application.

The KeyRequest resource should be used by SAEs to require QKD derived keys. Each KeyRequest must have a `master_SAE_ID` and a `slave_SAE_ID`, which defines the two peers for the request. The resource must have `size` and `number` properties if it comes from the master SAE or it must have a list of key identifiers in the `ids` parameter if it is a request coming from the slave SAE.

When a KeyRequest object is created the operator is triggered: it checks the properties and based on which are present and which are not understood if it is a request coming from the master SAE or the slave SAE. If retrieves the SAE credentials from the relative secret in the SAE namespace, perform the login to Keycloak and then require the keys from the QKS: if the resource

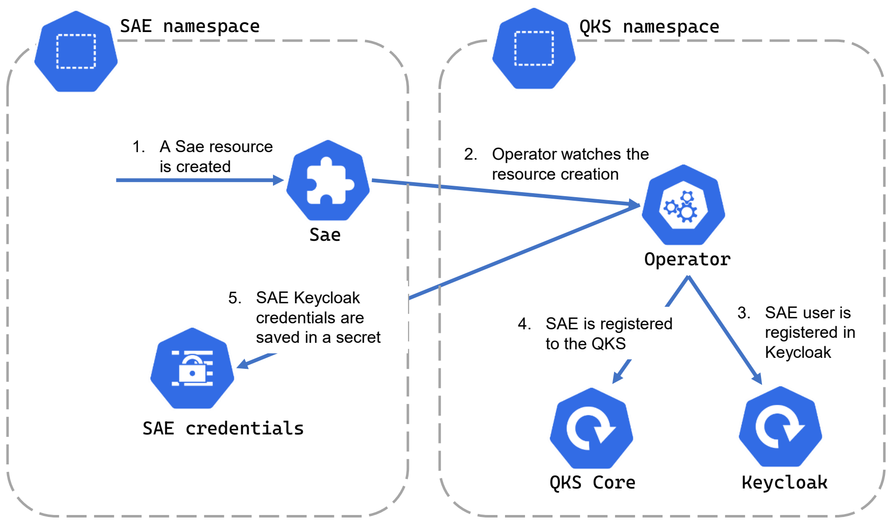

SAE namespace
QKS namespace
1. A Sae resource is created
Sae
2. Operator watches the resource creation
Operator
3. SAE user is registered in Keycloak
4. SAE is registered to the QKS
5. SAE Keycloak credentials are saved in a secret
SAE credentials
QKS Core
Keycloak

Figure 8.4. Sae resource workflow managed by the operator

```yaml
apiVersion: "qks.controller/v1"
kind: KeyRequest
metadata:
  name: requestA
spec:
  number: 1
  size: 128
  master_SAE_ID: sae_A01
  slave_SAE_ID: sae_B02
```

```yaml
apiVersion: "qks.controller/v1"
kind: KeyRequest
metadata:
  name: requestB
spec:
  ids:
    - 30b1ed31-561f-41dc
  master_SAE_ID: sae_A01
  slave_SAE_ID: sae_B02
```

Figure 8.5. Example of two KeyRequest resources

contains a list of identifiers the operator calls the `getKeyWithKeyIDs`, otherwise it calls the `getKey` specifying the requested key number and size. The returned object is saved in a secret in the SAE namespace, using the request name as the name for the secret. SAEs can retrieve their keys asynchronously, watching for the corresponding secret, once the operator has completed its task: differently from the usage of QKS APIs through direct HTTP calls, the creation of Kubernetes resources does not return any object to the application. This asynchronous behaviour allows preserving keys in a stateless scenario where pods can be restarted or substituted: even if the SAE pod crashes after performing the request, the returned keys can be securely retrieved by the new pod accessing the secret. The complete workflow of the KeyRequest resource creation is described in figure 8.6

Even if it is possible to automate with the same operator pattern also the creation of QKDMs, defining a custom resource for this task and extending the operator to watch also it, it should be considered that QKDMs in a production scenario are bound to a physical device and they must work in pairs: their initialization will fail until their peer is not up and running too, therefore it is complex to synchronize the behaviours of two separate cluster that has to manage the two peers. For this reason in this implementation QKDMs must be initialized manually by the cluster administrator and the registration procedure to the QKS is not carried on automatically.

8.2.1 Implementation details

Operators are brought in Kubernetes as standard stateless application: because they only react to operation on resources they do not have to keep a state in case of pod crash, therefore dey are instantiated with deployment resources. To ensure consistency the replica number should be set to one, avoiding duplicate actions triggered by the same resource. The operator runs outside

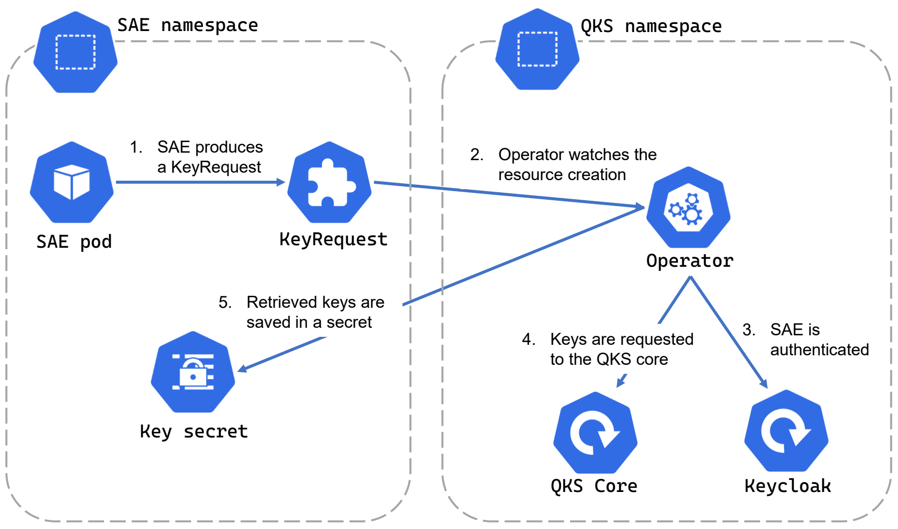

SAE namespace
QKS namespace
1. SAE produces a KeyRequest
SAE pod
KeyRequest
2. Operator watches the resource creation
Operator
5. Retrieved keys are saved in a secret
Key secret
4. Keys are requested to the QKS core
3. SAE is authenticated
QKS Core
Keycloak

Figure 8.6. KeyRequest resource workflow managed by the operator

of the control plane, but it must have access to credential secrets of all SAEs and should be able to create secrets in all SAEs namespaces, therefore it must have the proper rights and roles to perform all the required actions in the cluster. Its deployment can run either in the QKS namespace or in a separate one.

Despite Kubernetes having been developed with the Go programming language, the operator code can be developed in any programming language that supports Kubernetes API. This operator has been developed in Python 3 to use only one language for all the components in this work. The interaction with the Kubernetes API is performed through the official client for Python[1] and to reduce the complexity of the operator code the Kopf [2] framework has been used: it allows to reduce the required code only to functions that actively reacts to actions on resources and provide the communication with the cluster.

The operator has a corresponding Docker image that is used to deploy it in the cluster. It is currently set up in the cluster as a Deployment object, but it can be packaged to be deployed using the Operator Lifecycle Manager[3], a Kubernetes component that can automate the set up of controller and custom resources. For security reasons, the operator deployment should run in the same node as the control plane and the control loop, hence in the master node of the cluster. More details on the operator development can be found in appendix B

8.2.2 Key exchange workflow

This section wants to describe in detail how key exchanges are performed between SAEs deployed in different clusters and how the internal components interact among them. The QKS in each cluster is considered correctly configured, both in its internal components and regarding the Kubernetes resources.

Consider two SAEs, *SAE_A01* deployed into cluster *A* and *SAE_B01* into cluster *B*. Both of them have been correctly registered both in Keycloak and to their QKS, with the creation of the corresponding `Sae` resource as described in figure 8.3. When SAE_A01 wants to exchange a key with its peer it creates a `KeyRequest` resources with its name as the `master_SAE_ID` (figure 8.7.A1), its peer name as `slave_SAE_ID` and specifying the number and the size of needed

[1]https://github.com/kubernetes-client/python

[2]https://github.com/nolar/kopf

[3]https://github.com/operator-framework/operator-lifecycle-manager

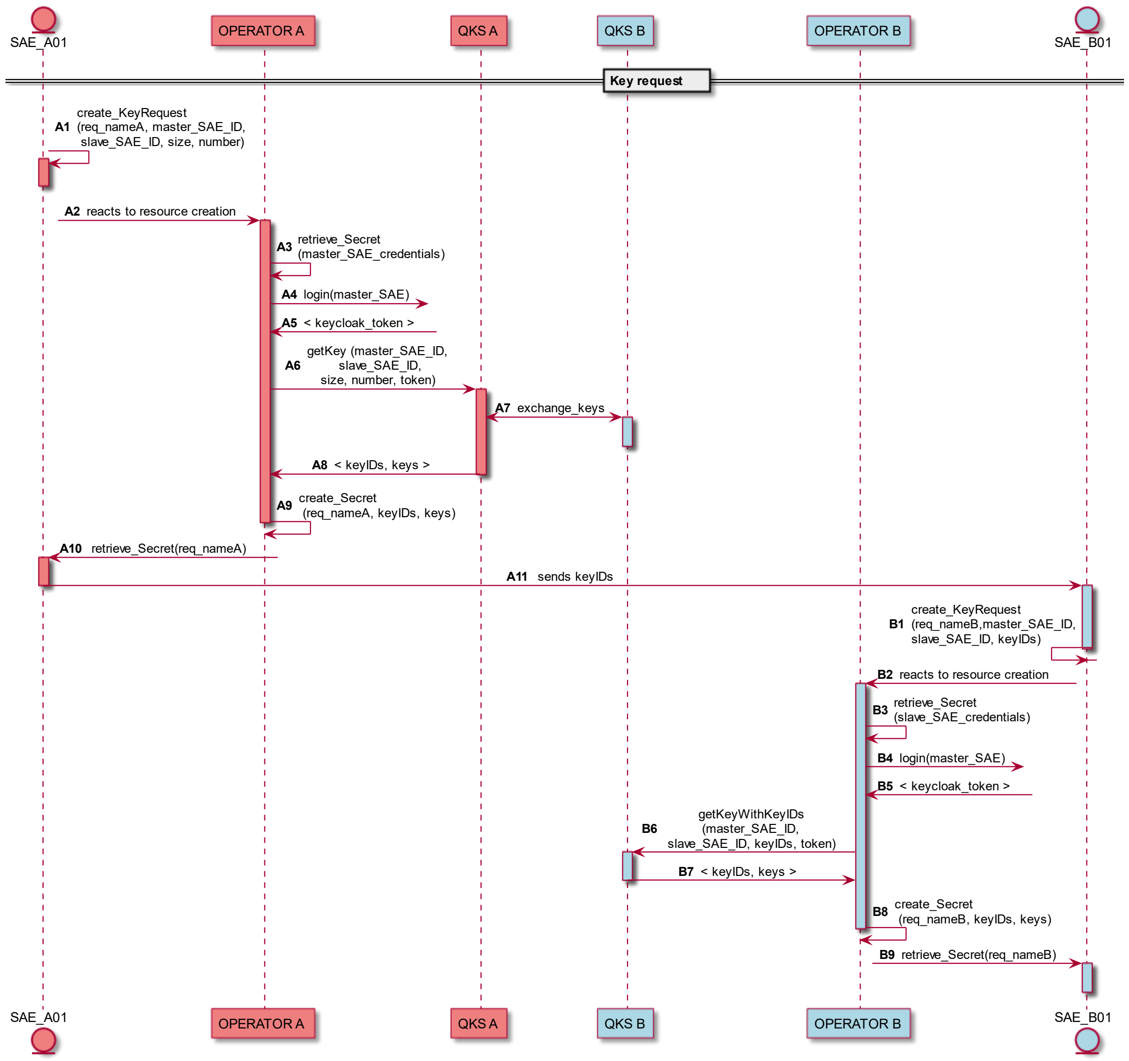

SAE_A01
OPERATOR A
QKS A
QKS B
OPERATOR B
SAE_B01
Key request
A1 create_KeyRequest (req_nameA, master_SAE_ID, slave_SAE_ID, size, number)
A2 reacts to resource creation
A3 retrieve_Secret (master_SAE_credentials)
A4 login(master_SAE)
A5 < keycloak_token >
A6 getKey (master_SAE_ID, slave_SAE_ID, size, number, token)
A7 exchange_keys
A8 < keyIDs, keys >
A9 create_Secret (req_nameA, keyIDs, keys)
A10 retrieve_Secret(req_nameA)
A11 sends keyIDs
B1 create_KeyRequest (req_nameB,master_SAE_ID, slave_SAE_ID, keyIDs)
B2 reacts to resource creation
B3 retrieve_Secret (slave_SAE_credentials)
B4 login(master_SAE)
B5 < keycloak_token >
B6 getKeyWithKeyIDs (master_SAE_ID, slave_SAE_ID, keyIDs, token)
B7 < keyIDs, keys >
B8 create_Secret (req_nameB, keyIDs, keys)
B9 retrieve_Secret(req_nameB)

Figure 8.7. Sae resource workflow managed by the operator

keys in the other parameter as described in figure 8.5 on the left. The operator is triggered and it looks at the new resources (A2), retrieves SAE_A01 Keycloak credentials from the corresponding secret, performs the login (A4) and uses the received token to authenticate to the QKS. The key request is carried to the QKS, calling the `getKey` method with the resource parameters (A6). The QKS performs the key exchange with the QKS where the slave SAE is registered (A7) as explained in section 7.7 and there is no difference from the SAE point of view if the keys are retrieved from a direct exchange or with a long-distance exchange that exploits multiple nodes in the QKD network. When the QKS returns to the operator the keys and their IDs it generates a new secret in the master SAE namespace and stores the data (A9), using as resource name the request one. If there are not enough available keys and the `getKey` the operator does not create the secret. SAE_A01 retrieves the secret asynchronously and must send to its peer the received key identifiers, in a way that is out of the scope of this work. SAE_B01, to retrieve the same keys must create a KeyRequest resource (B1) specifying the received IDs in the corresponding field (figure 8.5 on the right). The operator in cluster B performs the same actions as the other one but interacts with the QKS using the `getKeyWithKeyIDs` method (B6). After retrieving the secret with the shared keys (B9) the two SAEs can communicate securely. Note that it is not mandatory that the name of the two KeyRequest resources in the two clusters match.

8.3 Future works and alternative approaches

To bring the QKS to a production scenario it should be considered that the deployment should be plug and play and that the administrator could not have any knowledge on the QKS architecture and how it works. For this reason, an operator able to deploy the complete QKS architecture after the creation of a single resource can be implemented: defining a new resource type able to describe the server and its properties the operator could deploy the different components, managing all the initialization processes following the resource specifications. A similar approach for the deployment of the QKDM can be adopted too, to solve the problem of synchronization between module pairs without direct interaction with its APIs.

Currently, ConfigMaps are used to provide to the components their configuration files, but they can be substituted with environment variables injected via Secrets and ConfigMaps keys, not only to simplify the initialization procedure avoiding the need for volumes in containers but especially to avoid the presence of sensitive information written in clear in ConfigMaps. It is possible to shift the operator code to adopt the asynchronous pattern, in order to reach higher performances that will avoid a possible bottle-neck in the key request procedure: the used *kopf* library supports it, but alternative libraries must be used in place of the Kubernetes official client that does not provide this type of functionalities.

This work relies on the operator to interact with the QKS and on Kubernetes secrets for the key storage after they are retrieved, providing a middleware that interacts only with build-in resources and that can be adapted to match all the use cases that the QKS supports; but it is also possible to adopt different approaches such as using a third-party Key Management Systems (KMS) to directly store and manage keys securely and efficiently. It is possible to develop a plugin for a KMS that allows it to directly interact with the QKS through its APIs, which can in this way avoid the need for the operator.

The solution proposed in this work focuses on securing the communication between applications deployed into different clusters but does not address the problem of communication among nodes that are part of the same cluster. Large clusters can span over multiple nodes physical nodes that can be placed in different locations and that communicate over the network: QKD can be a solution to secure their communication in a scenario where quantum computing is a threat to standard encryption protocols. To solve this problem a QKS can be deployed in each node, bounding its components to the node, ensuring that each SAE can reach its QKS without traversing the public network, and securing the communication with other peers with QKD keys. The storage of keys should be addressed carefully: in the solution here proposed retrieved keys are stored in secrets, which are managed in a centralized way by the control plane, but this is not possible in a scenario where the standard communication between nodes is considered not secure enough, therefore other approaches should be investigated.

Chapter 9

Test and validation

Tests have been performed to verify the stack functionalities and analyze the performances and limitations of the proposed solution. Two different testbeds have been used, one to test the performances of the entire stack and the other for the routing algorithm. The former is composed of three machines: an HP EliteBook 8570p laptop and two Intel NUCs, each one with a Kubernetes cluster deployed on. The laptop has the following characteristics:

- CPU: Intel(R) Core(TM) i7-3520M 2.90GHz
- RAM: 8 GB
- OS: Ubuntu 20.04.2 LTS (Focal Fossa)
- Kubernetes cluster: K3s [1] version 1.21.1

The two NUCs features the same specifications:

- CPU: Intel(R) Core(TM) i5-5300U 2.30GHz
- RAM: 16 GB
- OS: Ubuntu 20.04.3 LTS (Focal Fossa)
- Kubernetes cluster: K3s version 1.21.4

The testbed used for the routing algorithm is a virtual machine with:

- CPU: 16 cores
- RAM: 40 GB
- OS: Ubuntu 18.04.5 LTS (Bionic Beaver)
- Docker version: 20.10.5

[1] https://k3s.io/

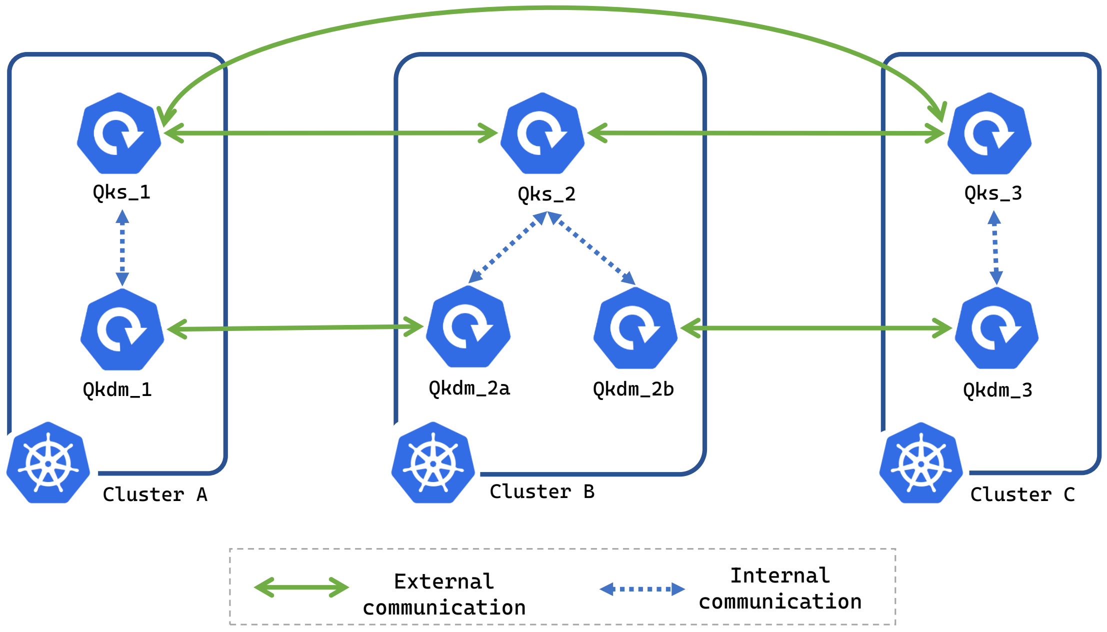


Figure 9.1. Configuration of the 3 nodes network testbed

9.1 Key-exchange tests

Several tests have been performed to analyze the performances of the QKS and the Kubernetes operator used to interact with it in the cluster. To test the key exchange and retrieval procedures three QKS stacks have been deployed in Kubernetes, one in each of the 3 nodes, as shown in figure 9.1 The three QKS have been connected with 2 pairs of QKDMs, which exchanged keys through a fake QKD exchange simulated through the *fakeKE* Python module. This algorithm exchanges random bits over two network sockets in blocks of fixed size without performing any other action on them and, despite it does not simulate any real QKD protocol, allows to reach speeds that are faster than what accurate simulations can guarantee. For all the tests performed the standard key size exchanged by the simulator was fixed to 128 bits.

The QKDM performances have been analyzed to discover its limits in the key exchange process. The *fakeKE* simulator has been set with no software limitation in the key exchange rate, hence the only limit in the key exchange was to the network connection between nodes. With this exchange in progress, the time required by the QKDM to exchange a key and save it into Vault has been computed and the key rate has been derived from it. The QKDM requires about 5.6 ms to perform the operation, which leads to a key rate limit of 178 keys per second, corresponding to a 22.8 kbit/s throughput with keys of 128 bits, with higher device speed the QKDM is not able to process received data fast enough and some information would be discarded. The measured throughput is enough to support the current QKD key rate of real commercial devices, which is set in the order of a few kbit/s but can become a limitation in the future with faster devices. The QKDM retrieves each key from the device and saves it into Vault, thus the latter is the bottleneck of the system: despite the QKDM task can reach a higher speed, the interaction with the secret engine limits it. To solve this issue it is possible to increase the number of Vault instances in the stack spreading the load among them all and to develop a better parallelization of requests. Because scaling Vault can be resource consuming and not always feasible, to support even higher key rates it is possible to increment the size of each exchanged key and therefore reduce the number of interactions with the secret engine. This operation should be addressed carefully because it can lead to higher key consumption from high-level applications if the requested key size is smaller than the new exchanged size: because each key returned by the QKS must be derived from at least one Vault key, even if it is smaller than the standard size, parts of the exchanged material can be unused and discarded.

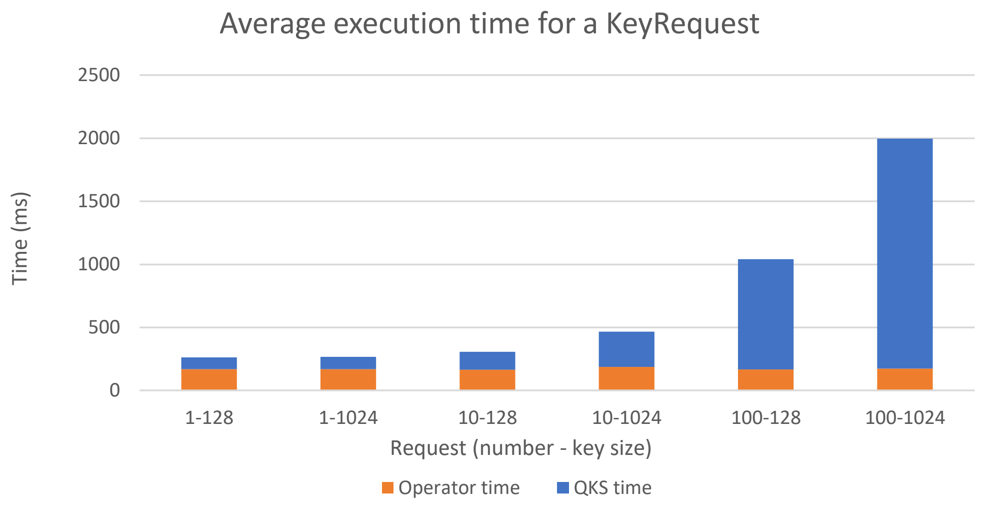


Figure 9.2. Average execution time for a KeyRequest resource with different parameter

To test the QKS and the operator different KeyRequests objects have been created on *cluster A* which hosted the master SAE varying both the key size and the number of requested keys, and the time required by the operator and the QKS to solve them have been retrieved. The key exchange rate of the simulated QKD devices has been set to the maximum one hence for each request enough keys were available in Vault and no failures have been encountered. The slave SAE was hosted on *cluster B* directly connected with a QKDM pair to the notebook node, as shown in figure 9.1. Figure 9.2 shows the average results obtained with different request types. The total execution time for each request has been measured in the operator from the triggering to the conclusion of its routine and for the *QKS time* from the getKey call to the response; the *Operator time* has been computed as their difference. The time required by the QKS to solve them varies depending both on the size and the number, with the same proportion: the execution time depends on the number of fixed length keys required to the QKDM, therefore doubling the key size or the key number does not change the QKS load. The QKS execution time grows with the growth of keys requested to the QKDM but is below linear thanks to the parallelization of calls both from the QKS to the QKDM and from the QKDM to Vault. For a small number of requests the module can handle them all in parallel without any significant slowdown (as can be seen for requests *1-128* and *1-1024* in the graph), but when the number grows over a certain threshold some requests are handled sequentially and performances are limited (as can be seen in the difference between requests *10-128* and *10-1024* in the graph) To reach the best performances retrieving a large set of keys it is better to produce a single request with a bigger number than generating several smaller requests that must be handled by the QKS as separate objects hence requiring more synchronization calls between nodes. Table 9.1 reports the time required to serve different requests and the corresponding retrieval rate per second. The operator overhead is constant in the cluster and is not influenced by the request parameters, it is considerable when requests are small both in size and in the number of keys but become less and less impacting when requests grow. In comparison with the first version of the QKS, which reached an average key retrieval rate of 1 key over 2 seconds, this version is faster than about an order of magnitude.

To test the QKS behaviour in a situation where the key exchange rate of the device is limited the *fakeKE* has been set to 2 kbit/s and 4 kbit/s rate, leading to a QKDM key rate slightly smaller due to the Vault overhead. Using the same setup of the previous paragraph different amounts of requests of 1 key of 1024 bit length per second have been performed and the success rate has been analyzed considering the number of Kubernetes secrets created by the operator. Keys saved in Vault have been deleted before each test to ensure that the results could not be influenced by data already stored. Figure 9.3 shows that the `getKey` success rate decrease with the number of requests per second received from the QKS and falls below 100% when the rate is higher than the device one and decrease proportionally to it. As expected the rate decreases faster when the

Key number - size	Requested size	QKS execution time (ms)	Retrieval bit rate (kb/s)
1 - 128	128 bit	93	1.4
1 - 1024	1024 bit	97	10.6
10 - 128	1280 bit	140	90.9
10 - 1024	10.2 kbit	280	365.0
100 - 128	12.8 kbit	873	1464.9
100 - 1024	102.4 kbit	1824	5613.1

Table 9.1. Key retrieval bit rate with different requests parameters

device rate is smaller and can guarantee a slower key saving rate to Vault. Once the limit is reached both the QKS and the QKDM can gracefully handle requests returning error messages without failures.

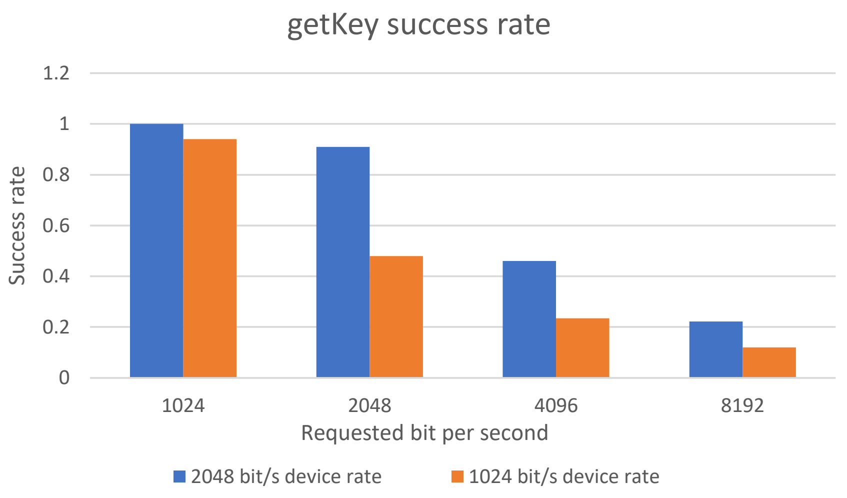


Figure 9.3. getKey success rate with 2 and 4 kbit/s device exchange rate

Tests have been performed to analyze the behaviour of the system in a multi-hop scenario, where SAEs in two different nodes want to share keys without being directly connected. *Cluster C* has been connected to *cluster B* with a QKD pair, in a 3 nodes network where the latter is connected to both the other peers as shown in figure 9.1. The master SAE has been registered to *cluster A* while the slave SAE has been registered in *cluster C*, in this way requests between these two SAEs are forwarded through *cluster B*. Exchanges have been performed asking for a key of 128 bits, which is the standard size provided by the simulated device. Figure 9.4 shows the average time required by the QKS to retrieve the keys both in the master and in the slave node, compared with the results obtained for point-to-point (PTP) exchanges. The time required to complete a `getKey` over 2 hops is higher than the PTP connection because all the operations for the key exchange must be repeated for every hop. Despite it was not possible to test the multi-hop behaviour over a higher number of hops, a linear increment is predictable because each hop executes the same routine until the end peer is reached. The obtained results demonstrated that the `getKeyWithKeyIDs` is not affected by the number of hops the exchanges spans over, and its execution time only depends on the time needed for the QKS and the QKDM to retrieve keys saved in Vault. For this reason, when requiring a lot of already received keys through `getKeyWithKeyIDs` the retrieve rate is limited by Vault performances and it is not impacted by the key rate of the QKD devices, thus increasing the number of Vault replicas can help in mitigating the issue. The figure also shows that the time required by the operator to handle each request does not depend nor on the number of hops neither on the request type (`getKey` or `getKeyWithKeyIDs`) because it is completely unaware about how the QKS manages the requests it receives.

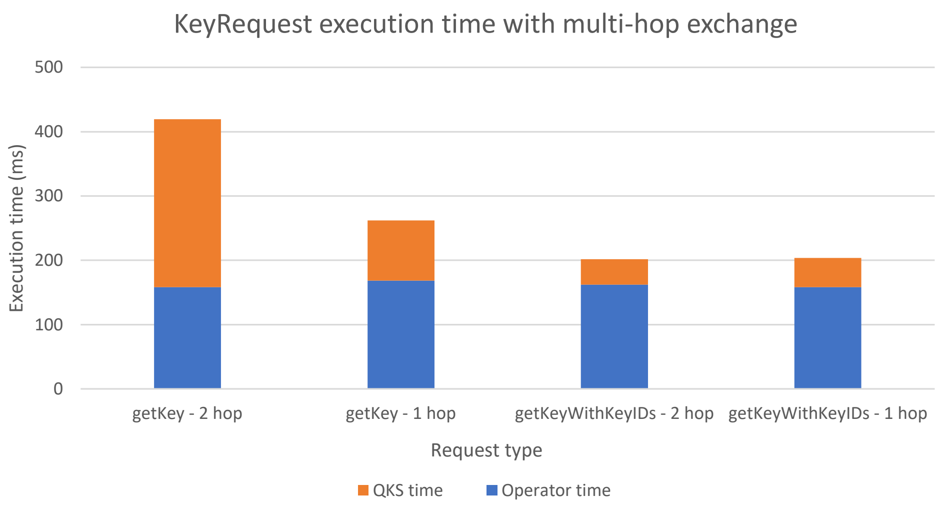


Figure 9.4. Average execution time for multi-hop requests compared with PTP ones

Tests performed on the key retrieval process showed that the QKS adds an overhead compatible with the current communication protocol when the amount of requested keys is limited and that it can support current QKD device rates in the PTP scenario without issues, but it does not scale unbounded due to Vault limitations and can expose some issues with faster devices. In multi-hop scenarios the execution time increases with the number of hops hence, despite long-distance exchanges being possible and working correctly, they should be used carefully because they can not guarantee the same speed as PTP ones.

9.2 Routing tests

To test the performances of the routing algorithm a different testbed have been deployed: it was not possible to use a number of physical nodes relevant for the test, hence a test version of the routing algorithm able to work independently, without the corresponding QKS core and attached components, has been developed and packaged in a Docker image. This algorithm version produces random events in the network, such as registration or removal of SAEs and QKDMs that triggers packets creation, to simulate the behaviour of real nodes. A set of these containers have been deployed on the virtual machine with some faked connection among them and the system has let been running producing random events and therefore sending packets and updating the network topology. Results proved that in a connected graph each node sees the same network topology and store the same network graph and that in a situation where nodes are not reachable any more links and SAEs are correctly removed from the graph after the specified threshold. The time required for a node to receive a packet and the number of travelled hops has been logged to produce data about the time required to analyze it with respect to the distance from the sender to the receiver. As shown in figure 9.5 the convergence time of the algorithm increases linearly with the number of nodes in the network, due to the transmission time hop-by-hop which has a prevalent impact also in this simulated scenario where containers are connected through a virtual bridge and thus packets do not travel over the network but are delivered at the software level. In a real scenario where each QKS is deployed on a different node the transmission time over the network will be higher and hence even more impacting concerning the algorithm execution time. In the worst-case scenario, where two nodes are at the two opposite edges of the network, the convergence time is nearly double the average one. With a network of 50 nodes, the average time to update the graph executing the Dijkstra algorithm and to compute the routing tables is 13 ms, hence smaller than the convergence one of about one order of magnitude.

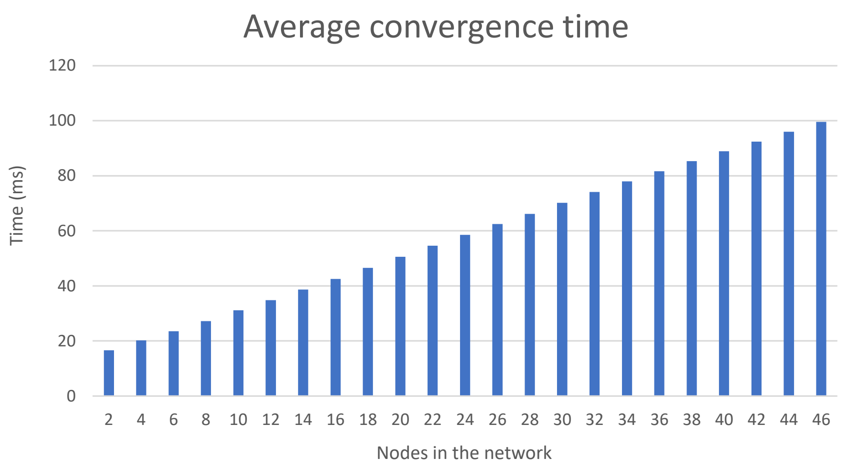


Figure 9.5. Average routing algorithm convergence time with several network sizes

Due to Dijkstra complexity of $O((E+V)\log V)$ (where V are the nodes and E is the number of links in the graph), the algorithm may overgrowth the transmission time in large networks, but it can support the current QKD network size of at most hundred of nodes. In the future, it can be updated with support to routing over different autonomous systems as in OSPFv2, reducing the complexity of the graph representation. The convergence time even in a network of tens of nodes is smaller than the time required by the QKS to deliver a key over 2 hops and this difference increases with the number of hops in the path, therefore the routing algorithm execution time is compatible with the current performances of the QKS. In a real scenario where nodes are deployed over different machines and the packet transmission requires more time the convergence time will be greater, but because the same overhead will be applied also to the other communication between nodes it will not exceed the key retrieval time. With a large network, the timer period for the route invalidation should be defined considering the number of nodes, ensuring that the worst-case convergence time is never higher than the longest path transmission time, ensuring ensure that each node can always receive updated information even from distant peers.

Chapter 10

Conclusions

This thesis work focused on the development of a complete software stack to bring QKD into real scenarios, addressing issues such as performances and scalability.

Tests showed that the QKS reaches an operating speed greater than one order of magnitude with respect to the previous version (*QKS 1.0*), thanks to its asynchronous behaviour and the new faster components. The speed of PTP exchanges is compatible with protocols already in use in real scenarios, especially if several keys are retrieved in the same request. Currently, performances are not limited by the stack itself but by QKD devices exchange rates and either in presence of faster devices the entire stack can be scaled horizontally adding replicas to support them without much effort for system administrators.

The development of the routing algorithm and the management of trusted repeaters to perform multi-hop key exchanges allows this solution to be integrated into a QKD network in its early stages when not a lot of point-to-point links are available. Trusted repeaters networks are not free from defects and the security of the exchange is preserved only if the middle nodes can be trusted, but they are the only available option while waiting for the development of quantum repeaters. Moreover, the speed of long-distance exchanges is significantly lower than PTP ones, hence keys shared among distant hops can not be retrieved guaranteeing the same performances, thus their effectiveness is limited.

The authentication of classical channels, both in the external interface and in the routing modules packets, must be addressed carefully: standard algorithms do not provide security in a quantum environment, hence quantum-resistant solutions are required. The same issue should be considered with the encryption of internal communication between SAEs and the QKS and between the QKS and the QKDM because they transmit sensitive data such as keys and credentials that must not be sent in clear. This version does not solve these issues, and thus future developers must address them before bringing the QKS into a production environment but the flexibility of the architecture and the interfaces allows simple integration of different methods, either based on PQC or QKD keys recycle.

The development of a first version of the Kubernetes operator shows that it is also possible to integrate the software stack in a cluster, supporting interaction with the QKS only through Kubernetes resources. This demonstrates that security applications can easily retrieve QKD derived keys that can be used in their tasks. In the future, it can be easily extended to support more functionalities and the automated deployment of the QKS and it can be packaged and published as a free to use and plug and play solution.

Bibliography

[1] R. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems”, Communications of the ACM, vol. 21, February 1978, pp. 120–126, DOI 10.1145/359340.359342

[2] National Institute of Standards and Technology, “Digital Signature Standard (DSS)” FIPS 186-4, July 2013, DOI 10.6028/NIST.FIPS.186-4

[3] L. C. E. Barker and, A. Roginsky, A. Vassilev, and R. Davis, “Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography” SP 800-56A Rev. 3, April 2018, DOI 10.6028/NIST.SP.800-56Ar3

[4] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3” RFC-8446, August 2018, DOI 10.17487/RFC8446

[5] P. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”, SIAM Journal on Computing, vol. 26, October 1997, pp. 1484–1509, DOI 10.1137/S0097539795293172

[6] C. Gidney and M. Ekerå, “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”, SIAM Journal on Computing, vol. 5, April 2021, p. 433, DOI 10.22331/q-2021-04-15-433

[7] H. Lo and H. F. Chau, “Unconditional security of quantum key distribution over arbitrarily long distances”, Science, vol. 283, March 1999, pp. 2050–2056, DOI 10.1126/science.283.5410.2050

[8] 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, August 2021, pp. 115270–115291, DOI 10.1109/ACCESS.2021.3102313

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

[10] 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

[11] X. Ma, X. Yuan, Z. Cao, B. Qi, and Z. Zhang, “Quantum random number generation”, npj Quantum Information, vol. 2, June 2016, DOI 10.1038/npjqi.2016.21

[12] 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. S. Shaari, M. Tomamichel, V. C. Usenko, G. Vallone, P. Villoresi, and P. Wallden, “Advances in quantum cryptography”, Advances in Optics and Photonics, vol. 12, June 2016, p. 1012, DOI 10.1364/AOP.361502

[13] F. Xu, X. Ma, Q. Zhang, H. Lo, and J. Pan, “Secure quantum key distribution with realistic devices”, Reviews of modern Physics, vol. 92, May 2020, p. 25002, DOI 10.1103/RevModPhys.92.025002

[14] A. Trizna and A. Ozols, “An overview of quantum key distribution protocols”, Information Technology and Management Science, vol. 21, December 2018, pp. 37–44, DOI 10.7250/itms-2018-0005

[15] C. H. Bennett and G. Brassard, “Quantum cryptography: Public key distribution and coin tossing”, Theoretical Computer Science, vol. 560, 2014, pp. 7–11, DOI h10.1016/j.tcs.2014.05.025

[16] M. Dusek, N. Lütkenhaus, and M. Hendrych, “Chapter 5 - Quantum cryptography”, Progress in Optics (E. Wolf, ed.), vol. 49, pp. 381–454, Elsevier, 2006, DOI https://doi.org/10.1016/S0079-6638(06)49005-3

[17] M. Mehic, M. Niemiec, S. Rass, J. Ma, M. Peev, A. Aguado, V. Martin, S. Schauer, A. Poppe, C. Pacher, and M. Voznak, "Quantum key distribution: A networking perspective", ACM Computing Surveys, vol. 53, September 2020, DOI 10.1145/3402192

[18] R. Alléaume, C. Branciard, J. Bouda, T. Debuisschert, M. Dianati, N. Gisin, M. Godfrey, P. Grangier, T. Länger, N. Lütkenhaus, C. Monyk, P. Painchault, M. Peev, A. Poppe, T. Pornin, J. Rarity, R. Renner, G. Ribordy, M. Riguidel, L. Salvail, A. Shields, H. Weinfurter, and A. Zeilinger, "Using quantum key distribution for cryptographic purposes: A survey", Theoretical Computer Science, vol. 560, December 2014, pp. 62–81, DOI https://doi.org/10.1016/j.tcs.2014.09.018

[19] P. S. S. Rass, "A unified framework for the analysis of availability, reliability and security, with applications to quantum networks", IEEE Transactions on Systems, Man, and Cybernetics, Part C, vol. 41, January 2011, pp. 107–119, DOI 10.1109/TSMCC.2010.2050686

[20] D. Collins, N. Gisin, and H. D. Riedmatten, "Quantum relays for long distance quantum cryptography", Journal of Modern Optics, vol. 52, no. 5, 2005, pp. 735–753, DOI 10.1080/09500340412331283633

[21] W. Dür, H. Briegel, J. I. Cirac, and P. Zoller, "Quantum repeaters based on entanglement purification", Physical Review A, vol. 59, January 1999, pp. 169–181, DOI 10.1103/PhysRevA.59.169

[22] 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, Karlsruhe (Germany), 2003, pp. 227–238, DOI 10.1145/863955.863982

[23] M. Peev, C. Pacher, R. Alléaume, C. Barreiro, J. Bouda, W. Boxleitner, T. Debuisschert, E. Diamanti, M. Dianati, J. F. Dynes, S. Fasel, S. Fossier, M. Fürst, J. Gautier, O. Gay, N. Gisin, P. Grangier, A. Happe, Y. Hasani, M. Hentschel, H. Hübel, G. Humer, T. Länger, M. Legré, R. Lieger, J. Lodewyck, T. Lorünser, N. Lütkenhaus, A. Marhold, T. Matyus, O. Maurhart, L. Monat, S. Nauerth, J. Page, A. Poppe, E. Querasser, G. Ribordy, S. Robyr, L. Salvail, A. W. Sharpe, A. J. Shields, D. Stucki, M. Suda, C. Tamas, T. Themel, R. T. Thew, Y. Thoma, A. Treiber, P. Trinkler, R. Tualle-Brouri, F. Vannel, N. Walenta, H. Weier, H. Weinfurter, I. Wimberger, Z. L. Yuan, H. Zbinden, and A. Zeilinger, "The SECOQC quantum key distribution network in Vienna", New Journal of Physics, vol. 11, July 2009, p. 075001, DOI 10.1088/1367-2630/11/7/075001

[24] T. Chen, J. Wang, H. Liang, W. Liu, Y. Liu, X. Jiang, Y. Wang, X. Wan, W. Cai, L. Ju, L. Chen, L. Wang, Y. Gao, K. Chen, C. Peng, Z. Chen, and J. Pan, "Metropolitan all-pass and inter-city quantum communication network", Optics Express, vol. 18, December 2010, pp. 27217–27225, DOI 10.1364/OE.18.027217

[25] J. Moy, "OSPF Version 2" RFC-2328, April 1998, DOI 10.17487/RFC2328

[26] 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 (E. J. Donkor, A. R. Pirich, and H. E. Brandt, eds.), 2005, pp. 138–149, DOI 10.1117/12.606489

[27] S. Wang, W. Chen, Z. Yin, H. Li, D. He, Y. Li, Z. Zhou, X. Song, F. Li, D. Wang, H. Chen, Y. Han, J. Huang, J. Guo, P. Hao, M. Li, C. Zhang, D. Liu, W. Liang, C. Miao, P. Wu, G. Guo, and Z. Han, "Field and long-term demonstration of a wide area quantum key distribution network", Optics Express, vol. 22, September 2014, pp. 21739–21756, DOI 10.1364/OE.22.021739

[28] M. Mehic, O. Maurhart, S. Rass, D. Komosny, F. Rezac, and M. Voznak, "Analysis of the public channel of quantum key distribution link", IEEE Journal of Quantum Electronics, vol. 53, no. 5, 2017, pp. 1–8, DOI 10.1109/JQE.2017.2740426

[29] A. Aguado, V. Lopez, D. Lopez, M. Peev, A. Poppe, A. Pastor, J. Folgueira, and V. Martin, "The engineering of software–defined quantum key distribution networks", IEEE Communications Magazine, vol. 57, July 2019, pp. 20–26, DOI 10.1109/MCOM.2019.1800763

[30] H. Wang, Y. Zhao, and A. Nag, "Quantum-key-distribution (QKD) networks enabled by software-defined networks (SDN)", Applied Sciences, vol. 9, May 2019, DOI 10.3390/app9102081

[31] Y. Cao, Y. Zhao, Y. Wu, X. Yu, and J. Zhang, "Time-scheduled quantum key distribution (QKD) over WDM networks", Journal of Lightwave Technology, vol. 36, May 2018, pp. 3382–3395, DOI 10.1109/JLT.2018.2834949

[32] A. Aguado, V. Lopez, J. Martinez-Mateo, T. Szyrkowiec, A. Autenrieth, M. Peev, D. Lopez, and V. Martin, “Hybrid conventional and quantum security for software defined and virtualized networks”, Journal of Optical Communications and Networking, vol. 9, October 2017, pp. 819–825, DOI 10.1364/JOCN.9.000819

[33] H. Zimmermann, “OSI reference model - the ISO model of architecture for open systems interconnection”, IEEE Transactions on Communications, vol. 28, no. 4, 1980, pp. 425–432, DOI 10.1109/TCOM.1980.1094702

[34] G. Meyer, “The PPP encryption control protocol (ECP)” RFC-1968, June 1996, DOI 10.17487/RFC1968

[35] M. Seaman, “802.1AE: MAC Security (MACsec)” IEEE Std. 802.1AE-2018, 2018, DOI 10.1109/IEEESTD.2018.8585421

[36] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, “Internet key exchange protocol version 2 (IKEv2)” RFC-7296, October 2014, DOI 10.17487/RFC7296

[37] F. T. Sufyan, “A novel extension of SSL/TLS based on quantum key distribution”, 2008 International Conference on Computer and Communication Engineering, Kuala Lumpur (Malaysia), May 2008, pp. 919–922, DOI 10.1109/ICCCE.2008.4580740

[38] P. S. S. Rass, a. Wiegele, “Building a quantum network: How to optimize security and expenses”, Journal of Network and Systems Management, vol. 18, September 2010, pp. 283–299, DOI 10.1007/s10922-010-9162-0

[39] K. Shimizu, T. Honjo, M. Fujiwara, T. Ito, K. Tamaki, S. Miki, T. Yamashita, H. Terai, Z. Wang, and M. Sasaki, “Performance of long-distance quantum key distribution over 90-km optical links installed in a field environment of tokyo metropolitan area”, Journal of Lightwave Technology, vol. 32, January 2014, pp. 141–151, DOI 10.1109/JLT.2013.2291391

[40] Z. Han, F. Xu, W. Chen, S. Wang, Z. Yin, Y. Zhang, Y. Liu, Z. Zhou, H. Li, D. Liu, and G. Guo, “An application-oriented hierarchical quantum cryptography network test bed”, Optical Fiber Communication Conference, San Diego (CA, USA), March 2010, DOI 10.1364/OFC.2010.OTuK4

[41] S. Wang, W. Chen, Z. Yin, Y. Zhang, T. Zhang, H. Li, F. Xu, Z. Zhou, Y. Yang, D. Huang, L. Zhang, F. Li, D. Liu, Y. Wang, G. Guo, and Z. Han, “Field test of wavelength-saving quantum key distribution network”, Optics Letters, vol. 35, July 2010, pp. 2454–2456, DOI 10.1364/OL.35.002454

[42] Q. Zhang, F. Xu, Y. Chen, C. Peng, and J. Pan, “Large scale quantum key distribution: challenges and solutions”, Optics Express, vol. 26, September 2018, pp. 24260–24273, DOI 10.1364/OE.26.024260

[43] J. Yin, Y. Cao, Y. Li, S. Liao, L. Zhang, J. Ren, W. Cai, W. Liu, B. Li, H. Dai, G. Li, Q. Lu, Y. Gong, Y. Xu, S-Li, F. Li, Y. Yin, Z. Jiang, M. Li, J. Jia, G. Ren, D. He, Y. Zhou, X. Zhang, N. Wang, X. Chang, Z. Zhu, N. Liu, Y. Chen, C. Lu, R. Shu, C. Peng, J. Wang, and J. Pan, “Satellite-based entanglement distribution over 1200 kilometers”, Science, vol. 356, June 2017, pp. 1140–1144, DOI 10.1126/science.aan3211

[44] European Telecommunications Standards Institute, “ETSI GS QKD 002: Quantum key distribution (QKD) use cases” `https://www.etsi.org/deliver/etsi_gs/qkd/001_099/002/01.01.01_60/gs_qkd002v010101p.pdf`, June 2010

[45] European Telecommunications Standards Institute, “ETSI GS QKD 003: Quantum key distribution (QKD) components and internal interfaces” `https://www.etsi.org/deliver/etsi_gr/QKD/001_099/003/02.01.01_60/gr_QKD003v020101p.pdf`, March 2018

[46] European Telecommunications Standards Institute, “ETSI GS QKD 004: Quantum Key Distribution (QKD) Application Interface” `https://www.etsi.org/deliver/etsi_gs/QKD/001_099/004/02.01.01_60/gs_QKD004v020101p.pdf`, August 2020

[47] European Telecommunications Standards Institute, “ETSI GS QKD 005: Quantum Key Distribution (QKD) Security Proofs” `https://www.etsi.org/deliver/etsi_gs/QKD/001_099/005/01.01.01_60/gs_QKD005v010101p.pdf`, December 2010

[48] European Telecommunications Standards Institute, “ETSI GS QKD 007: Quantum Key Distribution (QKD) Vocabulary” `https://www.etsi.org/deliver/etsi_gr/QKD/001_099/007/01.01.01_60/gr_QKD007v010101p.pdf`, December 2018

[49] European Telecommunications Standards Institute, “ETSI GS QKD 008: Quantum Key Distribution (QKD) QKD Module Security Specification” `https://www.etsi.org/deliver/etsi_gs/QKD/001_099/008/01.01.01_60/gs_QKD008v010101p.pdf`, December 2010

[50] European Telecommunications Standards Institute, "ETSI GS QKD 011: Quantum Key Distribution (QKD) Component characterization: characterizing optical components for QKD systems" https://www.etsi.org/deliver/etsi_gs/QKD/001_099/011/01.01.01_60/gs_QKD011v010101p.pdf, May 2016

[51] European Telecommunications Standards Institute, "ETSI GS QKD 012: Quantum Key Distribution (QKD) Device and Communication Channel Parameters for QKD Deployment" https://www.etsi.org/deliver/etsi_gs/QKD/001_099/012/01.01.01_60/gs_QKD012v010101p.pdf, February 2019

[52] European Telecommunications Standards Institute, "ETSI GS QKD 014: Quantum Key Distribution (QKD) Protocol and data format of REST-based key delivery API" https://www.etsi.org/deliver/etsi_gs/QKD/001_099/014/01.01.01_60/gs_QKD014v010101p.pdf, February 2019

[53] European Telecommunications Standards Institute, "ETSI GS QKD 015: Quantum Key Distribution (QKD) Control Interface for Software Defined Networks" https://www.etsi.org/deliver/etsi_gs/QKD/001_099/015/01.01.01_60/gs_QKD015v010101p.pdf, March 2021

[54] Telecommunication Standardiztion Sector of ITU, "Recommendation ITU-T Y.3800 - Overview on networks supporting quantum key distribution" https://www.itu.int/rec/T-REC-Y.3800-202004-I!Cor1, April 2020

[55] Telecommunication Standardiztion Sector of ITU, "Recommendation ITU-T Y.3801 - Functional requirements for quantum key distribution networks" https://www.itu.int/rec/T-REC-Y.3801-202004-I, April 2020

[56] Telecommunication Standardiztion Sector of ITU, "Recommendation ITU-T Y.3802 - Quantum key distribution networks - Functional architecture" https://www.itu.int/rec/T-REC-Y.3802-202104-I!Cor1, April 2021

[57] Telecommunication Standardiztion Sector of ITU, "Recommendation ITU-T Y.3803 - Quantum key distribution networks - Key management" https://www.itu.int/rec/T-REC-Y.3803-202012-I, December 2020

[58] Telecommunication Standardiztion Sector of ITU, "Recommendation ITU-T Y.3804 - Quantum key distribution networks - Control and management" https://www.itu.int/rec/T-REC-Y.3804-202009-I, September 2020

[59] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead", IEEE Access, vol. 7, April 2019, pp. 52976–52996, DOI 10.1109/ACCESS.2019.2911732

[60] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub", CODASPY '17: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale (AZ, USA), March 2017, pp. 269–280, DOI 10.1145/3029806.3029832

[61] M. P. Souppaya, J. Morello, and K. Scarfone, "Application container security guide", Special Publication (NIST SP), vol. 800, September 2017, DOI 10.6028/NIST.SP.800-190

[62] Cloud Native Computing Foundation, "Kubernetes documentation" https://kubernetes.io/docs/

[63] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices", 2020 IEEE Secure Development (SecDev), virtual conference, September 2020, pp. 58–64, DOI 10.1109/SecDev45635.2020.00025

[64] R. Mijumbi, J. Serratn, J. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges", IEEE Communications Surveys Tutorials, vol. 18, September 2016, pp. 236–262, DOI 10.1109/COMST.2015.2477041

[65] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey", Computer Networks, vol. 54, October 2010, pp. 2787–2805, DOI 10.1016/j.comnet.2010.05.010

[66] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey", Journal of Systems Architecture, vol. 98, September 2019, pp. 289–330, DOI 10.1016/j.sysarc.2019.02.009

[67] S. Josefsson, "The base16, base32, and base64 data encodings" RFC-4648, October 2006, DOI 10.17487/RFC4648

[68] O. Maurhart, "QKD networks based on Q3P", Applied Quantum Cryptography

(c. Kollmitzer and M. Pivk, eds.), pp. 151–171, Springer, 2010, DOI 10.1007/978-3-642-04831-9_8

[69] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. D. Jr., "Advanced encryption standard (AES)" Federal Inf. Process. Stds. (NIST FIPS) - 197, November 2001, DOI 10.6028/NIST.FIPS.197

Appendix A

User's manual

This chapter wants to explain all the steps required to deploy the entire QKS stack presented in this thesis work into a Kubernetes cluster. The first section describes how to deploy the stack while the second shows how to integrate it with the operator. For each step a correctly deployed K3s cluster is considered running on the used machine; refers to K3s official documentation [1] to find out how to install it. Because the QKS can run on its own and is not bound to Kubernetes, the last section of this chapter describes how to deploy the entire stack on Docker.

A.1 QKS deployment in Kubernetes

To deploy the stack on the K3s cluster a set of configuration files are required: an example set can be found in the project repository, in the *kubernetes* folder [2]. All the paths shown in this section refer to this repository in the textiasync branch. First of all a *namespace* in which the entire stack will be deployed must be choosen. To create a new namespace use the command:

```
kubectl create namespace <namespace_name>
```

To create any resource described in this chapter create the corresponding `.yaml` file and use the command:

```
kubectl apply -n <namespace_name> -f <path/to/the/file>
```

To deploy the stack you must first create the resources related to MongoDB, Vault, Redis and Keycloak and after that the QKS core and the routing module. Persistent volume resources are required both for Vault and for MongoDB and are the only resources that are not namespace scoped. You have to create them assigning a size compatible with the size of the network you are going to have. For the configuration proposed in the test chapter(9) 100 Mbit for each one was enough. An example file can be found in the GitHub folder at the path `/config/persistent_volume.yaml`

MongoDB

To deploy MongoDB create, in the corresponding namespace, a secret with two keys: `mongo-root-username` and `mongo-root-password` with the admin credential required by the QKS to access the database and to create users and databases for the QKDMs. Data in Kubernetes secret must be encoded in the *base64* format. An example file can be found in `/config/mongo_secret.yaml`. Now you have to create the mongoDB `statefulSet`, the corresponding `service` and the `persistentVolumeClaim` applying the resources in `/resources/mongo.yaml`. Because MongoDB database and collections are created only after the first object insertion and objects do not have a fixed structure it is not required to provide an initialization script.

[1] https://rancher.com/docs/k3s/latest/en/installation/

[2] https://github.com/ignaziopedone/qkd-keyserver/tree/async/kubernetes

Redis

To deploy Redis create in the namespace a `configMap` with the configuration file that should be injected in the Redis container. Refer to the example file in `/config/redis_configmap.yaml` to enable authenticated access and change the `password` at line 8. Redis `deployment` and `service` can be created applying the file in `/resources/redis.yaml` The topics to be used for messages and the database for routing tables are defined in the QKS core and routing configuration file.

Keycloak

Keycloak configuration requires both a `configMap` and a `secret` resources. The former should contains the JSON representation of the *realm* that will be used: it must contains the *client* object for the QKS and the roles for *admins*, *saes* and *QKDMs* as well as the mapping for the data returned in the login token. The JSON object can be exported from an already running instance and can be copied in the configMap. The secret object should contain admin credentials both for the QKS admin user and for the Realm admin, the QKS client ID and its secret (that should match the ones in the configMap). Keycloak deployment and service configurations can be found in `/resources/keycloak.yaml`. The `nodePort` service type is not mandatory and it can be converted into a `clusterIP` type if access from the outside is not necessary.

To retrieve the authentication token a user should interact with Keycloak with the following HTTP call:

```
Method: POST
URL: http://<keycloak_host>:8080
    /auth/realms/<realm>/protocol/openid-connect/token
Headers: Content-Type: application/x-www-form-urlencoded
Data: client_id=<client_id>&client_secret=<client_secret>&
    grant_type=password&scope=openid&username=<username>&password=<password>
```

If the login procedure is completed successfully the access token will be placed in the `access_token` field of the returned JSON object. With the proposed configuration the `realm` to be used is *qks* and the `client_id` is *qks*, but they can be changed in the configuration file. The token must be sent to the QKS in the *Authorization* header:

```
Authorization : Bearer <token>
```

Vault

The Vault container requires a configuration file to be injected to correctly set up the server. An example configMap which contain this file can be found in `/config/vault_configmap.yaml`. Deploy the Vault `statefulSet` and `service` through the file code/resources/vault.yaml Vault requires both inizialization and unsealing procedure before being accessed by the QKS. Initialize it executing the following command:

```
kubectl exec -n <namespace_name> --stdin --tty <vault_pod_name> -- vault
    operator init
```

and unseal it with:

```
kubectl exec -n <namespace_name> --stdin --tty <vault_pod_name> -- vault
    operator unseal
```

Use the parameters `--key-shares` and `--key-threshold` in the `init` command to specify the total amount of unsealing keys and how many of them are required to unseal the system after a reboot. Save the returned `root token` that must be injected in the QKS core configuration.

QKS core

The QKS core requires a configuration file containing the information on how to reach and access all the other external modules. The `.yaml` file can be injected in the container through a `configMap` as for the other components. The `root token` returned during the Vault initialization phase has to be placed in this configuration file as well as MongoDB and Redis credentials. A complete configuration example can be found in `/config/qks_configmap.yaml`, while figure A.1 shows an example of a section of a configMap for the core module. The deployment object and the corresponding service can be created applying the file `/resources/qks.yaml`. The NodePort used to expose the service can be changed to any other unused port or a `LoadBalancer` service type if needed.

```
data:
  qks-config: |
    qks:
      id: qks1
      ip: qks-service # service name to reach the core from the cluster
      port: 4000 # service port to reach the core from the cluster
      max_key_per_request: 20
      max_key_size: 512
      min_key_size: 128
      max_sae_id_count: 0
      indirect_max_key_count: 20
    mongo_db:
      host: mongodb-service # MongoDB service name
      port: 27017
      user: rootuser
      password: rootpwd
      auth_src : admin # database used for authentication
      db : qks # database to use for storing data
    vault:
      host : vault-service # Vault service name
      port : 8200
      token : s.KxoGNMbCu5Yvu7dImOvWlkjZ # root token
```

Figure A.1. Section of an example QKS core configMap.

QKS routing

As for the core component also the routing ones require a configuration file that can be injected in the container through a `configMap`. In the `qks` and `routing` parameters, it must contain the information on how to reach the core and the routing module from the outside of the cluster, which are the information that will be spread by the routing algorithm: the ports must correspond to the ones exposed through the NodePort services and the address must be public ones. The `redis` and `mongo_db` parameters should match the ones in the QKS core module configMap. Figure A.2 shows an example of a section of a configMap for the routing module. The module can be deployed through the resources in `/resources/routing.yaml`. As for the core module also the routing service can be changed to a LoadBalancer type if needed.

QKDM

The QKDM is the last component that should be deployed in the stack. Because it requires a registration procedure its deployment is not as straightforward as for the other components. If

```
data:
  routing-config: |
    qks:
      id: qks1
      ip: core.qks1.cluster1 # cluster address
      port: 30000 # node port
    routing:
      timer : 20
      ip: routing.qks1.cluster1 # cluster address
      port : 30500 # node port
```

Figure A.2. Section of an example QKS routing configMap.

the other peer the module is connected to is attached to a QKS which is not known in the current QKS it must be registered with the corresponding API (`POST /api/v1/qks`). Because Keycloak does not consider valid for interacting with the QKS tokens requested by users outside of the cluster network a pod containing management script has been developed and its resource file can be found in `/QKDMmanagement/management-pod.yaml`. To register the new QKS through the pod script execute it with the argument `3` followed by the QKS data (an example can be found in the pod file). You will need an account with the *admin* role on the QKS client to perform this operation and all the other management operations: you can register it through the Keycloak admin console or this pod script (with the argument `1`) The QKDM can be registered to the QKS either through its API (`POST /api/v1/qkdm/actions/attach`) or by an admin interacting directly with the QKS. The second option is preferred because it allows you to save returned information in a configMap; otherwise a persistent volume is required to safely store those data even in case of crashes because the QKDM pod can not modify configMap objects. To register it with the script launch the pod with argument `2` followed by the QKDM data; returned data will be printed on the standard output and can be copied in the QKDM configMap (an example can be found in `/config/qkdm_configmap.yaml`) The QKDM pod will fail if it controls the *sender* device and if the other peer is not already up: instantiate first the *receiver* device and then the sender. Pay attention to the fact that the *sender* in the `device` configuration parameter must contain the information on how to reach the receiver in the other node, while the latter must specify the `containerPort` it is listening on. Figure A.3 shows an example of a section of a configMap for a QKDM. An example of the `deployment` and `service` resource can be found in `/resources/qkdm.yaml`. A QKDM stream can be started by an administrator directly through the corresponding API to the QKS (`POST /api/v1/qkdms/<qkdm_id>/streams`) or with the script in the management pod (with argument `4`)

A.2 Kubernetes Operator integration

The operator can be deployed in the Kubernetes cluster to provide an easier interaction with the QKS from the SAEs point of view, but it is not mandatory: the QKS stack is completely functional also on its own but requires interaction through its REST interface. The files required to deploy the QKD operator in the cluster are located in the `/operator` folder. The `saes` and the `keyRequests` custom resources can be created applying the `/operator/saeCRD.yaml` and `/operator/keyRequestCRD.yaml` files, they do not require the namespace parameter in the command because resource definition is valid for the entire cluster even if the resources are namespace scoped. In an environment where Role-Based Access Control (RBAC) has been enabled the operator must be granted access to the resources it has to manage with cluster-wide permission because SAEs should operate in a different namespace from the operator. Apply the `clusterRole`, the `clusterRoleBinding` and the `serviceAccount` describe in `/operator/operatorRBAC.yaml` changing the clusterRole `subjects:namespace` parameter with the namespace the stack is running in. The operator can be deployed as a standard deployment object, which template can be

```
data:
  qkdm-config: |
    qkdm:
      id: qkdm1 # ID of this module
      dest_ID: qkdm2 # ID of the peer module
      dest_IP: qkdm.qks2.cluster2 # address of the peer module
      dest_port: 31000
      ip: qkdm-service # service name to reach the QKDM from the cluster
      port: 5000 # service port to reach the QKDM from the cluster
      key_size: 128
      max_key_count: 100
      protocol: fake # name of the used QKD protocol
      init: true
    qkd_device:
      role: sender # role of the device
      host: device.qks2.cluster2 # address of the peer device
      port: 32000
```

Figure A.3. Section of an example QKDM configMap.

found in `/operator/operator-deployment.yaml`. It is necessary to specify in the deployment object the name of the namespaces where the operator can find the secret in used to access keycloak with admin role to create new SAEs users in the `SECRET_NAMESPACE` environment variable as shown in the code reported here below:

```
spec:
    serviceAccountName: qkd-operator
    containers:
    - name: qkd-operator
        image: ignaziopedone/qkd:qks_operator
        env:
        - name: SECRET_NAMESPACE
        value: qkdns
```

Once the operator is deployed, SAEs can be registered creating a `sae` resource specifying the name and `true` in the `registration_auto` parameter. With the SAE correctly registered, its Keycloak credentials can be accessed by administrators in the secret `<sae_name>-credentials` in the namespace it is deployed into. To retrieve a key create a `keyRequest` object specifying the master and the slave SAEs and the key requested. To retrieve keys already reserved insert their IDs in the `ids` parameter. If the request is completed successfully a secret with the same name as the keyRequests will be created in the SAE namespace, in case of failure nothing will be created and the requests should be recreated; no *retry* behaviours have been implemented. Because resource names in Kubernetes have to be unique in a namespace each SAE should identify a way to produce unique names such as UUIDs.

A.3 QKS deployment in Docker

This section describes how to deploy the entire QKS stack in Docker. The same configuration files described in the Kubernetes chapter are required with Docker, despite they are not deployed through configMaps and secrets. For each pod that requires a configuration file injected through a configMap, the configuration file should be injected through a volume in the same path as the configMap. Environment variables injected through secrets here must be passed directly as variables in the container arguments or through Docker secrets. There are no differences in the parameters of the configuration files between the two deployment scenarios. To simplify the deployment a *docker-compose* file can be used, but it does not solve the issues related to the injection of configuration data and the required deployment sequence described in the Kubernetes section. If a docker-compose is used each container can reach the others via their container name, the mapping between the name and the IP address is performed directly by Docker when the container is created through Docker networking functionalities, while services must be replaced with the corresponding port mapping. An example docker-compose file can be found in the GitHub repository.

Appendix B

Developer's reference guide

This chapter wants to describe the main implementation choices, the libraries used to develop the proposed code and how to modify and extend the functionalities.

B.1 Quantum Key Server

The QKS code can be found in the `qkd-keyserver` GitHub repository in the `async` branch[1]. The *QKS core* code can be found in the `qks_core` folder while the *routing module* code in the `routing` one. Both have been developed in Python 3.9 to take full advantage of the *async* patter and to exploit *type hints* improving code readability. All the developed Docker images are available on DockerHub[2], tags have been used to describe them.

B.1.1 QKS core

The *qks_core* folder contains the following Python files:

server.py is the file that contains the Quart server which receives and manages the incoming HTTP calls, return formatted results and handles error messages. It can receive in the input parameters the name of the configuration file to use.

api.py contains all the functions called from the `server`, it contains the main application logic of the three interfaces and the management functions required to init the server.

asyncVaultClient.py is an asynchronous interface built to communicate with Vault which contains all the methods required by the QKDM and the QKS to interact with Vault.

These files have been packaged in a *Docker image* to simplify the deployment of the app. The Docker image can be built from the `Dockerfile` in the same folder as the code, with the command:

```
docker build -f <path/to/dockerfile> -t <image_name:image_tag>
```

Note that in a production environment the *Quart* web server should by run directly through the Python file, but an ASGI webserver (e.g. *Hypercorn*[3]) should be used in front of it. To run the server with Hypercorn use the command:

```
hypercorn server:app
```

The Docker image should be modified accordingly.

[1] https://github.com/ignaziopedone/qkd-keyserver/tree/async

[2] https://hub.docker.com/r/ignaziopedone/qkd/

[3] https://pgjones.gitlab.io/hypercorn/

Interaction with other components

The interaction with MongoDB is performed through the official `motor`[4] library, which provides support to asynchronous communication with the Python standard library `asyncio`.

The asynchronous interaction with Redis is carried on with the `aioredis`[5] library; a version equal or higher to the `2.0.0` is required to correctly perform all the operations.

The `asyncVaultClient` interface is built on top of the `async_hvac`[6] libraries used to interact with Vault. It contains all the methods required by the QKS and the QKDM, allowing to substitute Vault with another product without significant changes in the `api` code. It provides exceptions management and errors handling, returning empty values when the operation is not completed successfully.

The interaction with Keycloak, to validate the authorization token received in the `Authorization` header of requests over the northbound interface is performed through the `verifyToken` token function which receives it and forward it to the Keycloak endpoint through the Keycloak REST API with the following call:

```
Method: POST
URL: http://<keycloak_host>:8080
    /auth/realms/<realm>/protocol/openid-connect/userinfo
Headers:
    - Content-Type: application/json
    - Authorization: Bearer <access_token>
```

Detailed information on the available methods in the API can be found in the official documentation[7]. The authentication is not bound to Keycloak or the OIDC protocol, therefore this function can be modified to support any other validation techniques, either online or offline.

Logging

Loggin is performed with the standard Python library `logging`. The logger is configured before the Quart app initialization with the code:

```
logging.basicConfig(filename='qks.log', filemode='w', level=logging.INFO)
```

and logs are perfomed executing the proper method on the `app.logger` object, for example for a *warning* message:

```
app.logger.warning(message_string)
```

Data are both saved in the log file specified in the `basicConfig` method and printed on the standard output through the Hypercorn ASGI web server which runs with Quart. To save data persistently a persistent volume should be added to map the folder `/usr/app/qkd-keyserver/qks_core`

APIs payloads

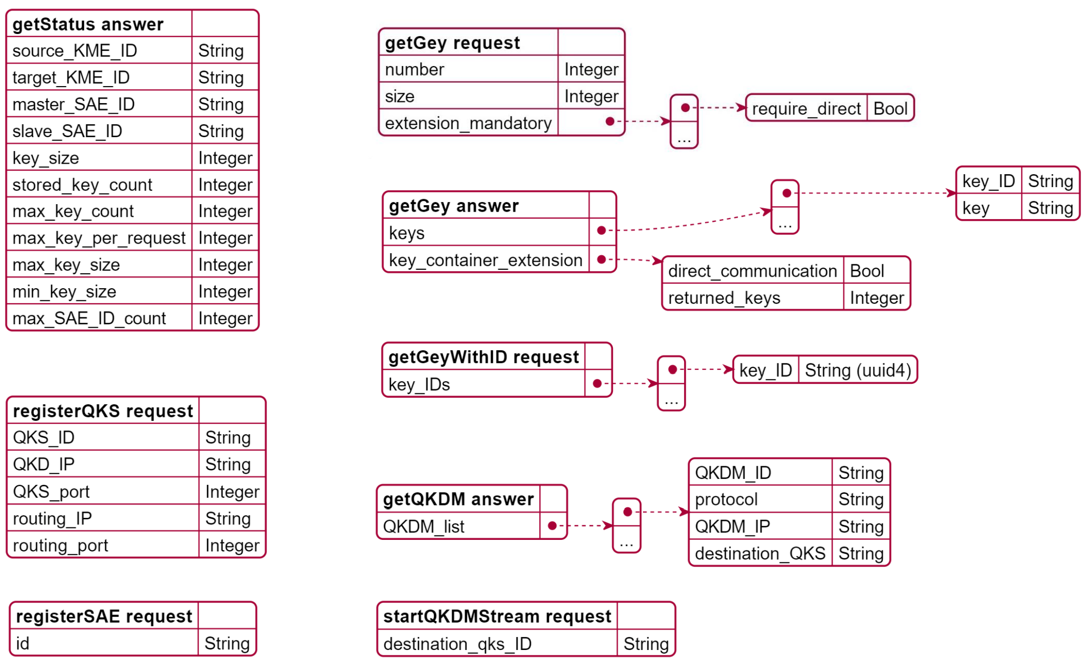
Figure B.1, B.2 and B.3 describe the body of REST requests and response respectively for the methods of the northbound, the southbound and the external interface of the QKS. When a request is received the server first checks if the sender is authenticated and then decode the received payload and check if all fields are present. In case of errors or exceptions in the handling of valid requests, a JSON payload is returned, with a message which describes the error.

[4] https://github.com/mongodb/motor

[5] https://github.com/aio-libs/aioredis-py

[6] https://github.com/Aloomaio/async-hvac

[7] https://www.keycloak.org/docs-api/15.0/rest-api/



getStatus answer	
source_KME_ID	String
target_KME_ID	String
master_SAE_ID	String
slave_SAE_ID	String
key_size	Integer
stored_key_count	Integer
max_key_count	Integer
max_key_per_request	Integer
max_key_size	Integer
min_key_size	Integer
max_SAE_ID_count	Integer

registerQKS request	
QKS_ID	String
QKD_IP	String
QKS_port	Integer
routing_IP	String
routing_port	Integer

registerSAE request	
id	String

getGey request	
number	Integer
size	Integer
extension_mandatory	●

require_direct	Bool

getGey answer	
keys	●
key_container_extension	●

key_ID	String
key	String

direct_communication	Bool
returned_keys	Integer

getGeyWithID request	
key_IDs	●

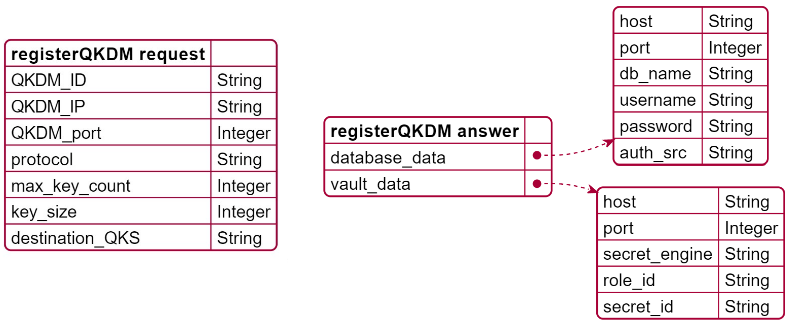
key_ID	String (uuid4)

getQKDM answer	
QKDM_list	●

QKDM_ID	String
protocol	String
QKDM_IP	String
destination_QKS	String

startQKDMStream request	
destination_qks_ID	String

Figure B.1. Northbound interface JSON bodies



registerQKDM request	
QKDM_ID	String
QKDM_IP	String
QKDM_port	Integer
protocol	String
max_key_count	Integer
key_size	Integer
destination_QKS	String

registerQKDM answer	
database_data	●
vault_data	●

host	String
port	Integer
db_name	String
username	String
password	String
auth_src	String

host	String
port	Integer
secret_engine	String
role_id	String
secret_id	String

Figure B.2. Northbound interface JSON bodies

B.1.2 Routing module

The *routing* folder contains the following Python files:

asyncRoutingApp.py is the file with the routing algorithm, that manages all the routing logic, the interaction with Redis and the computation of the cost for each link. It can receive in the input parameters the name of the configuration file to use.

lsaPacket.py contains the packet class, both in JSON and in the raw encoded version and the functions to encode and decode them.

qkdGraph.py contains the graph class and the classes for nodes and SAEs. It is used to represent the QKD network and the links between QKSs and to compute and return routing tables.

These files have been packaged in a *Docker image* that can be found in the `routing` folder, to simplify the deployment of the app.

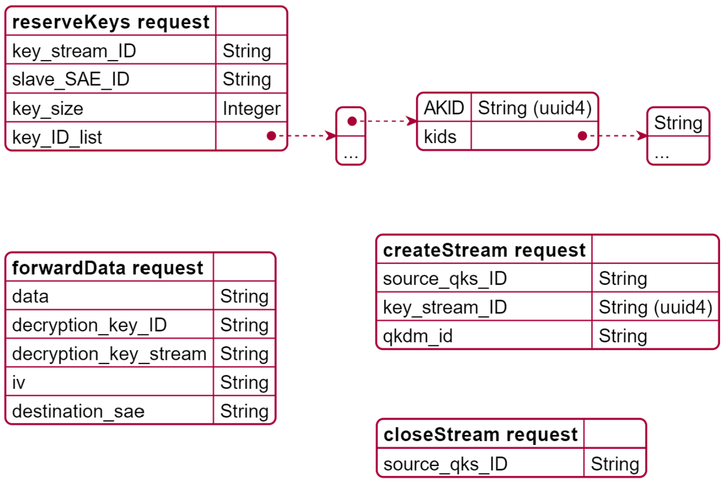


Figure B.3. Northbound interface JSON bodies

Cost function

The routing cost function can be found in the `asyncRoutingApp` file and computes the cost of each link in the following way:

```
cost_param = {'c0' : 100, 'c1' : -50, 'c2' : -25}
def routeCost(old: int, new: int, tot : int) -> int :
    global cost_param
    delta : int = ( new - old )
    cost : float = cost_param['c0'] + cost_param['c1'] * ( new / tot ) +
        cost_param['c2'] * ( delta / tot )
    return int(cost)
```

It receives the number of available keys in the QKDM at the previous interaction `old`, the current number of available keys (`new`) and the maximum number of storable keys (`tot`). `cost_parameter` is a global variable which contains the equation coefficients. This function can be updated and extended with other parameters without requiring any other change to the code. The only requirement is that the returned value must be greater than 0, otherwise, it can cause unexpected behaviour in the Dijkstra algorithm.

Redis messages

The updates from the QKS are received from Redis `PubSub` topics, in the following way:

```
async def listenForChanges() :
    pubsub : aioredis.PubSub = redis_client.pubsub()
    await pubsub.psubscribe(f"{config['redis']['topic']}-**")
    while True:
        message = await pubsub.get_message(ignore_subscribe_messages=True,
            timeout = 0.1)
        action, name = message['data'].split("-")
        # handle the message
```

Three different topics with the same root word have been used: `<config_name>-sae` for information related to SAEs, `<config_name>-qks` for new QKSs added to the network from the northbound interface method and `<config_name>-link` for added or removed QKD streams. The topic name must be specified in the configuration file and must match between the QKS core and the routing module. The received message is a string formatted with `<action>-<name>` where action can be "add" or "remove" while "name" is the subject to which the action refers. Routing tables are produced by the *graph* object and are pushed into Redis executing a `pipeline` in the `updateGraph` function, which allows performing several actions in a single call, reducing the transmission overhead.

Routing packet parameters

The packet content can be updated and easily extended modifying the parameters list global variable in the `lsaPacket.py` file:

```
element_list = ["version", "type", "source", "routing", "neighbors",
    "timestamp", "auth", "forwarder"]
```

The `encode` and `decode` functions of the JSON version does not require any update even if there are changes in the packet parameters, while the raw encoded version requires a code update to correctly handle changes. The size of each element in the raw encoded packet is defined in the `dims` dictionary global variable.

Logging

Loggin is performed with the standard Python library `logging`, and data are saved in the `/usr/app/qkd-keyserver/routing/routing.log` file. In a containerized environment, the file system of a container is not persisted by default, hence to keep logs after the pod destruction a persistent volume should be added in the routing pod mapping the corresponding folder.

B.2 QKD Module

The QKDM code can be found in the `qkd-module` GitHub repository in the `async` branch[8]. The code has been developed in Python 3.9 to take full advantage of the *async* patter as for the *QKS*, using only one programming language for the whole project.

The `qkdm_src` folder contains the source code of the project, organized in the following Python files and folder:

qkdm_server.py is the file that contains the Quart server which receives and manages the incoming HTTP calls and returns results. It performs the mapping between error codes and their meaning, returning human-readable messages in case of errors. It can receive in the input parameters the name of the configuration file to use.

api.py contains all the functions called from the `qkdm_server` and the management function required to exchange keys, initialize the module and handle the connected QKD device.

asyncVaultClient.py is an asynchronous interface built to communicate with Vault, equal to the one used in the QKS.

QKD_device is the folder that contains the `QKDcore` Python interface that should be extended to handle QKD devices. The `fakeKE.py` file contains the fake simulated protocol used for tests purposes.

These files have been packaged in a *Docker image* that can be found in the project root folder. Because the QKDM can work both as a standalone module and connected to a QKS, the `qkdm_server` file can receive the `server` input parameter, which is a boolean that defines if the QKDM should wait for the registration to a QKS or if it must start with the data received

[8]https://github.com/ignaziopedone/qkd-module/tree/async

in the configuration file. The `reset` input parameter can be used to delete any registration data present in the configuration file, thus deleting the previous registration. Note that in a production environment the *Quart* web server should by run through an ASGI webserver. You can run the server with Hypercorn modifying the Docker image with the following command:

```
hypercorn qkdm_server:app
```

B.3 Kubernetes integration

The code of the operator can be found in the `/kubernetes/operator/operator.py` file in the repository folder. Thanks to the *kopf* framework all the modules, components and configuration required to interact with Kubernetes are automatically managed and the operator code is developed in a single file.

There are three main functions related to action on resources in the operator:

keyreq_on_create is triggered when a `keyRequest` resource is created. It watches the parameters, perform the login for the `master_SAE` into Keycloak and then calls the `getKey` or `getKeyWithKeyIDs` on the QKS. If the requests is completed successfully it creates the corresponding secret.

sae_on_create is triggered when a `sae` resource is created. It perform login with QKS admin credentials into Keycloak and calls the `registerSAE` method to the QKS. If the parameter `registration_auto` is set to *true* it register the SAE into Keycloak. An error is returned if a user with the same username as the SAE is already present; to register a SAE already present in Keycloak the `registration_auto` parameter must be set to *false*

sae_on_delete is triggered when a `sae` resource is deleted. It calls the `unregisterSAE` method to the QKS but does not delete the user from Keycloak.

The operator does not perform any action when a `keyRequest` object is deleted. In the proposed solution it does not have the authorization to watch resource changes, therefore even if a keyRequest gets modified after its creation the operator will not react. This ensures consistency in secrets and avoids the risk of overwriting keys already returned.

To create a function that will be triggered after action on a resource an annotation should be added on top of it specifying the group, the version and the name of the resource; e.g. for the `keyreq_on_create` function:

```
@kopf.on.create('qks.controller', 'v1', 'keyrequests')
def keyreq_on_create(namespace, spec, body, name, **kwargs):
```

Authorization management

The operator must have access to the custom resources `saes` and `keyRequests` to watch their creation, to `event` to discover when a new object has been created and to `secrets` to retrieve SAEs credentials and to save retrieved keys. A basic implementation of the operator cluster role is reported here:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: qkd-operator
rules:
- apiGroups: ["qks.controller"]
  resources: ["keyrequests"]
  verbs: ["*"]
- apiGroups: ["qks.controller"]
```

```
    resources: ["saes"]
    verbs: ["*"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["*"]
  - apiGroups: [""]
    resources: ["secrets"]
    verbs: ["*"]
```

Image building

The code can be packaged in a Docker image using the Dockerfile provided in the operator folder. The operator code must be run not as a standard Python file, but through the following command:

```
kopf run /path/to/the/operator.py
```

which allows the *kopf* framework to run the code required to interact with Kubernetes and to load all the needed libraries, embedding the provided Python code into a controller loop.