F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

Design of a Key Management System Architecture for Quantum Key Distribution

Vojtěch Sobotka

May 2024
Supervisor: doc. Ing. Leoš Boháč, Ph.D.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Sobotka** Jméno: **Vojtěch** Osobní číslo: **507425**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Návrh architektury systému správy klíčů pro kvantovou distribuci klíčů

Název bakalářské práce anglicky:

Design of a Key Management System Architecture for Quantum Key Distribution

Pokyny pro vypracování:

Cílem této bakalářské práce je navrhnout komplexní architekturu systému pro správu kryptografických klíčů v sítích QKD a implementovat část tohoto systému. Student by měl provést analýzu současných přístupů ke správě kryptografických klíčů v QKD sítích, provést architektonický návrh KMS, a implementovat alespoň tyto části celého systému:
a) napojení na zařízení QKD přes protokol ETSI
b) zajištění správy lokální klíčů (bezpečné uložení, buffering, zobrazení stavu zásobníku)
c) zajištění generování klíčů preš důvěryhodné uzly
d) přehledný WEB rozhraní pro konfiguraci a řízení přenosu klíčů
e) možnost přenosu klíčů k třetím stranám (uživatelům)
Splnění zadání práce bude provedeno ověřením funkčností výše uvedených bodů.

Seznam doporučené literatury:

K. -S. Shim, Y. -h. Kim, I. Sohn, E. Lee, K. -i. Bae and W. Lee, "Design and Validation of Quantum Key Management System for Construction of KREONET Quantum Cryptography Communication," in Journal of Web Engineering, vol. 21, no. 5, pp. 1377-1417, July 2022
O. Shirko and S. Askar, "A Novel Security Survival Model for Quantum Key Distribution Networks Enabled by Software-Defined Networking," in IEEE Access, vol. 11, pp. 21641-21654, 2023
R. Kaewpuang et al., "Cooperative Resource Management in Quantum Key Distribution (QKD) Networks for Semantic Communication," in IEEE Internet of Things Journal, vol. 11, no. 3, pp. 4454-4469, 1 Feb.1, 2024

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Leoš Boháč, Ph.D. katedra telekomunikační techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **07.02.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

doc. Ing. Leoš Boháč, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgement

I would like to extend my sincere gratitude to all those who have supported me throughout the process of researching and writing this thesis.

I am deeply grateful to my supervisor, doc. Ing. Leoš Boháč, Ph.D., for providing an interesting topic, his time spent on consultations, and his invaluable feedback. I also thank Ing. Jiří Weiss for his helpful feedback and commentaries.

Special thanks to my family and girlfriend for their unwavering support and encouragement throughout this journey.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of the university thesis.

24 May 2024, Prague

....

Abstrakt / Abstract

Vzhledem k rozvoji kvantových počítačů se běžné metody používané v počítačové kryptografii, jako je RSA, stávají zranitelnými a potenciálně nebezpečnými k dalšímu používání. Kvantová distribuce klíčů (QKD) je alternativním přístupem ke kryptografii. Namísto asymetrické kryptografie se zaměřuje na náhodné šifrovací klíče neodhalitelné pro třetí stranu, která by chtěla komunikaci zachytit.

Tato bakalářská práce se zaměřuje na realizaci kvantové distribuce klíčů ve větším měřítku. Cílem práce je navrhnout a vytvořit proof of concept kvantové distribuční sítě klíčů.

V rámci přípravy implementace jsem prošel existující specifikace k tématu a definoval nutné komponenty ke správnému chodu sítě.

Závěrečná část mé práce je implementace s demonstrací základních principů a funkcionalit souvisejících s kvantovou distribucí klíčů.

Klíčová slova: Distribuce kvantového klíče, Software-defined networking, Java, Spring Boot, REST,

In the face of the development of quantum computers, common methods used in computer cryptography, like RSA, are becoming vulnerable and potentially not safe to use anymore.

Quantum key distribution (QKD) is an alternative approach to computer cryptography, where rather than focus on asymmetric cryptography, attention is turned to random key generation for symmetric cryptography, which should be more resilient against eavesdropping.

My thesis focuses on enabling Quantum key distribution on a larger scale. The thesis aims to design and create a proof of concept of the Quantum key distribution network.

To achieve this, I reviewed numerous existing specifications related to the topic and designed the interactions and security measures for the system components.

The final part of my thesis is an implementation with a demonstration of the basic principles and functionalities related to the quantum key distribution.

Keywords: Quantum key distribution, Software-defined networking, Java, Spring Boot, REST,

Contents /

1 Introduction **1**
1.1 Aims of the thesis 1
1.2 Why is the Quantum key distribution developed and used? 1
1.3 Basics of Quantum key distribution 2
1.4 Introduction to Quantum key distribution networks 3

2 Requirements for the QKD network **5**
2.1 Functionalities 5
2.1.1 Requirement: Identification of QKD-Secured Applications 5
2.1.2 Requirement: Network Registration and Monitoring Requirements for Trusted Nodes in QKD Networks 5
2.1.3 Requirement: Creation of key exchange session for the key relay 5
2.1.4 Requirement: Production of key on demand 6
2.2 Security requirements 6
2.2.1 Fulfilling of base IT security principles 6

3 Design of QKD SDN network **8**
3.1 Software-defined network general idea 8
3.2 Software-defined network for QKD networks 9
3.3 SD-QKD node 9
3.3.1 Node agent 9
3.4 Key management system 10
3.5 Network controller 10
3.5.1 Control plane service . . . 10
3.5.2 Secured application service 11
3.5.3 Authorisation manager . . 11

4 Design proposal of the Key management system **12**
4.1 Required functionalities for KMS 12
4.1.1 Key relay 12
4.1.2 Application key lifecycle inside of KMS 13
4.1.3 API for client application . 15
4.1.4 Control interface for Software-defined network . 15
4.1.5 Communication with QKD modules 15
4.2 Database of KMS 15

5 Proof of concept and implementation **17**
5.1 Minimal configuration components 17
5.2 Selection of technologies for implementation 18
5.2.1 Spring Boot 18
5.2.2 H2 database 18
5.2.3 Memory safety of Java . . . 19
5.2.4 Application deployment using Docker container 19
5.3 Database of the components . . 19
5.3.1 Implemented database of KMS 20
5.3.2 Implemented database of the controller 20
5.4 Communication between components 21
5.5 Declaration of non-specified communication between modules 22
5.5.1 API between trusted nodes for key relay 22
5.5.2 Control interface for routing setup by the SDN controller on the individual KMSs. 23
5.5.3 Communication from the trusted node to the KMS about routing 23
5.6 Sequence for creating QKD application session 23
5.7 Event-based programming on the side of the KMS 25
5.7.1 Problem of sequential key relay 26

5.7.2 Asynchronous key forwarding 26
5.7.3 Event-based key forwarding proposal 27
5.8 Implementation and code details 27
5.8.1 Reduction of boilerplate code 28
5.8.2 Declarative generation of API servers and clients . 28
5.8.3 Safe deletion of application keys 29
5.8.4 Own implementation of Dijkstra's algorithm for path finding 30
5.8.5 Implementation of manipulation of keys with the XOR function 31
5.9 Connection for the QKD secured application to the Trusted nodes 33
5.10 Connection of the KMS to QKD modules for encryption key fetching 34
5.11 Interface to manage the key relay 36

6 Quality assurance of developed components 39
6.1 Test levels 39

7 Demonstration of proof of concept 41
7.1 Configuration of a demonstration network 41
7.1.1 Operation with mocked QKD modules 42
7.1.2 Operation of the network . 42
7.2 Demonstration of the Key Relay 43
7.2.1 Testing scenario 43
7.2.2 Execution of demonstration 43
7.2.3 Description of logs of transfer node 46
7.3 Demonstration code 48

8 Conclusion 49
8.1 Thesis summary and contributions 49
8.2 Fulfilment of the requirements . 49
8.3 Future plans 50

References 51

A Abbreviations 53
A.1 Abbreviations 53

B Source code 54
B.1 Source code on the GitLab . . . 54

C Created API specifications 55
C.1 ETSI 014 API 55

/ Figures

1.1 Example of QKD module. 3
1.2 Scheme of quantum key distribution network. 4
4.1 Diagram of multihop key relay . 13
4.2 Lifecycle of the application keys inside of a KMS. 14
5.1 Scheme of components for proof of concept 17
5.2 Implemented database of the KMS 20
5.3 Implemented database of the SDN controller 20
5.4 Application components and their communication interfaces 21
5.5 Sequence diagram of QKD application session creation 24
5.6 Sequence of events inside of the KMS 27
5.7 Example of YAML OPEN API specification 28
5.8 Controller resolving the API for QKD secured applications . 34
5.9 Interface to fetch key by master 37
5.10 Interface to fetch key by slave . 37
5.11 Interface to open new QKD session 38
7.1 Diagram of QKD network demonstration 41
7.2 Request for opening the QKD application session 44
7.3 Response to request to open a new QKD application session . . 44
7.5 Response to request to fetch a key by server QKD secured application 45
7.6 Request and response for fetch key by ID for the client QKD secured application 46
7.7 Application log during session registration and forwarding 47
7.8 Log of deletion of key on BETA 48

Chapter 1
Introduction

The main aim of this work is the implementation of the key management system for the quantum key cryptography network. Quantum key distribution is now a popular topic, and it is constantly developing and becoming more used in industries where security is crucial.

There are already existing commercial vendors of the Quantum key distribution hardware that provide software solutions for the Quantum key distribution network. These solutions are often the trade secret of the producers, so we cannot simply see how their solution is implemented. This is a problem, for example, for system security certification because one can't trust the solution and does not know many details about implementation.

Hence, the main goal of my thesis is to develop a working prototype of a Quantum key cryptography Key management system. To achieve this goal, I followed these steps:

1. Gather a complete set of functional requirements for the QKD network.
2. Select the most suitable architecture for the QKD network.
3. Select the software components that will be part of the QKD network.
4. Design the core software components of the QKD network following existing specifications and industry standards.
5. Implement the proof of concept with the system's key components working together.

1.1 Aims of the thesis

The main tasks are related to the design and implementation of the QKD network and are defined by the following requirements (references to the corresponding sections with solutions are provided):

1. Establishing connections to QKD modules using ETSI protocols.
2. Managing keys locally.
3. Generating keys through trusted nodes.
4. Creating a web interface to control key generation.
5. Enabling key relay to third-party applications.

1.2 Why is the Quantum key distribution developed and used?

Quantum Key Distribution (QKD) is a solution to the threat that quantum computers pose to asymmetric cryptography. We can simplify the problem by stating that a quantum computer could compute the private key from the public key in real time. By this, the third party could decrypt intercepted communication encrypted by asymmetric cryptography like RSA.

RSA is based on the idea that the factorization of a large integer, which is a product of two large prime numbers, is very difficult and practically impossible within a reasonable time frame. As a result, asymmetric ciphers like RSA depend on the impracticality of determining the original prime numbers from the public key.

The development of quantum computers could threaten this. A well-known quantum algorithm is the Shor's algorithm. It was created by Peter Shor in 1994 as an example of an algorithm that could break the RSA [1]. However, to this day, it has not been proven on a large integer.

Two possible solutions to this threat are now considered [2]:

- *Post-quantum cryptography* - Development of algorithms focused on being resistant against quantum computers and related algorithms.
- *Quantum key distribution* - Method focused on the randomness of the key and maximizing the disability to derive the key by brute force attack.

1.3 Basics of Quantum key distribution

Quantum cryptography uses the laws of quantum physics to provide the highest possible level of security. The quantum cryptography addressed here is based on the superposition of quantum states and the probability of measuring certain states. The underlying fact is that by observing a particle, we cannot always find its original state. Moreover, the states are disturbed by the measurement itself, and with this knowledge, we can detect if a third party eavesdropped or intercepted the key.

My thesis focuses on the Quantum key distribution technology. As stated in its name, it provides secure generation of the key rather than encrypting the data, which must be sent secretly. When we use Quantum key distribution, we ensure that both sides get a random key that is not eavesdropped by a third party. The encryption of the secure communication is then the responsibility of the client side, typically comprising of the symmetric cryptosystem, e. g. Advanced Encryption Standard (AES), for secure data encryption.

The first protocol for Quantum key distribution was BB84, which was created in 1984 by Charles H. Bennett and Gilles Brassard [3]. This protocol encodes logical bits into a sequence of photons, which are polarized in 4 directions (two polarization states within one measurement basis) and sent from one module to another. The receiving module then attempts to measure bases of polarized photons and exchanges data about measurement bases using a public channel (not the sequence of photons itself). The random key, which is available for applications and enables data relay, is composed of the photons where both QKD modules agree on the basis.

Figure 1.1. Example of a QKD module, which is sold by company ID Quantique [4].

I would like to introduce a few essential terms for quantum key distribution.

Definition 1.1. *QKD module* — The Hardware module runs the QKD protocol connected to another QKD module. These modules exchange the keys using the QKD protocol.

Definition 1.2. *QKD link* — Connection of two QKD modules, which are connected and exchange the keys between themselves.

Definition 1.3. *Random key* — Truly random key generated by the QKD modules using the QKD protocol.

1.4 Introduction to Quantum key distribution networks

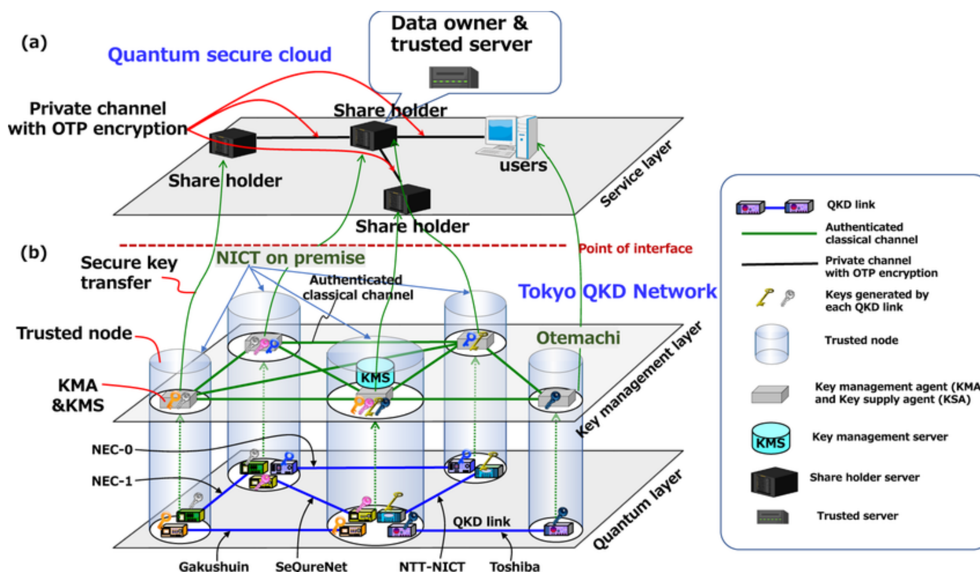

Figure 1.2. Scheme of quantum key distribution network [5].

One downside of the quantum key distribution is the limited distance on which the quantum link can operate correctly [6]. To fix this issue and enable the generation of the random key for longer distances, it is necessary to make a setup consisting of more quantum links.

However, because the quantum link always consists of a pair of QKD modules, it is necessary to combine multiple quantum links when generating a key between two modules that are not directly connected by a quantum link. This operation happens in a place called a trusted node. A trusted node has a secure environment to exchange keys between modules without any security risks that a standard connection would pose.

Definition 1.4. *Trusted node* — Secure location, where one or more *QKD modules* are located. It also serves as the start or end point of the whole key relay operation.

Definition 1.5. *QKD network* — Combination of connected *trusted nodes* by *QKD link*

A trusted node needs to store the received keys before they are sent to another trusted node. It is also necessary for the trusted node to have information about the routing of the keys. The component of the trusted node responsible for this is called a Key management system.

Definition 1.6. *Key management system* — Component of the trusted node responsible for storing and forwarding random keys.

Random keys are fetched from the trusted nodes by the clients. The clients always exist in pairs; one fetches the key from the start node and the other from the end node. The encryption of the data is the client's responsibility. The QKD network is responsible for delivering the secure random keys only.

Definition 1.7. *QKD secured application* — Entity connected to the trusted node, fetching the random keys from the QKD network.

Definition 1.8. *QKD application session* — Combination of two *QKD secured applications*that want to fetch the same *random keys* from the QKD network.

Chapter 2
Requirements for the QKD network

The first step of creating a functional QKD network is to establish the requirements for the design phase of development. These requirements should consider the needs of the clients, represented by the QKD-secured applications, which consume the keys from the network, as well as the operational aspects of the network. The network needs to have the key generation running, correctly handle error states, and notify about them. [7].

2.1 Functionalities

The basic functionality of the QKD network is to connect two QKD-secured applications and supply both with single-use keys. The keys should be transferred without interception. If intercepted, the affected part is excluded from the key. If the intercepted part is too significant, the generation is considered unsuccessful, and both sides acknowledge that.

2.1.1 Requirement: Identification of QKD-Secured Applications

The network must be able to identify and authenticate QKD-secured applications to ensure their identity is accurately verified. Each QKD-secured application must have a unique identifier within the network to facilitate this identification and authentication process.

2.1.2 Requirement: Network Registration and Monitoring Requirements for Trusted Nodes in QKD Networks

The trusted node must be registered within the network and provide information about connections to other trusted nodes. The trusted node must be able to notify the network about closed or malfunctioning connections between the node and the other nodes. The trusted node must also inform the network about its own malfunctions or any malfunction inits QKD modules. Additional information, e.g., the relay capacity of the node, the bandwidth, and the availability of keys within the node, should also be provided to the network.

2.1.3 Requirement: Creation of key exchange session for the key relay

The network must be able to provide a key exchange session for two QKD-secured applications connected to different trusted nodes. These two applications must be verified, as mentioned in the first requirement. Session creation includes finding a path within the network and notifying all trusted nodes within the path about creating the new session. In case of an error occurring in some of the nodes, the route must be replanned, or the applications on both sides must be notified about the unavailability of the desired session. The session must also be correctly closed, and all nodes along the path must be informed to increase capacity.

2.1.4 Requirement: Production of key on demand

When a QKD application requests to supply the key for both sides of an open session, the key must be fetched from a QKD module, or a new key has to be generated using the single-use key generator. Then, the key has to be transmitted to the trusted node connected to the other QKD-secured application. If there is a problem during the transfer, a new key can be resent via a new route, or both applications must be notified that the key relay is unavailable.

2.2 Security requirements

Due to the highly secure nature of the QKD, we need to ensure that the network is protected from any interference. Security must also be ensured for the hardware, which is composed mainly of QKD modules, servers providing the network control, and the software running inside the servers, acting as a network control layer [5].

- **Physical security** - The hardware must be located in secured locations with restricted access for authorized personnel only to protect unauthorized access to network equipment.
- **Security of deployed services** - All running services should be locked inside containers to restrict access to their interfaces. All secrets or certificates of confidential character should be located inside an external key store, to which the service will connect on startup.
- **Security of communication between services** - All control layer communication between the services is vulnerable and needs to be secured. Also, the author of the message must always be verified to prevent false communication towards any network component. Another critical point is the transfer of the random key between the trusted node and the QKD-secured application. The safest option for this is a physical connection within a secured location. Still, the possibility of sending the key through a public channel with appropriate encryption must be considered.

The purpose of the quantum key distribution is to replace asymmetric cryptography. Therefore, we need to use single-use keys for communication.

2.2.1 Fulfilling of base IT security principles

The so-called CIA triad, consisting of confidentiality, integrity, and availability, is one of the basic cryptography models [8].

- **Confidentiality** - All data should be available for authorized entities only. The decrypted data should be inaccessible to others. The network must supply keys only to secured applications inside the session, and the keys must be deleted securely after finishing the generation. When a key or its part is intercepted during a transfer, the intercepted part is excluded if possible, or the whole key is flagged as not confidential and discarded. The network supplies the keys with the highest level of security. Confidentiality of client application data is in their scope of control.
- **Integrity** - The data is trustworthy and free from tampering. We can ensure that the keys coming from the network were only changed by the QKD modules if there was a reason to do so.
- **Availability** - The service of key generation must always be available. This is a critical point and the hardest to fulfill. The QKD network needs to be protected from attacks that can disable its ability to provide keys. This is done on the side

of application interfaces by authorizing the secured applications. On the quantum layer of the relay, there may be situations where a high interception rate results in an insufficient number of generated keys. Additionally, some nodes or modules may malfunction, and due to the network topology, the network graph may become disconnected, preventing the relay of keys between certain points.

Chapter 3
Design of QKD SDN network

This chapter aims to use the requirements gathered in the previous chapter to design the QKD network, specifying the components and their functionalities.

The chosen architecture for this network is a software-defined network (SDN). This decision is supported by the existence of relevant ETSI protocols, which provide architectural concepts and even communication schemes for the network components [9] [10].

3.1 Software-defined network general idea

Software-defined network is a common type of network architecture [11]. The core idea of software-defined networking is splitting data flow delivery and routing information. These things are split into two separate planes (a plane is a part of the architecture separated from the other parts of the system). We called these planes the Data Plane, responsible for data delivery, and the Control plane, responsible for network topology and routing tables. Structurally, a Software-defined network consists of three components.

- **SDN Controller** - The Controller is a central component of the software-defined network. It manages the control plane centrally in opposition to traditional solutions like switches.
  - The *Northbound interface* of the controller provides API for SDN applications that want to relay data through the network controlled by the controller.
  - The *Southbound interface* transfers data between the controller and the networking devices. It is used to manage the control plane of the network.
- **SDN networking device** - The networking device forwards data and processes the data plane.
- **SDN Application** - The application is a client entity that sends data through the SDN network. It sends API requests to the SDN controller to relay data.

3.2 Software-defined network for QKD networks

Quantum key distribution already has a functional solution for the software-defined network. The Swiss company ID Quantique offers solutions based on the SDN [12]. There is also an existing ETSI standard for the Software-defined network in Quantum key distribution. These existing projects supported my choice to use the Software-defined network as the architecture for this project.

3.3 SD-QKD node

A software-defined enabled QKD node represents a location within the network where one or more QKD modules are located. The node is connected to other SD-QKD nodes using its KMS modules. The Key management system serves as a gateway between these modules and provides an interface for the Node agent. It also provides the interface for the QKD-secured applications to pull the keys. The node agent provides control communication with the SDN.

The node is physically located in a secured location. The infrastructure of the node will most likely be located in one rack.

3.3.1 Node agent

The *Node agent* of a node is a service responsible for communication with the SDN controller. Its main purpose is to create an abstraction of a single communication interface that represents multiple QKD modules. As a central point for the communication of the QKD node, it is also responsible for forwarding messages from the KMS to the controller. The messages inform about creating or managing sessions and recognizing the secured applications.

The node agent should provide the following functions:

- **Initialization** - On the first node startup, the node sends a heartbeat to the (predefined) location of the controller, obtains the security credentials, and then sends information about its topology (relations with other nodes). Technical parameters essential for routing, like bandwidth or relay capacity, are also sent.
- **Health checks** - The node regularly transmits heartbeats to report its current operational status, which can be categorized as UP or DOWN. This mechanism is primarily designed to monitor the node agent and the key management system. Additionally, these updates help in assessing and sorting the operational states of various QKD modules.
- **Updates** - The node is able to update information about the QKD modules and their linked counterparts. If a link is not active, the controller should be notified. It is also possible to send information about, e.g., the storage of the keys, as an update to enhance routing.
- **Acknowledgements** - The node can react to the network's controller orchestration. This typically involves information about receiving and registering sessions between two applications. The information about the remaining capacity of the node should be sent as part of acknowledgments to support the load-balancing of the network.

3.4 Key management system

The *key management system* facilitates communication between the QKD modules and supplies the keys to the secured applications. It represents the network's data plane responsible for forwarding the keys.

It must provide the following functions:

- **Creating sessions** - the KMS should be able to forward a message from the QKD-secured application. The message contains a request to open a new QKD application session. The message is forwarded to the node agent and then to the controller.
- **Registering sessions running through node** - The KMS will be informed about opening a new session using the node. The message will typically consist of the ID of the session and information about the previous and next node.
- **Providing key fetching for QKD-secured application** - Upon request from an application, the key is fetched from the KMS via a selected link. This key is then assigned to the open QKD application session. Once the application receives the key at the link's endpoint, it will temporarily store the key, allowing it to be accessed before it is discarded after a predetermined period.

3.5 Network controller

The SDN controller is the centerpiece of the QKD Software-defined network. It incorporates standard SDN functionalities, enhanced with specific QKD-related features. The first key functionality registers any incoming QKD application, authorizes it to use the network, and provides credentials for secure communication with network components. This is done by the Secured application service and the Authorization service.

The second functionality is the management of the network topology itself. The controller must have a real-time database of working nodes and their connections and manage the requests for new sessions. The controller is the main point of failure of the network, making it the most vulnerable part of the network. It needs to be properly secured and have redundant computation power to ensure the highest availability of the network.

3.5.1 Control plane service

The network's main service is in charge of running the control plane of the session. All node agents must be able to connect to this service. The service must have the following functions:

- **Registering SDN-QKD nodes** - The service must be able to register the node to the network. This means as the node finishes its initialization at a startup, the first heartbeat or the discovery call to register itself to the network. After that, he will receive its unique node ID and can participate within the relays in the network.
- **Healtchecks and updates of the QKD links** - The service must maintain the actual topology of the network. While an SDN-QKD node agent is running, the state of its QKD modules or their counterparts can change. The service must, therefore, perform period health checks, including the QKD modules' states.
- **Creating sessions** - When there is a request from two QKD-secured applications to open a key session between them, their existence and configuration are first

checked in the Secured applications service. Once verified, the route is calculated, and the nodes along the path are notified.

Certain parameters need to be taken into consideration when computing a path. It is necessary to focus on particular specifications of the keys, such as the bandwidth, and the network specifications, e.g., the relay capacity of the network.

3.5.2 Secured application service

This service is responsible for recognizing the QKD-secured applications and enabling them to use the network. It should contain the database of registered secured applications, and in an attempt to connect, it should provide them with their unique application ID. There is also an interface for the network administrator, which enables registering more secured applications into the network or removing them.

If more organizations use the network for key relay, possibly as QAAS (QKD as a service) [13], this service will be extended to provide usage statistics and, for example, services such as billing.

3.5.3 Authorisation manager

The authorization manager is part of the network controller. However, it should be physically separate from the other two components since it does not usually communicate with them directly. This service provides credentials based on the certificate to connect to network components.

Chapter 4
Design proposal of the Key management system

The Key management system is an orchestrating component of the SDN-QKD node. It is responsible for manipulating the keys from QKD modules and delivering them to QKD-secured applications.

In this chapter, I would like to present a more precise design of the KMS.[1]

4.1 Required functionalities for KMS

To design the network, I need to create a set of functionalities to be implemented in my KMS. I have based these functionalities on the requirements formulated in the Requirements chapter 2, ensuring they align with the ETSI QKD standards and requirements.[9] [10].

- **Key relay** - The so-called multihop or key forwarding is a core function of the KMS, providing the path for synchronizing the key for two QKD-secured applications.
- **Key management** - Disclosure of the keys is the worst-case scenario for the whole QKD SDN network. Therefore, the keys must be stored securely within the KMS. They must also be safely deleted when not used for a certain period. Key information is also stored within the KMS, such as whether the key was synchronized or delivered to the final trusted node.
- **Database synchronization** - Due to the nature of the network, KMS is also a distributed database. It is necessary to ensure that the keys and linked data shared with the connected trusted nodes are equivalent, especially the data about delivery, applications, states, and deletion.
- **Quality of service management** regarding the availability of the network and the node usage balancing - The KMS should provide information about the usage of available keys to enhance route planning. Additional services such as logging should also be implemented.

4.1.1 Key relay

Definition 4.1. *Application key* — A random key to be delivered to a QKD-secured application.

Definition 4.2. *Encryption key* — A random key to be used to encrypt an *Application key* when the key is sent between trusted nodes.

A key relay, or so-called multihop, is needed when the application key path contains more than two trusted nodes.[2] To create a multihop, it is necessary to use at least

[1] Related literature also uses the term Q-KMS for KMS in QKD.

[2] Two trusted nodes are the minimal QKD configuration containing just one combination of connected QKD modules.

one node as a transfer point. The following scheme is applied to secure key transfer across multiple nodes.

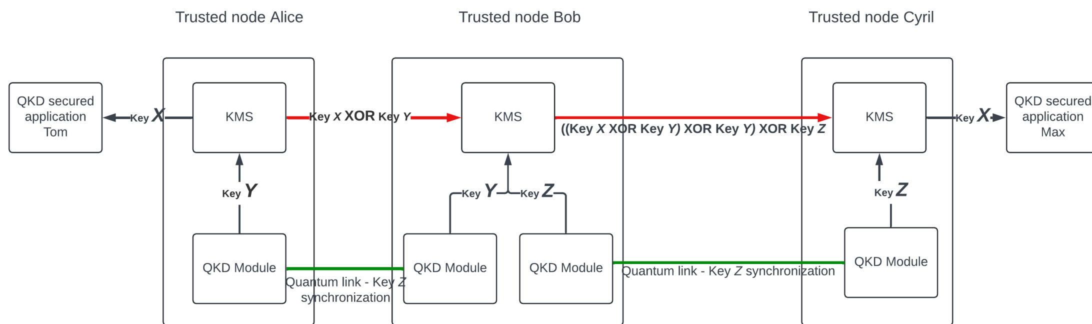

Figure 4.1. Example of multihop key relay

The picture presents a typical multihop relay inside the network, where secured applications Tom and Max want to fetch an application key X.

The fetching process is as follows:

1. The relay will start in a trusted node, Alice, connected to the QKD-secured application Tom. The SDN controller had already set up the path, and all nodes along the path have correct information in their routing tables.
2. Application key X in trusted node Alice is obtained either from the QKD module, a random key generator connected directly to Alice's KMS, or delivered by QKD secured application Tom.
3. The key X is sent to the trusted node Bob. To do this, Alice's KMS obtains the synchronized application key X from the QKD module, directly linked to the trusted node Bob. Alice's KMS encrypts the application key X using the obtained encryption key Y with the XOR function and sends it to Bob's KMS.
4. When Bob's KMS receives the message with the key encrypted by XOR, the KMS obtains the key Y from its QKD module linked to Alice using the key ID in message metadata and decrypts the application key X using the XOR function with the encryption key Y.
5. Bob's KMS then needs to send the application key X to the trusted node Cyril. To do this, Bob's KMS obtains the encryption key Z from the QKD module linked to Cyril, just as Alice did.
6. Bob's KMS then sends the key, which is a combination of the previously decrypted application key X and the newly obtained encryption key Z synchronized with Cyril, all encrypted by XOR, to Cyril's KMS.
7. When Cyril's KMS receives the encrypted key from Bob, it retrieves the corresponding encryption key Z from its QKD module. Cyril's KMS then applies the XOR function to the encryption key Z to decrypt the original application key X from the message.
8. Then Cyril notifies Alice about the successful relay. The application key X is marked as synchronized and available to be fetched by both QKD-secured applications of Tom and Max.

4.1.2 Application key lifecycle inside of KMS

Two types of random keys are used to relay random keys for different purposes.

- **Application keys** for QKD-secured application key delivery - Random keys are ultimately used by the QKD-secured application for encryption. These keys are relayed across the network, and encrypted using the XOR function with encryption keys when passing between trusted nodes. Application keys can either be obtained from the QKD modules or delivered using the QKD-secured application.
- **Encryption keys** - Used by the QKD-secured applications for the XOR encryption of application keys relayed between trusted nodes. Encryption keys are used for the XOR encryption between trusted nodes. These synchronized encryption keys are always fetched from the linked QKD modules related to the two trusted nodes, directly linked to perform the XOR function for encryption and decryption of the application key relayed across the network.

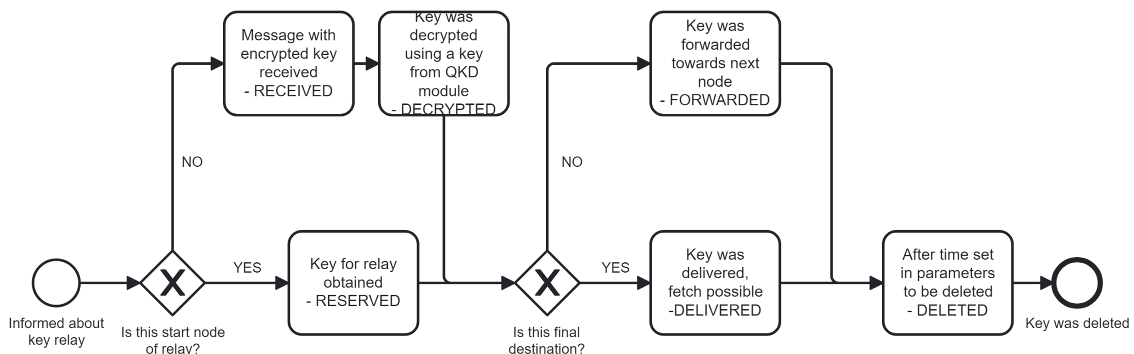

Figure 4.2. Diagram of the application keys states inside of KMS

The core functionality of the KMS is to store keys in a database. The states of these keys are determined by the flow of incoming messages. The procedures described in the previous section are executed with the keys to manage the network and ultimately deliver keys to QKD-secured applications. Throughout these operations, the keys transition through various states, as illustrated in the sequence flow diagram shown in the picture.

The sequence begins when the KMS is informed about a key relay. This notification can come from either a QKD-secured application requesting a key fetch or from a trusted node, which is the preceding node in the path, requesting the KMS to send an encrypted key for relay.

The possible states shown in the diagram are explained below:

- **RESERVED** - Used when a new key is requested to be transmitted. The key for the relay can be obtained from one of the QKD modules or delivered by the application requesting the relay. The key is reserved when assigned to some session and is to be transmitted soon.
- **RECEIVED** - Used when the encrypted key has been received in a message from the preceding node in the path. Then, the encrypted key from a recognized session is stored in a database.
- **DECRYPTED** - Used when the encrypted key has been received in a message from the preceding node in the path. The encryption key used by the preceding node in the path is fetched and used to decrypt the application key from the preceding node using the XOR function.

  The encryption key is fetched from the QKD module related to the preceding node and is based on the key ID provided by the incoming message metadata.

- **FORWARDED**- Used when the current node is not the final node and the key must be forwarded. Forwarded means the key was successfully sent to the next trusted node in the path. The forwarded key is sent encrypted using the XOR function with a random key generated using the QKD modules. The information about the ID of an encryption key for decryption is also sent in the metadata.
- **DELIVERED** - Used when the node is the destination. The key is stored inside the KMS and ready to be fetched to the QKD-secured application.
- **DELETED** - After the key has been stored in a database for the time specified in the session metadata, the key is deleted from the database. Periodic checks of the database typically do this. This is essential for the system security.

4.1.3 API for client application

Client applications that connect to the key management system in order to consume the key typically use the ETSI QKD 014 specification [9]. This specification uses the REST API-based key delivery with a mutual TLS handshake to secure the key delivery API.

I created a standardized OpenAPI YAML file representing a computer-readable API specification. This specification allows me to generate the API client without the need to specify the endpoints manually and create the API using one of the numerous available OpenAPI libraries.

4.1.4 Control interface for Software-defined network

The ETSI GS QKD 015 [10] specification defines an interface for controlling the network by the SDN controller.

4.1.5 Communication with QKD modules

The original ETSI-specified API for communication with QKD modules was the ETSI 04 [14]. However, it has since been simplified into ETSI 014 [9], which now serves as the standardized API for connecting with QKD modules. In my work, this API is utilized for communication from the KMS towards the QKD module and follows a standard REST architecture, using HTTPS and JSON objects for communication. The QKD modules require the mTLS authentication, ensuring that both sides authenticate themselves.

4.2 Database of KMS

The KMS database has to store two different types of content or two different entities. The data is suitable for traditional SQL databases since it can be stored in rows.

The first table is the routing table. It contains the following information to enable key forwarding.

- **Application ID** - The ID of an opened QKD session assigned by the SDN controller. This value serves as the primary key.
- **Previous node identifier** - Identifier of the preceding trusted node in the path. Set to null if the node is the first node in the path.
- **Next node identifier** - Identifier of the next trusted node in the path. Set to null if the node is the destination node.
- **Identifier of QKD secured application A** - Identifier of the application connected to the first node.
- **Identifier of QKD secured application B** - Identifier of the application connected to the last node.
- **Additional metadata about the connection** - Additional parameters regarding the open session, e.g., the key bandwidth.

The second core entity is the application key stored in the database. The keys can be in various states, as described in [4.1.2].

The application key table must have the following properties:

- **Key ID** - Unique key identifier
- **Key material** - The application key itself, encoded to String.
- **Assigned application ID** - ID of the application session to which the key is assigned. If null, the key is not assigned to any session.
- **Timestamp delivered** - Date and time of the key delivery to KMS
- **Timestamp last update** - Information about last updates or fetches from KMS, essential for deleting the keys.
- **Additional Metadata** - Information about the key parameters or, e.g., the time required to delete the key.

Chapter 5
Proof of concept and implementation

The primary goal of my thesis is to create and test the Key management system for the QKD network. In this chapter, I want to show the minimal configuration for the software-defined quantum key distribution network. To do this, I will specify minimal components and related communication interfaces, which must rely on ETSI specifications [9] [10].

5.1 Minimal configuration components

When designing a minimal proof of concept implementation, excluding components necessary for a larger-scale network only is important. However, the design must still meet the ETSI specifications and ensure that components are replaceable. This way, future replacements can be made without disrupting the functionality of the remaining system components.

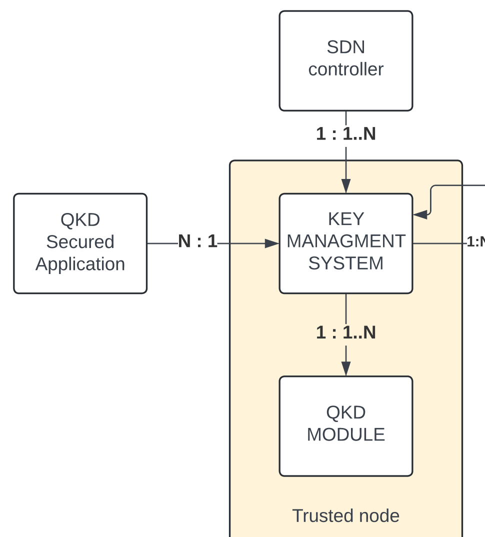

Figure 5.1. Diagram of components needed for minimal proof of concept implementation with multiplicities of component relations.

The diagram illustrates the components needed for the minimal implementation. I define a component as a part of the system that is implemented separately, communicating with others through a communication interface. Some components mentioned in the previous chapters are omitted because they are unnecessary for the proof of concept.

Most of the reductions had to be made on the controller side of the project. The authentication manager is replaced with pre-shared keys and certificates inside components delivered before deployment. While a more dynamic security approach would

be essential for a larger-scale network, it would significantly increase the complexity of the proof of concept. Instead, a simpler yet fully functional and secure solution is used. Also, the central service with a QKD-secured application is omitted in my demonstration. The application would most likely be a REST client with additional authorization.

The last reduction is a full-sized trusted node agent, which is combined with a key management system to provide a single point of communication of the trusted node for the SDN controller.

The diagram in Figure 5.1 shows the components with the multiplicities of the dependent components:

- **SDN controller** — This component must be implemented.
- **Key management system** — This component must be implemented. It includes the functionality of the node agent as well.
- **QKD secured application** — An abstraction of an application or entity fetching application keys from the network. For my demonstration, I use a REST API client (Insomnia or similar) with a prepared workspace to fetch the keys, mimicking the behavior of a secured application
- **QKD module** — A Simulation of a QKD module is used.

5.2 Selection of technologies for implementation

To implement the key management system and the software-defined network controller, I used a microservice architecture [15] because it is ideal for creating small services with a standardized API. Another reason for choosing a microservice architecture is modularity. I need the whole system to be modular so that new modules can be implemented or the old ones can be replaced.

5.2.1 Spring Boot

I decided to use Spring Boot, a Java framework providing easy creation of microservices. Spring Boot is an extension of the Spring framework [16], a Java application framework with the intervention of a control container. Spring Boot is ideal for this use case since it provides numerous libraries for working REST APIs and enhanced security libraries, offering many options for desired security levels, such as mTLS (Spring Security). Also, it facilitates database manipulation, as running queries is available through Spring JPA.

I will use the latest version of Spring 3 combined with Java 21 to provide an application with long-term software support and good security of used frameworks and other components.

5.2.2 H2 database

The H2 database is selected for use in the KMS. H2 is an in-memory database, meaning that data persist only in the application memory and not anywhere else [17]. The in-memory database is ideal for the KMS because, with each system restart, all data becomes outdated, requiring the retrieval of new keys and related information from scratch. Since this database exists only in the memory of the running program (as an embedded database) inside the Docker container, it does not require encryption, as it is not exposed externally outside the runtime environment.

5.2.3 Memory safety of Java

Java has a managed memory, preventing direct memory control. This can pose problems with safe deallocation and immediate disposal of private data stored in memory (mainly application and encryption keys).

While new objects are created using the *new* keyword, the dereferenced objects and no longer needed variables are managed by the garbage collector [18], which disposes data and cleans memory. Especially with Spring JPA that uses Hibernate to manage data, we could experience problems with data structures still existing in memory after being dereferenced in code.

The problem arises mainly because we cannot directly control data deletion from memory. On the other hand, we can call the garbage collector from the code. To destroy and deallocate unused data structures, we call *System.gc()*. This command instructs the Java runtime environment[1] to clear memory. Deallocation is in its power and cannot be directly manipulated.

I combine two approaches to prevent this vulnerability:

- **Safe value deletion** - I overwrite all sensitive values (application and encryption keys) in my code with nulls before deletion and then carefully dereference the sensitive objects. Then, I instruct the database to perform the operations immediately using the *flush* function. In the case of an in-memory database, this is still under the control of runtime.
- **Other safety measures** - These include locating the trusted node in a safe and protected environment and running the software safely inside the docker container, with access permitted to only API endpoints and additional external monitoring services (Prometheus, logs, etc.).

5.2.4 Application deployment using Docker container

Application deployment is done using Docker. On initialization, the application should connect to the key store of the system using AES. A key store is a secure place that provides storage for all the secrets (certificates and keys for service-to-service authorization) needed to operate the network. The KMS should be set up with a file including the locations of the node's QKD modules and the location of the pre-shared secrets to communicate with them.

5.3 Database of the components

Each component has its own database. The databases are managed by the Spring JPA framework, which utilizes the Hibernate framework. With these technologies, it is possible to declare database queries without writing SQL. Writing custom SQL queries is still possible and even necessary when the need for a more complicated database query arises.

The data used for demonstration (network topology and QKD-secured applications) are added to the database after the Spring Boot application is successfully initialized, using the interface *ApplicationRunner*.

[1] Different virtual machines exist to execute Java programs. I am using GraalVM.

5.3.1 Implemented database of KMS

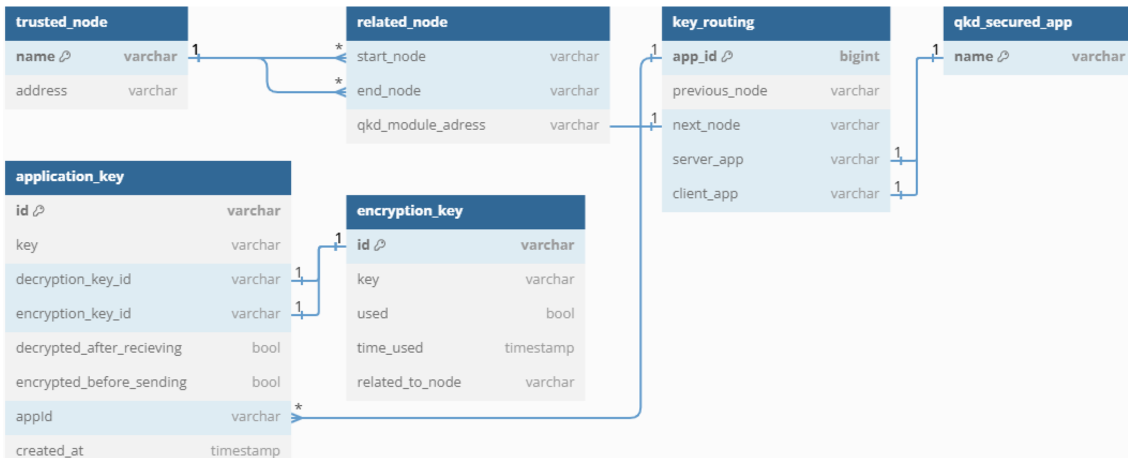

Figure 5.2. Diagram illustrating the structure of the KMS database

The main task of the KMS is to generate the application keys. A simplified database (illustrated in Figure 5.3.1) stores the application keys along with all supporting information needed for the key relay.

The table *key_routing* stores the information about opened QKD application sessions, delivered by the controller. The *app_id* value is used to identify related QKD applications. The rest of the properties are analogically related to nodes and QKD-secured applications.

Application keys in table *application_key* are stored using *UTF-8* encoding. Each application key inside the database holds the reference to related encryption keys. Correct encryption keys are identified using information about related sessions and nodes.[2] Table *encryption_key* serves as an abstraction for the QKD modules.

5.3.2 Implemented database of the controller

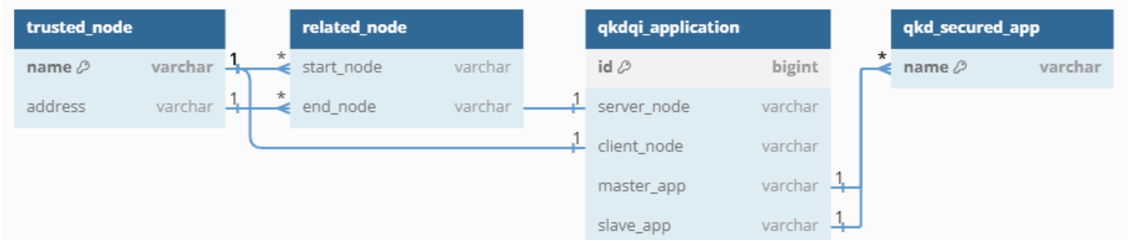

Figure 5.3. Diagram shows the structure of the SDN controller database

The controller's database contains data needed to set up the multi-hop path. The data about trusted nodes and network topology is the main content of the controller database. This data is predominantly used to determine the path of the key relay.

The essential entity of the database is the *qkdqi_application*[3], which is equivalent to the QKD application session. It stores all crucial information about the QKD application session. I have omitted more detailed settings to keep a simple proof of concept.

[2] Key routing will identify the right QKD module from which the encryption key will be obtained.
[3] I borrowed the name from ETSI 015 [10], which sometimes uses different terminology than other literature regarding QKD topic [11]).

A vital part of this database is generating the unique identifiers of the QKD application session. The database generator generates the Application ID. The ID is then distributed along with the routing information.

5.4 Communication between components

The components of the system must interact with each other. The network components (both hardware and software) run separately, so we must provide the following communication types:

- **Communication with routing information** from the SDN controller towards the trusted nodes.
- **Messages from QKD-secured applications** towards the KMS and forwarded to the SDN controller with requests to open QKD application sessions.
- **Information about trusted nodes** and their connection to the SDN controller is needed to provide actual information about the livability and topology of the network.
- **Communication with key material** between components of the system. This group exchanges application keys between trusted nodes and QKD-secured applications, and KMS fetches the encryption keys from related QKD modules.

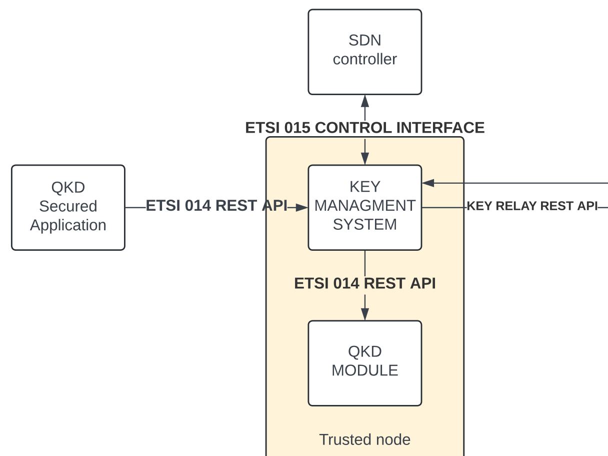

Figure 5.4. Diagram of components of the application and their communication interfaces

The diagram shows the communication between components, along with specific communication interfaces:

- **ETSI QKD 014 REST API** — Fully specified communication interface in the format of REST API, used for fetching and requesting keys by both QKD-secured applications to fetch the application keys from the KMS as well as for the KMS tp fetch the encryption keys from the QKD modules.
- **ETSI QKD 015 control communication interface** [10] — Control interface for the SDN controller for communication towards the trusted nodes. Also, communication regarding the states of the KMS and QKD modules is included in this specification. Specification is made using the *yang* model of an object holding

the necessary data and messages, suggested in a sequence diagram. This interface needs to be more specified for implementation and will be the subject of the next section.

- **KEY RELAY API** — Key relay API is an interface forwarding encrypted application keys between the trusted nodes' key management systems. This communication interface was not described in any material I could find, so I had to design it myself.

5.5 Declaration of non-specified communication between modules

Some communication interfaces are not specified in standards or literature known to me. I had to declare the following interfaces myself:

- Between the trusted nodes while exchanging the key during a multi-hop relay.
- The API provided by trusted node KMS to specify routing within the QKD modules inside the trusted node.
- The API hosted by the SDN controller to register new sessions by the trusted nodes. The trusted node forwards this information from the QKD-secured application that wants to open a new session.

I chose to use a standard REST API because it is suitable for service-to-service communication, which is also my case. Although a GraphQL API could be beneficial due to its ability to selectively retrieve specific data, its implementation requires a more complex library for both the client and server sides.

In my implementations of REST API clients and servers, I decided to use the declarative approach. Therefore, I always need to prepare OPEN API specifications for the REST API containing the information about paths in URL, available queries, and the related data structures needed for the REST API (request and responses bodied). I generate the API clients and servers using these specifications. This approach helps to keep the code shorter and more readable because all API-related files are kept outside the source code as the generated files.

5.5.1 API between trusted nodes for key relay

As defined in the section related to multihop key relays relay, trusted nodes KMS have to communicate directly with each other using encrypted keys. I did not find any standard or specification for this communication, so it is necessary to create the communication format.

The messages sent from one KMS to another must contain the application key, encrypted with the XOR function using the encryption key. The message between two KMSs with the encrypted application key must contain the identification of the used encryption key.

The REST API for the key relay consists of just one method, which is the *POST* message with a container in the request body with the following content:

- **Encrypted application key** — In string format and encoded by *UTF-8* in my specific case
- **Application key ID** — A unique identification of the application key must be the same in all trusted nodes along the path of the specific key relay.

- **Encryption key ID** — Unique identification of the encryption key. The encryption key ID is relevant to the sender and receiver KMS and related QKD modules.
- **QKD application ID** — Identification of the QKD application session between two QKD-secured applications related to the relayed key.

5.5.2 Control interface for routing setup by the SDN controller on the individual KMSs.

Routing messages are used to set up the path for the QKD application session in each trusted node along the key relay path. These messages are sent after the SDN controller gets a request from the QKD-secured server and client applications. The routing message has to contain the following properties:

- **QKD application session ID** — Unique identification of the QKD application session assigned by the SDN controller.
- **Previous trusted node ID** — Unique identification of the previous trusted node in the key relay path. The previous trusted node is the one from which the incoming key relay will originate. If this identifier is null, it indicates that the receiver of the message is the first trusted node in the path.
- **Next trusted node ID** — Unique identification of the next trusted node in the key relay path. The next trusted node is the one to which the incoming key is forwarded. If null, it means that the receiver of the message is the last trusted node in the path.
- **QKD application object** — Object with all necessary details about the QKD application.

This message container is distributed from the SDN controller towards each node contained in a particular multi-hop key relay path. The container is sent by classical *POST* request. The answer is a simple *OK* if the routing was added to the table. When the SDN controller gets *OK* from all involved trusted nodes, the server marks the QKD application as operational.

5.5.3 Communication from the trusted node to the KMS about routing

ETSI 015 [10] specification presents a relatively large scale model designed for a much bigger scale network than the QKD network used for testing proof of concept. I have created an ETSI 015 *yaml* specification describing the models required for large-scale networks. This API is implemented to exchange the topology data between the KMS and the SDN controller.

For the proof of concept, I have created a small repository containing all related data to create the path, overriding the KMS to control messages about topology. However, the full-scale implementation of ETSI 015 was done to enable the project to be updated if more complex information about topology is needed.

5.6 Sequence for creating QKD application session

Creating a QKD application session is a complicated sequence of multiple tasks spread over multiple divided system components. I created a sequence diagram [5.5] of the messages between components.

Creating the QKD application session always starts with invoking the QKD-secured applications that want to obtain the QKD application session ID, which they then use to fetch the application keys from related application modules.

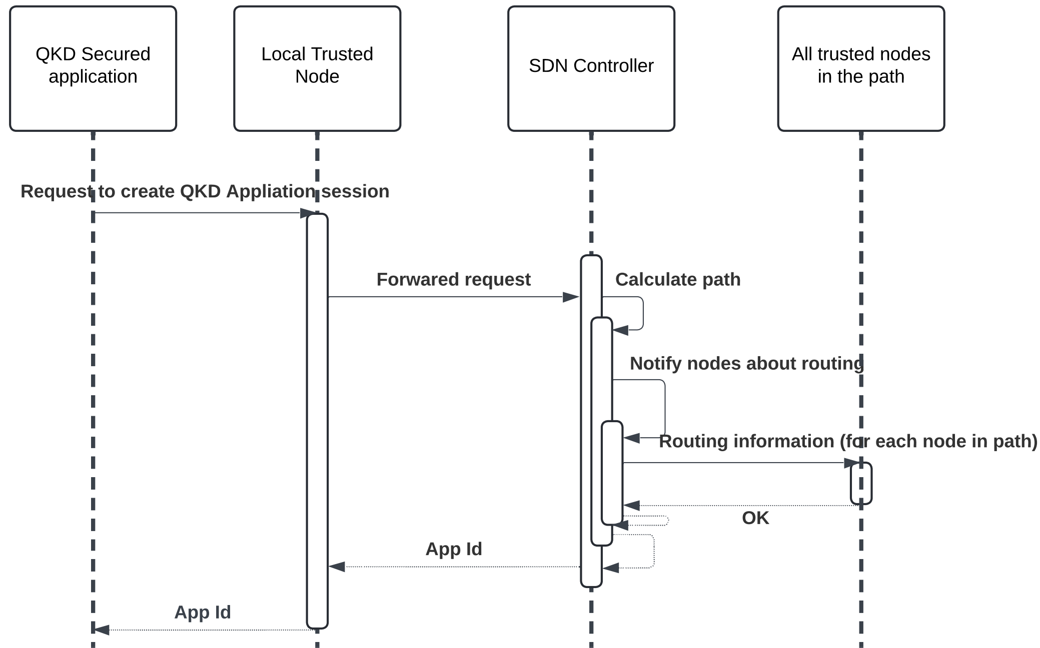

Figure 5.5. Sequence diagram of QKD application session creation

Explanation of the sequence elements shown in the diagram:

- **QKD secured application** — The Application initializes the request for a QKD application session. By ETSI 015 [10], two applications are needed to open the QKD application session. However, the ETSI 014 [9] does not mention the session opening. So, I decided that the session opening would be done by an external API on the SDN controller or trusted node.
- **Local trusted node to secure application** — The node from which the QKD-secured application wants to fetch keys after the session is initialized. In the session creation sequence, it is responsible for resending requests to the SDN controller.
- **SDN controller**— Takes this sequence's biggest portion of tasks. First, it must validate all entities' existence in the request. Then, it needs to find and set the path for the key relay and finally return the information about the opened session.
- **All trusted nodes located in the path** — Nodes included in the path, including the local nodes for the QKD-secured application, must set their routing tables to forward keys in the right direction.

My main concern regarding the session creation sequence is the length of the sequence, especially for the QKD-secured applications. It could take a relatively long time to differentiate from normal timeouts while retrieving the QKD application session ID using REST communication. Solutions to this could be the following:

- **Splitting the original request** into multiple requests. The first request contains data required to open a QKD application session. The sender is a QKD-secured application. Response to this request is a simple *ACCEPTED*, meaning the request

was received and the path is now being created. When the path is created, the app_id is sent to the QKD-secured application that requested the session, meaning that the session is ready.

- **Using parallelism** when it is suitable. The exact use case of this in the session creation sequence is to send the routing messages toward trusted nodes in parallel. The procedure, as proposed earlier, uses the normal sequential requests from the controller towards the nodes, which can take a long time. Processing the request inside the node is fast, but creating, sending, and receiving the HTTP request takes much longer. Therefore, sending the individual routing information could be done in parallel using the Spring Async technology, which could simplify the process quite a bit.

Finally, I describe the whole sequence 5.5 step-by-step:

1. The QKD application sends the request to the trusted node. The request contains specifications of the requested session to be opened. The request must contain the names of both QKD-secured applications and the nodes at the start and end of the path (Server and client nodes).
2. The request from the QKD-secured application is forwarded by the trusted node only. Additionally, depending on the implementation of the network, the trusted node can verify and check the settings of the secured application.
3. SDN controller checks if all entities mentioned in the request exist. Otherwise, it throws an exception with the specification of the missing entity.

   Some additional checks of the application settings and ordered services could be done.
4. The following step is searching for a path within the network. The *Dijkstra's algorithm* is used for this task. In this context, the graph nodes are represented by the trusted nodes, and the graph edges correspond to the QKD links connecting the nodes. The weight of all edges is set to 1. The result of this stage is a sequence of nodes that form the path.
5. Then, trusted nodes along the path are informed about the new routing. It is necessary to provide information about the session, including technical information, session identification, previous node, and most importantly, the next node in the path. The distribution of the routing information could be done in parallel, as described earlier in this section.
6. Each trusted node that receives the message adds the information about the session to its routing table. Then, it sends an *OK* in response to the SDN controller.
7. When the SDN controller receives all responses with *OK*, it will mark the session as *OPERATION* and send the ID of the newly opened session to the application or trusted node that made the original request.

5.7 Event-based programming on the side of the KMS

The Key management system within the trusted node has many functionalities that must run in long sequences, which could lead to a potentially disastrous problem with long timeouts.

5.7.1 Problem of sequential key relay

The main problem arises during the invocation of the following sequence of methods:

1. The previous node in the path invokes the key relay API. A message with an encrypted application key is forwarded to the service layer of the KMS.
2. Get the encryption key from the QKD module to decrypt the application key.
3. Retrieve the next trusted node in the path of the key from the routing table.
4. Decrypt the key and fetch the application key from the QKD module.
5. Send the encrypted application to the next key in the path.
6. Wait for a response that the next module in the path finished the same procedure,
7. Send an OK response to the previous node in the path – response on a call first point.

The problematic fact is that the trusted node, which invokes the API of the next node in the path, is waiting for a response until the next node in the path is finished. Ultimately, this means that when the sequential solution is used, the first node in the path has to wait for the last node, and during this time, timeouts can occur.

5.7.2 Asynchronous key forwarding

I want to propose a better solution by implementing the asynchronous server. It is possible to split the original request into two parts. One of them will end after a short initialization procedure, and the other will invoke further actions related to the key relay. The procedures are the following:

- Correctly receive the message and save the application key to the database.
- Run the following tasks related to the current node and the next nodes within the path (decryption, routing, encryption, and a message to the next trusted node in the path of the key relay).

We can respond to the message sent by the previous trusted node once all the data from the message has been saved to the local database. Before sending an OK response to the previous node, we need to invoke the second (asynchronous) procedure to relay the key to the next trusted node.

There are two main approaches to implement the asynchronous server in the Spring framework:

- **Spring Async** - Library that enables us to run more threads with the server context simultaneously. The declaration is straightforward - we can specify methods that should be executed asynchronously with *@Async* annotation.

  The problematical part of Spring Async is passing the authorization for invoking APIS to the threads. I am afraid that the implementation complexity is too high, especially with passing the TLS certificates and encryption keys. This can be fixed to a great degree by setting the thread with data regarding security passed in constructors, but some issues might still occur.

- **Spring events** - Using this technique, we can create events that will be claimed by the job executor of the server asynchronously.

  It is possible to publish an event inside the running method. The event is captured by an interceptor and asynchronously executed when the event propagates to the top of the queue. The Interceptor then executes the method on the service layer of the server. This solution does not require a server with multiple threads, so there are no problems with passing request context to threads.

5.7.3 Event-based key forwarding proposal

I decided to use the event-based approach because I consider it easier to implement. I propose the following flow of events:

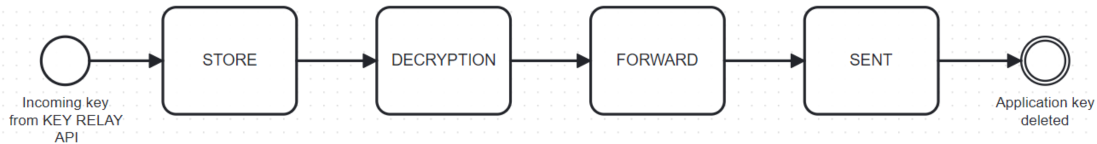

Figure 5.6. Sequence of events inside the KMS when the encrypted application key is received.

The diagram shows the sequence of events that are executed when a key is received by the KMS using the KEY RELAY API. The sequence is composed of events, which are the original parts of the sequence split into logical blocks. The first event is executed together with processing the incoming request. The others are invoked by *Spring Events*.

Responsibilities of each event section are the following:

- **STORE** — The encrypted application key is received by the KEY RELAY API and stored inside the KMS database. Also, the additional information from the key relay message, especially the encryption key ID, is essential for the next step.
- **DECRYPTION** — During this phase, the KMS fetches the encryption key from the related QKD module. The encryption key then decrypts the application key using the XOR function. This phase can take a relatively long time, based on the availability of the QKD module.
- **FORWARD** — The routing table is checked to find the application key's next hop (Trusted node). Then, the KMS fetches the encryption key from the corresponding QKD module, and the application key is encrypted with a fresh encryption key using the XOR function.

  The message for the KEY RELAY API is created and composed of the encrypted application key, its ID, the ID of the encryption key, and the ID of the related QKD application session. Then, the message is passed to the KEY RELAY API client, which finds the right address of the next node in the path and sends the message.
- **SENT** — In the final stage of the sequence, a counter is used to eventually safely delete the key from the KMS. After that, the key is flagged as deleted and is subsequently deleted from the database.

Using this event sequence, the methods located in the KMS server's service layer are invoked by events published by other methods. The invocation is done by event listeners, which are located in the controller layer. Service layer methods are invoked from there as a standard REST API request.

5.8 Implementation and code details

The proof of concept was implemented in *Java 21*, using the *Spring Boot 3.2* framework. In this section, I provide particular parts of code essential for the operation of the network. I also explain and discuss some of the best practices used for the implementation.

For the development, I have used the IntelliJ Idea integrated development environment. [4]

5.8.1 Reduction of boilerplate code

Implementation in Java brings a lot of boilerplate code[5] An example may be the accessing and modifying methods to the object's properties (Getters and Setters). Also, long constructors of objects make the code less readable.

Therefore, I use the Lombok [6] library, which adds annotations to generate the desired methods upon compiling the code.

5.8.2 Declarative generation of API servers and clients

Two common approaches to developing REST API in Spring Boot exist:

- **Code first approach** — The API is first implemented, and the documentation of the API is generated.
- **Design first approach** — The API documentation is created first. Then, the API implementation is generated from the documentation.

For my APIs, I am using the design-first approach, which enables the use of the Maven Open API Generator [7]. Using this technique, I am able to utilize the already prepared *yaml* files, created in the *OPEN API* format, to generate the API and all supportive files during the compilation. This is another step towards reducing the length of the code and enhancing its cleanliness and readability.

```yaml
/api/v1/keys/{slave_SAE_ID}/enc_keys:
  get:
    summary: Get key
    description: Retrieves an encrypted key for a key identified by {slave_SAE_ID}.
    parameters:
      - name: slave_SAE_ID
        in: path
        required: true
        description: URL-encoded SAE ID of the slave SAE.
        schema:
          type: string
      - name: number
        in: query
        required: false
        description: Number of keys requested.
        schema:
          type: integer
      - name: size
        in: query
        required: false
        description: Size of each key in bits.
        schema:
          type: integer
    responses:
      '200':
        description: Successful response
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/KeyContainer'
```

Figure 5.7. Example of ETSI 014 specification [9] in OPEN API format in YAML, which I have created.

[4] https://www.jetbrains.com/idea/
[5] Often repeated code that varies just a little.
[6] https://projectlombok.org/
[7] https://github.com/OpenAPITools/openapi-generator/blob/master/modules/openapi-generator-maven-plugin

5.8.3 Safe deletion of application keys

One of the main security concerns regarding the implementation is the safe deletion of old application keys from the KMS database. To enable the safe deletion of keys, I follow the procedure designed in the section 5.2.3:

1. Overwrite the vulnerable value of the application key with nulls.
2. Dereference the key by deleting it from the database.
3. Flush the database to force the immediate execution of deletion of the old key.
4. Finally, the garbage collector will be urged to dispose of the dereferenced data structures.

This technique is involved in the database by a scheduled event, implemented using *Spring Scheduling*, which runs the deletion procedure every 2 minutes. For the proof of concept implementation, the keys older than 10 minutes are considered old and removed during the procedure.

The following application code snippet shows the exact procedure for deleting old keys from the KMS. A functional approach is used to filter the keys for removal.

```
// Header with frequency, in which is the method invoked (2 minutes)
  @Scheduled(fixedRate = 120000)
  public void deleteOldKeys() {
    log.info("Running key deletion procedure");
    // Fetch of keys to enable the filterer of keys with
    // functional interface used to commit
    // operation with exact dates in SQL(2 minutes)
    List<ApplicationKey> keys =
    applicationKeyRepository.findAll();

    // Functional finding of keys older than 10 minutes.
    keys =
        keys.stream()
            .filter(key ->
            key.getCreatedAt().isBefore(LocalDateTime
            .now()
            .minusMinutes(10)))
            .toList();

    for (ApplicationKey key: keys) {
      log.info("Deleting key " + key + " because it is
      outdated!");

      //Overwrite the key
      key.setApplicationKey(NULL_KEY);
      applicationKeyRepository.save(key);

      //Deletion and dereference of the key
      applicationKeyRepository.deleteById(key.getId());

      //Force of execution of the previous operation
      applicationKeyRepository.flush();
```

```
    }
    //Invocation of the garbage collector
    System.gc();
  }
```

5.8.4 Own implementation of Dijkstra's algorithm for path finding

The SDN Controller uses Dijkstra's algorithm to create a key relay path. Dijkstra's algorithm is an algorithm for finding the shortest route in a weighted graph [19].

The algorithm input consists of the start and end nodes of the relay. During its execution, the algorithm requests the node relations from the repository.

To keep the appropriate level of complexity, I have assigned weight 1 to all QKD association links. In the future, the weight of the edge may reflect, e.g., the usage of related nodes and their QKD modules.

As opposed to a classical Dijkstra algorithm returning just the length of the shortest path, I need to obtain the whole sequence of nodes since it is essential for setting up the path. I have included this adjustment in my implementation. Therefore, the algorithm's output is a full sequence of nodes forming the path.

Later, the routing information is sent to the Trusted nodes from this sequence. Here, I present the detailed implementation of my Dijkstra algorithm.

```
  private List<String> createRoutingPath(String startNodeName,
  String endNodeName) {
    log.info("Create Routing Path " + startNodeName + " " +
    endNodeName);
    // Initialization
    Queue<List<String>> queue = new LinkedList<>();
    List<String> solution = new ArrayList<>();
    List<String> visited = new ArrayList<>();
    visited.add(startNodeName);
    // Initialize queue with paths from the start node
    for (String node : getConnectedNodesNames(startNodeName)) {
      List<String> arr = new ArrayList<>();
      arr.add(startNodeName);
      arr.add(node);
      queue.add(arr);
    }
    //Get the first path in queue, if exists
    while (!queue.isEmpty()) {
      List<String> nodes = queue.poll();
      // Skip if the last node in the path was already visited
      if (visited.contains(nodes.getLast())) {
        continue;
      } else {
        visited.add(nodes.getLast());
      }
      // End if solution was found
      if (nodes.contains(endNodeName)) {
```

```
        solution = nodes;
        break;
      } else {
        String lastNode = nodes.getLast();
        // Insert new paths from node
        getConnectedNodesNames(lastNode)
            .forEach(
                x -> {
                  ArrayList<String> addition = new ArrayList<>
                  (nodes);
                  addition.add(x);
                  queue.add(addition);
                });
      }
    }
    // If solution was not found throw an error
    if (solution.isEmpty()) {
      throw new IllegalStateException("Path " + startNodeName + "
      " +
      endNodeName + " not found");
    }
    log.info("Solution route is " + solution.toString());
    // Return solution
    return solution;
  }

  private TrustedNode getTrustedNodeByName(String name) {
    return trustedNodeRepository
        .findById(name)
        .orElseThrow(() -> new NoSuchNodeExistsException(name));
  }

  private List<String> getConnectedNodesNames(String nodeName) {
    return
    getTrustedNodeByName(nodeName).getConnectedNodes().stream()
        .map(TrustedNode::getName)
        .toList();
  }
```

5.8.5 Implementation of manipulation of keys with the XOR function

Encryption of the application key with the utilization of the XOR function is one of the core concepts of the QKD network. To apply the XOR function, the application keys must be in the form of bits. However, these keys are transferred as strings[8] when relayed between trusted nodes and QKD-secured applications. Additionally, the keys are stored as strings in the KMS database. Therefore, a reliable method is required to convert strings to bit sequences and back.

[8] For my implementation, I am exclusively using UTF-8 encoding for strings.

The first idea about changing formats was to use Java standard library *BitSet*, which should enable straightforward parsing of *BitSet* to string and back, as shown in the code snippet.

```
    //String to a bit set
    BitSet stringToBitSet(String string) {
        return
        BitSet.valueOf(string.getBytes(StandardCharsets.UTF_8));
    }

    //Bit set to string
    String bitSetToString(BitSet bitSet) {
        return new
        String(bitSet.toByteArray(),StandardCharsets.UTF_8);
    }
```

Contrary to my expectation, a String inputted and converted to *BitSet* and back to string, as shown in the snippet, was never the same as the original inputted String. So, I had to develop a different, more complicated solution.

I tested several prototypes until I found a robust solution. My current solution is converting *UTF-8* to a string composed of *0* and *1* bit by bit. This solution is robust, and I have tested that the string can always be transferred to a bit sequence and back. Utilizing this method, I have also implemented the XOR, as shown in a code snippet.

```
public static String bitStringToString(String key) {
    char[] chars = new char[32];
    for (int i = 0; i < 32; i++) {
      //Slice the string of bits to bytes
      String curSeq = key.substring(i * 8, i * 8 + 8);
      //Transfer bits to int
      int codePoint = Integer.parseInt(curSeq, 2);
      //Get a char from int
      String utf8Char = new String(Character.toChars(codePoint));
      chars[i] = utf8Char.charAt(0);
    }
    return new String(chars);
  }

  public static String stringToBitString(String key) {
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < 32; i++) {
      //Read the character at a certain position
      String binaryString = Integer.toBinaryString(key.charAt(i));
      //Transfer char to binary
      binaryString = String.format("%8s", binaryString)
      .replace(' ', '0');
      //Append to string
      builder.append(binaryString);
    }
```

```
    return builder.toString();
  }

  public static String xor(String applicationKey,
                           String decryptionKey){
    applicationKey = stringToBitString(applicationKey);
    decryptionKey = stringToBitString(decryptionKey);

    StringBuilder result = new StringBuilder();

    // Ensure both binary strings have the same length
    int length = Math.min(applicationKey.length(),
                          decryptionKey.length());
    for (int i = 0; i < length; i++) {
      // Perform XOR operation on each pair of corresponding bits
      char bit1 = applicationKey.charAt(i);
      char bit2 = decryptionKey.charAt(i);
      char xorResult = (bit1 != bit2) ? '1' : '0';
      result.append(xorResult);
    }

    return bitStringToString(result.toString());
  }
```

5.9 Connection for the QKD secured application to the Trusted nodes

The QKD-secured application is connected to the trusted node by the ETSI 014 API [9]. This API should be secured with a mutual TLS handshake.

I present the OPEN API specification for this API in Appendix C.1. The API can be created from the specification using the generator described in section 5.8.2.

The generated API interface for the server part of the API (API will be invoked by clients in QKD-secured applications) is then implemented in the application's controller layer. The particular services to process the corresponding requests are injected into the controller layer. All data structures required for the API are generated as well.

```
@Service  Sobotka, Vojtěch *
@RequiredArgsConstructor
@Log
public class KeyDeliveryApiDelegateImpl implements ApiApiDelegate {
  private static final String SLAVE_SAEID_DEFAULT = "SLAVE";  1 usage
  private static final String MASTER_SAEID_DEFAULT = "MASTER";  1 usage
  private final KeySupplyService keySupplyService;

  @Override  1 usage  Sobotka, Vojtěch
  public ResponseEntity<KeyContainer> apiV1KeysSlaveSAEIDEncKeysGet(
      String slaveSAEID, Integer number, Integer size) {
    return ResponseEntity.ok(
        keySupplyService.getKeysToMaster(MASTER_SAEID_DEFAULT, slaveSAEID, number, size));
  }

  @Override  1 usage  Sobotka, Vojtěch
  public ResponseEntity<KeyContainer> apiV1KeysMasterSAEIDDecKeysGet(
      String masterSAEID, String keyID) {
    return ResponseEntity.ok(
        keySupplyService.getKeyToSlave(masterSAEID, SLAVE_SAEID_DEFAULT, keyID));
  }
}
```

Figure 5.8. Controller resolving the API for QKD secured applications. Also, the injection of the related service layer class is shown

The main difference between ETSI 014 and ETSI 015 [10] lies in omitting the QKD application session ID in ETSI 014, which makes no sense for the key fetching. So, instead, I have to use the identification of both server and client QKD applications (Described as **MASTER** and **SLAVE** in ETSI 014) to find the application session ID in the routing table of KMS.

I have preregistered two applications, **MASTER** and **SLAVE**, to the network to simplify the demonstration of fetching the application keys. I also have implemented two essential methods to show the key relay from this specification (As shown in the snippet of the implementation of API); other methods could be invoked, but *NOT_IMPLEMENTED* response will be returned.

Due to the recurring problems with the mTLS certificates, I have turned off the authentication to ensure a smooth demonstration of the core principles of the QKD network for the proof of concept. However, I have implemented the required code regarding mTLS (Keystores and authorization interceptors).

5.10 Connection of the KMS to QKD modules for encryption key fetching

QKD modules provide API that can be invoked by KMS to fetch keys from the QKD module. Typically, one KMS module will be connected to more than one QKD module, so some abstraction above the QKD modules is needed.

QKD modules and the KMS provide the same API for key fetching. The ETSI 014 API is described in the previous section 5.9. The OPEN API generator is provided with the specification for generating the API client instead of the API server.

As the physical infrastructure (QKD modules) was not ready for testing at the time of finishing my thesis, I needed to find a solution to simulate the functionality of QKD modules. This simulation is described in the demonstration section 7.1.1.

I have created a universal interface (Utilizing the service interface pattern) to fetch encryption keys from the QKD modules.

```
@Service
public interface EncryptionKeyService {
  /*
   * Get a new encryption key from the QKD module
   */
  EncryptionKey getEncryptionKeyForRelayString(String toNode);

  /*
   * Get the decryption key by ID and related node
   */
  String genEncryptionKeyIdById(String id, String fromNode);
}
```

This interface is implemented for both real and mock QKD module services. The mocked QKD module service is used for demonstration, but a real QKD module service for connection to real QKD modules is implemented and prepared for service. Below, I present the core logic of service.

```
// @Service Uncomment when connected to the QKD modules
@RequiredArgsConstructor
public class RealQkdModuleService implements EncryptionKeyService {
  private final QkdModuleClient qkdModuleClient;

  /*This method is by the client calling the
  GET /api/v1/keys/{slave_SAE_ID}/dec_keys
  to fetch a new key
  */
  @Override
  public EncryptionKey getEncryptionKeyForRelayString(String
  toNode)
  {

    KeyContainer keyContainer =
    qkdModuleClient.getOneEcryptionKey(toNode);
    Key key =
        Optional.ofNullable
        .orElse(Collections.emptyList()).stream()
            .findFirst()
            .orElseThrow(() -> new IllegalStateException("Module
            return no encryption key"));
    EncryptionKey encryptionKey = new EncryptionKey();
    encryptionKey.setKeyId(key.getKeyID());
    encryptionKey.setEncryptionKey(key.getKey());
    return encryptionKey;
  }

  /*
  This method is by the client calling the
   GET /api/v1/keys/{master_SAE_ID}/dec_keys
  to get a decryption key by ID.
```

```
     */
    @Override
    public String genEncryptionKeyIdById(String id, String
    fromNode)
    {
      String returnedKey =
      qkdModuleClient.getOneEncryptionKeyByIdAndRelatedAdress(id,
      fromNode);
      if (returnedKey == null || returnedKey.isEmpty()) {
        throw new NoKeyFoundException(id);
      } else {
        return returnedKey;
      }
    }
  }
```

5.11 Interface to manage the key relay

One of the tasks in my thesis assignment was to provide an interface that enables the control and configuration of the QKD network.

The users of the proof of concept will be individuals with knowledge of QKD networks who are interested in observing the functionalities of the network.

The interface aims to provide the following methods to enable the demonstration of the QKD network:

- Enable the MASTER QKD-secured app to fetch the application keys. Master requests the parameters of keys and the number of keys to be generated.
- Enable the SLAVE QKD-secured app to fetch the application keys by ID.
- Set up the routing path on the SDN controller.

Due to the large scale of my project, I decided to use a simple Swagger interface rather than implementing a full-scale web application. Spring Boot offers the Swagger API explorer with documented Rest API as an interface to test the API.

Library *SpringDoc OpenAPI Starter WebMVC UI* [9] is added to both the KMS and the controller. The interface is located at `http://basepath/swagger-ui/index.html##`. The basepath is relevant to the component that provides the API.

[9] https://github.com/springdoc/springdoc-openapi

GET /api/v1/keys/{slave_SAE_ID}/enc_keys Get key

Retrieves an encrypted key for a key identified by {slave_SAE_ID}.

Parameters

Name	Description
slave_SAE_ID * required string (path)	URL-encoded SAE ID of the slave SAE. SLAVE
number integer($int32) (query)	Number of keys requested. 1
size integer($int32) (query)	Size of each key in bits. 256

Execute

Figure 5.9. Interface of request made towards the KMS to fetch the keys as **MASTER** QKD secured application

GET /api/v1/keys/{master_SAE_ID}/dec_keys Get key container for specified simple requests

Retrieves keys matching those previously delivered to a remote master SAE based on the Key IDs supplied from the remote master SAE. Only for specified simple requests.

Parameters

Name	Description
master_SAE_ID * required string (path)	URL-encoded SAE ID of master SAE. MASTER
key_ID * required string (query)	Key ID in UUID format. ID

Execute

Figure 5.10. Interface of request made towards the KMS to fetch the keys as **SLAVE** QKD secured application

Control Control interface for the QKD SDN Controller

POST /createApplication

Parameters

No parameters

Request body


```
{
  "startNodeId": "ALPHA",
  "endNodeId": "GAMMA",
  "clientAppId": "SLAVE",
  "serverAppId": "MASTER",
  "qos": {
    "maxBandwidth": 0,
    "minBandwidth": 0,
    "jitter": 0,
    "ttl": 0,
    "clientsSharedPathEnable": true,
    "clientsSharedKeysRequired": true
  }
}
```


Execute

Figure 5.11. Interface of request on the controller to open new QKD application session with parameters

Chapter 6
Quality assurance of developed components

All components of the QKD network must be carefully tested to deliver the required quality of service. The problematic part of testing the QKD network's components is the network's distributed nature, which significantly complicates testing the network as a unit. Moreover, because the network components should be modifiable, even more focus should be put on testing the individual components.

In this chapter, I would like to focus on quality assurance for the components delivered to the QKD network. I decided to write test levels for my components. The test levels demonstrate all aspects of testing that should be done to prepare the components for production operation.

6.1 Test levels

The test levels, or, more precisely, software testing levels, demonstrate the approach to testing and quality assurance of software [20]. Software testing levels are methodologies or procedures that split the software's development into multiple levels, which are tested individually.

The following test levels are considered:

- **Unit testing** — Lowest level testing used in software development. The focus should lie on small parts of code, like individual functions, not complicated software logic. The creation of unit tests is also important for future refactoring and replacing of the logic because the refactored code must pass the same test as the old code.

  This layer of tests could be easily tested within the Spring Boot with frameworks for unit testing, such as *JUnit*. Unit tests could also be a part of the *CI/CD* pipelines, which can run the tests before deployment.

- **Integration testing** — Integration testing focuses on testing the interaction between components. At this level, components on the module level are tested together to prevent issues like problematic inputs from one module to another. Also, the microservice API should be tested at this level. This means that the REST API provided by one of the system components in the form of a microservice will be tested from outside.

  This is the final part of testing for my proof of concept implementation. It focuses on testing the services as a whole unit. Possible software for this part of testing is the usage of frameworks like *Selenium*, which can call the REST API of the microservice from the web browser.

- **System testing** — System testing focuses on evaluating the entire integrated system to ensure it functions correctly. This level of testing is typically conducted by a quality assurance team and can be considered black-box testing, where the emphasis is on verifying inputs and outputs without knowledge of the internal

workings of the system. The testers performing system testing usually have no prior knowledge of the system's internal functions.

In relation to my implementation, system testing will be done when the whole system, including physical parts, is prepared.

- **User Acceptance Testing** — The last level of testing will be conducted before the application is deployed to the production environment. Acceptance testing will be purely black-box, focusing on several aspects:
  - **Security audit** — Due to the system's role in providing cryptographic and secured services, it must be audited by a qualified individual or organization to verify the security of all system components.
  - **Business requirements testing** — Relevant stakeholders should conduct tests to ensure the system meets the projected business requirements. Any discrepancies or insufficiencies should be identified and reported.
  - **User testing** — Real users should test the network for key generation, covering all possible network states, including potential error states. During this phase, tests for network availability and relay capacity should also be performed to ensure the system can handle the required load and scenarios.

Chapter 7
Demonstration of proof of concept

As the final part of my thesis, I want to demonstrate the fulfillment of my thesis assignment. The practical output of my thesis is the proof-of-concept demonstration of a functional software-defined network for a quantum key distribution network. The network was implemented as proposed in the previous chapters, adhering to the communications interfaces specified in the ETSI [9][10] standards while addressing the gaps or ambiguities in these specifications.

7.1 Configuration of a demonstration network

To create a demonstration setup as close as possible to the future testing deployment of the whole network, I replicated the same setup with the same components. In my implementation, QKD modules are simulated. Still, the interchangeable setup enables quick change of simulated QKD modules for the real ones with an already created service according to interface [9].

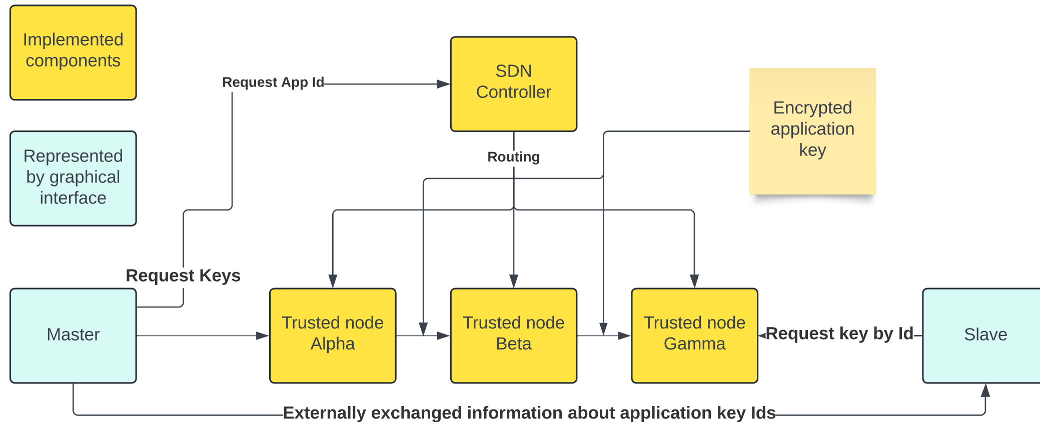

Figure 7.1. Diagram of the setup of QKD distribution network used for testing and demonstration of proof of concept implementation

The diagram illustrates the network, including all its components, the direction of sent requests, and the communication between the components. It is essential to mention that the direction of the request is changeable, and any node can serve as the start or the endpoint of an application key relay.

The following components are used for the testing setup:

- **SDN Controller** — Provides an interface for routing requests and topological information of the trusted nodes. However, for the proof of concept implementation, the nodes, their addresses, and connections were loaded inside the startup of the controller.

- **Trusted nodes** — Provide all required APIs, mainly the API for the QKD-secured applications to request keys and require opening a session. APIs for communication with the controller are provided but not implemented.

  There is no full coverage of the required data yet (some data structures are initialized to default values). The last but essential API is the key relay API, which has been implemented for both the client and server to send messages containing encrypted application keys.

- **QKD secured applications** — QKD secured application invoke the ETSI 014 API [9], for key relays on the individual trusted nodes. The default Swagger API explorer is used as a graphical interface for the key relay because of its simplicity.

7.1.1 Operation with mocked QKD modules

The demonstration is done with the simulated QKD modules. This brings a few things that need to be solved.

- Generation of application keys within the network and their usage for XOR encryption.
- Generation or supply of random application keys.

Mocking the QKD modules is problematic because synchronized encryption keys are needed in linked trusted nodes. My solution is to use pre-shared keys for each direction of each link between QKD modules. This key is retrieved from the service that creates the abstraction of the QKD modules. The only thing that needs to be done to run the demonstration with physical QKD modules (except for adding the certificates) is replacing the mocked service with the clients for real QKD modules. To simplify this process, both services implement the same interface.

The generation of random keys within the trusted node is solved with the standard Java library *SecureRandom*. This library should provide a safe and non-deterministic generation of random values according to the standard RFC 1750 [21]. Using this library, the random key is generated as a sequence of boolean values. In my implementation, 256-bit long keys are exchanged for demonstration purposes. The KMS database stores these bit sequences as strings with *UTF-8 encoding*. This encoding is also used to provide the application keys to the QKD-secured applications.

7.1.2 Operation of the network

To start the network, download the code and build both services as described in the **README** file with the code in the appendix B.1. By default, the components are located at the following addresses:

Component	address
Controller	http://localhost:8080
ALPHA	http://localhost:8081
BETA	http://localhost:8082
GAMMA	http://localhost:8083

All endpoints provided by the components are located at these addresses. For simplicity of use, the address *basepath/swagger-ui/index.html* is located in the Swagger API explorer. All endpoints can be invoked from this graphical interface, and detailed descriptions of the invoked APIs are also provided. Additionally, it is possible

to fetch the full specifications of each component on the path *basepath/v3/api-docs* for use in external API clients like Insomnia or Postman.

All system components log their current states and operations to the console or file if configured.

7.2 Demonstration of the Key Relay

The final part of my thesis demonstrates the key generation in the QKD-secured application. To present the functionality of the key generation, I demonstrate how both applications fetch the same application key from different nodes. I create a scenario for the key relay to demonstrate this functionality and present screenshots of send requests towards components along with their responses. Additionally, I provide a detailed description of the log of the trusted node Beta.

7.2.1 Testing scenario

The testing scenario involves generating an application key between three trusted nodes. The application invoking the relay of the key is named **MASTER** and is responsible for sending the key to the application **SLAVE**. The overall configuration of the testing network is shown in diagram 7.1.

1. Session creation is invoked on the **Controller**. The request creates a QKD application session between two QKD-secured applications MASTER and SLAVE, connected to the trusted nodes **ALPHA** and **GAMMA**. The container with appId is returned.
2. One application key is fetched from ALPHA with the target application SLAVE stated in the request. In return, the MASTER will get one key with ID.
3. SLAVE will invoke the API of the trusted node GAMMA to receive the application from the trusted node GAMMA with the ID of the application key that the master received from ALPHA. In return, the SLAVE will receive the desired key, identical to the one received by MASTER.

Verifying the successful end of the scenario involves comparing the keys fetched by both the MASTER and SLAVE. If they are the same, the base functionality of the network is verified.

7.2.2 Execution of demonstration

In this section, I present screenshots of the demonstration with descriptions.

1. Request to create a session is made towards the Controller. An error is thrown if an unknown node or QKD-secured Application is input. Additionally, an error could be thrown if there is no connection between known nodes or some node is unresponsive to routing messages.

Curl

```
curl -X 'POST' \
  'http://localhost:8080/createApplication' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "startNodeId": "ALPHA",
  "endNodeId": "GAMMA",
  "clientAppId": "SLAVE",
  "serverAppId": "MASTER",
  "qos": {
    "maxBandwidth": 0,
    "minBandwidth": 0,
    "jitter": 0,
    "ttl": 0,
    "clientsSharedPathEnable": true,
    "clientsSharedKeysRequired": true
  }
}'
```

Request URL

```
http://localhost:8080/createApplication
```

Figure 7.2. Curl request to open a new QKD application session

2. Response with information about an opened session with appId is returned, meaning that the multi-hop path for the key relay was set.

Server response

Code	Details
201	Response body

```
{
  "appId": "1",
  "appStatus": "OPERATIONAL",
  "qos": null,
  "appType": null,
  "serverAppId": "MASTER",
  "clientAppId": [
    "SLAVE"
  ],
  "backingQkdlId": null,
  "localQkdnId": "ALPHA",
  "remoteQkdnId": "GAMMA",
  "appPriority": null,
  "creationTime": "2024-05-19T15:55:15.5567188+02:00",
  "expirationTime": "2024-05-20T03:55:15.5567188+02:00",
  "appStatistics": null
}
```

Figure 7.3. Curl request to open a new QKD application session

3. The MASTER requests to fetch the number of application keys. The size of the keys should be arbitrary, but the network is demonstrated with keys 256 bits long. This request is specified by ETSI 014 QKD API [9], where an *appId* is not a part of the request. So, the node finds the *appId* in its database.
4. Standardized key container with *UTF-8*-formatted decrypted application key is returned. The MASTER must send the key ID outside the network towards the slave.

Curl

```
curl -X 'GET' \
  'http://localhost:8081/api/v1/keys/SLAVE/enc_keys?number=1&size=256' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:8081/api/v1/keys/SLAVE/enc_keys?number=1&size=256
```

Server response

Code	Details
200	Response body


```
{
  "keys": [
    {
      "key_ID": "5aa14206-aa0a-4363-a58d-ed880f4c4115",
      "key": "'t(L÷îÞa6g\u0015¶î :X?ýv?¸g?J?ÄOHR\f¯?"
    }
  ]
}
```


Figure 7.5. Response to request to fetch a key by MASTER QKD secured application

5. The SLAVE now fetches the key with specification of the server QKD-secured application MASTER and the ID of the application key are to be fetched. If the application related to the QKD secured application is not found, an error is thrown.

6. As we can see in Figure 7.6, the received application key is identical to the application key received by MASTER.

Curl

```
curl -X 'GET' \
  'http://localhost:8083/api/v1/keys/MASTER/dec_keys?key_ID=5aa14206-aa0a-4363-a58d-ed880f4c4115' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:8083/api/v1/keys/MASTER/dec_keys?key_ID=5aa14206-aa0a-4363-a58d-ed880f4c4115
```

Server response

Code	Details
200	Response body


```
{
  "keys": [
    {
      "key_ID": "5aa14206-aa0a-4363-a58d-ed880f4c4115",
      "key": "'t(L÷îÞa6g\u0015¶î :X�ýv�¸g�J�ÄOHR\f¯�"
    }
  ]
}
```


Response headers

```
connection: keep-alive
content-type: application/json
date: Sun,19 May 2024 19:28:04 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

Figure 7.6. Request and response of fetch of the application key by the client QKD secured application.

7.2.3 Description of logs of transfer node

I want to present what happens inside the network during the key relay. I decided to show the application log of the Trusted node BETA KMS and explain what is happening inside the trusted node.

During the previous scenario, the trusted node BETA had to set the key routing, receive the encrypted application key from ALPHA, and forward it to GAMMA.

In the following section, the raw application log was captured during the execution of the demonstration. The path for the key relay was already set up when the log was captured.

```
 [nio-8082-exec-3] c.c.fel.qkd.kms.service.KeyRelayService  : Received key relay request: class
 KeyRelayRequest {
    keyId: 5aa14206-aa0a-4363-a58d-ed880f4c4115
    encryptedKey: F2L¨¬°yPÔƒHq(ãœçÝ-äß°<9=JÅÎ
    encriptionKeyId: ALPHA_BETA_ENCRYPTION_KEY_ID
    appId: 1
}
2024-05-19T21:27:04.321+02:00  INFO 23300 --- [Key Management system for the SD - QKD network]
 [nio-8082-exec-3] c.c.f.qkd.kms.service.QkdModuleService   : decryptKey:
 5aa14206-aa0a-4363-a58d-ed880f4c4115
2024-05-19T21:27:04.348+02:00  INFO 23300 --- [Key Management system for the SD - QKD network]
 [nio-8082-exec-3] c.c.f.qkd.kms.service.QkdModuleService   : decryptedKey:
 aFdLcBdaOrEbmhKpyanzeJgVKtsqoFjW
2024-05-19T21:27:04.359+02:00  INFO 23300 --- [Key Management system for the SD - QKD network]
 [nio-8082-exec-3] c.c.f.qkd.kms.service.QkdModuleService   : encryptKey:
 5aa14206-aa0a-4363-a58d-ed880f4c4115
2024-05-19T21:27:04.366+02:00  INFO 23300 --- [Key Management system for the SD - QKD network]
 [nio-8082-exec-3] c.c.f.qkd.kms.service.QkdModuleService   : encryptedKey:
 5aa14206-aa0a-4363-a58d-ed880f4c4115
2024-05-19T21:27:04.367+02:00  INFO 23300 --- [Key Management system for the SD - QKD network]
 [nio-8082-exec-3] c.c.fel.qkd.kms.service.KeyRelayService  : Forwarding key:
 5aa14206-aa0a-4363-a58d-ed880f4c4115
2024-05-19T21:27:04.396+02:00  INFO 23300 --- [Key Management system for the SD - QKD network]
 [nio-8082-exec-3] c.c.fel.qkd.kms.client.KeyRelayClient    : Sending key relay request: class
 KeyRelayRequest {
    keyId: 5aa14206-aa0a-4363-a58d-ed880f4c4115
    encryptedKey: F6˙¨¹)_-~úƒnè®ÉÍ1ôí.)3hÎø
    encriptionKeyId: BETA_GAMMA_ENCRYPTION_KEY_ID
    appId: 1
} to http://localhost:8083
```

Figure 7.7. Application log of trusted node BETA regarding the session initialization and forwarding of one application key. Each line starts with the timestamp and specification of the logger, followed by the application's name and a detailed name of the execution. This data is not crucial. Essentials are the following: the name of the class from which the log came and the payload, which specifies what is happening during the process of key forwarding.

1. First, the encrypted application key is received from the ALPHA. We can see that a different key was received than what was mentioned in the previous section. That happened because the key is encrypted using the encryption key, whose ID is also part of the message.

2. Next, we can see logs of events and procedures related to the key processing. This shows how the application key passes through a sequence of events. Logs of this topic mainly indicate where errors happened during processes inside the KMS and when some error state occurred.

3. The last part of the forwarding process is singing the application key to the next trusted node. We can see that the encrypted application key is sent differently than received because it was encrypted using the XOR function with a new encryption key related to the QKD modules between which the forward happens.

```
2024-05-19T21:44:48.479+02:00  INFO 23300 --- [Key Management system for the SD - QKD network] [
 scheduling-1] c.c.f.q.kms.service.KeyDeletionService   : Running key deletion procedure
2024-05-19T21:44:48.484+02:00  INFO 23300 --- [Key Management system for the SD - QKD network] [
 scheduling-1] c.c.f.q.kms.service.KeyDeletionService   : Deleting key ApplicationKey
 (id=5aa14206-aa0a-4363-a58d-ed880f4c4115, applicationKey=F6'"¹)_-~úƒnè®�ÉÍ1ô�í�.)3hÎø,
 decryptedAfterReceiving=true, decryptionKeyId=ALPHA_BETA_ENCRYPTION_KEY_ID,
 encryptedBeforeSending=true, encryptionKeyId=BETA_GAMMA_ENCRYPTION_KEY_ID, appId=1,
 createdAt=2024-05-19T21:27:04.309301) because it is outdated!
```

Figure 7.8. A log showing the deletion of the old keys from the KMS database. Essential are the same messages as in the previous log. They show what is happening inside the KMS.

4. After forwarding the key, it is deleted in 10 minutes (this default interval is set in the system) from the KMS by the automatic procedure that runs inside the KMS every 2 minutes to dispose of old keys.

Based on my demonstration, I want to state that the implemented components of the QKD network (KMS and controller) work correctly.

7.3 Demonstration code

The code used for the demonstration, along with instructions on how to set and execute the demonstration, is part of Appendix B.1.

Chapter 8
Conclusion

I have successfully designed and implemented the proof of concept implementation of the quantum key distribution network. The implementation satisfies the core requirements of quantum key distribution while focusing on the path creation for the key relay and the generation of application keys by the trusted nodes.

8.1 Thesis summary and contributions

Initially, I thoroughly researched the QKD network functionalities' requirements, summarized in Chapter 2.

The design is divided into two phases. The first phase 3 is more theoretical, focusing on selecting QKD network architecture and providing definitions of the main components of the network.

In the second design phase 4, I focus on the design of the Key management system. My main contribution towards the topic is creating a state machine diagram of the phases of the application key lifecycle 4.1.2.

Based on the design proposals, I implement a proof of concept in Chapter 5. This part is vital since it focuses on creating an asynchronous event-based concept for the key forwarding 5.7 (illustrated by the event flow diagram 5.6) and on its implementation. Finally, in Chapters 6 and 7, I propose the QKD software testing methods and demonstrate the desired functionalities in different scenarios.

8.2 Fulfilment of the requirements

The system created (Implementation of the KMS and the network controller) fulfils all the assignment requirements. I have implemented the client protocol for communication with QKD modules 5.9. Unfortunately, the physical part of the system was not yet ready for testing, so I created a simulation of the QKD modules 7.1.1.

I have implemented the KMS system, which enables storing the keys 5.7 5.3.1.

The most complex part of the implementation was the key generation across multiple nodes within the network. This combines the creation and setup of the path 5.5.2, communication between the trusted nodes 4.1.1 and management of the keys inside the KMS 5.7.

The WEB interface for control of the key relay for creating routing paths was provided 5.11.

The last implementation requirement is the key delivery towards the QKD-secured application, which was implemented for the basic demonstration of functionality 5.10.

For demonstration purposes, mTLS authorization was disabled because of problems regarding the certificates' formats.

All these functionalities were demonstrated in my thesis during the demonstration part 7, which shows the basic scenario for the key generation.

8.3 Future plans

The future plans include finishing deploying the proof of concept, integrating it with the physical QKD modules, and further testing. This also includes enabling the mTLS authorization for the QKD client applications.

After the deployment, this work could serve as a demonstration of the QKD network for educational purposes at the Faculty of Electrical Engineering at CTU.

References

[1] Unathi Skosana, and Mark Tame. Demonstration of Shor's factoring algorithm for N = 21 on IBM quantum processors. *Scientific Reports*. 2021, 11 (1). DOI https://doi.org/10.1038/s41598-021-95973-w.

[2] Guobin Xu, Jianzhou Mao, Eric Sakk, and Shuangbao Paul Wang. An Overview of Quantum-Safe Approaches: Quantum Key Distribution and Post-Quantum Cryptography. *2023 57th Annual Conference on Information Sciences and Systems (CISS)*. 2023. DOI https://doi.org/10.1109/ciss56502.2023.10089619.

[3] Charles H. Bennett, and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*. 2014, 560 7–11. DOI https://doi.org/10.1016/j.tcs.2014.05.025.

[4] *clavis xg qkd system*. `https://www.idquantique.com/quantum-safe-security/products/clavis-xg-qkd-system/`.

[5] M. Sasaki, M. Fujiwara, H. Ishizuka, W. Klaus, K. Wakui, M. Takeoka, S. Miki, T. Yamashita, Z. Wang, A. Tanaka, K. Yoshino, Y. Nambu, S. Takahashi, A. Tajima, A. Tomita, T. Domeki, T. Hasegawa, Y. Sakai, H. Kobayashi, and T. Asai. Field test of quantum key distribution in the Tokyo QKD Network. *Optics Express*. 2011, 19 (11), 10387. DOI https://doi.org/10.1364/oe.19.010387.

[6] Miralem Mehic, Marcin Niemiec, Stefan Rass, Jiajun Ma, Momtchil Peev, Alejandro Aguado, Vicente Martin, Stefan Schauer, Andreas Poppe, Christoph Pacher, and Miroslav Voznak. Quantum Key Distribution. *ACM Computing Surveys*. 2020, 53 (5), 1–41. DOI https://doi.org/10.1145/3402192.

[7] Paul James, Stephan Laschet, Sebastian Ramacher, and Luca Torresetti. *Key Management Systems for Large-Scale Quantum Key Distribution Networks*. In: 2023.

[8] Spyridon Samonas, and David Lewis Coss. *The CIA Strikes Back: Redefining Confidentiality, Integrity and Availability in Security*. In: 2014. `https://api.semanticscholar.org/CorpusID:215838643`.

[9] ETSI. *Quantum Key Distribution (QKD); Protocol and data format of REST-based key delivery API*. ETSI GS QKD 014, 2019.

[10] ETSI. *Quantum Key Distribution (QKD); Control Interface for Software Defined Networks* . ETSI GS QKD 015, 20202.

[11] Sumit Badotra, and S. N. Panda. *Software-Defined Networking: A Novel Approach to Networks*. In: Brij B. Gupta, Gregorio Martinez Perez, Dharma P. Agrawal, and Deepak Gupta, eds. *Handbook of Computer Networks and Cyber Security: Principles and Paradigms*. Cham: Springer International Publishing, 2020. 313–339. ISBN 978-3-030-22277-2. `https://doi.org/10.1007/978-3-030-22277-2_13`.

[12] *ID Quantique product range.*
https://www.idquantique.com/quantum-safe-security/products/##quantum_key_distribution.

[13] Ittipong Khemapech. *Quantum as a Service (QaaS) in Digital Disruption Era Quantum as a Service (QaaS) in Digital Disruption Era Ittipong Khemapech.* 2024.

[14] ETSI. *Quantum Key Distribution (QKD); Application Interface.* ETSI GS QKD 004, 2020.

[15] Chris Richardson. *Microservices patterns : with examples in Java.* Shelter Island, New York: Manning Publications, 2019. ISBN 9781617294549.

[16] *Spring framework documenattion.* 2017.
https://spring.io/projects/spring-framework.

[17] *H2 database engine.*
https://www.h2database.com/html/main.html.

[18] Alexander Obregon. *Understanding Java's Garbage Collection.* 2023.
https://medium.com/@AlexanderObregon/understanding-javas-garbage-collection-bc141a2ef31f.

[19] Chung-Yang (Ric) Huang, Chao-Yue Lai, and Kwang-Ting (Tim) Cheng. *CHAPTER 4 - Fundamentals of algorithms.* Boston: Morgan Kaufmann, 2009. ISBN 978-0-12-374364-0.
https://www.sciencedirect.com/science/article/pii/B9780123743640500114.

[20] Mubarak Albarka Umar. *Comprehensive study of software testing: Categories, levels, techniques, and types.* 2020.
https://www.techrxiv.org/users/662669/articles/675874-comprehensive-study-of-software-testing-categories-levels-techniques-and-types.

[21] Steve Crocker, Donald E. Eastlake 3rd, and Jeffrey I. Schiller. *Randomness Recommendations for Security.* RFC 1750. Request for Comments. 1994.
https://www.rfc-editor.org/info/rfc1750.

Appendix A
Abbreviations

A.1 Abbreviations

AES Advanced Encryption Standard - A widely used symmetric encryption algorithm for securing sensitive data.

API Application Programming Interface - A set of rules and protocols for building and interacting with software applications.

ETSI European Telecommunications Standards Institute - An organization responsible for defining telecommunications standards in Europe.

JSON JavaScript Object Notation - A lightweight data interchange format commonly used for transmitting data between a server and a web application.

mTLS Mutual Transport Layer Security - A security protocol that provides authentication and encrypted communication between client and server.

QKD Quantum Key Distribution - A method for securely sharing encryption keys using the principles of quantum mechanics.

KMS Key Management System - A software service for managing security keys.

REST Representational State Transfer - An architectural style for designing networked applications, often used in web services development.

RSA Rivest-Shamir-Adleman - A public-key asymmetric encryption algorithm widely used for secure data transmission.

SDN Software-Defined Networking - A programming architecture allowing network administrators to control network behaviour programmatically.

SD-QKD Software-Defined Quantum Key Distribution - A framework for implementing quantum key distribution protocols in software-defined networks.

SQL Structured Query Language - A standard language for managing and manipulating relational databases.

Appendix B Source code

B.1 Source code on the GitLab

The source code for proof of concept implementation is located on GitLab `https://gitlab.fel.cvut.cz/sobotvo2/qkd-sdn-network`. The repository also contains build components with instructions for the demonstration, as described in chapter 7.

Appendix C
Created API specifications

C.1 ETSI 014 API

I present the created ETSI 014 *YAML* specification. I created this specification to enable the API generation from the OPEN API specification. The source for this was the original ETSI text specification [9]

```
    openapi: 3.0.0
info:
  title: ETSI GS QKD 014 V1.1.1 REST API
  version: 1.0.0
  description: OpenAPI Specification for ETSI GS QKD 014 V1.1.1
  (2019-
  02)
paths:
  /api/v1/keys/{slave_SAE_ID}/status:
    get:
      summary: Get status
      description: Retrieves status information from a KME to the
      calling SAE.
      parameters:
        - name: slave_SAE_ID
          in: path
          required: true
          description: URL-encoded SAE ID of the slave SAE.
          schema:
            type: string
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Status'
        '400':
          description: Bad request format
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
        '401':
          description: Unauthorized
```

```
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
        '503':
          description: Error on the server side
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
  /api/v1/keys/{slave_SAE_ID}/enc_keys:
    get:
      summary: Get key
      description: Retrieves an encrypted key for a key
      identified by
      {slave_SAE_ID}.
      parameters:
        - name: slave_SAE_ID
          in: path
          required: true
          description: URL-encoded SAE ID of the slave SAE.
          schema:
            type: string
        - name: number
          in: query
          required: false
          description: Number of keys requested.
          schema:
            type: integer
        - name: size
          in: query
          required: false
          description: Size of each key in bits.
          schema:
            type: integer
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/KeyContainer'
        '400':
          description: Bad request format
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
        '401':
```

```
          description: Unauthorized
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
        '503':
          description: Error on the server side
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
    post:
      summary: Get key with options
      description: >-
        Retrieves an encrypted key with additional options for a
        key
        identified
        by {slave_SAE_ID}.
      parameters:
        - name: slave_SAE_ID
          in: path
          required: true
          description: URL-encoded SAE ID of the slave SAE.
          schema:
            type: string
      requestBody:
        description: Key request data format
        required: false
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/KeyRequest'
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/KeyContainer'
        '400':
          description: Bad request format
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
        '401':
          description: Unauthorized
          content:
            application/json:
```

```
              schema:
                $ref: '#/components/schemas/Error'
        '503':
          description: Error on the server side
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'

  /api/v1/keys/{master_SAE_ID}/dec_keys:
    post:
      summary: Get key container
      description: Retrieves keys matching those previously
      delivered
      to a remote master SAE based on the Key IDs supplied from
      the
      remote master SAE.
      parameters:
        - name: master_SAE_ID
          in: path
          required: true
          description: URL-encoded SAE ID of master SAE.
          schema:
            type: string
      requestBody:
        description: Key IDs data format
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/KeyIDs'
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/KeyContainer'
        '401':
          description: Unauthorized
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'

    get:
      summary: Get key container for specified simple requests
      description: Retrieves keys matching those previously
      delivered
```

```
        to a remote master SAE based on the Key IDs supplied from
        the
        remote master SAE. Only for specified simple requests.
      parameters:
        - name: master_SAE_ID
          in: path
          required: true
          description: URL-encoded SAE ID of master SAE.
          schema:
            type: string
        - name: key_ID
          in: query
          required: true
          description: Key ID in UUID format.
          schema:
            type: string
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/KeyContainer'
        '401':
          description: Unauthorized
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
components:
  schemas:
    Status:
      type: object
      properties:
        source_KME_ID:
          type: string
        target_KME_ID:
          type: string
        master_SAE_ID:
          type: string
        slave_SAE_ID:
          type: string
        key_size:
          type: integer
        stored_key_count:
          type: integer
        max_key_count:
          type: integer
        max_key_per_request:
```

```
        type: integer
      max_key_size:
        type: integer
      min_key_size:
        type: integer
      max_SAE_ID_count:
        type: integer
      status_extension:
        type: object
  KeyRequest:
    type: object
    properties:
      number:
        type: integer
        description: Number of keys requested. The default
        value is
        1.
      size:
        type: integer
        description: >-
          Size of each key in bits. Default value is defined as
          key_size in
          Status data format.
      additional_slave_SAE_IDs:
        type: array
        items:
          type: string
        description: >-
          Array of IDs of slave SAEs. Used for specifying two
          or
          more slave
          SAEs to share identical keys.
      extension_mandatory:
        type: array
        items:
          type: object
        description: >-
          Array of extension parameters specified as name/value
          pairs that KME
          shall handle or return an error.
      extension_optional:
        type: array
        items:
          type: object
        description: >-
          Array of extension parameters specified as name/value
          pairs that KME may ignore.
  KeyContainer:
    type: object
```

```yaml
    properties:
      keys:
        type: array
        items:
          $ref: '#/components/schemas/Key'

  KeyIDs:
    type: object
    properties:
      keys:
        type: array
        items:
          type: string
  Key:
    type: object
    properties:
      key_ID:
        type: string
      key:
        type: string
  Error:
    type: object
    properties:
      message:
        type: string
        description: Error message
      details:
        type: array
        items:
          type: object
        description: >-
          Array to supply additional detailed error information
          specified as
          name/value pairs.
```