NetXPTO - LinkPlanner - KML

Armando Nolasco Pinto

October 26, 2023

Contents

Chapter 1

Case Studies

1.1 Key Management Layer

Goal	:	Design and implementation of a quantum key management layer
Directory	:	sdf/key_management_layer
Contributors	:	Armando Nolasco Pinto (2023/03/– —–/–/–)
		Diogo Matos (2023/03/14 - —–/–/–)

1.1.1 Introduction

Quantum Key Distribution (QKD) enables the negotiation of cryptographic keys in a secure manner without relying on computational complexity to achieve its security. On top of the QKD systems a Key Management Layer (KML) need to exist in order to mediate the key provisioning to the applications. The key manager provides keys with characteristics requested by the apps and bond to a pre-negotiated quality of service. The proposed system follows Discretion's and ETSI standards [1][2][3]. The system differentiates itself in the fact that mostly (or only) receives quantum oblivious keys from the QKD devices. A future version of the key manager should be able to offer classical, post-quantum and quantum keys, helping its integration into existing networks and providing more flexibility to its clients.

1.1.2 Notes and considerations

- During this section, quantum oblivious keys will be divided into two types, A and X. In any connection involving a oblivious keys, one of the parties will receive keys of type A and the other of type X. Keys of type A are whole keys, and keys of type X are keys that are a representation, in a oblivious key fashion, of half of its respective type A key.

- In this first version, only the quantum part of the system is specified, though it's our intention to support classical and post-quantum keys too. Nevertheless the system from scratch should be modular enough to facilitate this integration.

- This section will be gradually updated even during the implementation of the system. Some dependencies, like the SDN Agent and the QKD Devices, and issues only found during the implementation might trigger the need to change/enrich this specifications.

1.1.3 Architecture

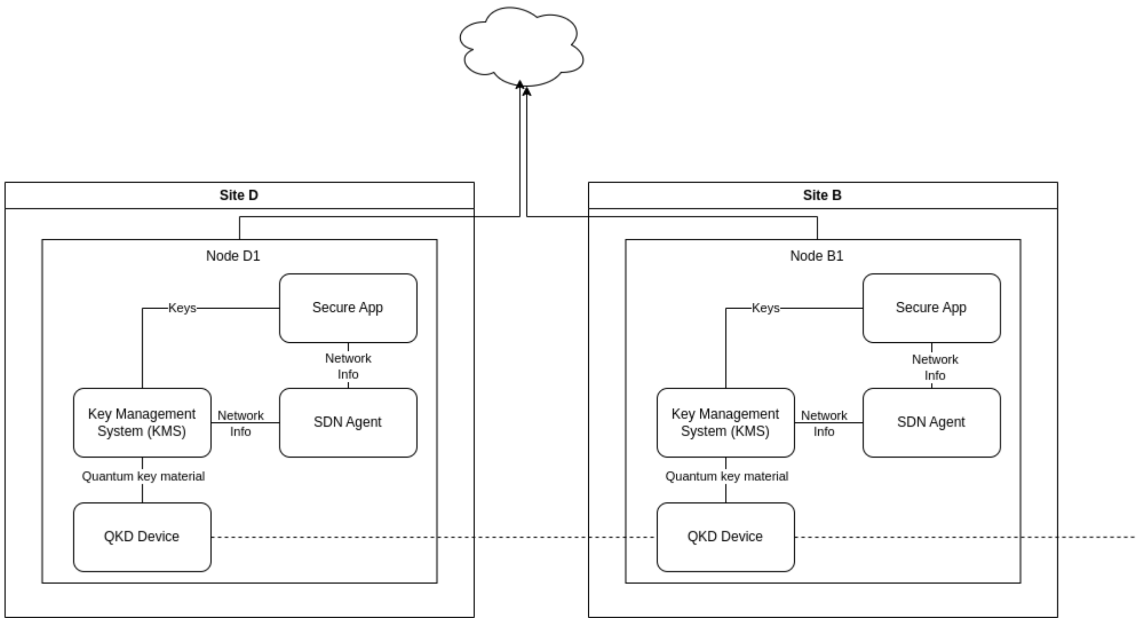


Figure 1.1: System architecture

In the diagram above (Figure 1.1 a simple overview over the whole system in a two node scenario where the Key Management System (KMS) is integrated can be seen. The dashed line represents a quantum link and the cloud is the black network where, for example, the Software Defined Network (SDN) Controller is located. Each secure site have one or more SD-QKD nodes each identified by its *qkdn_id*. More information on SD-QKD nodes, QKD interfaces, QKD key association links can be found in the standard ETSI QKD 015 [2]. The KMS is connected to a physical layer that provides keys to it. These keys can be symmetrical or oblivious and can be generated in distinct manners - classical, post-quantum or quantum (our main focus). Multiple applications are connected to it in order to retrieve keys. Both interfaces, to the applications and to the QKD devices, are based on ETSI QKD 004 standard [1]. Each KMS is connected to at least one KMS that is considered to be its peer since they receive matching keys from the physical layer. Also, the SDN agent (SDN Controller representative inside the node) is connected to the KMS and is used, from the KMS perspective, to retrieve information about the network, provide QoS parameters and to receive command orders.

The type of key sources connected to the KMS might limit the keys that this one can provide. The connection to a oblivious key source is enough to provide both symmetrical and oblivious keys and even random numbers. Though the key rate might be lower because in order generate a symmetrical key using oblivious keys, twice of the size of wished key will be used.

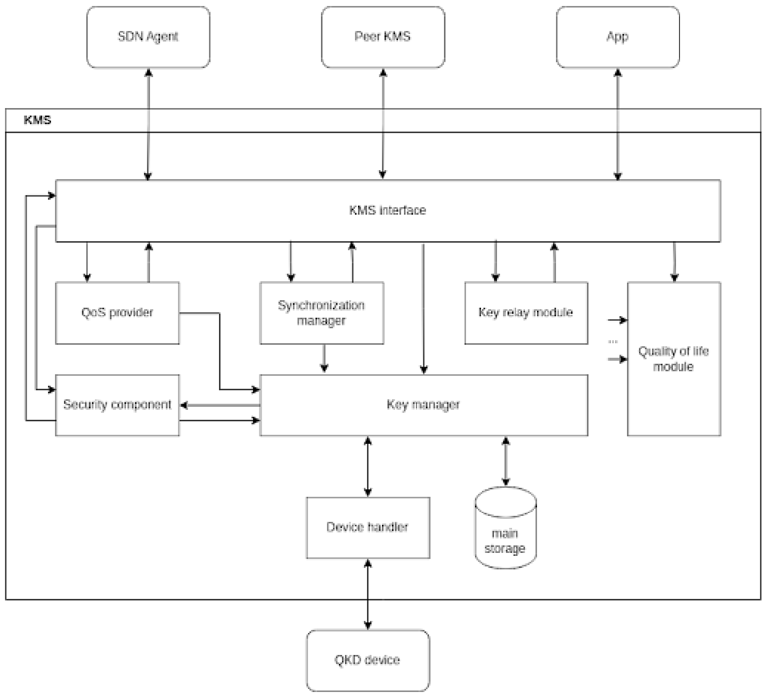


Figure 1.2: KMS logical modules

Apart from just receiving and sending keys through its South and North bound interfaces respectively, the KMS need to store, maintain, synchronize, derive and relay (forward) keys. The KMS is divided into various logical modules (Figure 1.2), namely:

- KMS interface (north bound interface)

  ETSI QKD 004 based interface to the applications(see 1.1.4). Interface to the SDN Agent (to be defined) and to its peer KMSs.

- QoS provider

  Gathers and provides all QoS parameters to be used to negotiate with the applications and to send them to the SDN Agent when requested.

- Synchronization manager

  Handles the synchronization protocol (see 1.1.8)

- Security component

  Implements all cryptographic algorithms used during the operation of the KMS.

- Key relay module

  Handles the key relay(forwarding) process (see 1.1.9).

- Key manager

  Maintains and handles keys. Is mainly used for key material retrieval, storage, etc. It's involved in almost any activity related with keys.

- Base functionality

  Handles several quality-of-life features such as logging and configuration.

- Device handler

  Handles devices connected to the KMS (such as the QKD devices).

1.1.4 KMS interface to the apps

This interface follows the ETSI QKD 004 [1] standard in pull mode, it was presented before in 1.3 where more detailed information can be found, nevertheless here is made a brief overlook of the API. During the implementation all error cases should be considered and handled has defined in the standard.

The interaction between KMS and app is based on three request, namely:

```
OPEN_CONNECT(in source, in destination, inout QoS, inout Key_stream_ID, out status)
GET_KEY(in Key_stream_ID, inout index, out Key_buffer, inout Metadata, out status)
CLOSE(in Key_stream_ID, out status)
```

The OPEN_CONNECT request is made by an app to create a key stream with the characteristics specified in the QoS field. A Key Stream Id (KSID) is returned. If the OPEN_CONNECT request is made with a value in the Key_stream_ID field, it connects to an existing Key Stream instead of creating a new one.

The GET_KEY request is used by the apps to retrieve keys. It can be made with or without an index. With a given index, the KMS will return the key, if exists, with that index. Otherwise, it return the oldest key that was created by its peer by not retrieved yet, if there is no keys to be return, a new one will be created and returned alongside with its index.

The CLOSE request is used to terminate a Key Stream connection. The peer still can retrieve already created and unexpired keys.

Since the proposed system is able to offer different types of keys, to the Quality of Service field in the OPEN_CONNECT was added one extra parameter called 'Key_type'. After the changes, the QoS parameters are:

- Key_type

  Key type for the key stream. 0: classical symmetrical, 1: P.Q. symmetrical, 2: quantum symmetrical, 3/4: classical oblivious (A/X), 5/6: P.Q. oblivious (A/X), 7/8: quantum oblivious (A/X).

- Key_chunk_size

  Length of the key buffer requested by the application. 32 bit unsigned integer.

- Max_bps

  Maximum key rate requested in bits per seconds. 32 bit unsigned integer.

- Min_bps

  Minimum key rate requested in bits per seconds. 32 bit unsigned integer.

- Jitter

  Maximum expected deviation, in bps, for key delivery. 32 bit unsigned integer.

- Timeout

  Time, in msec, after which the all will be aborted, returning an error.

- TTL (Time to Live)

  Time after which the keys corresponding to this Key_stream_ID shall be erased. 32 bit unsigned integer.

- Metadata mimetype

  Field that defines the format of the metadata on each subsequent GET_KEY call. Char array of size 256. JSON by default.

1.1.5 South bound interface (interface to the physical layer)

In order for the KMS to provide keys to the applications, it first needs to receive key material from the physical layer (a set of QKD devices). The interface between those two is based on ETSI QKD 004 [1] in push mode (the KMS is always receiving keys without making individual requests for each one). This interface is basically the same as the one provided by the KMS for the applications to use but now the KMS will act like an application requesting key material. The QoS parameters are the ones defined by ETSI's since usually each QKD device can only provide one key type. The KMS will start by making an OPEN_CONNECT request to create a connection to the QKD device. Then does one GET_KEY request and from that moment forward it will receive key material with the characteristics and pace specified in the QoS field until it makes a CLOSE request to terminate the key stream. The Quantum Key Distribution (QKD) device provide key material to the KMS by continuously sending key_buffers with its respective metadata.

1.1.6 KMS interface to the SDN Agent and app registration

Every time a new application, any entity requesting keys, connects to the KMS, it needs to be properly registered. The existence of all applications and properties inside the QKD network is only known by the SDN Controller. The following steps need to be made after a OPEN_CONNECT request from an application to the KMS:

1. Before any further action, the QoS parameters need to be checked, if the KMS cannot provide the QoS proposed by the application, it responds with a status code equal to 7 (OPEN_CONNECT failed because requested QoS settings could not be met, counterproposal has been included). This process repeats until both agree on a specific QoS.

2. The SD-QKD node (the KMS in this case), as it does not have information about the peer application or its SD-QKD node (remote), informs the SDN controller, via the SDN agent, of this new request, forwarding the same information as in the application request (Src app, Dst app, QoS), also including the local SD-QKD node ID.

3. If the peer application is not yet registered in the SDN controller, the controller sends back an acknowledgement, but without further information of the peer SD-QKD node. As specified in ETSI QKD 004 [1] an OPEN_CONNECT blocks, with a threshold defined in the timeout parameter in the QoS, until both applications are connected. If the peer is already connected, move to point number 5.

4. In the other secure node, the peer application connects to the KMS. The SD-QKD node does not have the peer application or SD-QKD node information, so it informs the SDN controller, via the SDN agent, of the incoming application and its requirements.

5. Now with both applications connected and registered, the SDN controller detects it, created a globally unique *app_id* (also known as *ksid* to the KMS), and sends to both SD-QKD nodes all the necessary information to configure the application at each endpoint (applications and nodes IDs, the *app_id* and the QoS requirements).

This process is for when the nodes have a physical quantum link, otherwise the process would have some slight differences that are explored in 1.1.9.

The specification of the SDN Controller and Agent is out of the scope of this section, but an interface between the KMS and the SDN Agent is starting to get some shape just in order to correctly carry out the process described above.

Name	From	To	Arguments	Notes
NEW_APP	KMS	SDN Agent	Src, Dst, QoS, Node_id	Replied with a REGISTER_APP or an acknowledge with a status code informing that the peer application is not connected
REGISTER_APP	SDN Agent	KMS	Src, Dst, QoS, Node_src, Node_dst, app_id	None

Table 1.1: KMS/SDN Agent requests related to application registration.

Additionally, the KMS should be able to be configured by the SDN Controller. Mainly for creation, modification and deletion of links. The manipulation of logical links is crucial for the process of key forwarding (1.1.9).

1.1.7 Key storage

All keys are stored in a Database (DB) that's accessed by the key manager in order to create and retrieve keys. As most as possible the DB should have mechanisms to create an abstraction layer, such as stored procedures and triggers, making it more resilient, secure and self sustaining. This same DB might be used to save other useful information as connected devices, application information and a basic network topology.

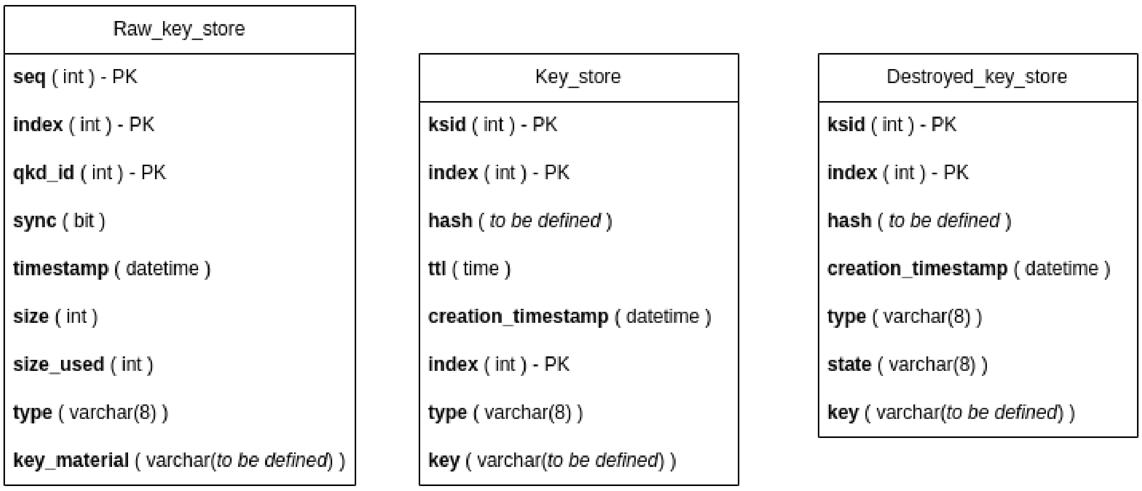


Figure 1.3: Database entity relationship diagram

As can be seen in diagram above (Figure 1.3), the DB is divided into three main tables.

The *Raw_key_store* has key material as received from the physical layer. This key material is not considered internally to be a already created key, instead will be used in the future to create keys when requested by the applications. This table is organized into the following fields:

- seq (int)(PK) - Sequence number attached to the frame sent by the QKD device.
- index (int)(PK) - Index of the key material chunk after being split. The size of the data field in the frames sent by the QKD devices is variable, so it might be too large to or inconvenient to store it all in one entry in the DB.
- qkd_id (int)(PK) - QKD device identifier.
- sync (bit) - Boolean value that tells if the key material is in sync with the peer.
- timestamp (datetime) - Date and time of when key material was save.
- size (int) - key material size in bytes.
- size_used (int) - number of bytes already used.
- type (int) - key type. 0: classical symmetrical, 1: P.Q. symmetrical, 2: quantum symmetrical, 3/4: classical oblivious (A/X), 5/6: P.Q. oblivious (A/X), 7/8: quantum oblivious (A/X).

- key_material (varchar(4096)) - key material.

The table shall have a maximum number of rows, after the table is full new keys must be discarded. Let say that the maximum is set to one million entries, the DB can store about 4 Gb of raw key material. Both this value and the maximum size of the key material might be changed depending on expected the key rate and the key material sent by the physical layer.

The *Key_store* table has the created keys. This table should have at least the following fields:

- ksid (int)(PK) - Key stream id that the key is related to. For internal keys, set to the symmetric value of the id of the peer KMS which the key is shared.
- index (int)(PK) - index of the key in the context of the key stream (sequential).
- hash (tbd) - hash value or some kind of checksum used in order to check the integrity of the key with the peer KMS.
- expiration_timestamp (datetime) - Expiration date.
- suspended (bit) - boolean value that tells if the key is in the suspended state.
- creation_timestamp (datetime) - Timestamp of when the key was created.
- type (int) - Key type. Classical oblivious (A/X), 5/6: P.Q. oblivious (A/X), 7/8: quaSntum oblivious (A/X).
- used (bit) - Boolean value that tells if the keys was already given to the app. It's used in order for a GET_KEY request is made without a given index and the the oldest key not provided yet should be returned is exists.
- key_material - key material.

The final table related to keys is the *Deleted_key_store* that its only purpose is to reduce the size of the *Key_store* by moving keys in the deleted state to this table.

Any key created has to go through its life-cycle. The states and timings are specified by NIST(NIST SP 800-57) and a summary of the standard can be found here. The key state can be extracted from the *expiration_timestamp* and *creation_timestamp*. The only state that cannot be derived from the timestamps is the suspended mode, for that purpose the *suspended* parameter in the *Key_store* table is used.

In terms of the mechanisms to keep the database to a certain degree clean, at least two scheduled events (scripts that run periodically) should be deployed in the DB - one to delete old not synced keys from the *Raw_key_store* and other to move keys to *Destroyed_key_store* based on the timestamps.

1.1.8 Key synchronization

The keys stored by peer KMSs have to match, not necessarily be identical, in order to correctly providing them to the applications. A key is synchronized if both peers have received it and both are aware of that fact.

A KMS to inform its peer of what keys it has received from the physical layer, a *key_synch* message is sent with a set of *index*s of the keys received since the last message of the same type. The number of *index*s in one message it may vary from KMS to KMS, as an example a KMS with a higher key rate might have a higher number of SEQ_ID's per message. For this process to work properly the *Key_chunk_size* agreed on between each KMS and the respective QKD device need to be same, otherwise it's impossible to synchronize the keys using this method.

Each time a new key_buffer is received by a KMS, an action will be performed based on the following criteria:

- current index was mentioned in a KEY_SYNC message sent by the peer KMS, and the number of distinct indexs was reached in order to send a KEY_SYNC message:

  action: send KEY_SYNC message to peer including the last received index and store key material with sync field at 1 (true).

- current index was not mentioned by its peer KMSâ€™s KEY_SYNC messages and the last notified seq is higher than the current seq:

  action: discard keys

- default:

  action: store keys with sync field at 0 (false)

The sync field of a key in the *raw_key_store* might be updated later based on the received *key_synch* messages. In order to set it to 1, both the following conditions need to be checked:

1. A KEY_SYNC message was received from the peer with the index of the key material associated with the key.
2. The KMS has sent a *key_synch* message notifying the peer the reception of the key material from where the key was took.

Not synced keys should be discarded after a high enough time period for a well functioning KMS to send at least one *key_synch* message. The value of this time period must be configurable. The removal of keys from the DB is made by a routine programmed in the DB that runs periodically (see 1.1.7).

1.1.9 Key relay

Since the QKD devices only generate matching key material between two directly connected devices, but matching keys are required to be present at two arbitrary nodes, a process to take a key from a KMS to other KMS (from a node to another node) not directly connected need to be available. A QKD Key Association Link is a logical key association between two remote SD-QKD nodes, each one being of one of two types: direct, if there is a quantum channel connecting the nodes, or virtual if keys are forwarded(key relay) through multiple trusted nodes to form an end-to-end association.

The key relay process, the creation and configuration of virtual links, is managed by the SDN Controller and the key relay is performed by a set of KMSs. The KMS / SDN Agent interaction during this process is not much specified in the standards (ETSI QKD 004 [1] and 015 [2]), though we know that each relay node is configured with the previous and next node in the path, the respective *ksid* and relay method to be used. In some specific cases where there is redundant direct links between nodes, it might also specify the link for the relay. This configuration is done between the NEW_APP and APP_REGISTRATION requests (see 1.1.6). Those properties are used by the SDN Agent to configure the KMS.

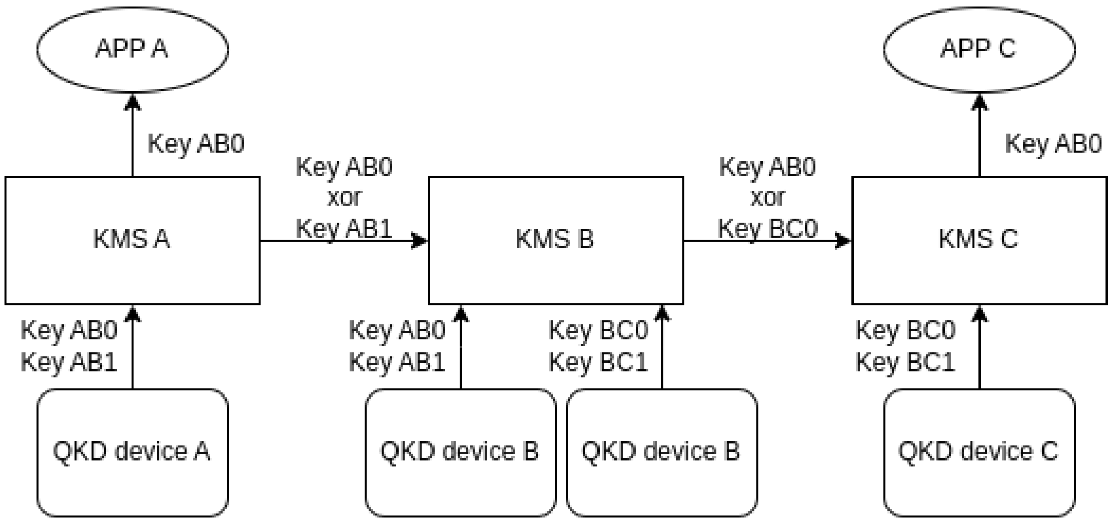


Figure 1.4: Key relay process

The basic key relay method (see Figure 1.4) is done by using One Time Pad (OTP) performing a XOR between the key to be relayed and a key that both peer KMSs know, this operation is done each hop. This key relay method requires all nodes to be trusted, in some specific cases that might be a problem. In a future implementation of the system more relay method can be added and used based on the security requirements on each end-to-end connection. Some other methods can be found in ETSI QKD 014 [4] and in this paper [5].

1.1.10 Communication between KMSs

The communication directly between peer KMSs is done over a low fidelity channel used only for synchronization purposes. There's two types of massages, namely:

- NEW_KEY

  - **Parameters** - Key Stream Id (Ksid), index, hash
  - **Goal** - Establish the creation of a key.
  - **Observations** - The peer KMS should create the key too and acknowledge it. The KMS that sends this message should only create the key after receiving the acknowledgement. If the message is not acknowledge during a valid time period (timeout defined in the QoS field in the OPEN_CONNECT request) no key should be created. The *index* field is not the index of the key to be created but the index of the raw key material from where the key will be extracted and it's optional.

- NEW_INTERNAL_KEY
  - **Parameters** - index, hash
  - **Goal** - Establish the creation of a key to be used internally by both peer KMSs. Usually, this keys are used for authentication and integrity check of messages when communicating with each other.

- KEY_SYNC
  - **Parameters** - Received key indexs (received_indexs)
  - **Goal** - Notify peer of what key material it was received from the physical layer. See 1.1.8.

- KEY_RELAY
  - **Parameters** - Multiple encrypted keys, their respective IDs and the ID of the relay process (used to know the next hop)
  - **Goal** - Forward keys to the next hop in the key relay process enabling key share between two not directly connected nodes.

1.1.11 Quality of service providing

In order for the QKD network to work effectively, the SDN Controller need to be aware of some basic QoS parameters of the KMS. The KMS at any point can receive a request from the SDN Agent asking for the current QoS metrics, for that the KMS has a dedicated module (QoS provider) that keeps a updated set of metrics of the current state of the system. This parameters are not specified explicitly in the specifications, but some can be extracted from some of the operations done by the SDN Controller like the proposed optimal key relay path computation algorithm. Each QoS parameter that might vary from key type to key type (e.g. key_availability) must be provided such way that distinction can be made. For now, the QoS metrics are:

- key_availability

  Total size of key material ready to be used (in bytes).

- key_rate

  Rate at which keys are received from the QKD devices.

- effective_key_rate

  Rate at which keys can be provided to an application (in bps). Does not include key that are used internally by the KMS.

- average_key_consuption_per_link

  Average key consumption (in bps). Very useful to compute optimal paths (in bps).

1.1.12 Algorithm security

To assure Information-Theoretic Security (ITS) of the system as a whole, every module need to use ITS algorithms to promise security against an adversary with unbound computational power.

1.1.12.1 Authentication and integrity

Can be achieved by using a Message Authentication Code (MAC), such as UMAC or Poly1305. Both the transmitter and the receiver have a shared secret, used by the first to create the MAC and by the other to verify the authenticity of the sender and the integrity of the message. This must be used in all communication where is possible to have shared keys, such between KMSs where it can take an advantage from QKD. But in order to assure both this aspect in a communication such as between the KMS and a secure app the use of asymmetrical algorithms would be perfect to bypass the QKD, but classical Key Encapsulation Mechanism (KEM) is not IT secure. The use of post-quantum and hybrid algorithms could be used to authenticate and share a keys between them. There's a very limited amount of ITS post-quantum algorithms, right now being advised to use hybrid algorithms instead. The use of certificates would simplify authentication and authorization allowing all entities in the system to identify themselves and others.

1.1.12.2 Encryption

Encryption is most importantly used in the communication between KMSs since all data is transmitted over the black network. Idealizing the use of QKD shared keys, the encryption of data can be as simple as a XOR, that is IT secure as long a different key is always used and its size it's equal to the size of the data to encrypt.

1.1.13 Sequence Diagrams

1.2 Two node basic scenario

In this scenario we consider a two nodes, Node A and Node B, directly connected in two distinct sites. App A and App B want to communicate, in order to get keys they connect to their respective KMS. This scenario can be considered one the most simple, representing the backbone of the system.

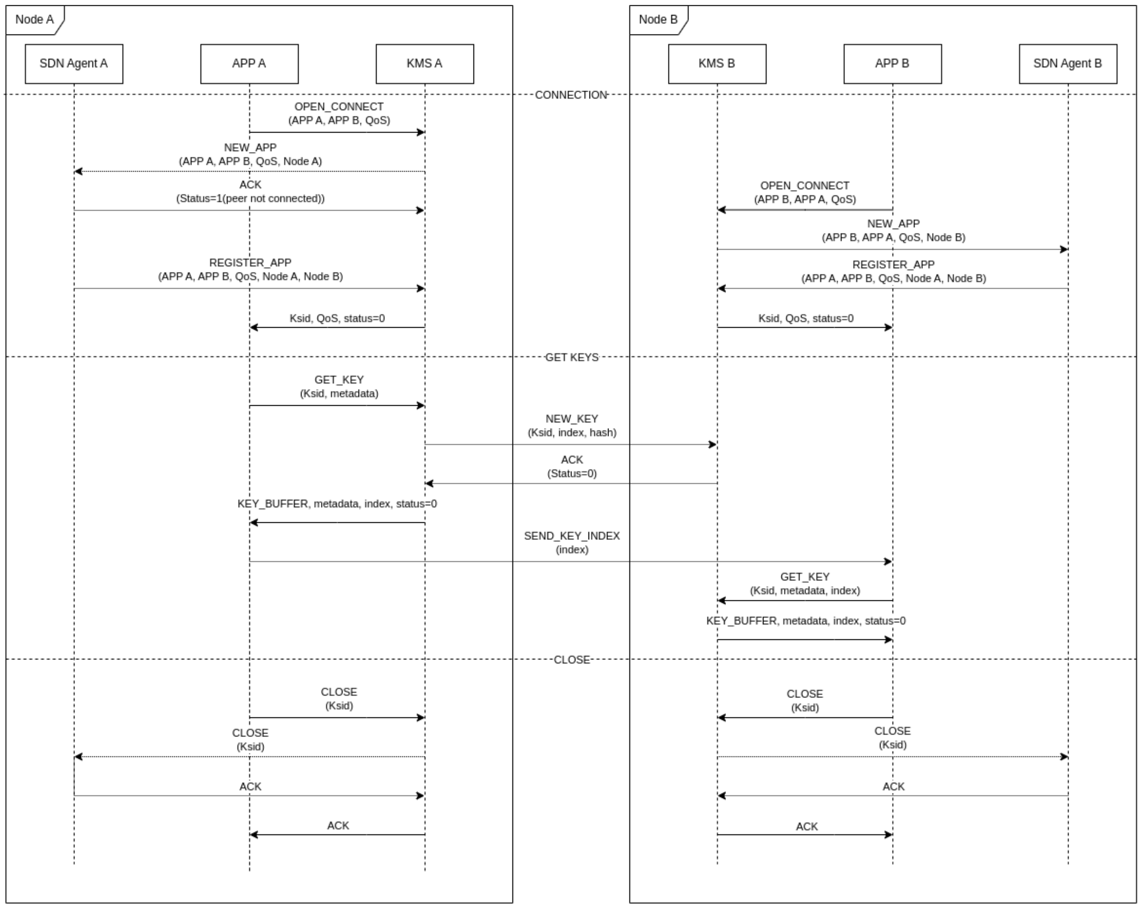

Node A
SDN Agent A
APP A
KMS A
Node B
KMS B
APP B
SDN Agent B
CONNECTION
OPEN_CONNECT (APP A, APP B, QoS)
NEW_APP (APP A, APP B, QoS, Node A)
ACK (Status=1(peer not connected))
OPEN_CONNECT (APP B, APP A, QoS)
NEW_APP (APP B, APP A, QoS, Node B)
REGISTER_APP (APP A, APP B, QoS, Node A, Node B)
REGISTER_APP (APP A, APP B, QoS, Node A, Node B)
Ksid, QoS, status=0
Ksid, QoS, status=0
GET KEYS
GET_KEY (Ksid, metadata)
NEW_KEY (Ksid, index, hash)
ACK (Status=0)
KEY_BUFFER, metadata, index, status=0
SEND_KEY_INDEX (index)
GET_KEY (Ksid, metadata, index)
KEY_BUFFER, metadata, index, status=0
CLOSE
CLOSE (Ksid)
CLOSE (Ksid)
CLOSE (Ksid)
CLOSE (Ksid)
ACK
ACK
ACK
ACK

Figure 1.5: Sequence diagram for a basic scenario with two directly connected nodes.

The workflow description associated with the sequence diagram (Figure 1.5) is as follows:

- In node A, APP A makes an OPEN_CONNECT request in order to create a connection to the KMS. The goal of APP A is to communicate to APP B and vice-versa, so the *source* is APP A, the *destiny* is APP.
- KMS A agrees on the QoS parameters proposed by the APP A. In order to know the location of APP B, sends a NEW_APP request to the SDN Agent A.
- In the instant that the SDN Controller is informed of the connection of APP A to KMS A in node A, APP B is to yet connected, so the SDN Agent acknowledges the NEW_APP request with a status code informing that the peer app is not connected.
- In node B, APP B makes a OPEN_CONNECT and the SDN Controller is notified via SDN Agent B.

- With both applications connected the SDN Controller creates a global unique *ksid* and, since Node A and Node B are directly connected, simply informs both KMSs via the SDN Agents of the *ksid* and *qkdn_id* of both nodes.
- Both KMS answer to the OPEN_CONNECT request made previously by indicating the QoS and *ksid* inherent to the connection.
- App A makes a GET_KEY request, giving the ksid and the metadata field only with the Metadata_size that dictates the maximum size of the metadata buffer.
- KMS A notifies KMS B of the creation of a new key for a specific *ksid*. The *index* field is not the index of the key to be created but the index of the raw key material from where the key will be extracted and it's optional. The *hash* field is used for KMS B to check the integrity of the key that will be created. KMS B creates the key and acknowledges with a successful status code.
- KMS A creates the key too and sends it to APP A alongside its metadata and index. The metadata might include creation and expiration timestamps and key type.
- APP B receives a key index (how it's sent is out of the scope of the KMS) and requests the key to KMS B that responds with a key matching the one received previously by APP A. It's important to note that the result of this GET_KEY request would be the same if no index would be provided by the application, since the oldest active key attached to this key stream is the one withe that index.
- Both applications send a CLOSE request to their respective KMSs to terminate the given key stream. The SDN Agents (consequently the SDN Controller too) are notified. Finally, both applications CLOSE request is acknowledge.

Acronyms

KML	Key Management Layer	4
KMS	Key Management System	5
KSID	Key Stream Id	7
SDN	Software Defined Network	5
DB	Database	10
OTP	One Time Pad	13
ITS	Information-Theoretic Security	15
MAC	Message Authentication Code	15
KEM	Key Encapsulation Mechanism	15

1.2.1 References

1. ETSI GS QKD 004, V2.1.1, 2020-08
2. ETSI GS QKD 015, V2.1.1, 2022-04
3. Discretion D3.1 SDN Preliminary Design Report, 2023
4. ETSI GS QKD 014, V1.1.1, 2019-02
5. Multiple stochastic paths scheme on partially-trusted relay quantum key distribution network, Hao, 2009

1.3 ETSI QKD 004

Goal	:	Implementation of the ETSI GS QKD 004
Directory	:	sdf/dv_qkd_ldpc
Contributors	:	Armando Nolasco Pinto (2021/01/22 -)
		Diogo Silva (2021/03/29 -)

1.3.1 Introduction

The security of cryptographic keys negotiated through an insecure communication channel is only ensured by the amount of computational power required to break it, to the point where it becomes unpractical to try to break them. This is done with using one-way functions. One-way functions are functions where it is easy to compute its output given a certain input but it is hard to do the reverse process. This makes it so that if someone eavesdrop the key negotiation channel and wants to get the negotiated key they have to test every possibility until they find a match. So if the keys are big enough they are considered secure, although the security of these keys is only assured by the difficulty of the reverse process, which is not mathematically proven. Usually the security of the keys is increased by making them bigger. However, with the presence of quantum computers, these methods become insecure. This is due to quantum computers being very fast at factoring large numbers. Because of this there is a need to create cryptographic key negotiation methods which do not become obsolete in the presence of quantum computers. ETSI Quantum Key Distribution protocol is a method for negotiating cryptographic keys whose security is ensured by the laws of physics, so they cannot be broken either by classic computers or quantum computers. These keys will be negotiated through a quantum channel controlled by specific equipment. The keys can then be distributed to applications via a key management layer which communicates directly with the QKD system. This key management layer is defined in the ETSI QKD 004 standard.

1.3.2 Key Management Layer

Although the QKD system manages the exchange of cryptographic keys, those keys need to be distributed and synced with the various applications that will use them. This is done by the Key Management Layer. This layer will ensure that the applications that communicate with each other at both ends use the same cryptographic key. The Key Management Layer provides the applications with an API (Application Programming Interface) for them to interact with the QKD System. The interface must have the following functions:

```
OPEN_CONNECT(in source, in destination, inout QoS, inout Key_stream_ID, out status)
GET_KEY(in Key_stream_ID, inout index, out Key_buffer, inout Metadata, out status)
CLOSE(in Key_stream_ID, out status)
```

- OPEN_CONNECT(in source, in destination, inout QoS, inout Key_stream_ID, out status)

  This function associates a Key_stream_ID to a set of future keys at both ends of the QKD system and establishes a set of parameters for the key service specified by the QoS structure. This function blocks until the peers are connected or the Timeout value is exceeded.

- GET_KEY(in Key_stream_ID, inout index, out Key_buffer, inout Metadata, out status)

  This function returns the required amount of key material requested for this specific Key_stream_ID. In case it does not return the key material it should return a error message. The QKD Management Layer should return an index value for the specified key. This index is used for synchronization purposes. The key position will be the result of multiplying the index value by the Key_chunk_size value.

- CLOSE(in Key_stream_ID, out status)

  This function verifies if the QKD link is available and the key_handle association is synchronized at both ends of the link. This function does not block and thus returns immediately indicating if both sides of the link are synchronized or if an error has occurred.

1.3.2.1 Parameters

The parameters used by the API functions are the following:

```
Key_stream_ID
Key_buffer
Index
Source
Destination
QoS
Metadata
Status
```

- Key_stream_ID

  The Key_stream_ID is a unique identifierfor the group of bits provided by the QKD Key Manager to the application. It is a 16 bytes UUID (Universally unique identifier).

- Key_buffer

  The key_buffer is a buffer containing the current stream of keys. It should be stored as an array of bytes.

- Index

  Position of the key to be accessed within the reserved key store for the application. Should be stored as a 32 bit unsigned integer

- Source

  Identifier of the source application connection to the QKD key management layer. The identifier is structured as an URI.

- Destination

  Identifier of the destination application connecting to the QKD key management layer. The identifier is structured as an URI.

- QoS

  The QoS parameters is a structure composed by the following parameters:

  – Key_chunk_size
    Length of the key buffer requested by the application. 32 bit unsigned integer.
  – Max_bps
    Maximum key rate requested in bits per seconds. 32 bit unsigned integer.
  – Min_bps
    Minimum key rate requested in bits per seconds. 32 bit unsigned integer.
  – Jitter
    Maximum expected deviation, in bps, for key delivery. 32 bit unsigned integer.
  – TTL (Time to Live)
    Time after which the keys corresponding to this Key_stream_ID shall be erased. 32 bit unsigned integer.
  – Metadata mimetype
    Field that defines the format of the metadata on each subsequent GET_KEY call. Char array of size 256.

- Metadata

  – Metadata_size Size of metadata buffer in characters. 32 bit unsigned integer.
  – Metadata_buffer Buffer for the returned metadata.

- status

  32 bit unsigned integer that indicates the success or failure of the request. The output values are the following:

  – 0: Successful
  – 1: Successful connection, but peer not connected.
  – 2: QKD_GET_KEY failed because insufficient key available.
  – 3: QKD_GET_KEY failed because peer application is not yet connected.
  – 4: No QKD connection available.

  - 5: OPEN_CONNECT failed because the Key_stream_ID is already in use.
  - 6: TIMEOUT_ERROR. The timeout has been exceeded.
  - 7: OPEN failed because requested QoS settings could not be met, counter proposal included in return has occurred.
  - 8: GET_KEY failed because metadata field size insufficient. Returned Metadata_size value holds minimum needed size of metadata.

1.3.2.2 Notes

Some additional considerations to be taken into account are:

- An application may request various different keys and there is no limit to the number of key streams a given application can request. The only requirement is that the Key_stream_ID must be different.
- The Key_stream_ID must uniquely identify the key material and the key material cannot be derived from it.
- The Key_stream_ID may or may not be provided by the application. In case the application does not provide a Key_stream_ID the Key Management Layer should generate one and return it to the application.

1.3.3 Extending QKD to oblivious keys

ETSIs QKD protocol could be extended to use oblivious keys instead. For that we need to add API calls that deal with QOKD instead of QKD.

```
QOKD_OPEN_CONNECT(in source, in destination, inout QoS, inout Key_stream_ID, out status)
QOKD_GET_KEY(in Key_stream_ID, inout index, out Key_buffer, inout Metadata, out status)
QOKD_CLOSE(in Key_stream_ID, out status)
```

There is also a additional parameter needed, `role` , that serves the purpose of indicating if the application will play the role of Alice or Bob in the oblivious transfer because both parties follow different routines in the transfer and the Key Manager needs that information. The status parameter may return an additional error, `1024`, that indicates conflicting roles, in case both APPA and APPB try to play the same role in the OT.

1.3.4 Use Case Examples

- APPA: Application at host A
- APPB: Application at host B
- KMA: QKD key manager at host A
- KMB: QKD key manager at host B

- QKSA: QKD system at host A
- QKSB: QKD system at host B

1.3.4.1 QKD with undefined key_handle

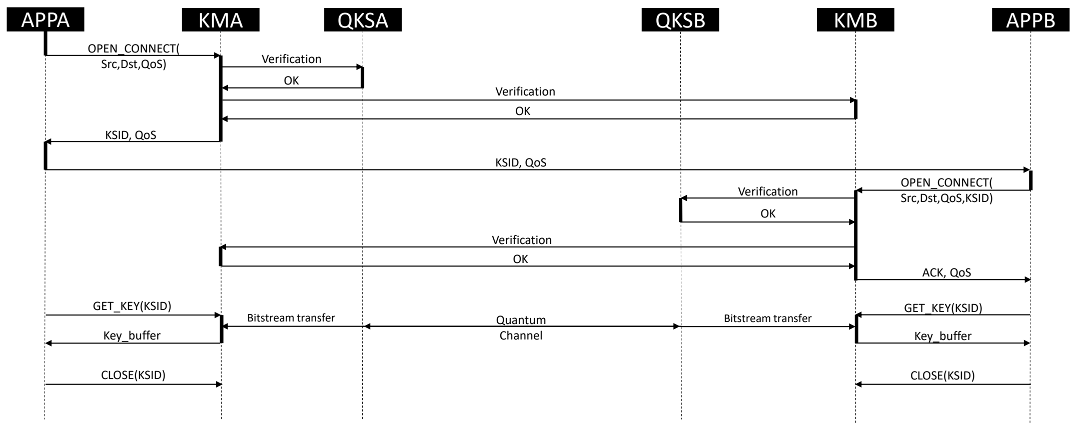


Figure 1.6: QKD with undefined Key_stream_ID

In this example APPA does not provide a Key_stream_ID. Because of that the Key Manager returns a generated Key_stream_ID, which the application has to send to the APPB to make sure that the KSID is synchronized at both ends.

1.3.5 Block Diagram

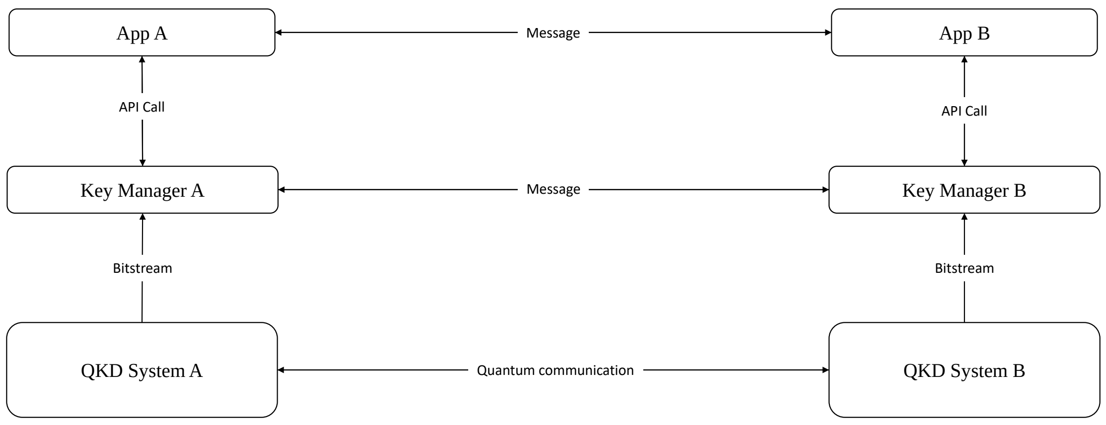


Figure 1.7: Communications Block Diagram

The objective of this system is to provide the same symmetric and oblivious keys to both applications (APP A and APP B). These keys will come from the QKD systems. The Key

Management Layer is responsible to get the keys from the physical QKD system. From there there is a need for the Key Manager to communicate with its peer Key Manager for key synchronization purposes. The way this is done is left to the implementer. One simple example is a socket communication. From there both Apps need to get the keys from their respective Key Manager. This should be done through calls to an API. The technology used for this API is left to the implementer. Two examples of ways to implement this API are REST and a socket. The only requirement is that the API accepts data in json format. The communication between both Apps is left to the implementer. This need for this communication is to make sure that both Apps have the same Key_stream_ID.

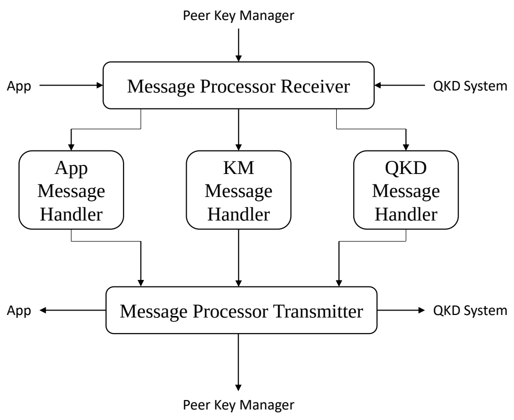


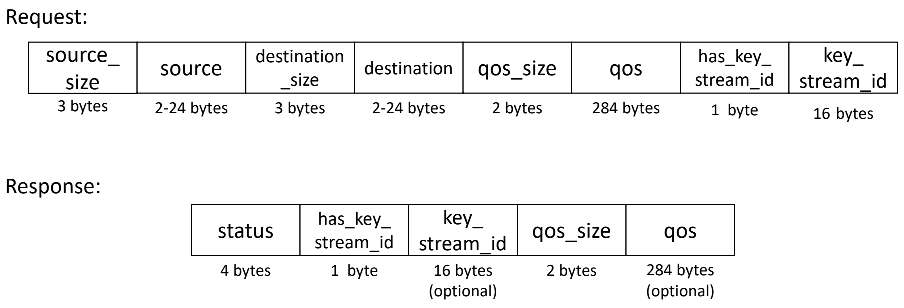
Figure 1.8: Message Processor

Above is a simple diagram that illustrates the message processor of each Key Manager. The Receiver gets its input from the App, the Peer Key Manager and the QKD System. These inputs might come in different formats. The receiver than relays that information to the appropriate message handler, depending on who the input came from. Once the handlers have processed the information and return an answer, that information is send to the Message Processor Transmitter, whose job is to then send the response to the adequate system.

1.3.6 Madrid Example

A example was provided from our partners in Madrid. This example is written in Python 3.6. It is a generic 004 implementation that has no hardware connection to the QKD system, and so it simulates it by generating a random key. There is a script that simulates two applications exchanging messages encrypted with the keys provided by the LKMS. The LMKS's can be run without the applications and used with our own application.

1.3.7 Socket Description



Request:

source_size	source	destination_size	destination	qos_size	qos	has_key_stream_id	key_stream_id
3 bytes	2-24 bytes	3 bytes	2-24 bytes	2 bytes	284 bytes	1 byte	16 bytes

Response:

status	has_key_stream_id	key_stream_id	qos_size	qos
4 bytes	1 byte	16 bytes (optional)	2 bytes	284 bytes (optional)

Figure 1.9: Open Connect Socket

- source

  URI of the source

- source_size

  Size of the source URI

- destination

  URI of the destination

- destination_size

  Size of the destination URI

- qos_size

  Size of the QoS structure

- qos

  QoS structure

- has_key_stream_id

  Informs if the request provides a KSID

- key_stream_id

  KSID provided by the application (optional)

- status

  Status code returned by the KMS

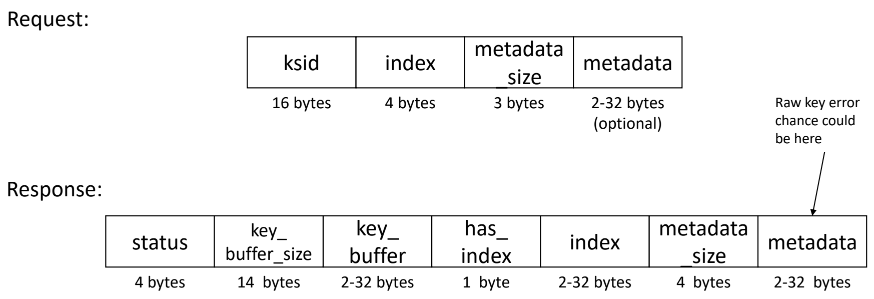


Figure 1.10: Get Key Socket

- ksid

  KSID

- index

  Position of the key to be accessed within the reserved key store for the application

- metadata_size

  Size of the metadata parameter

- metadata

  Metadata about the key requested/provided

- status

  Status code returned by the KMS

- key_buffer_size

  Size of the key buffer

- key_buffer

  Key Buffer

- has_index

  Field that indicates if the key index is provided

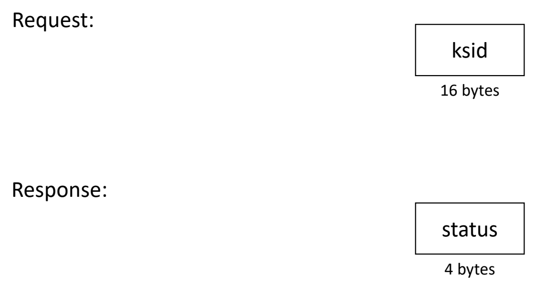


Figure 1.11: Close Socket

- ksid

  KSID

- status

  Status code returned by the KMS

1.3.7.1 Setting up the virtual environment and dependencies

Before running the example we need to make sure we are using a virtual environment (to have a clean installation), the right python version and the its dependencies. The first step is to install python 3.6 and gcc, running the following command:

```
sudo apt install -y python3.6-dev gcc virtualenv
```

To make sure the right version of python is installed we can run python3 with the –version argument:

```
python3 -version
```

Having python 3.6 installed we can proceed. The next step is to head over to the example directory, `sdf/etsi_qkd_004/madrid_example/` and open a terminal there so we can make a virtual environment. To install the virtual environment simply run the command:

```
virtualenv -p python3 qkd004
```

This will create a virtual environment with the name `qkd004`. After installing it we need to activate it. Run the command:

```
source qkd004/bin/activate
```

Upon the virtual environment we can install the needed python dependencies by running the command:

```
pip install -r requirements.txt
```

At this point, provided there were no errors, everything should be set for running the example.

The scripts should be used inside the virtual environment. After using it, to leave the virtual environment, simply type the command:

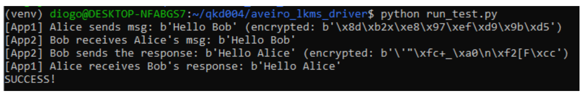
```
deactivate
```

1.3.8 Running the example

Two run the example simply run the command:

```
python run_test.py
```

This example will spawn two LKMS and two applications using threads. These applications will exchange messages encrypted with symmetric keys provided by the LMKS.



```
(venv) diogo@DESKTOP-NFABGS7:~/qkd004/aveiro_lkms_driver$ python run_test.py
[App1] Alice sends msg: b'Hello Bob' (encrypted: b'\x8d\xb2x\xe8\x97\xef\xd9\x9b\xd5')
[App2] Bob receives Alice's msg: b'Hello Bob'
[App2] Bob sends the response: b'Hello Alice' (encrypted: b'\'"\xfc+_\xa0\n\xf2[F\xcc')
[App1] Alice receives Bob's response: b'Hello Alice'
SUCCESS!
```

Figure 1.12: Madrid Example

1.3.9 Running only two LKMS

To run the LKMS without running the applications we must first install the qkd004 implementation as a python module by running the following command:

```
pip install etsi_qkd_004/
```

After that simply run the script `run_lkms.py`:

```
python run_lkms.py
```

This will spawn two LMKS on separate threads, one in port 44441 and another in port 44442, both on localhost. To change the port or address simply edit lines 5 and 6 of the script.

1.3.10 Using the Application driver

The Madrid example includes a driver for the application. This driver includes functions that implement the socket communication needed to interact with the LMKS API. To use this driver you also need to install the etsi_qkd_004 module as described above. Note that you should import the driver at the start of the python script with the following line:

```
from aveiro_driver.app_004 import *
```

This driver really simplifies the application implementation as there is only the need to initialize the application driver object. From then on if there is the need to encrypt messages just use the encrypt and decrypt method of the application driver.

First declare the variables needed (lkms address, source URI, destination URI and QoS):

```
lkms_node1_address = ("localhost", 44441)
source =
URI("qkd://Application1@aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa")
destination =
URI("qkd://Application2@bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb")
qos = QoS(32, 32, 32, 0, 0, 0, 0)
```

Being the QoS fields the key_chunk_size, max_bps, min_bps, jitter, priority, timeout and ttl, respectively.

Then, to initialize the driver:

```
node1 = App004(lmks_node1_address, source, destination, qos)
```

After that, to encrypt a message simply use:

```
node1.encrypt(message)
```

and to decrypt:

```
node1.decrypt(message)
```

Note that the message should be in bytes format.

1.3.11 Running Alice, Bob and Charlie symmetric key distribution

There is also an example where 3 parties exchange symmetric keys between each other. This example is present in `sdf/etsi_qkd_004/qkd004/`. To run this example simply set up the virtual environment and dependencies like explained before and run (in this order):

In Alice:

`python3 run_alice.py`

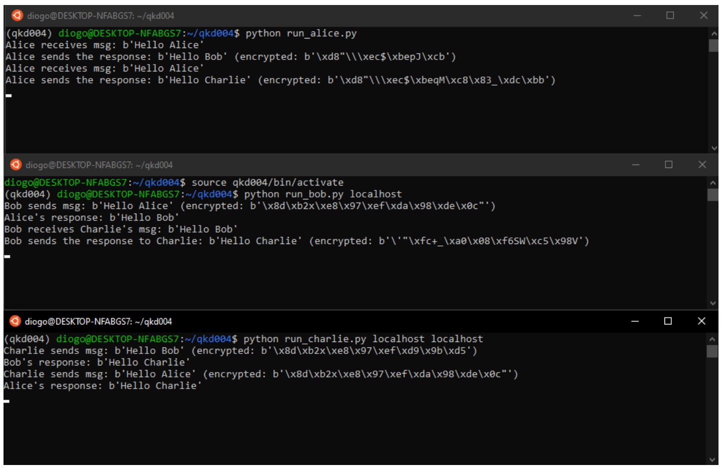
In Bob:

`python3 run_bob.py aliceIP`, using alice's IP address as argument

In Charlie:

`python3 run_charlie.py aliceIP bobIP`, replacing with Alice's and Bob's IP

You should see in each computer an exchange of message between peers.


diogo@DESKTOP-NFABGS7: ~/qkd004

```
(qkd004) diogo@DESKTOP-NFABGS7:~/qkd004$ python run_alice.py
Alice receives msg: b'Hello Alice'
Alice sends the response: b'Hello Bob' (encrypted: b'\xd8"\\\xec$\xbepJ\xcb')
Alice receives msg: b'Hello Alice'
Alice sends the response: b'Hello Charlie' (encrypted: b'\xd8"\\\xec$\xbeqM\xc8\x83_\xdc\xbb')
```

diogo@DESKTOP-NFABGS7: ~/qkd004

```
diogo@DESKTOP-NFABGS7:~/qkd004$ source qkd004/bin/activate
(qkd004) diogo@DESKTOP-NFABGS7:~/qkd004$ python run_bob.py localhost
Bob sends msg: b'Hello Alice' (encrypted: b'\x8d\xb2x\xe8\x97\xef\xda\x98\xde\x0c"')
Alice's response: b'Hello Bob'
Bob receives Charlie's msg: b'Hello Bob'
Bob sends the response to Charlie: b'Hello Charlie' (encrypted: b'\'"\xfc+_\xa0\x08\xf6SW\xc5\x98V')
```

```
(qkd004) diogo@DESKTOP-NFABGS7:~/qkd004$ python run_charlie.py localhost localhost
Charlie sends msg: b'Hello Bob' (encrypted: b'\x8d\xb2x\xe8\x97\xef\xd9\x9b\xd5')
Bob's response: b'Hello Charlie'
Charlie sends msg: b'Hello Alice' (encrypted: b'\x8d\xb2x\xe8\x97\xef\xda\x98\xde\x0c"')
Alice's response: b'Hello Charlie'
```

Figure 1.13: Running the examples all on the same machine

1.3.12 Raw Key Distribution

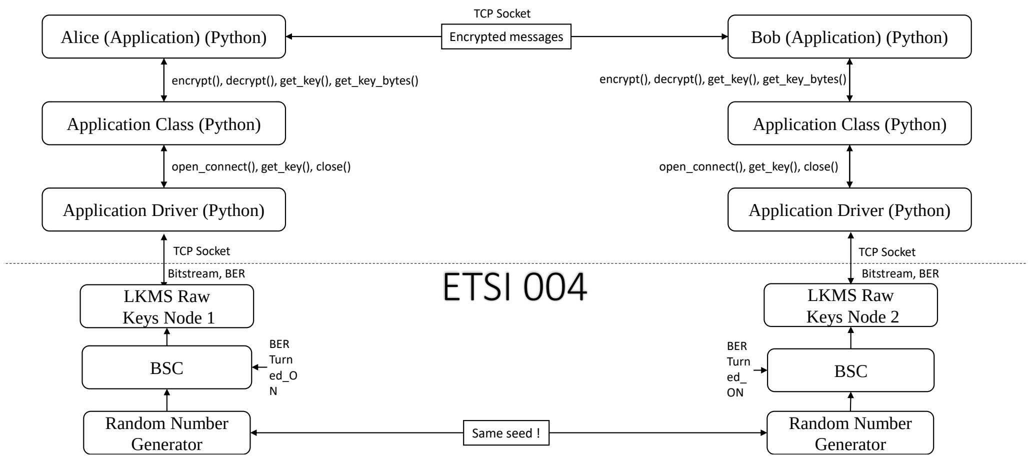

TCP Socket
Encrypted messages
Alice (Application) (Python)
Bob (Application) (Python)
encrypt(), decrypt(), get_key(), get_key_bytes()
encrypt(), decrypt(), get_key(), get_key_bytes()
Application Class (Python)
Application Class (Python)
open_connect(), get_key(), close()
open_connect(), get_key(), close()
Application Driver (Python)
Application Driver (Python)
TCP Socket
Bitstream, BER
TCP Socket
Bitstream, BER
ETSI 004
LKMS Raw Keys Node 1
LKMS Raw Keys Node 2
BSC
BER Turned_ON
BER Turned_ON
BSC
Random Number Generator
Same seed !
Random Number Generator

Figure 1.14: Raw Key Distribution

We have adapted the Madrid QKD example to also distribute raw keys. The way it works is when the LKMS Simulator is deployed, a seed and an error chance is associated with it. To deploy the LMKS use the following function:

```
LKMSSimulator(("Address",Port),Seed,Error_chance)
```

The error chance should be an integer from 0 to 100

To connect to the Raw Key Manager you should use the same procedure as the symmetric key manager:

```
app004 = App004(lkms_address, source, destination, qos,
KeyStreamId)
```

The key can then be retrieved with:

```
key = app004.get_key_bytes(size)
```

An example called "test_raw_keys.py" is available in `/sdf/etsi_qkd_004/qkd004/`

1.3.13 Development plan

- Socket Driver

  Develop the socket driver, responsible for translating the API functions into bytes and parsing its responses.

  Classes:

  - ksid
  - uri
  - qos
  - key_buffer
  - metadata

Functions:

- openConnectRequest_tobytes
- openConnectRequest_frombytes
- openConnectResponse_tobytes
- openConnectResponse_frombytes
- getKeyRequest_tobytes
- getKeyRequest_frombytes
- getKeyResponse_tobytes
- getKeyResponse_frombytes
- closeRequest_tobytes
- closeRequest_frombytes
- closeResponse_tobytes
- closeResponse_frombytes

• Application-KMS Driver Class

  Create an Application driver class, responsible for storing and managing important information like QoS and KSID that can be used in its calls to the API using the Socket Driver.

  Attributes:

  - kms_address

  Methods:

  - KmsDriver(address)
    Application-Kms driver constructor
  - KmsDriver
    Application-Kms driver destructor
  - void open_connect(source, destination, qos, ksid, status)
    Open Connect function
  - void get_key(ksid, index, metadata, status, key_buffer)
    Get Key function
  - void close(ksid,status)
    Close function

• Application Class

  Create an Application Class, responsible for abstracting the programmer from inner workings of the ETSI protocol, enabling him to provide the parameters and get the Key Buffer in return.

  Attributes:

- source
- destination
- qos
- ksid
- index
- key_buffer

Methods:

- App004(address, source, destination, qos, ksid)
  Application Class constructor
- App004
  Application Class destructor
- keyBuffer get_key(int size)
  Gets key buffer as a KeyBuffer object
- char* get_key_bytes(int size)
  Gets key buffer as an array of bytes
- char* encrypt(char* message)
  Encrypts a message using XOR method
- char* decrypt(char* message)
  Decrypts a message using XOR method

Example:

```
#include <iostream>
#include <string>
#include "app_class.h"

int main(){

// LKMS Address and port

std::string lkms_address = "192.168.1.100";
uint32_t lkms_port = 44440;

// Source and destination URI

std::string source_uri =
 "qkd://Application1@aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa";
std::string destination_uri =
 "qkd://Application2@bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb";
// QoS - key_chunk_size, max_bps, min_bps, jitter, priority, ttl,
// metadata_mimetype

QoS qos = QoS(32,32,32,0,0,0,"raw+oblivious");

// KSID

std::string ksid = "12345678-1234-1234-1234-123456789012";

// Initialize App class

App004 alice_app = App004(lkms_address, lkms_port, source_uri,
 destination_uri, qos, ksid);

// Request a key from the App class

char* key_buffer = alice_app.get_key_bytes(20, RAW_KEY);

// Close connection

alice_app.close()

}
```

1.3.14 Madrid cpp 004 example

1.3.14.1 Introduction

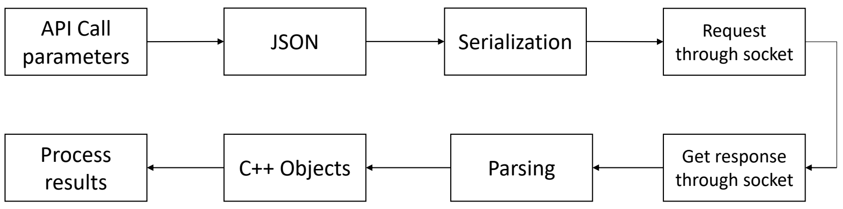


Figure 1.15: Madrid cpp example

The figure above tries to ilustrate how the cpp madrid example works. To start the API call parameters are defined. Once those parameters are defined (by the user), the API call function can be called. This function procedes to convert these parameters into a JSON object. The JSON object is then serialized and sent through a socket to the KMS. A response should be sent from the KMS through that same socket, which is then parsed and converted into a response struct and returned to the user that can then interpret these results as he wishes to.

1.3.14.2 Example

An example on how to use the madrid 004 cpp library was provided:

```
#include <iostream>
#include <nlohmann/json.hpp>
#include "etsi_qkd_004.hpp"
#include <stdexcept>
#include <thread>

void print_key_buffer(const etsi_qkd_004::KeyBuffer& key_buffer,
const std::string& name = "KEY BUFFER") {
std::cout << name << ": [ ";
std::for_each(key_buffer.cbegin(), key_buffer.cend(),
[](const unsigned char &c) {
std::cout << int(c) << " "; });
std::cout << "]" << std::endl;
}


int main(int argc, char **argv) {
if (argc - 1 != 2)
throw std::invalid_argument("ERROR: Number of arguments " +
std::to_string(argc - 1) +
"\narg1 should be IP (ex: \"127.0.0.1\")"
"\narg2 should be the PORT (ex: 1234)\n");

etsi_qkd_004::LKMSAddress address = {argv[1], (unsigned short) atoi(argv[2])};
std::cout << "Using address: " << address.ip << ':'
<< address.port << std::endl;

// CONFIGURE DESIRED KEY (CHUNK_SIZE * CHUNK_NUMBER in bytes)
// Number of key bytes retrieved per GET_KEY
const unsigned int CHUNK_SIZE = 5;
// Number of key chunks you want to get
```

```
const unsigned int CHUNK_NUMBER = 8;

// ----- OPEN_CONNECT -----
// Classic key
//    std::string source = "qkd:Application2@cccccccc-cccc-cccc-cccc-cccccccccc
// Classic key
//    std::string destination = "qkd:Application1@bbbbbbbb-bbbb-bbbb-bbbb-bbbb
// Raw key
std::string source = "raw:Application2@cccccccc-cccc-cccc-cccc-cccccccccccc";
// Raw key
std::string destination = "raw:Application1@bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbb

etsi_qkd_004::QoS qos = {CHUNK_SIZE, 32, 32, 0, 0, 150000, 0, ""};
auto oc_response = etsi_qkd_004::open_connect(address,
source, destination, qos);
std::cout << "OPEN CONNECT STATUS: " << oc_response.status << std::endl;

// ----- GET_KEY -----
etsi_qkd_004::KeyBuffer cumulative_key_buffer;
unsigned int index = 0;
while (index < CHUNK_NUMBER) {
std::this_thread::sleep_for(std::chrono::milliseconds(1000));
auto gk_response = etsi_qkd_004::get_key(address,
oc_response.key_stream_id, index);
if (gk_response.status == etsi_qkd_004::Status::SUCCESSFUL) {
print_key_buffer(gk_response.key_buffer);

// Extend cumulative_key_buffer
cumulative_key_buffer.reserve(cumulative_key_buffer.size() +
std::distance(gk_response.key_buffer.cbegin(),
gk_response.key_buffer.cend()));
cumulative_key_buffer.insert(cumulative_key_buffer.cend(),
 gk_response.key_buffer.cbegin(),
gk_response.key_buffer.cend());
// Increment chunk index (only if gk_response.status is SUCCESSFUL)
++index;
}
else
;//std::cout << "ERROR: Status " << gk_response.status << std::endl;
}
print_key_buffer(cumulative_key_buffer, "CUMULATIVE KEY BUFFER");
```

```
// ----- CLOSE -----
auto cl_response = etsi_qkd_004::close(address, oc_response.key_stream_id);
std::cout << "CLOSE STATUS: " << cl_response.status << std::endl;


return 0;
}
```

This example can be found at sdf/etsi_qkd_004/cpp/ .

It is important to note that the type of key requested is defined by the scheme in the URI.

To compile it use cmake:

```
sudo apt-get install uuid-dev build-essential libssl-dev
sudo snap install cmake --classic
mkdir cmake-build
cd cmake-build
cmake ..
```

1.3.15 Requesting keys

The type of keys requested are defined by the scheme in the URI. Example of QOKD URI: `"qokd:Application2@cccccccc-cccc-cccc-cccc-cccccccccccc"`.

Sample bytes received:

Alice: 10 10 10 00

Bob: 01 01 01 00

The first bit is the selection byte and the second one is the corresponding bit.

To change the role played in QOKD use the role parameter in open_connect. Example:

```
std::string role = "bob";
auto oc_response = etsi_qkd_004::open_connect(address, source, destination, qo
```

The system also provides symmetrical and raw keys:

`"qkd:Application2@cccccccc-cccc-cccc-cccc-cccccccccccc"`
`"raw:Application2@cccccccc-cccc-cccc-cccc-cccccccccccc"`

1.3.16 Changing raw bit error rate

To change the LKMS error rate edit the second parameter of LKMSSimulator on /sdf/etsi_qkd_004/lkms/run_lkms.py :

```
def run_lkms_simulator(lkms_address):
LKMSSimulator(lkms_address, 50)
```

In this example the bit error rate is set to 50

1.3.17 How to get QKD, QOKD and RAW keys

For the key delivery service to work we need two parts: the LKMS (Key Management Service) and the client. The LKMS generates a different set of keys for each KSID (with the correct type, specified by the user). To start the LKMS execute `python3 run_lkms.py` at `/sdf/etsi_qkd_004/lkms/`. If you want to change the raw key error rate open the run_lkms.py script and change the second argument of LKMSSimulator. Example of 50

```
def run_lkms_simulator(lkms_address):
LKMSSimulator(lkms_address, 50)
```

After starting the LKMS Simulator it should be ready to provide keys. To test this you can use the example cpp program or write your own. The program is located in `/sdf/etsi_qkd_004/cpp/`. If all is working then the cpp program should print out the keys provided by the LKMS. There are a couple of things that can be changed in the program:

- KSID You can provide your own KSID. In case you don't the LKMS should generate one and return it on the open_connect call.
- Type of keys The type of keys provided depends on the scheme of the source URI. Example:

  ```
  "qkd:Application2@cccccccc-cccc-cccc-cccc-cccccccccccc"
  "raw:Application2@cccccccc-cccc-cccc-cccc-cccccccccccc"
  ```

  The first one provides qkd keys and the second on raw. There are also qokd keys.
- Role of qokd keys

  To change the role (alice or bob) when getting qokd keys add a argument to the open_connect function:

  ```
  std::string role = "bob";
  auto oc_response = etsi_qkd_004::open_connect(address, source, destinatior
  , qos,"b0dd24e6-94cb-4f32-abfe-50e194783a48", role);
  ```

To compile the example you first need to install the dependencies:

```
sudo apt-get install uuid-dev build-essential libssl-dev
sudo snap install cmake --classic
```

After that go to the folder of the example:

```
mkdir cmake-build
cd cmake-build
cmake ..
make
```

1.3.18 Further Discussion

1.3.19 Open Issues

1.3.20 References

- ETSI GS QKD 004 V2.1.1 (2020-08)

Acronyms

QKD	Quantum Key Distribution	8