ETSI GS QKD 015 V1.1.1 (2021-03)

Quantum Key Distribution (QKD); Control Interface for Software Defined Networks

Disclaimer

The present document has been produced and approved by the Quantum Key Distribution (QKD) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference

DGS/QKD-015_ContIntSDN

Keywords

control interface, quantum cryptography, Quantum Key Distribution, Software-Defined Networking

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00   Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

The present document can be downloaded from:
http://www.etsi.org/standards-search

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format at www.etsi.org/deliver.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at
https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx

If you find errors in the present document, please send your comment to one of the following services:
https://portal.etsi.org/People/CommiteeSupportStaff.aspx

Copyright Notification

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.
The content of the PDF version shall not be modified without the written authorization of ETSI.
The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2021.
All rights reserved.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members.
3GPP™ and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.
oneM2M™ logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners.
GSM® and the GSM logo are trademarks registered and owned by the GSM Association.

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

Foreword

This Group Specification (GS) has been produced by ETSI Industry Specification Group (ISG) Quantum Key Distribution (QKD).

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Executive summary

The present document deals with the interface between a SDN-QKD node and a SDN controller. It describes the flow of information between both entities, The SDN-QKD node part being embodied by an SDN-Agent that collects the local node information. The information model is given in YANG [1] and [2], a language well suited and widely used for these purposes. The information model is agnostic from the vendor and the implementation, permitting to control any type of QKD systems, whilst also enabling to the centralized SDN controller to build an end-to-end view of the network for managing and optimizing the transmission of quantum signals and also to deliver the QKD-derived keys.

Introduction

Quantum Key Distribution relies on the creation, transmission and detection of signals at the quantum level. This is difficult to achieve if the network used for the transmission is also in use with classical signal, which are much more powerful. On the other hand, the quantum transmission can be neither amplified nor regenerated - at least without quantum repeaters, which are not feasible with current technology - implying a limited reach for quantum communications and the need to resort to trusted repeaters to increase the distance. To optimize the transmission of quantum signals together with classical communications - whether they share the same physical media or not - over a network and manage the key relay required for longer distances, it is necessary to integrate the QKD systems such that they can communicate with the network control and also receive commands from it. These network-aware QKD systems have to be integrated at the physical level (e.g. to allocate spectrum for the quantum channel, dynamically change the peer, or use a new optical path, etc.), but also logically connected to the management architectures. To achieve this integration, the required capabilities of the QKD devices have to be described to the network controller. YANG [1] and [2] is the major modelling language used to describe network elements. Any new elements, services or capabilities being defined usually come together with a YANG model for enabling a faster integration into management systems.

The purpose of the information model presented in the present document, regardless of the protocol chosen to implement the control channel, is to simplify the management of the QKD resources by implementing an abstraction layer described in YANG. This will allow to optimize the creation and usage of the QKD-derived keys by introducing a central element through the SDN controller. This is a standard component of SDN networks that has a global view of the whole network. This abstraction layer will enable the SDN controller to simultaneously manage both, the classical and quantum parts of the network. The integration has the added benefit of using well-known mechanisms and tools in the classical community, which will facilitate its adoption and deployment by the telecommunications world.

1 Scope

The present document provides a definition of management interfaces for the integration of QKD in disaggregated network control plane architectures, in particular with Software-Defined Networking (SDN). It defines abstraction models and workflows between a SDN-enabled QKD node and the SDN controller, including resource discovery, capabilities dissemination and system configuration operations. Application layer interfaces and quantum-channel interfaces are out of scope.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] IETF RFC 6020 (October 2010): "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)".

[2] IETF RFC 7950 (August 2016): "The YANG 1.1 Data Modeling Language".

[3] IETF RFC 6241 (June 2011): "Network Configuration Protocol (NETCONF)".

[4] IETF RFC 8040 (January 2017): "RESTCONF Protocol".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI GR QKD 007: "Quantum Key Distribution (QKD); Vocabulary".

3 Definitions of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms apply:

NOTE: Where possible, the definitions from ETSI GR QKD 007 [i.1] are used.

entity: set of hardware, software or firmware components providing specific functionalities

QKD application: entity consuming QKD-derived keys from the key management system

NOTE: They can be either external applications (similar to SAE, see below) or internal applications running in the QKD system.

QKD-derived key: secret key derived from QKD system(s) operating QKD protocol(s) over a QKD link

QKD interface: interface that is a high-level abstraction of a QKD system

NOTE: A QKD interface defines only attributes that are relevant from the point of view of the network. These attributes are revealed to a SDN controller to establish and manage QKD.

QKD link: set of active and/or passive components that connect a pair of QKD modules to enable them to perform QKD and where the security of symmetric keys established does not depend on the link components under any of the one or more QKD protocols executed

QKD module: set of hardware, software or firmware components contained within a defined cryptographic boundary that implements part of one or more QKD protocol(s) to be capable of securely establishing symmetric keys with at least one other QKD module

QKD network: network comprised of two or more QKD nodes

QKD node: set of QKD modules installed in the same location within the same security perimeter

QKD protocol: operations on quantum and classical signals that allow two parties to agree on commonly shared bit strings between two ends of a QKD link that remain secret

QKD system: pair of QKD modules connected by a QKD link designed to provide Quantum Key Distribution functionality using QKD protocols

quantum channel: communication channel for transmitting quantum signals

Quantum Key Distribution (QKD): procedure involving the transport of quantum states to establish symmetric keys between remote parties using a protocol with security based on quantum entanglement or the impossibility of perfectly cloning the transported quantum states

SDN agent: entity that is responsible of managing one or more QKD Systems (through their respective QKD interfaces) within a secure location, abstracting the information of QKD resources under its control and communicating with a SDN controller for the QKD network

SD-QKD node: logical and abstracted representation of the QKD resources under the responsibility of a single SDN Agent

Secure Application Entity (SAE): entity that requests one or more keys from a Key Management System for one or more applications running in cooperation with one or more other Secure Application Entities

secure location: location assumed to be secured against access by adversaries

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ACK	Acknowledgement
API	Application Programming Interface
CRUD	Create, Read, Update and Delete
DoS	Denial of Service
HSM	Hardware Security Module
HTTP	Hypertext Transfer Protocol
ID	IDentifier

IETF	International Engineering Task Force
JSON	JavaScript Object Notation
NBI	North Bound Interface
NMS	Network Management System
PHYS	Physical link
QKD	Quantum Key Distribution
QoS	Quality of Service
RFC	Request For Comments
SAE	Secure Application Entity
SDN	Software-Defined Networking
SD-ONC	Software-Defined Optical Network Controller
SD-QKD	Software-Defined Quantum Key Distribution
SD-QNC	Software-Defined Quantum Network Controller
SKR	Secure Key generation Rate
SSH	Secure Shell
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
XML	Extensible Markup Language
YANG	Yet Another Next Generation

4 Software-Defined Quantum Key Distribution

4.1 Introduction

The parametrization and modelling defined in the present document relates to the management interface of QKD modules (one or multiple) that connects them to a SDN controller. The requirements for such an interface and the further integration is described as a YANG model and as associated workflows for the main functional use cases (see Annex A). This architectural design permits a controller to centrally orchestrate the QKD resources to optimize the key allocation per link based on demands and automate the creation of either direct (physically connected through an uninterrupted quantum channel) or virtual (multi-hop-based) QKD links, where the keys are relayed from one hop (direct QKD link) to the next in the chain connecting the initial with the final points. The workflows described in Annex A are thought to be implemented by using any of the well-accepted network management protocols used in SDN architectures, which are based on YANG information models for their internal data structures. However, it is out of the scope of the present document to define which specific protocol, data structures or specific implementation is chosen to carry the YANG-structured information defined in the present document. These specifics are left aside to permit some flexibility during the system design and implementation phases.

In addition, this YANG model is designed to be a base or core model for the integration of QKD technologies in operator's management architectures, but it is not closed for experimentation and for further extensions, as YANG provides such flexibility to easily integrate new capabilities inside a given model. Future revisions of the present document may include additional parameters.

4.2 SD-QKD node

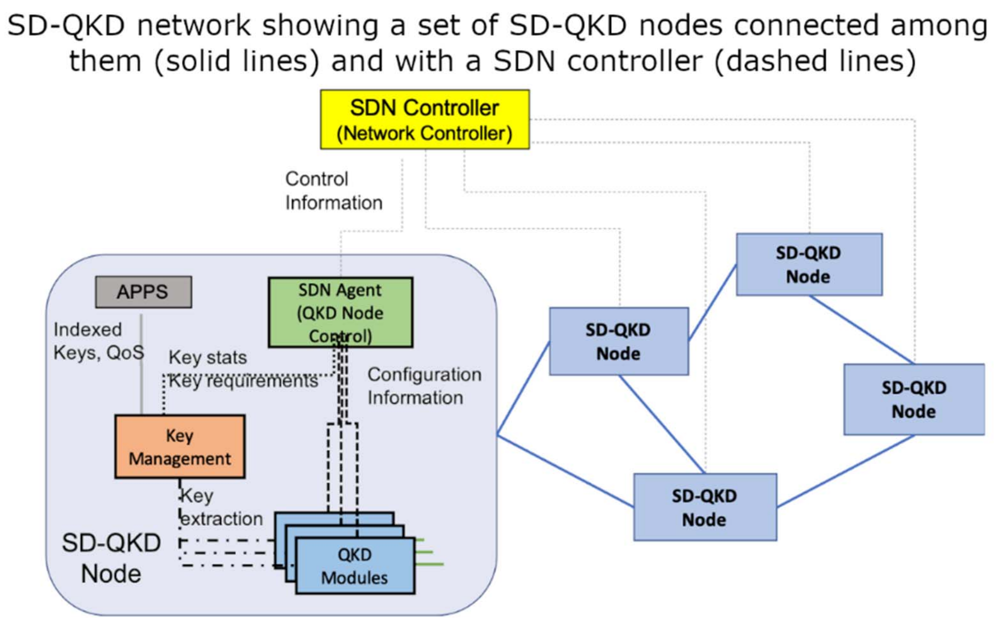
A Software-Defined Quantum Key Distribution (SD-QKD) node is an aggregation of one or multiple QKD modules that interface with a SDN controller using standard protocols (i.e. it is SDN-enabled). The connection between node and controller allows information to be retrieved from the QKD domain and dynamically and remotely configure the behaviour of the QKD systems to create, remove or update key associations (either through a quantum channel, or via multi-hop) between remote secure locations. A SD-QKD node shall be compliant with some specific requirements:

- A SD-QKD node shall comprise at least one QKD module, exposed to the controller as a QKD interface.
- It may comprise multiple QKD modules, creating an abstraction of a single node with multiple interfaces.
- It shall be located within a secure location.
- A single location may comprise one or multiple SD-QKD nodes.

- A SD-QKD node shall contain a key management system aggregating the key material from the different key associations. The key management system may be implemented using multiple logical key stores to distinguish groups of applications.
- It shall provide a single access for applications to retrieve keys from the key store via an API.
- It shall connect to (at least) one SDN controller, via standard protocols and mechanisms to enable discovery, control and telemetry and statistics streaming.
- It should expose applications information to the controller, for discovery purposes and to better optimize the utilization of QKD keys.
- It should expose QKD interface with QKD system information to the SDN controller, and allow to configure some parameters of the systems to create the quantum channel.
- It should expose information of the key associations (links) with other SD-QKD systems.
- It should expose the classical channel requirements for each of the systems within the node (e.g. attenuation, supported wavelengths, etc.).

The modelling defined in the next clauses provides an abstracted view of the QKD domain. It can abstract the QKD systems within a secure location as interfaces of a network element. This network element, the SD-QKD node, is able to communicate with its neighbours and with the central controller to create end-to-end services, or key associations. When possible (enough QKD systems, reachability over the physical media), these associations are created over a direct quantum channel. In other cases, a multi-hop link or key association is created granting a fully connected QKD network. Also, the information exchanged across the control plane is not critical (e.g. keys are not forwarded to the controller). Therefore, the introduction of the SDN paradigm for QKD networks, should not imply any further security risks different from the already known in trusted node QKD networks (e.g. DoS attacks). In particular, the aim of this abstraction model is to:

- Enable centralized management of the QKD resources based on demands, capabilities and network (quantum and classical) status.
- Aggregate different QKD systems within a secure perimeter under a single key management, to better detect demands and provision the necessary links or key associations.
- Reduce the complexity of operating separately all the QKD modules within a secure location and handling statistics from the QKD systems.
- Abstract the complexity of managing low level parameters and behaviour of each QKD system and technology, as each node can take the responsibility of low-level configurations.
- Optimize the configuration and the distribution of QKD links in the QKD network to cope with all demands, based on application's QoS information and generation rate statistics of each link.
- Coordinate quantum and classical channels (the configuration of the optical network), whether they share the same physical media or not, to enhance the performance of the QKD systems.


SD-QKD network showing a set of SD-QKD nodes connected among them (solid lines) and with a SDN controller (dashed lines)
SDN Controller (Network Controller)
Control Information
APPS
SDN Agent (QKD Node Control)
Indexed Keys, QoS
Key stats
Key requirements
Configuration Information
Key Management
Key extraction
QKD Modules
SD-QKD Node
SD-QKD Node
SD-QKD Node
SD-QKD Node
SD-QKD Node

NOTE: The SD-QKD Nodes are connected among them (solid lines) and with the SDN controller (dashed lines). One of the nodes is detailed to show a typical set of components and the type of information that flows among them. In particular the SDN Agent that connects the node with the SDN controller is shown. The present document deals with this connection. See the text for additional information.

Figure 1: Depiction of a SD-QKD network showing a set of SD-QKD nodes

Figure 1 shows, in a high-level design, a SD-QKD network as a set of connected nodes under the control of the SDN controller. One of the nodes is shown in more detail with the fundamental components which are required to build a SD-QKD node in order to illustrate the typical flow of information between components. Note that the figure is for illustrative purposes and does not imply a mandatory node structure. The Applications are included as part of the node to illustrate that they are contained in the same security perimeter. At the hardware level, the SD-QKD system shall comprise at least one QKD module (in the example figure, there are three modules). These modules are used to physically connect the SD-QKD node to other remote peers through a quantum link, composing a QKD system for key generation purposes (note that the scheme can be easily extended to include other services allowed by the quantum device, like entanglement distribution). The generated keys are pushed (or extracted) to a key management system, which is responsible for maintaining and distributing them. The key management system registers incoming applications and their QoS, and monitors the real demands of each of them. It also exposes the parameters needed to monitor the utilization of the QKD-derived keys for each link. This information allows to optimize the planning of the QKD network.

The following clauses describe the different data structures (YANG grouping) to be handled by the SD-QKD node and the SDN controller. The YANG data model for the SD-QKD node is divided in four main structures (groupings): SD-QKD node capabilities, QKD applications, QKD links (or key associations) and QKD interfaces (or systems). In addition, YANG notifications are also included for server (node) to client (controller) communications.

4.3 SD-QKD node capabilities

The SD-QKD node capabilities structure contains essential parameters to expose to the SDN controller its support for some basic functionalities. An example is the capability (or policy) of exporting statistics about the key usage, or if the node is allowed (capable) of forwarding keys (key relay) in a multi-hop environment. Other submodules could also include their own capabilities, while this clause just refers to the capabilities of the node as a whole.

Table 1: SD-QKD node capabilities

Name	Type	Details	Description
link_stats_support	boolean	Default: true	If true, this node exposes link-related statistics (secure key generation rate-SKR, link consumption, status, QBER).
application_stats_support	boolean	Default: true	If true, this node exposes application related statistics (application consumption, alerts).
key_relay_mode_enable	boolean	Default: true	QKD node support for key relay mode services.

4.4 QKD Interfaces

As described in the introductory clause, QKD interfaces are an abstraction of the QKD systems which are contained within a secure location as part of a SD-QKD nodes. This abstraction allows a SDN controller to identify all the QKD systems within a location and aggregate them as a single network element with multiple interfaces (e.g. as a switch or a router, with very different capabilities).

Due to interoperability issues, the current version of the model shall specify the QKD technology implemented by the device and the vendor and model, as mix-matching different QKD modules in the current state of development will lead to inoperative links with no key generation.

The QKD interfaces within a SD-QKD node shall include the following parameters:

Table 2: Parameters of QKD interfaces

Name	Type	Details	Description
qkdi_id (interface ID)	ietf_yang_types:uuid	None	Interface id. It is described as a locally unique number, which is globally unique when combined with the SD-QKD node ID.
capabilities	container	None	Capabilities of the QKD system (interface).
capabilities/ role_support	etsi-qkdn-types: QKD-ROLE-TYPES	None	QKD node support for key relay mode services.
capabilities/ wavelength_range	etsi-qkdn-types: QKD-ROLE-TYPES	None	Range of supported wavelengths (nm) (multiple if it contains a tunable laser).
capabilities/ max_absorption	uint32	None	Maximum absorption supported (in dB).
model	string	None	Device model (vendor/device).
type	etsi-qkdn-types: QKD-TECHNOLOGY-TYPES	None	Interface type (QKD technology).
att_point	container	None	Interface attachment point to an optical switch.
att_point/ device	string	None	Unique ID of the optical switch (or passive component) to which the interface is connected.
att_point/ port	uint32	None	Port ID from the device to which the interface is connected.

4.5 QKD Key Association Links

A QKD Key Association Link is a logical key association between two remote SD-QKD nodes. These links associations can be of two different types: direct (also called physical), if there is a direct quantum channel through which keys are generated i.e. a physical QKD link connecting the pair of QKD modules, or virtual if keys are forwarded (key relay) through several SD-QKD -trusted- nodes to form an end-to-end key association. i.e. there is no direct quantum channel connecting the end points, and a set of them have to be concatenated such that for each a secret key is produced and then used to relay a key from the initial to the end point in a multi-hop way.

Any new key association link created in a SD-QKD node has to be tracked, labelled and isolated from other links. Virtual links are also registered as internal applications, as they make use of QKD-derived keys from other QKD key association links for the key transport.

The information exported to the SDN controller should be kept minimal but sufficient to allow analysis and optimization of the deployed links and applications, while other crucial information (e.g. the QKD-derived keys) shall be kept within the SD-QKD node security perimeter. The parameters that define a QKD key association link within the SD-QKD node abstraction are:

Table 3: Parameters of a QKD key association link

Name	Type	Details	Description
link_id	ietf_yang_types:uuid	None	Unique ID of the QKD link (key association).
enable	boolean	Default true	This value allows to enable of disable the key generation process for a given link.
local/qkd_node	ietf_yang_types:uuid	None	Unique ID of the local SD-QKD node.
local/interface	uint32	None	Interface used to create the key association link.
remote/qkd_node	ietf_yang_types:uuid	None	Unique ID of the remote QKD node. This value is provided by the SDN controller when the key association link request arrives.
remote/interface	uint32	None	Interface used to create the link.
type	etsi-qkdn-types: QKD-LINK-TYPES	None	Key Association Link type: Virtual (multi-hop) or Direct.
state	etsi-qkdn-types: LINK-STATUS-TYPES	None	Status of the QKD key association link.
applications	List: ietf_yang_types:uuid	None	Applications which are consuming keys from this key association link.
prev_hop	ietf_yang_types:uuid	(if type=VIRTUAL)	Previous hop in a multihop/virtual key association link config.
next_hop	Leaf-list: ietf_yang_types:uuid	(if type=VIRTUAL)	Next hop(s) in a multihop/virtual key association link config. Defined as a list for multicast over shared sub-paths.
bandwidth	uint32	(if type=VIRTUAL)	Required bandwidth (in bits per second) for that key association link. Used to reserve bandwidth from the physical QKD links to support the virtual key association link as an internal application.
channel_att	uint8	(if type=PHYS)	Expected attenuation on the quantum channel (in dB) between the Source/qkd_node and Destination/qkd_node.
wavelength	etsi-qkdn-types: wavelength	(if type=PHYS)	Wavelength (in nm) to be used for the quantum channel. If the interface is not tunable, this configuration could be bypassed.
qkd_role	etsi-qkdn-types: QKD-ROLE-TYPES	(if type=PHYS)	Transmitter/receiver mode for the QKD module. If there is no multi-role support, this could be ignored.
Performance/ expected_consumption	uint32	Config false	Sum of all the application's bandwidth (in bits per second) that are on this particular key association link.
Performance/skr	uint32	Config false	Secret key rate generation (in bits per second) of the key association link.
Performance/eskr	uint32	Config false	Effective secret key rate (in bits per second) generation of the key association link available after internal consumption.
Performance/ phys_perf	list	(if type=PHYS) Config false Key "type";	List of physical performance parameters.
Performance/ phys_perf/ type	etsi-qkdn-types: PHYS-PERF-TYPES	(if type=PHYS) config false;	type of the physical performance value to be exposed to the controller.

Name	Type	Details	Description
Performance/ phys_perf/ value	uint32	(if type=PHYS) config false;	Numerical value for the performance parameter type specified above.

4.6 QKD Applications

From the perspective of the SD-QKD node, a QKD application is defined as any entity requesting QKD-derived keys from the key manager within the node. These applications might be external (e.g. an end-user application, a Hardware Security Module (HSM), a virtual network function, an encryption card, security protocols, etc.) or internal (keys used for authentication, to create a virtual link - for key transport, like e.g. a forwarding module, etc.). From the software perspective, an application is a concrete running instance or process consuming keys at a given point of time. A single instance or process may also require to open different isolated sessions (with a different unique ID) with the SD-QKD node.

The SDN controller takes two roles related to application management: the first one is to register any incoming application and, more specifically, their QoS requirements to better optimize the usage of the QKD network based on the real demands; the second is to handle the complexity of finding the remote SD-QKD peer node for QKD applications. Forcing applications to know about the QKD network (or to exchange information about their local QKD nodes/systems) brings an undesirable complexity to the application layer. To avoid such situations, the SDN controller acts as a central repository where new applications are registered and where SD-QKD nodes can access to find the peer SD-QKD node for a given application. Any application just knows where its node's key manager is located (ip/port), while the control plane will take care of registering and coordinating new applications (see clause 5.2).

The set of parameters that compose the QKD applications structure is as follows.

Table 4: Parameters of a QKD application

Name	Type	Details	Description
app_id	ietf_yang_types:uuid	None	This value uniquely identifies a pair of applications extracting keys from a QKD key association link. This value is similar to a key ID or key handle.
QoS	Container	None	Requested Quality of Service.
QoS/max_bandwidth	uint32	None	Maximum bandwidth (in bits per second) allowed for this specific application. Exceeding this value will raise an error from the local key store to the appl. This value might be internally configured (or by an admin) with a default value.
QoS/min_bandwidth	uint32	None	This value is an optional QoS parameter which enables to require a minimum key rate (in bits per second) for the application.
QoS/jitter	uint32	None	This value allows to specify the maximum jitter (in msec) to be provided by the key delivery API for applications requiring fast rekeying. This value can be coordinated with the other QoS to provide a wide enough QoS definition.
QoS/ttl	uint32	None	This value is used to specify the maximum time (in seconds) that a key could be kept in the key store for a given application without being used.
QoS/ clients_shared_path_enable	boolean	Default false	If true, multiple clients for this application might share keys to reduce service impact (consumption).
QoS/ clients_shared_keys_required	boolean	Default false	If true, multiple clients for this application might share keys to reduce service impact (consumption).

Name	Type	Details	Description
type	etsi-qkdn-types: QKD-APP-TYPES	None	Type of the registered application. These values, defined within the types module, can be client (if an external application is requesting keys) or internal (if the application is defined to maintain the QKD - e.g. multi-hop, authentication or other encryption operations).
server_app_id	inet:URI	None	ID of the server application connecting to the local key server.
client_app_id	inet:URI	None	ID of the client application connecting to the local key server of the peer SD-QKD node.
backing_link_id	Leaf-list: ietf_yang_types:uuid	None	Unique ID of the key association link which is providing QKD keys to these applications.
local_qkdn_id	ietf_yang_types:uuid	None	Unique ID of the local SD-QKD node which is providing QKD keys to the local application.
remote_qkdn_id	ietf_yang_types:uuid	None	Unique ID of the remote SD-QKD node which is providing QKD keys to the remote application. While unknown, the local SD-QKD will not be able to provide keys to the local application.
priority	uint32	default 0	Priority of the association/application this might be defined by the user, but usually handled by a network administrator.
creation_time	date-and-time	Config false	Date and time of the service creation.
expiration_time	date-and-time	None	Date and time of the service expiration.
Statistics/statistic	Container/list	None	
Statistics/statistic end_time	date-and-time	Config false	End time for the statistic period.
Statistics/statistic start_time	date-and-time	Config false	Start time for the statistic period.
Statistics/statistic consumed_bits	uint32	Config false	Consumed secret key amount (in bits) for a given period of time.

4.7 Notifications

YANG notifications are used to allow device to controller communications, usually to expose information based on changes or by-time subscriptions. Within YANG notifications, two ways of exposing data have been identified: the first one, based on ad-hoc and very specific notifications, which have to be appropriately modelled using YANG constructs; and the second one, where notifications are generically structured and then exposed as topic-based subscriptions. The approach described in the present document specifies a minimal set of base notifications for application, interface and link management while, with the purpose of providing a structure for new events to be integrated, a generic structure for events and alarms is also included. Such generic structure is divided in three main categories of elements within the model: applications, links and interfaces.

The specific notifications considered so far have also being structured by applications, links and interfaces. The list of defined notifications is as follows:

Applications:

- *sdqkdn_application_new*: Defined for the controller to detect new applications requesting keys from a QKD node. This maps with the workflow shown in clause 5.2 "QKD Application Registration". Parameters such as client and server app IDs, local QKD node identifier, priority and QoS are sent in the notification.

- *sdqkdn_application_qos_update*: Notification that includes information about priority or QoS changes on an existing and already registered application.

- *sdqkdn_application_disconnected*: Includes the application identifier to inform that the application is no longer registered and active in the QKD node.

Interfaces:

- *sdqkdn_interface_new*: Includes all the information about the new QKD system installed in the secure location of a given QKD node.
- *sdqkdn_interface_down*: Identifies an interface within a QKD node which is not working as expected, allowing additional information to be included in a "reason" string field.
- *sdqkdn_interface_out*: Contains the ID of an interface which is switch off and uninstall from a QKD node. This information can be gathered from this notification or from regular polling from the controller's side.

Links:

- *sqdkdn_link_down*: As in the interface down event, this notification contains the identifier of a given link which has gone down unexpectedly. In addition, further information can be sent in the "reason" field.
- *sqdkdn_link_perf_update*: This notification allows to inform of any mayor modification in the performance of an active link. The identifier of the link is sent together with the performance parameters of the link.
- *sqdkdn_link_overloaded*: This notification is sent when the link cannot cope with the demand. The link identifier is sent with the expected consumption and general performance parameters.

In addition to the specific and generic notifications described above, most popular network protocols based on YANG allow to subscribe to any information within the node's datastore. In this sense, any of the parameters subject to configuration or operational changes can be informed in real time, if the selected protocol used together with the YANG model has this capability. Clause 6 describes the most popular management protocols used in the networking area.

5 Sequence Diagrams and Workflows

5.1 Introduction

This clause provides an overview of some fundamental use cases that are meant to be handled by a SD-QKD network. For each use case, this clause includes the workflow (shown as a sequence diagram) to show the interactions and exchange of information between the SD-QKD node and the SDN controller (SD-QNC). The base use cases identified relate to the management of incoming requests (new applications requesting keys) from the network perspective and the creation of new key associations, either virtual or physical.

5.2 QKD Application Registration

New applications requesting QKD-derived keys from the SD-QKD node's key manager shall be registered in both, the local node and the SDN controller. This allows the SDN controller to keep track of the real-time key consumption as well as to have an estimate of the maximum key rate reserved for each link at a time. These values are then used to appropriately plan new links in the SD-QKD network.

Similarly, applications aiming to use QKD-derived keys shall know about their node's key manager endpoint (ip/port), but they should not be required to know about any other node within the QKD network (not even the ID of the SD-QKD node within the domain of the peer application). For that reason, this clause provides a detailed sequence diagram and workflow description on how to register incoming applications within the context of a SD-QKD network.

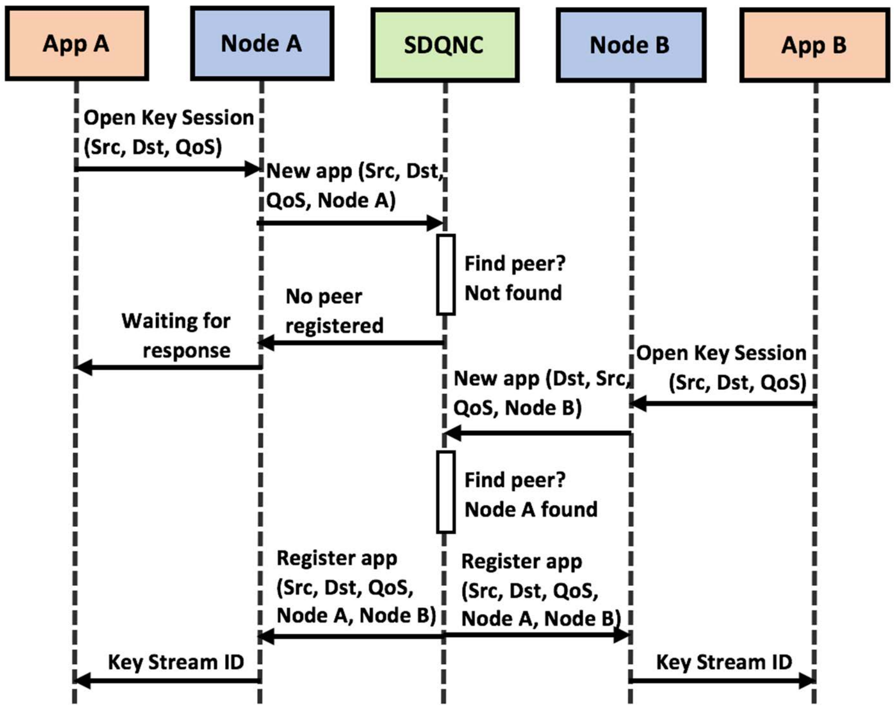

App A
Node A
SDQNC
Node B
App B
Open Key Session (Src, Dst, QoS)
New app (Src, Dst, QoS, Node A)
Find peer? Not found
No peer registered
Waiting for response
Open Key Session (Src, Dst, QoS)
New app (Dst, Src, QoS, Node B)
Find peer? Node A found
Register app (Src, Dst, QoS, Node A, Node B)
Register app (Src, Dst, QoS, Node A, Node B)
Key Stream ID
Key Stream ID

Figure 2: Sequence diagram of the applications registration in a SD-QKD network

Figure 2 shows the exchange of messages between application and their local SD-QKD nodes, as well as the SD-QKD nodes with the SDN controller. The workflow is as follows:

- Initially, an application within the security domain A connects to the local SD-QKD node (key manager) via key delivery API. In this connection request, the application sends its own unique ID, the unique ID of the peer application and some QoS requirements.
- The SD-QKD node, as it does not have information about the peer application or its SD-QKD node (remote), it informs the SDN controller of this new request, forwarding the same information as in the application request, including the local SD-QKD node ID.
- If this pair of applications are not yet registered in the SDN controller, the controller sends back an acknowledgement, but without further information of the peer SD-QKD node.
- In the security domain B, the peer application connects to the SD-QKD node's key manager to start gathering QKD-derived keys.
- The SD-QKD node at B does not have information of the peer application or SD-QKD node, so it informs the SDN controller of the incoming application and its requirements.
- The SDN controller detects the application already registered, creates a unique ID for the service (key ID), and it sends to both SD-QKD nodes (A and B) all the necessary information to configure the application at each endpoint (applications and nodes IDs, the unique key ID and the QoS requirements).
- The nodes are now capable of sending the unique key ID to the applications and reserve keys for this application. Applications will use this unique ID to further request keys from their local key stores.

5.3 QKD Physical (Direct) Link creation

The integration of existing QKD systems in communications networks is currently a complex task. New QKD systems require to be manually installed, usually over an exclusive use dark fibre in the shape of a fixed QKD link connecting two sites. Network operators have already started to analyse different integration strategies for the QKD resources. Some of these strategies directly study the costs of deploying a completely parallel and isolated QKD network. On the other hand, the advent of disaggregation for optical networks and the already well-studied (and partially adopted by operators) software-defined optical network architectures are key technologies that enable high configurability in the optical domain. These technologies are the base to bring coexistence of classical and quantum signals and dynamically manage the creation of classical and quantum channels integrated in the same network and even sharing the same physical media.

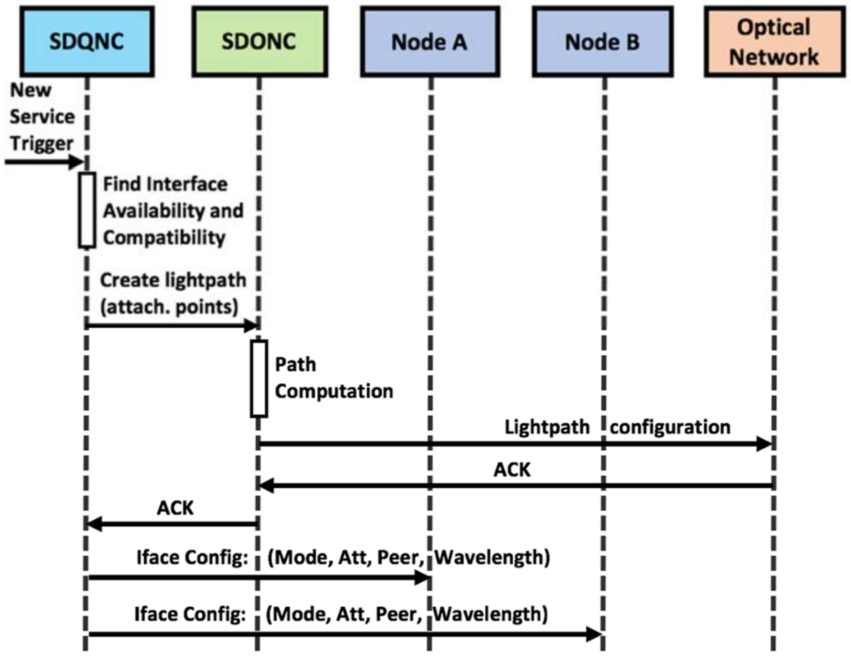

SDQNC
SDONC
Node A
Node B
Optical Network
New Service Trigger
Find Interface Availability and Compatibility
Create lightpath (attach. points)
Path Computation
Lightpath configuration
ACK
ACK
Iface Config: (Mode, Att, Peer, Wavelength)
Iface Config: (Mode, Att, Peer, Wavelength)

Figure 3: Sequence diagram for the dynamic creation of a physical QKD link

The workflow description associated with the physical QKD link creation sequence diagram is as follows:

- Initially, the software-defined QKD network controller (SD-QNC) receives a trigger, either from the detection of key demands from the nodes, or from the controller's North Bound Interface (NBI).
- The SD-QNC finds from the two locations two available and compatible interfaces to be the endpoints of the quantum channel.
- After that, the SD-QNC sends a request for light-path creation to the software-defined optical network controller (SD-ONC), with the specific constraints of the quantum channel. If both controllers are integrated in one, this operation is internally handled.
- If the path computation succeeds, the SD-ONC proceeds to configure the optical devices and creates the light-path.
- When the SD-QNC receives an ACK for the light-path creation, it connects to both SD-QKD nodes to configure each involved interface, including the characteristics of the optical channel.

5.4 QKD Virtual Link creation

A virtual link is defined as a key association between two SD-QKD nodes which are connected through multiple SD-QKD nodes in relay mode and multiple physical QKD links (multi-hop). This key association acts as an internal application which consumes QKD-derived keys for the end-to-end key transport operations, as well as a new link providing keys for the two remote endpoints. If this type of operations is not managed appropriately, the QKD network will not perform optimally and virtual links could potentially end up causing congestion over other physical connections.

A SDN controller collecting performance information of the QKD network can help mitigating this issue, as it has a view of the QKD network as a whole, keeping track of consumption demands, key store availability and generation rates. When a new trigger or demand is received by the controller (as in the previous clause) it is the responsibility of the SDN controller to allocate the new optimal path over existing links to reduce the impact over other running applications and key associations.

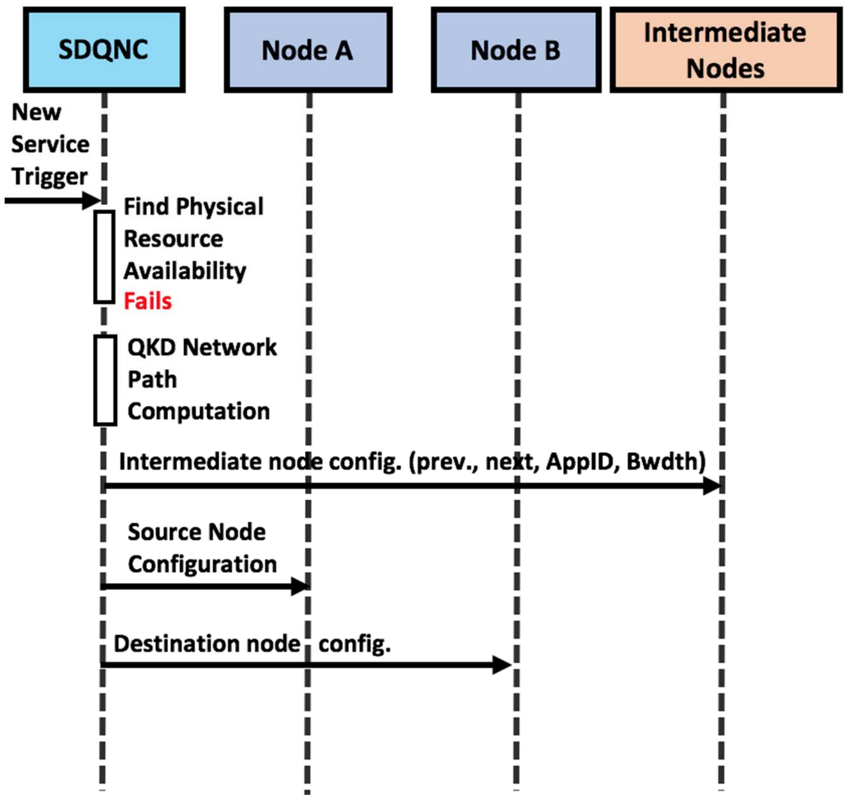

SDQNC
Node A
Node B
Intermediate Nodes
New Service Trigger
Find Physical Resource Availability Fails
QKD Network Path Computation
Intermediate node config. (prev., next, AppID, Bwdth)
Source Node Configuration
Destination node config.

Figure 4: Sequence diagram for the dynamic creation of a virtual (multi hop) link

The workflow associated to the creation of a virtual QKD link is as follows:

- Initially, the SDN controller will receive a trigger (initiated by a network administrator or based on the demands).
- The SD-QNC fails on finding available resources, either from the SD-QKD node side (not available and compatible interfaces) or from the optical network side (there is not an available light-path fulfilling the quantum channel requirements).
- The SD-QNC computes a path over the existing QKD network e.g. based on a weighted graph built using the key availability, key generation rates and consumption for each link.
- If succeed, the SD-QNC sends the configuration to each intermediate node in key relay mode informing of the previous and next nodes in the path. If there are multiple links, the controller could also specify the link to be used for the relay.
- The controller finally configures the endpoints of the virtual link/key association, notifying the previous or next hop, the bandwidth to be provided and other optional information.

Note that in figure 4 some messages, such as acknowledgements, are not displayed to improve readability and to have a clearer image.

6 Security considerations

The process of installing new QKD systems, regardless of having a SDN management layer or not, requires some initial steps to be taken for authenticating and have initial security schemes guaranteeing that the introduction of the new systems does not break the security (e.g. a trusted identity, pre-shared secret key, used for a short period of time and just during installation, etc.). These schemes should follow the standard security procedures used in this type of installation. After this process, any further communication can then be secured using the QKD-derived keys.

Assuming that the SDN controller for the QKD network is within a secure location, the installation of a new QKD system (from the abstraction perspective, a new interface) will have to follow a similar procedure. In this sense, two different situations are possible:

- A new system being installed in an existing secured area: this case is simple, as any channel between controller and systems can be secured using QKD-derived keys from other existing links.
- A new secure location is integrated in the SDN network: this procedure will need a security scheme to present the new local SDN Agent to the SDN controller such that the rest of the network would trust in it. As mentioned previously, there exist mechanisms in conventional security to perform this task. Once a first authentication has been performed QKD-derived keys can be used for further authentication and encryption.

Despite of the considerations described above, the description of any particular security implication and the associated solution for each one is out of the scope of the present document. This clause aims to clarify that any security scheme valid for the installation of QKD systems in any network or as a standalone pair is also valid for a software-defined quantum key distribution network.

7 Protocol Considerations

The goal of using YANG for data modelling is double: firstly, it is positioned as the main modelling language for network elements, systems and services while the main network control plane protocols already use it to structure their internal data; secondly, it is easy to define, read and extend base on the needs from new technologies and services. The YANG modules in the present document (Annex A) use YANG version 1.0 [1] and are checked for compatibility with version 1.1 [2].

The main two protocols being used for network management that are based on YANG are:

- RESTCONF (IETF RFC 8040) [4]: usually used at the SDN controller and other platforms' northbound interface (NBI), it is based on HTTP in order to implement CRUD operations (create, read, update and delete) over data defined in YANG. Any call to this interface requires, among other parameters or headers, an URL and a body (usually encoded in JSON or XML).
- NETCONF (IETF RFC 6241) [3]: in this case, it is usually implemented between a network element and a SDN controller or Network Management System (NMS). Its encoding is usually XML (it also accepts JSON) and the protocol is based on transactions, while the transport is usually over secure protocols, such as SSH or TLS.

The current definition of each of these protocols allows to manipulate the data defined via YANG and stored in the devices' datastore in different ways. Also, additional capabilities of these protocols (e.g. subscription not just to notifications but to any modification on configuration or operational data within the datastore) are being revised and developed as standards within the IETF.

Annex A (normative): SD-QKD node YANG Model

A.1 ETSI QKD SDN node module

A file containing the YANG module in this clause is also available at the following URL: https://forge.etsi.org/rep/qkd/gs015-ctrl-int-sdn/blob/v1.1.1/etsi-qkd-sdn-node.yang.

```
/* Copyright 2021 ETSI
Licensed under the BSD-3 Clause (https://forge.etsi.org/legal-matters) */


module etsi-qkd-sdn-node {

  yang-version "1";

  namespace "urn:etsi:qkd:yang:etsi-qkd-node";

  prefix "etsi-qkdn";

  import ietf-yang-types { prefix "yang"; }
  import ietf-inet-types { prefix "inet"; }
  import etsi-qkd-node-types { prefix "etsi-qkdn-types"; }

  // meta
  organization "ETSI ISG QKD";

  contact
    "https://www.etsi.org/committee/qkd
    vicente@fi.upm.es";

  description
    "This module contains the groupings and containers composing
    the software-defined QKD node information models
    specified in ETSI GS QKD 015 V1.1.1";

  revision "2020-09-30" {
    description
      "First definition based on initial requirement analysis.";
  }

  container sdqkd_node {
    description
      "Top module describing a software-defined QKD node (SD-QKD node).";

    leaf node_id {
      type yang:uuid;
      mandatory true;
      description
        "This value reflects the unique ID of the SD-QKD node.";
    }

    leaf location_id {
      type string;
      default "";
      description
        "This value enables the location of the secure
        area that contains the SD-QKD node to be specified.";
    }

    container qkd_capabilities {
      uses qkd_capabilities_top;
      description "Capabilities of the SD-QKD node.";
    }

    container qkd_applications {
      uses qkd_applications_top;
      description "List of applications that are currently registered
        in the SD-QKD node. Any entity consuming QKD-derived keys (either
        for internal or external purposes) is considered an application.";
    }

    container qkd_interfaces {
      uses qkd_interfaces_top;
      description "List of physical QKD modules in a secure location,
        abstracted as interfaces of the SD-QKD node.";
    }

    container qkd_links {
      uses qkd_links_top;
      description "List of (key association) links to other SD-QKD nodes in the network.
        The links can be physical (direct quantum channel) or virtual multi-hop
        connection doing key-relay through several nodes.";
    }
  }

  grouping qkd_capabilities_top {

      leaf link_stats_support {
        type boolean;
        default true;
        description
          "If true, this SD-QKD node exposes link-related statistics (key
          generation rate, link consumption, status).";
      }

      leaf application_stats_support {
        type boolean;
        default true;
        description
          "If true, the SD-QKD node exposes application related
          statistics (application consumption, alerts).";
      }

      leaf key_relay_mode_enable {
        type boolean;
        default true;
        description
          "If true, the SD-QKD node supports key relay (multi-hop) mode services.";
      }

  }

  grouping qkd_applications_top{

      list qkd_application {
        key "app_id";

        leaf app_id {
          type yang:uuid;
          description
            "This value uniquely identifies a pair of applications
            extracting keys from a QKD key association link. This value is similar
            to a key ID or key handle.";
        }

        container qos {

          leaf max_bandwidth {
            type uint32;
            description "Maximum bandwidth (bps) allowed for this specific
              application. Exceeding this value will raise an error
              from the local key store to the application. This value might
              be internally configured (or by an administrator).";
          }

          leaf min_bandwidth {
            type uint32;
            description "This value is an optional QoS parameter
              indicating the minimum key rate that the application is expected
              to request (in bits per second).";
          }

          leaf jitter {
            type uint32;
            description "This value allows a maximum time
              jitter (msec) of keys provided by the key delivery API to be specified, for
              applications requiring fast rekeying. This value can
              be coordinated with the other QoS to provide a wide
              enough QoS definition.";
          }

          leaf ttl {
            type uint32;
            description "Time To Live (sec) specifies the maximum
              time a key can have been stored before delivery to
              an application.";
          }

          leaf clients_shared_path_enable {
            type boolean;
            default false;
            description "If true, multiple clients for this application
              might share paths across the network (rather than having
              disjoint paths) to reduce service impact (consumption).";
          }

          leaf clients_shared_keys_required {
            type boolean;
            default false;
            description "If true, multiple clients for this application
              might share keys to reduce service impact (consumption).";
          }

        }

        leaf type {
          type identityref {
            base etsi-qkdn-types:QKD-APP-TYPES;
          }
          description "Type of the registered application. These
            values, defined within the types module, can be client
            (external applications requesting keys)
            or internal (application is defined to maintain
            the QKD network - e.g. multi-hop, authentication or
            other encryption operations).";
        }

        leaf server_app_id {
          type inet:uri;
          description "ID of the server application connecting to
            the key store.";
        }

        leaf-list client_app_id {
          type inet:uri;
          description "ID of the client application connecting to
            the key server of the peer SD-QKD node. It is considered
            as the client from the security perspective.";
        }

        leaf-list backing_link_ids {
          type yang:uuid;
          description "Universally Unique IDs of the (key association) links providing
            QKD keys to these applications.";
        }

        leaf local_qkdn_id {
          type yang:uuid;
          description "Universally Unique ID of the local SD-QKD node that
            is providing QKD keys to the local application.";
        }

        leaf remote_qkdn_id {
          type yang:uuid;
          description "Universally Unique ID of the remote SD-QKD node that
            is providing QKD keys to the remote application. While
            unknown, the local SD-QKD will not be able to provide
            keys to the local application.";
        }

        leaf priority {
          type uint32;
          default 0;
          description "Priority of the association/application.
            High value indicate high priority. This might be defined by
```

```
            the user, but will usually assigned by a network administrator.";
        }

        leaf creation_time {
          config false;
          type yang:date-and-time;
          description "Date and time of the service creation.";
        }

        leaf expiration_time {
          type yang:date-and-time;
          description "Date and time of the service expiration.";
        }

        container statistics {

          description "Statistical information relating to a
            specific statistic period of time.";

          list statistic {

            key "end_time";
            config false;

            leaf end_time {
              type yang:date-and-time;
              config false;
              description "End time for the statistics collection period.";
            }

            leaf start_time {
              type yang:date-and-time;
              config false;
              description "Start time for the statistics collection period.";
            }

            leaf consumed_bits {
              type uint32;
              config false;
              description "Consumed secret key amount (bits)
                for a statistics collection period of time.";
            }

          }
        }
      }
  }

  grouping qkd_interfaces_top {
      list qkd_interface {
        key "qkdi_id";
        description "QKD Interface ID.";

        uses qkd_interface_common;

      }
  }

  grouping qkd_interface_common {

    leaf qkdi_id {
      type uint32;
      description "Interface ID. A locally unique number, which
        is globally unique when combined with the SD-QKD node ID.";
    }

    container capabilities {
      description "Capabilities of the QKD system (interface).";

      leaf role_support {
        type identityref {
          base etsi-qkdn-types:QKD-ROLE-TYPES;
        }
        description "Support for transmit, receive or both.";
      }


      leaf wavelength_range {
        type etsi-qkdn-types:wavelength-range-type;
        description "Range of supported wavelengths (nm) (multiple
          if it contains a tunable laser).";
      }

      leaf max_absorption {
        type uint32;
        description "Maximum absorption supported (dB).";
      }
    }

    leaf model {
      type string;
      description "Device model (vendor/device).";
    }

    leaf type {
      type identityref {
        base etsi-qkdn-types:QKD-TECHNOLOGY-TYPES;
      }
      description "Interface type (QKD  technology).";
    }

    container att_point {
      description "Interface attachment point to an optical switch.";

      leaf device {
        type string;
        description "Unique ID of the optical switch (or
        passive component) to which the interface is connected.";
      }

      leaf port {
        type uint32;
        description "Port ID of the device to which the interface
        is connected.";
      }
    }
  }

  grouping qkd_links_top {

    list qkd_link {
      key "link_id";

      leaf link_id{
        type yang:uuid;
        description "Universally Unique ID of the QKD link (key association).";
      }

      leaf enable {
        type boolean;
        default true;
        description "This value allows the key generation process for a
          given link to be enabled or disabled. If true, the key generation
          process is enabled.";
      }

      container local {
        description "Source (local) node of the QKD link.";

        leaf qkd_node {
          type yang:uuid;
          description "Universally Unique ID of the local qkd-node.";
        }

        leaf interface {
          type uint32;
          description "Interface used to create the key association link.";
        }
      }

      container remote {
        description "Destination (remote) unique QKD node ID.";

        leaf qkd_node {
          type yang:uuid;
          description "Universally Unique ID of the remote QKD node.
          This value is provided by the SDN controller as part
          of the request to establish the key association link.";
      }

      leaf interface {
        type uint32;
        description "Interface used to create the key association link.";
      }
    }

    leaf type {
      type identityref {
        base etsi-qkdn-types:QKD-LINK-TYPES;
      }
      description "Link type: Virtual (multi-hop) or physical.";
    }

    leaf state {
      type identityref {
        base etsi-qkdn-types:LINK-STATUS-TYPES;
      }
      description "Status of the QKD key association link.";
    }

    leaf-list applications {
      type yang:uuid;
      description "Universally Unique IDs of Applications which
        are consuming keys from the QKD key association link.";
    }

    uses virtual_link_spec {
      when "../type = 'VIRT'" {
        description "Virtual key association link specific configuration.";
      }
    }

    uses physical_link_spec {
      when "../type = 'PHYS'" {
        description "Physical key association link specific configuration.";
      }
    }

    container performance {

      uses common_performance;

      uses physical_link_perf {
        when "../../type = 'PHYS'" {
          description "Performance of the specific physical link.";
        }
      }
    }

  }
}

grouping common_performance {

  leaf expected_consumption {
      type uint32;
      description "Sum of the bandwidths (bps) of the application's that are
        consuming keys from the key association link.";
  }

  leaf skr {
      type uint32;
      description "Total secret key rate (bps) generation of the key association
         link.";
  }

  leaf eskr {
      type uint32;
      description "Effective secret key rate (bps) generated by the key association
         link. i.e. the available rate to the applications after internal
         consumption.";
  }

}
```

```
grouping physical_link_perf {

  list phys_perf {
    key "type";

    leaf type {
      type identityref {
          base "etsi-qkdn-types:PHYS-PERF-TYPES";
        }
        description "Type of the physical performance value to be
          exposed to the controller.";
    }

    leaf value {
      type uint32;
      description "Numerical value for the performance parameter.";
    }
  }

}

grouping virtual_link_spec {

    leaf prev_hop {
      type yang:uuid;
      description "Universally Unique ID of the previous hop in
          a multi-hop/virtual link configuration.";
    }

    leaf-list next_hop {
      type yang:uuid;
      description "Universally Unique IDs of the Next hop(s) in
          a multi-hop/virtual key association link configuration. List will
          contain multiple entries for multicast over shared sub-paths.";
    }

    leaf bandwidth {
      type uint32;
      description "Required bandwidth (bps)for the key association link. Used to
        reserve bandwidth from the physical link to support the virtual link
        as an internal application.";
    }
}

grouping physical_link_spec {

    leaf channel_att {
      type uint8;
      description "Expected attenuation on the quantum channel (dB).";
    }


    leaf wavelength {
      type etsi-qkdn-types:wavelength;
      description "Wavelength (nm) to be used for the quantum channel. If
        the interface is not tunable, this parameter can by bypassed.";
    }

    leaf qkd_role {
      type identityref {
        base "etsi-qkdn-types:QKD-ROLE-TYPES";
      }
      description "Transmitter/receiver mode for the QKD system. If
        there is no multi-role support, this could be ignored.";
    }
}

notification sdqkdn_application_new {

  container qkd_application {

    leaf server_app_id {
      type inet:uri;
      description "ID of the server application connecting to
        the key store.";
    }
```

```
       leaf-list client_app_id {
         type inet:uri;
         description "ID of the client application connecting to
           the key server of the peer SD-QKD node. It is considered
           as the client from the security perspective.";
       }

       leaf local_qkdn_id {
         type yang:uuid;
         description "Universally Unique ID of the local SD-QKD node which
           is providing QKD keys to the local application.";
       }

       leaf priority {
         type uint32;
         default 0;
         description "Priority of the association/application.
           Higher values indicate higher priority. This might be
           defined by the user, but usually handled
           by a network administrator.";
       }

       container qos {

         leaf max_bandwidth {
           type uint32;
           description "Maximum bandwidth (bps) allowed for this specific
             application. Exceeding this value will raise an error
             from the local key store to the application. This value
             might be internally configured (or by an administrator).";
         }

         leaf min_bandwidth {
           type uint32;
           description "This value is an optional QoS parameter
             that enables an application to request a minimum key
             rate (in bits per second).";
         }

         leaf jitter {
           type uint32;
           description "This value allows to specify the maximum
             jitter (msec) of keys provided by the key delivery API, for
             applications requiring fast rekeying. This value can
             be coordinated with the other QoS parameters to provide a wide
             enough QoS definition.";
         }

         leaf ttl {
           type uint32;
           description "Time To Live specifies the maximum
             time (secs) a key can have been stored delivery to
             and application.";
         }

         leaf clients_shared_path_enable {
           type boolean;
           default false;
           description "If true, multiple clients for this application
             might share paths to reduce service impact (consumption).";
         }

         leaf clients_shared_keys_required {
           type boolean;
           default false;
           description "If true, multiple clients for this application
             might share keys to reduce service impact (consumption).";
         }

       }

       leaf type {
         type identityref {
           base etsi-qkdn-types:QKD-APP-TYPES;
         }
         description "Type of the registered application. These
           values, defined within the types module, can be client
           (if an external application is requesting keys)
```

```
          or internal (if the application is defined to maintain
          the QKD network - e.g. multi-hop, authentication or
          other encryption operations).";
      }
    }
  }

  notification sdqkdn_application_qos_update {

    container qkd_application {

      leaf app_id {
        type yang:uuid;
        description
          "This value uniquely identifies a pair of applications
          extracting keys from a QKD link. This value is similar
          to a key ID or key handle.";
      }

      container qos {

        leaf max_bandwidth {
          type uint32;
          description "Maximum bandwidth (bps) allowed for this specific
            application. Exceeding this value will raise an error
            from the local key store to the appl. This value might
            be internally configured (or by an admin).";
        }

        leaf min_bandwidth {
          type uint32;
          description "This value is an optional QoS parameter
            which enables to require a minimum key rate (bps)
            for the application.";
        }

        leaf jitter {
          type uint32;
          description "This value allows to specify the maximum
            jitter (msec) to be provided by the key delivery API for
            applications requiring fast rekeying. This value can
            be coordinated with the other QoS to provide a wide
            enough QoS definition.";
        }

        leaf ttl {
          type uint32;
          description "This value is used to specify the maximum
            time that a key could be kept in the key store (sec) for a
            given application without being used.";
        }

        leaf clients_shared_path_enable {
          type boolean;
          default false;
          description "If true, multiple clients for this application
            might share keys to reduce service impact (consumption).";
        }

        leaf clients_shared_keys_required {
          type boolean;
          default false;
          description "If true, multiple clients for this application
            might share keys to reduce service impact (consumption).";
        }
      }

      leaf priority {
        type uint32;
        default 0;
        description "Priority of the association/application. Higher numbers
          mean higher priority. It might be defined by the user, but usually handled
          by a network administrator.";
      }
    }
  }

  notification sdqkdn_application_disconnected {
    container qkd_application {

      leaf app_id {
        type yang:uuid;
        description
          "This value uniquely identifies a pair of applications
          extracting keys from a QKD key association link. This value is similar
          to a key ID or key handle.";
      }
    }
  }

  notification sdqkdn_interface_new {

    container qkd_interface{
      uses qkd_interface_common;
    }

  }

  notification sdqkdn_interface_down {

    container qkd_interface {

      leaf qkdi_id {
        type uint32;
        description "Interface id. It is described as a locally
          unique number, which is globally unique when combined
          with the SD-QKD node ID.";
      }

      leaf reason {
        type string;
        description "Auxiliary parameter to include additional
          information about the reason for interface failure.";
      }
    }
  }

  notification sdqkdn_interface_out {

    container qkd_interface {

      leaf qkdi_id {
        type uint32;
        description "Interface id. It is described as a locally
          unique number, which is globally unique when combined
          with the SD-QKD node ID.";
      }
    }
  }

  notification sqdkdn_link_down {

    container qkd_link {

      leaf link_id {
        type yang:uuid;
        description "Unique ID of the QKD link (key association).";
      }

      leaf reason {
        type string;
        description "Auxiliary parameter to include additional
          information about the reason for link failure.";
      }
    }
  }

  notification sqdkdn_link_perf_update {

    container qkd_link {

      leaf link_id{
        type yang:uuid;
        description "Unique ID of the key association QKD link.";
      }
      container performance {

        uses common_performance;

        uses physical_link_perf;

      }
    }
  }

  notification sqdkdn_link_overloaded {

    container qkd_link {

      leaf link_id{
        type yang:uuid;
        description "Unique ID of the key association QKD link.";
      }

      container performance {

        uses common_performance;

      }
    }
  }

  notification alarm {

    container link {

        leaf link_id {
          type yang:uuid;
          description "Placeholder for link_id of an Alarm.";
        }

        leaf status {
          type identityref {
            base "etsi-qkdn-types:LINK-STATUS-TYPES";
          }
          description "Placeholder for status of an Alarm.";
        }

        leaf message {
          type string;
          description "Placeholder for message of an Alarm.";
        }

        leaf severity {
          type identityref {
            base "etsi-qkdn-types:SEVERITY-TYPES";
          }
          description "Placeholder for the severity of an Alarm.";
        }
    }

    container interface {

        leaf qkdi_id {
          type uint32;
          description "Placeholder for a new interface.";
        }

        leaf status {
          type identityref {
            base "etsi-qkdn-types:IFACE-STATUS-TYPES";
          }
          description "Placeholder for the status of the new interface.";
        }

        leaf message {
          type string;
          description "Placeholder for the message.";
        }

        leaf severity {
          type identityref {
            base "etsi-qkdn-types:SEVERITY-TYPES";
          }
          description "Placeholder for the severity.";
        }
    }

    container application {

        leaf app_id {
          type yang:uuid;
          description "Placeholder for the description of the new App.";
        }

        leaf status {
          type identityref {
            base "etsi-qkdn-types:APP-STATUS-TYPES";
          }
          description "Placeholder for the status of the new App.";
        }

        leaf message {
          type string;
          description "Placeholder for the message associated to the new App.";
        }

        leaf severity {
          type identityref {
            base "etsi-qkdn-types:SEVERITY-TYPES";
          }
          description "Placeholder for the severity.";
        }
    }

  }

  notification event {

    container link {

        leaf link_id {
          type yang:uuid;
          description "Placeholder for the description of the new event.";
        }

        leaf status {
          type identityref {
            base "etsi-qkdn-types:LINK-STATUS-TYPES";
          }
          description "Placeholder for the status.";
        }

        leaf message {
          type string;
          description "Placeholder for the message.";
        }

        leaf severity {
          type identityref {
            base "etsi-qkdn-types:SEVERITY-TYPES";
          }
          description "Placeholder for the severity.";
        }
    }

    container interface {

        leaf qkdi_id {
          type uint32;
          description "Placeholder for the QKD ID.";
        }

        leaf status {
          type identityref {
            base "etsi-qkdn-types:IFACE-STATUS-TYPES";
          }
          description "Placeholder for the status.";
        }
        leaf message {
          type string;
          description "Placeholder for the message.";
        }

        leaf severity {
          type identityref {
            base "etsi-qkdn-types:SEVERITY-TYPES";
          }
          description "Placeholder for the severity.";
        }
    }

    container application {

        leaf app_id {
          type yang:uuid;
          description "Placeholder for the app_id.";
        }

        leaf status {
          type identityref {
            base "etsi-qkdn-types:APP-STATUS-TYPES";
          }
          description "Placeholder for the status.";
        }

        leaf message {
          type string;
          description "Placeholder for the message.";
        }

        leaf severity {
          type identityref {
            base "etsi-qkdn-types:SEVERITY-TYPES";
          }
          description "Placeholder for the severity.";
        }
    }

  }

}
```

A.2 ETSI QKD SDN node types module

A file containing the YANG module in this clause is also available at the following URL: https://forge.etsi.org/rep/qkd/gs015-ctrl-int-sdn/blob/v1.1.1/etsi-qkd-node-types.yang.

```
/* Copyright 2021 ETSI
Licensed under the BSD-3 Clause (https://forge.etsi.org/legal-matters) */


module etsi-qkd-node-types {

  yang-version "1";

  namespace "urn:etsi:qkd:yang:etsi-qkd-node-types";

  prefix "etsi-qkdn-types";

  organization "ETSI ISG QKD";

  contact
    "https://www.etsi.org/committee/qkd
    vicente@fi.upm.es";

  description
    "This module contains the base types created for
    the software-defined QKD node information models
    specified in ETSI GS QKD 015 V1.1.1
    - QKD-LINK-TYPES
    - QKD-TECHNOLOGY-TYPES
    - QKD-ROLE-TYPES
    - QKD-APP-TYPES
    - Wavelength
    ";

  revision "2020-09-30" {
    description
      "First definition based on initial requirement analysis.";
  }

  identity QKD-TECHNOLOGY-TYPES {
    description "Quantum Key Distribution System base technology types.";
  }

  identity CV-QKD {
    base QKD-TECHNOLOGY-TYPES;
    description "Continuous Variable base technology.";
  }

  identity DV-QKD {
    base QKD-TECHNOLOGY-TYPES;
    description "Discrete Variable base technology.";
  }

  identity DV-QKD-COW {
    base QKD-TECHNOLOGY-TYPES;
    description "COW base technology.";
  }

  identity DV-QKD-2Ws {
    base QKD-TECHNOLOGY-TYPES;
    description "2-Ways base technology.";
  }

  identity QKD-LINK-TYPES {
    description "QKD key association link types.";
  }

  identity VIRT {
    base QKD-LINK-TYPES;
    description "Virtual Link.";
  }

  identity PHYS {
    base QKD-LINK-TYPES;
    description "Physical Link.";
  }

  identity QKD-ROLE-TYPES {
    description "QKD Role Type.";
  }

  identity TRANSMITTER {
    base QKD-ROLE-TYPES;
    description "QKD module working as transmitter.";
  }

  identity RECEIVER {
    base QKD-ROLE-TYPES;
    description "QKD module working as receiver.";
  }

  identity TRANSDUCER {
    base QKD-ROLE-TYPES;
    description "QKD System that can work as a transmitter or receiver.";
  }

  identity QKD-APP-TYPES {
    description "Application types.";
  }

  identity CLIENT {
    base QKD-APP-TYPES;
    description "Application working as client.";
  }

  identity INTERNAL {
    base QKD-APP-TYPES;
    description "Internal QKD node application.";
  }

  identity PHYS-PERF-TYPES {
    description "Physical performance types.";
  }

  identity QBER {
    base PHYS-PERF-TYPES;
    description "Quantum Bit Error Rate.";
  }

  identity SNR {
    base PHYS-PERF-TYPES;
    description "Signal to Noise Ratio.";
  }

  identity LINK-STATUS-TYPES {
    description "Status of the key association QKD link (physical and virtual).";
  }

  identity ACTIVE {
    base LINK-STATUS-TYPES;
    description "Link actively generating keys.";
  }

  identity PASSIVE {
    base LINK-STATUS-TYPES;
    description "No key generation on key association QKD link but a pool of keys
    are still available.";
  }

  identity PENDING {
    base LINK-STATUS-TYPES;
    description "Waiting for activation and no keys are available.";
  }

  identity OFF {
    base LINK-STATUS-TYPES;
    description "No key generation and no keys are available.";
  }

  ///

  identity IFACE-STATUS-TYPES {
    description "Interface Status.";
  }

  identity UP {
    base IFACE-STATUS-TYPES;
    description "The interfaces is up.";
  }

  identity DOWN {
    base IFACE-STATUS-TYPES;
    description "The interfaces is down.";
  }

  identity FAILURE {
    base IFACE-STATUS-TYPES;
    description "The interfaces has failed.";
  }

  identity APP-STATUS-TYPES {
    description "Application types.";
  }

  identity ON {
    base APP-STATUS-TYPES;
    description "The application is on.";
  }

  identity DISCONNECTED {
    base APP-STATUS-TYPES;
    description "The application is disconnected.";
  }
  identity OUT-OF-TIME {
    base APP-STATUS-TYPES;
    description "The application is out of time.";
  }

  identity ZOMBIE {
    base APP-STATUS-TYPES;
    description "The application is in a zombie state.";
  }

  identity SEVERITY-TYPES {
    description "Error/Failure severity levels.";
  }

  identity MAJOR {
    base SEVERITY-TYPES;
    description "Major error/failure.";
  }

  identity MINOR {
    base SEVERITY-TYPES;
    description "Minor error/failure.";
  }

  typedef wavelength {
        type string {
                pattern "([1-9][0-9]{0,3})";
            }
            description
                "A WDM channel number (starting at 1). For example: 20";
  }

  //Pattern from "A Yang Data Model for WSON Optical Networks".
  typedef wavelength-range-type {
            type string {
                pattern "([1-9][0-9]{0,3}(-[1-9][0-9]{0,3})?" +
                        "(,[1-9][0-9]{0,3}(-[1-9][0-9]{0,3})?)*)";
            }
            description
                "A list of WDM channel numbers (starting at 1)
                 in ascending order. For example: 1,12-20,40,50-80";
  }
}
```

Annex B (informative): Bibliography

- IETF RFC 7951 (August 2016): "JSON Encoding of Data Modeled with YANG".
- IETF DRAFT 9 (November 2017): "A Yang Data Model for WSON Optical Networks".
- ETSI GS QKD 004 (V2.1.1): "Quantum Key Distribution (QKD); Application Interface".
- ETSI GS QKD 014 (V1.1.1): "Quantum Key Distribution (QKD); Protocol and data format of REST-based key delivery API".
- ETSI GS QKD 007 (V1.2.1): "Quantum Key Distribution (QKD); Vocabulary".

Annex C (informative): Change History

Date	Version	Information about changes
May 2018	0.0.1	Early draft.
May 2018	0.0.2	Minor corrections over 0.0.1 (correct numbering and contributors).
November 2018	0.0.3	Updates on notifications and security and protocol considerations. Other minor fixes.
March 2019	0.0.4	Minor fixes. Typos corrections.
May 2019	0.0.5	Changes in the Introduction, Executive Summary, Terms and References.
May 2020	0.0.6	Refinement of initial definitions. Fix some typos. SD-QKD node YANG Model minor changes. Bibliography.
May 2020	0.0.7	Inclusion of YANG basic types.
September 2020	0.0.10	Clarified text to avoid misunderstanding with QKD Link. YANG revised.
November 2020	0.0.12	Clarified text multi-hop/key relay. YANG descriptions further clarified for key association QKD link. Incorrect example for local/interface and remote/interface in QKD key association link parameters removed.

History

Document history		
V1.1.1	March 2021	Publication