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

Quantum Key Distribution (QKD); Application Interface

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
RGS/QKD-004ed2_ApplIntf

Keywords
API, quantum cryptography, quantum key distribution, security, use case

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

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.

1 Scope

The present document is intended to specify an Application Programming Interface (API) between a QKD key manager and applications. The function of a QKD key manager is to manage the secure keys produced by an implementation of a QKD protocol and to deliver the identical set of keys, via this API, to the associated applications at the communication end points.

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] IANA Media Type registry list of Directories of Content Types and Subtypes.

NOTE: Available at http://www.iana.org/assignments/media-types/.

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] OASIS Standard (22 November 2017): "Key Management Interoperability Protocol Specification Version 1.4". Editor by Tony Cox.

NOTE: Available at http://docs.oasis-open.org/kmip/spec/v1.4/os/kmip-spec-v1.4-os.html.

[i.2] ETSI GS QKD 014 (V1.1.1): "Quantum Key Distribution (QKD); Protocol and data format of REST-based key delivery API".

3 Definition of terms, symbols and abbreviations

3.1 Terms

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

Application Programming Interface (API): interface implemented by a software program to be able to interact with other software programs

Key Management Interoperability Protocol (KMIP): protocol for the communication between enterprise key management systems and encryption system

NOTE: The KMIP is directed by the OASIS initiative.

key management layer: abstraction in a layered model including physically distributed key managers in two or more network nodes

NOTE: The key management layer sits between QKD modules and various applications. It manages the synchronization and deletion of keys, etc. as well as key delivery to applications.

link encryptor: device performing link encryption, i.e. the communication security process of encrypting / decrypting information between two peers on the data link level

Organization for the Advancement of Structured Information Standards (OASIS): global consortium that drives the development, convergence and adoption of e-business and web service standards, including KMIP

QKD application interface: interface between a QKD key manager and one or more application

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 and software and/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 protocol: list of steps including the transport of quantum states that have to be performed by QKD modules to establish symmetric keys between remote parties with security based on quantum entanglement or the impossibility of perfectly cloning the transported quantum states

Quality of Service (QoS): description or measure of the overall performance of a service provided to an application after any management policies for prioritizing different applications or users have been applied

synchronization: function to ensure that symmetric keys in two key managers are identical

Transport Layer Security (TLS): cryptographic protocols used to encrypt the segments of network connections above the Transport Layer, using symmetric cryptography for privacy and a keyed message authentication code for message reliability

3.2 Symbols

Void.

3.3 Abbreviations

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

AAA	Authentication, Authorization and Accounting
API	Application Programming Interface
APPA	Application at host A
APPB	Application at host B
APPZ	Application at host Z
CORBA	Common Object Request Broker Architecture
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
ITS	Information Theoretical Secure
JSON	JavaScript Object Notation
KIMP	Key Management Interoperability Protocol
KM	Key Manager
KMIP	Key Management Interoperability Protocol
KSID	Key stream ID
OASIS	Organization for the Advancement of Structured Information Standards

PAP	Password Authentication Protocol
QKD	Quantum Key Distribution
QKDA	QKD key management layer peer at host A
QKDB	QKD key management layer peer at host B
QoS	Quality of Service
REST	Representational State Transfer
SDN	Software Defined Networking
SSL	Secure Socket Layer
TLS	Transport Layer Security
TTL	Time To Live
URI	Uniform Resource Identifier

4 Introduction

The present document is intended to specify an Application Programming Interface (API) between QKD Key Managers (KMs) and applications. The function of a QKD KM is to manage secure keys produced by an implementation of a QKD protocol and to deliver, on demand, identical sets of keys, via this API, to peer applications at the communication end points. It should be noted that there may be multiple levels of key managers (e.g. one at the QKD link level and another one at the QKD network level) and the API specified in the following clauses may be implemented in every QKD key manager level and that such key interchange is accomplished within the user's local security perimeter.

Manufacturers may implement this API wherever a QKD key manager is provided. Manufacturers may provide additional APIs and expanded functionality as they deem fit, realizing that these additions may be incompatible with versions provided by other vendors. Also, ther high level APIs can be built on top of this primary API, such as the REST API specified in ETSI GS QKD 014 [i.2].

5 QKD Application Interface Specification Description

The present document encompasses the ability to have multiple levels of key managers (e.g. one at the QKD link level and another one at the QKD network level), see figure 1 and figure 2, and the API specified in the following clauses may apply to every QKD key manager level and that this interchange is accomplished within the user's local security perimeter, referred to as sites. The methods by which a key manager accomplishes these functions is beyond the scope of the present document, but it is expected that external communication between key managers at different sites will be necessary. In addition, some communication between the peer applications may also be necessary to communicate a common key association.

The simplest example, figure 1, shows a single QKD link, with end points at Site A and Site B. Each site has a single application (shaded in yellow) and a single QKD module, enclosed by a blue box, which implements its part in a QKD protocol (shaded in red) to produce QKD keys that are managed by a QKD key manager peer (shaded in green). The single peer application in this case, uses the QKD application interface to acquire identical sets of secure keys in the two sites on demand.

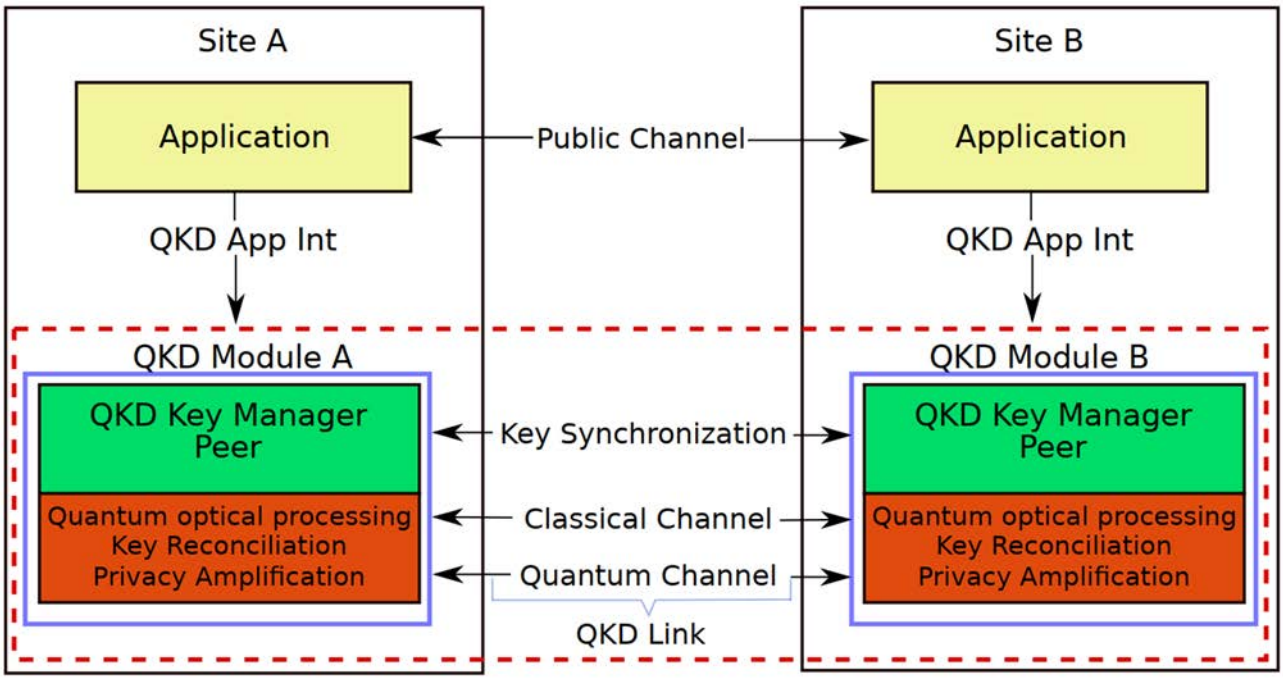


Figure 1: QKD Application Interface and peer relationships
Sites A and B represent security perimeters at each site

A more general example of two sites in a QKD network is shown in figure 2. Here two Sites of many in the network that contain a number of applications are shown. Site A and Site B are connected via a single QKD link between QKD Module A3 and QKD Module B1. The end points of the other QKD modules are not shown. In addition, network layer key managers, referred to as a QKD key servers are also depicted. The function of the QKD key servers is to manage keys between end points and deliver identical sets of keys to the applications at these end points. Note that the QKD application interface specified in the present document may be used to deliver keys from the QKD key management peers to the QKD key servers as well as to deliver keys from the QKD key servers to applications.

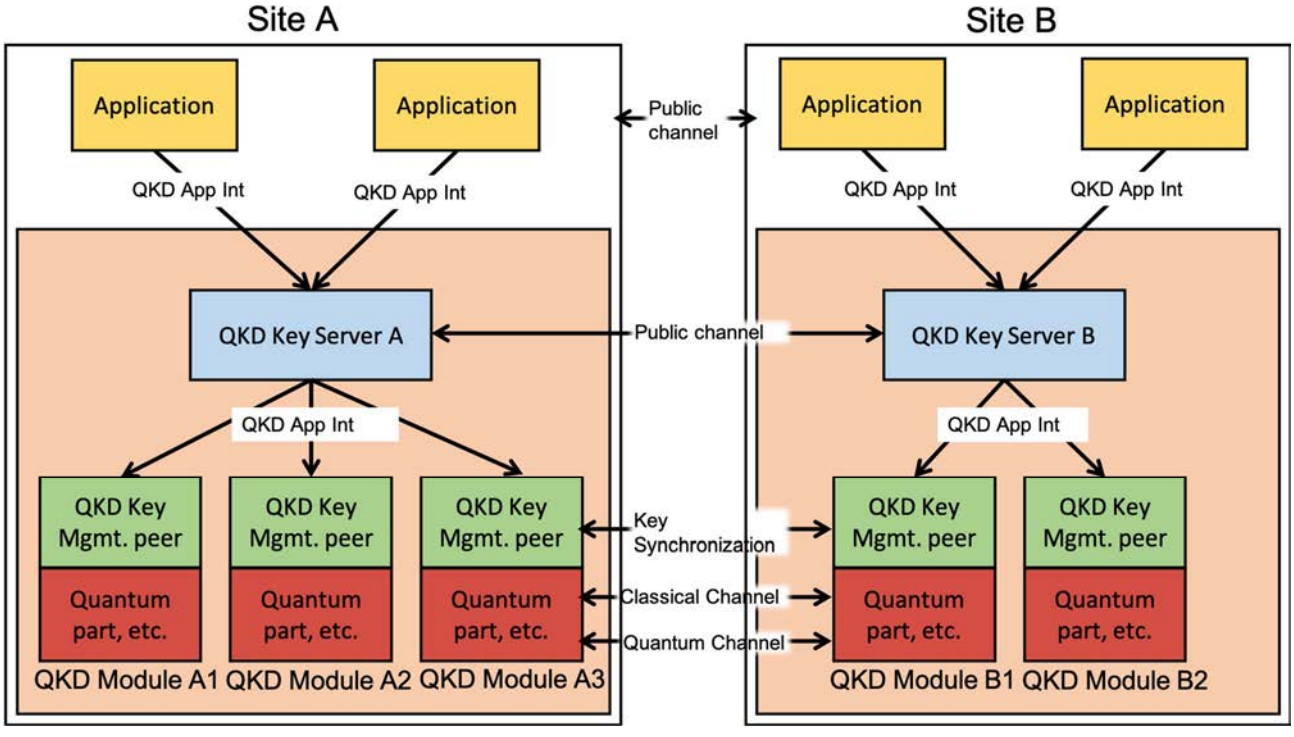


Figure 2: QKD Application Interface at two levels and peer relationships for Sites within a QKD Network. Sites A and B represent security perimeters at each location

6 QKD Application Interface API Specification

6.1 General

A QKD key manager implementing this application interface shall provide the following API functions.

Table 1

Name	Description
OPEN_CONNECT	Reserve an association (Key_stream_ID) for a set of future keys at both ends of the QKD link through this distributed QKD key management layer and establish a set of parameters that define the expected levels of key service. This function shall block until the peers are connected or until the Timeout value in the QoS parameter is exceeded.
CLOSE	Terminate the association established for this Key_stream_ID. No further keys shall be allocated for this Key_stream_ID after the association has been closed. Due to timing differences at the other end of the link this peer operation will happen at some other time and any unused keys shall be held until that occurs and then discarded or the TTL (Time To Live QoS parameter) has expired.
GET_KEY	Obtain the required amount of key material requested for this Key_stream_ID. Each call shall return the fixed amount of requested key, in the key_buffer parameter, or an error message indicating why it failed. Together with the requested key, the QKD key manager peer shall return an index value, or allow the application to specify the position of the key to be retrieved for synchronization purposes via the index parameter. The key position will be the result of multiplying the index value by the Key_chunk_size value within the QoS parameters. The call will return information about the chunk of key in the Metadata_buffer parameter, if the Metadata_size parameter is not zero. If it is zero, no Metadata will be returned. If the Metadata_size is not zero and this information cannot fit into the buffer provided, the call will fail without providing any key, but will return the needed buffer size in the Metadata_size parameter and will set a flag in the status parameter. Thus, the application can reissue the call with the buffer size set correctly or to zero, without loss of key. This function may be called as often as desired, but the QKD key manager only needs to respond at the bit rate requested through the QoS parameters, or at the best rate the system can manage. The QKD key manager is responsible for reserving and synchronizing the keys at the two ends of the QKD link through communication with its peers. This function always returns with the status parameter indicating success or failure, depending on the request made via the OPEN_CONNECT function. The Timeout value for this function is specified in the OPEN_CONNECT() function.

The syntax of these functions are as follows:

```
Interface QKD{
    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);
}
```

NOTE: The parameter "Key_stream_ID" is an output parameter when the caller is initially opening the connection and the value passed in is "NULL". If the value is not "NULL" the value is assumed to be either a "Key_stream_ID" recently established by the peer end or a specifically desired/predefined "Key_stream_ID". For use in all other functions "Key_stream_ID" is an input. If the "Key_stream_ID" is already in use by some other application, the OPEN_CONNECT function will return with an error. On return, the structure QoS is the closest QoS that the system can provide (best effort). If on input QoS is "NULL" on a certain field of the structure, the system is free to provide the one that it considers most appropriate. In particular, if a KSID has been already registered, the QoS structure will be filled with the QoS values previously assigned to that KSID, independently of any value of the QoS in the OPEN_CONNECT call.

The function parameters are described in table 2.

Table 2: Description of API function parameters

Name			Description	Type	Comments
Key_stream_ID (KSID)			A unique identifier for the group of synchronized bits provided by the QKD Key Manager to the application	UUID_v4 16 bytes (128 bits)	It contains a reference to the necessary information to locate a key, but does not contain any key material. Key material cannot be derived from the Key_stream_ID. It shall be represented as a character array (char *), with the high order octet being the lowest significant octet. It may be passed to and stored in other applications at other sites. It shall be unique and both peers will use that same value to reference their application key stream. It can be previously agreed upon between the peer applications or sent between them by a public channel.
Key_buffer			Buffer containing the current stream of keys.	Array of bytes (octets)	Key buffer is an array of bits packed into octets (char*) ordered such that bit[0] of octet[0] is the 1st bit and bit[7] of octet[n] is the 8*n+8th bit.
Index			Position of the key to be accessed within the reserved key store for the application	32 bit unsigned integer	For client/server synchronization purposes, the index value allows one to specify which position within the reserved key stored is to be accessed. The actual position will be calculated as the multiplication of the index value by Key_chunk_size from the QoS parameters.
Source			Source identifier defined as a uniform resource identifier	URI	Identifier of the source application connecting to the QKD key management layer. The identifier is structured as a URI. If a client/server scheme is defined, the source will integrate the URI of the server.
Destination			Destination identifier defined as a uniform resource identifier	URI	Identifier of the destination application connecting to the QKD key management layer. The identifier is structured as a URI. If a client/server scheme is defined, the source will integrate the URI of the client.
QoS	Structure describing the characteristics of the requested key source				
	Key_chunk_size		Length of the key buffer, in Bytes, requested by the application	32 bit unsigned integer	If the requested key amount cannot be provided, an error shall be returned in the status parameter.
	Max_bps		Maximum key rate, in bps, requested by the application	32 bit unsigned integer	This is intended to provide guidance to the key management layer on expected maximum demand. If the peer cannot meet the requested rate, it may choose to do its best, which may be lower than the requested rate, or reject the request.
	Min_bps		Minimum key rate, in bps, required by the application	32 bit unsigned integer	This is intended to provide an estimation of the minimum key rate needed by the application to the QKD key management layer, so that an application could maintain its security channels. If the peer cannot meet the requested rate, it may reject the request.
	Jitter		Maximum expected deviation, in bps, for key delivery	32 bit unsigned integer	This value allows applications to specify flow variation of key bits with a minimum deviation for the delivery rate.
	Priority		Priority of the request	32 bit unsigned integer	This is intended to provide guidance to the QKD Key Management layer about the priority level of the request. The handling of this information is left to the specific implementation.
	Timeout		Time, in msec, after which the call will be aborted, returning an error.	32 bit unsigned integer	If this much time has passed and the one of the API functions has not completed, the function will return a TIMEOUT_ERROR in the status parameter This value shall be expressed in milliseconds.

Name		Description	Type	Comments
QoS	Structure describing the characteristics of the requested key source			
	Time to Live (TTL)	Time, in seconds, after which the keys for this KSID shall be erased from the application's dedicated key store, for security reasons.	32 bit unsigned integer	This value sets the life-time of keys reserved for a given KSID. This is meant as a clean-up operation if for any reason the application crashes or leaves unused key behind.
	Metadata mimetype	The mimetype of the metadata to be delivered by the KM on each GET call.	char array of size 256	This field defines the format of the metadata on each subsequent GET_KEY call. The mimetype value shall be one of the Media Types defined in the Internet Assigned Numbers Authority (IANA) Media Type registry [1]. Any KM claiming conformance to the present document shall support the metadata mimetype "application/json". Any KM may implement additional mimetypes. If no mimetype is defined, i.e. the first byte of the metadata mimetype is 0, then the KM shall not report metadata in the GET_KEY call associated with this key stream. If a mimetype is specified, but the current KM implementation does not support this particular mimetype then OPEN_CONNECT shall fail but the metadata mimetype field shall return with the KM's preferred mimetype. Any KM compliant to the present document shall not reject "application/json" mimetype even if it is not the KM's preferred mimetype.
Metadata	A Structure containing a 32 bit unsigned integer and a character array. The integer specifies the size of the character array.			
	Metadata_size	Size of buffer	32 bit unsigned integer	Size of metadata buffer in characters
	Metadata_buffer	Data buffer	Array of bytes (octets)	Buffer for returned metadata
Status		Success/Failure of the request	32 bit unsigned integer	Status values shall have the following meanings: 0 ➔ Successful. 1 ➔ Successful connection, but peer not connected. 2 ➔ GET_KEY failed because insufficient key available. 3 ➔ GET_KEY failed because peer application is not yet connected. 4 ➔ No QKD connection available. 5 ➔ OPEN_CONNECT failed because the "KSID" is already in use. 6 ➔ TIMEOUT_ERROR The call failed because the specified TIMEOUT. 7 ➔ OPEN failed because requested QoS settings could not be met, counter proposal included in returnhas occurred. 8 ➔ GET_KEY failed because metadata field size insufficient. Returned Metadata_size value holds minimum needed size of metadata. Other values can be added in the future if needed.

Important considerations:

1) An application may manage several independent key streams through different "key stream IDs" (e.g. one set of keys for output messages and another for input messages, where each set is identified by a separate "Key_stream_ID"). There is no limit on the number of "Key_stream_IDs" that one application can request.

2) The Key_stream_ID shall be generated in such a way that uniquely identifies the key material and that key material cannot be derived from it.

3) It has not been considered how the internal operation of this QKD KM service is to be accomplished, except for error messages back to the application through the status parameter. This will be left to the implementer.

4) It has not been considered how the QoS priority parameter shall be handled by the internal operation of this QKD KM service. This shall be left to the implementer.

5) The lifetime of the keys in the QKD bit pool that is managed by this layer is not specified nor is any action specified at that time. This shall be left to the implementer.

6) It is mandatory that the API shall implement the source/destination pair as URIs, however, for the sake of compatibility with other APIs, it may also interpret other types of data.

7) Note that this API may manage Authentication, Authorization and Accounting (AAA) by integrating some information into the URI (e.g. an OAuth token), however, the API is not required to process this information, but it can pass it to the subsystem in charge of processing AAA information.

8) The metadata field is designed to provide information to help with the key management duties. The API does not deal with key management.

9) The metadata QoS consists of a Metadata_size value and a character block provided by and under the authority of the client calling the KM. The client is responsible for allocating and freeing this memory block. The KM shall check the size of this memory block on each GET_KEY call and if this value is insufficient, then place the current minimum value instead into the size field on return. The call itself is then returned with an error value and the key shall not be delivered to the client.

Table 3

"age"	Time since the chunk of key was made available to be delivered.	32 bit unsigned integer	Measured in msec.
"hops"	Number of trusted nodes visited to deliver the chunk of key	32 bit unsigned integer	0 = direct connection (no trusted nodes in between).

10) Some examples of metadata that may be returned from the KM are:

- Every KM vendor may add additional values as desired, respecting the format of the specified mimetype.

6.2 Sequence diagrams for QKD Application Interface

6.2.1 General

In all sequences, there are five actors:

1) Application at host A, APPA.

2) Application at host B, APPB and at host Z, APPZ.

3) QKD key manager at host A (in Module A of the QKD system connecting Site A to Site B), QKDA.

4) QKD key management layer peer at host B (in module B of the QKD system connecting Site A to Site B), QKDB.

5) QKD key manager in a QKD network at host A (in module A), QKD_SERVA and at host Z (in module Z), QKD_SERVZ.

Five different sequences are shown:

1) APPA and APPB do not have a predefined key stream ID and execute successfully in a single link scenario.

2) APPA and APPB do not have a predefined key stream ID and timeout on the GET_KEY() call in a single link scenario.

3) APPA and APPB both know the predefined key stream ID that they will use for the key association and execute successfully in a single link scenario.

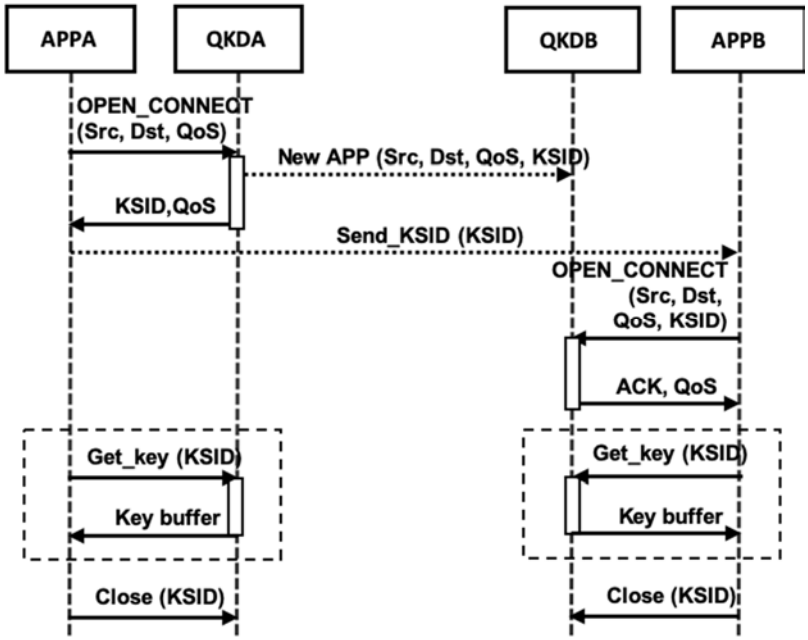
4) APPA and APPB both know the predefined key stream ID that they will use for the key association and timeout on the GET_KEY() call in a single link scenario.

5) APPA and APPZ do not have a predefined key stream ID and execute successfully over a QKD network.

For the scenarios covered in this clause, one may assume that the communication between local applications and the QKD key manager is accomplished within the user's local security perimeter.

6.2.2 Case 1: Undefined KSID in a single link scenario

In this case, APPA starts the sequence with a call to OPEN_CONNECT() indicating application source, destination and the desired QoS (and a NULL value in the Key_stream_ID parameter). As this is a single link scenario and QKD systems do not need to discover their peers, OPEN_CONNECT() directly returns a Key_stream_ID value and status information indicating success. APPA sends this Key_stream_ID value to APPB on the public channel and then starts gathering keys or waits for APPB to also connect. APPB is waiting on the public channel from APPA for a message with the KSID that they will share. Once APPB gets the KSID it shall call OPEN_CONNECT() to register its Key_stream_ID but it does not need to repeat the level of services (QoS) desired as long as APPA has already done so. APPB shall check the status parameter returned by OPEN_CONNECT for success and rendezvous with APPA. At this point the key association has been verified and established and if the QKD system is operational the status shall be returned as successful. After that, both APPA and APPB start making calls to GET_KEY() until they have no more need for secure keys and call the CLOSE() function in order to terminate the Key_stream_ID association in a timely manner.

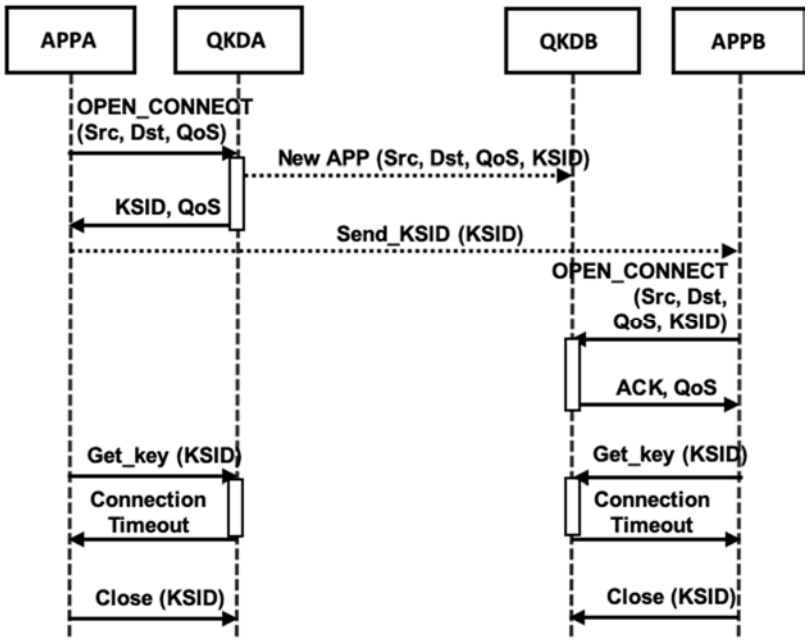


NOTE: SEND_KSID() or New APP are not in the scope of the QKD Application Interface. They are included to illustrate the functionality of the case.

Figure 3: Case 1 - Undefined KSID

6.2.3 Case 2: Undefined KSID and failed get key call in a single link scenario

This is the same as in case 1, except for some reason QKDA cannot exchange the necessary Key information with QKDB in the time specified by the TIMEOUT parameter of OPEN_CONNECT (). Both APPA and APPB receive a TIMEOUT ERROR in the status parameter from their calls to GET_KEY().



NOTE: SEND_KSID() or New App are not in the scope of the QKD Application Interface. They are included to illustrate the functionality of the case.

Figure 4: Case 2 - Undefined KSID and failed get key call

6.2.4 Case 3: Predefined KSID in a single link scenario

In this case, it is irrelevant who starts the sequence. Both APPA and APPB call OPEN_CONNECT() indicating the level of service desired and a predefined (first interaction) value in the Key_stream_ID parameter. OPEN_CONNECT() verifies that the specified Key_stream_ID is not already in use and returns status information indicating success. No Key_stream_ID information needs to be sent between APPA and APPB after the OPEN_CONNECT call. QKDA and QKDB might exchange the information within the OPEN_CONNECT call for synchronization and verification purposes. At this point the key association has been verified and established between QKDA and QKDB, and if the QKD system is operational the status will be returned as successful. After that, both APPA and APPB start making calls to GET_KEY() until they have no more need for secure key and call the CLOSE() function that terminates the Key_stream_ID association in a timely manner.

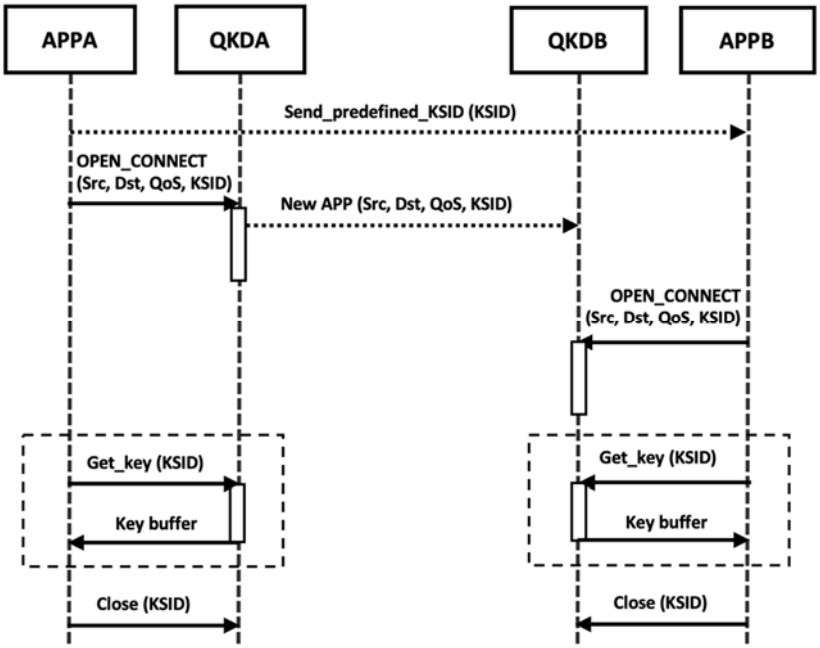


Figure 5: Case 3 - Predefined KSID and get key call

6.2.5 Case 4: Predefined KSID and failed get key call in a single link scenario

This is the same as case 3, except for some reason QKDA cannot exchange the needed information with QKDB in the time specified by the TIMEOUT parameter of OPEN_CONNECT (). Both APPA and APPB receive a TIMEOUT ERROR in the status parameter from their calls to GET_KEY().

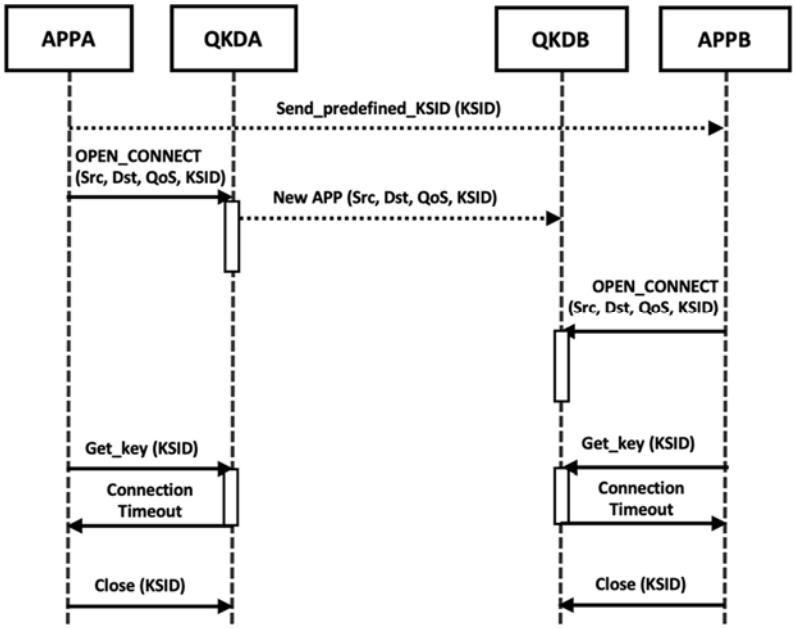


Figure 6: Case 4 - Predefined KSID and failed get key call

6.2.6 Case 5: Application discovery in a QKD network

The integration with network applications follow a workflow outlined below that is similar to the ones described above, where applications shall connect to the QKD Key Server via OPEN_CONNECT(). Additionally, there is an intermediate step for the QKD Key Servers to find its end-to-end peer for the requested session. This is out of the scope of the present document and may happen in different manners: signalling among QKD sites for application discovery or notifying an intermediate central repository or management application (e.g. a central key management system or a software-defined networking -SDN- controller). Also, for key relay purposes, Information Theoretical Secure (ITS) mechanisms shall be integrated in any communication channel between QKD Key Servers, but it is out of the scope of the present document to define any of these security mechanisms.

This case is defined when a QKD network (rather than a single point-to-point QKD link) is in place. An application APPA wants to obtain a series of shared secret bits with an application APPZ. Therefore, as there is no direct link between QKD modules at locations A and Z, some support facility needs to be provided by the system to identify the QKD endpoints of APPZ on demand. In figure 7 these facilities are depicted as QKD_SERVA and QKD_SERVZ. When APPA sends an OPEN_CONNECT() to QKD_SERVA, QKD_SERVA likewise does not know to which QKD endpoint APPZ is connected. The first operation to be done, is the generation of a KSID for the session that APPA wants to create. Then, QKD_SERVA needs to advertise this new application request, indicating source (APPA), destination (APPZ), the generated KSID, the local QKD module (QKDA or QKD_SERVA) and QoS information. This advertisement process could be performed using traditional routing schemes, where this new application and its requirements are broadcasted over the QKD network, or via a centralized management system that takes care of these notifications and provides the QKD module peers with the appropriate information. When this information is forwarded, QKD_SERVA sends to APPA the generated KSID. APPA shall then inform its peer, APPZ, of the KSID return by the QKD_SERVA. When APPZ connects to its QKD endpoint (QKD_SERVZ), the same situation happens, with the difference that (in this implementation) QKD_SERVZ already knows about the application connected to QKD_SERVA (because of the intermediate advertisement procedure) and that APPZ knows the valid KSID, previously shared by APPA. Note that this workflow is an example for a given implementation and may vary based on architectural decisions (e.g. if a centralized management system is in place).

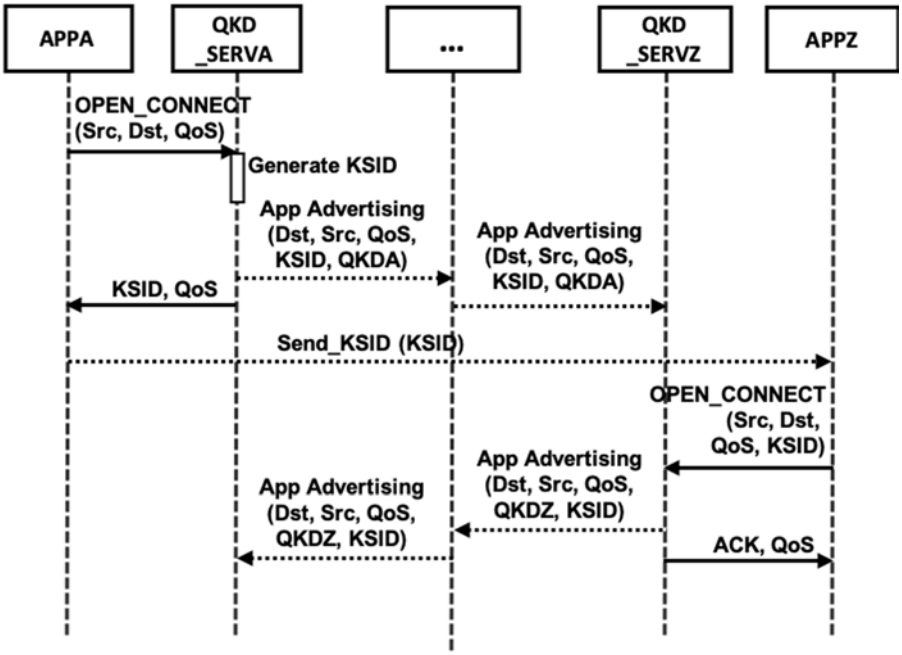


Figure 7: Case 5 - Workflow for Application Discovery in a QKD network

Applications may or may not predefine the KSID to be used, as long as it is globally unique. Any issue with connection timeouts will be locally managed and treated as an error, similar to cases 2 and 4 in clauses 6.2.3 and 6.2.5 respectively.

Annex A:
Void

Annex B (informative): Conventional Key Management Systems

Conventional key management systems were investigated for use within QKD systems. One such protocol, the KMIP initiative within the OASIS open standards consortium https://www.oasis-open.org was considered. It specifies a protocol and interface for communication between clients and a key server. QKD instead operates under a distributed paradigm, where the keys are already distributed to the QKD end points (QKD Modules) but demultiplexing them into independent synchronized streams and distributing them to various applications, at those communication end points, as needed.

Some thought was given to adopting the KMIP framework for QKD systems as well as using QKD to strengthen the KMIP as follows:

1) To enhance security of KMIP authentication protocols. KMIP does not specify, as part of the protocol, the mechanisms by which key management systems and cryptographic clients identify themselves to each other. Rather, KMIP relies on existing standards for mutual authentication that specify how this identification is to be established. KMIP currently defines two authentication profiles, the first based on TLS, the second on HTTPS. In both profiles, digital certificates are used by the client and the server to identify themselves as participants in KMIP requests and responses. Registration mechanisms by which the enterprise key manager learns the identity of cryptographic clients are not defined in KMIP. The credentials used by the cryptographic client to identify itself can be included in the protocol as part of a request message, to simplify processing of the request by the key management system. However, the credential element is not guaranteed to be authenticated and is therefore not intended for use in authentication. Message integrity for KMIP exchanges, as well as entity authentication, is provided by TLS. Other mechanisms (for example with the inclusion of QKD inside TLS or HTTPS key exchange process) that could also be used for enhanced security of KMIP messages are not currently defined for KMIP.

2) To enhance security of a Key Management system using QKD one-time-pad tunnels for the interchange of KMIP messages.

To include QKD keys as a third party source of keys for KMIP servers.

Key Management Interoperability Protocol Specification Version 1.4 is an OASIS Standard [i.1].

Annex C (informative): Relationship of this API to ETSI GS QKD 014 "Protocol and Data Format of REST-based Key Delivery API"

This API is designed to be as small as possible while being highly flexible. The underlying design is to have an API able to create a continuous stream of keys from two (or more) connected remote applications in order to maximize key throughput with a minimal overhead. This also means that the interface is able to provide sessions to which a given Quality of Service (QoS) can be associated and negotiated between the application and the underlying QKD Networks. Based on QoS requests, the network can organize all the requests to maximize the use of the QKD based services, for example, assigning priorities and organizing the key provisioning and reservation for future requests. Together with the QoS the present document introduces the management of network metadata, that it is useful to extract knowledge of the network, for example to have network statistics or debug information about QKD devices.

This API is also implementation agnostic. It is independent of the programming language, libraries, or other tools required for its implementation, leaving its choice to the implementer to better satisfy the service requirements. This means that the protocol stack may be chosen to suit the needs of the concrete implementation. i.e. a small stack without much footprint for resource-limited devices, a protocol which provides easy parsing and checking for an (usually small) implementation that is easier to evaluate and certify for security or a complex protocol stack (like Corba, REST over HTTP(S)) which eases integration into other applications. It is possible to use this API from simple raw sockets to more secure communications methods like SSL/TLS tunnels or ZeroMQ/Protobuf, etc.

The use of URIs also makes possible the use of different protocols of Authentication (e.g. PAP, Kerberos, OAuth, etc.), Authorization and Accounting (AAA). URIs also makes possible the use of different routing protocols from any pair of applications, including multi-hop (with key relay) connections.

In contrast, the 014 Key Delivery API follows a completely different design principle (Additional information on the relationship between the 014 and 004 API can be found in ETSI GS QKD 014 [i.2], annex A). ETSI GS QKD 014 [i.2] is conceived as a single key request API, where applications always need to ask for keys using REST request, individually, not as a key stream and are responsible for transmission and handling of the key identifier on their own. Also, it does not have the concept of sessions.

ETSI GS QKD 014 [i.2] API specifies the use of the HTTPS protocol and the JSON data formats, this eases the implementation but also presents some constraints, since HTTPS and JSON require high level libraries or frameworks for implementation. In contrast, the 004 API can be implemented without any libraries, in a very compact way and completely in standard C language, simplifying the security analysis.

ETSI GS QKD 014 [i.2] API can be seen as a simplification, built with high level tools (web based) and without the QoS of the 004 API. ETSI GS QKD 014 [i.2] API can be implemented on top of the present document API, although the QoS needs to be set to a default constant. The basic approach is the webserver processing the REST requests to issue the present document API OPEN_CONNECT() call on startup and the corresponding API CLOSE() call when it stops. For each REST request related to keys, a simple (the present document) API GET_KEY() call is needed. This approximation leaves all the high-level processing to the Web side and all the Key management to the present document API. The other way around is not always possible (without additional synchronization), because the session concept of the present document allows the users of the API to get multiple corresponding keys by simple requests with only a counter as input, whereas with ETSI GS QKD 014 [i.2] each key gets a new unique key identifier which has to be transferred by the user.

Annex D (informative): Bibliography

- C.H. Bennett and G. Brassard: "Quantum Cryptography: Public Key Distribution and Coin Tossing", Proceedings of IEEE International Conference on Computers Systems and Signal Processing, Bangalore India, December 1984, pp 175-179.

NOTE: Available at http://www.research.ibm.com/people/b/bennetc/bennettc198469790513.pdf.

- Nicolas Gisin, Grégoire Ribordy, Wolfgang Tittel and Hugo Zbinden: "Quantum cryptography, Reviews of Modern Physics", vol 74, 145-195 (2002).

- Valerio Scarani, Helle Bechmann-Pasquinucci, Nicolas J. Cerf, Miloslav Dušek, Norbert Lütkenhaus and Momtchil Peev: "The security of practical quantum key distribution", vol. 81, 1301-1351 (2009).

NOTE: Available at http://arxiv.org/abs/0802.4155.

- D. Gottesman, H.-K. Lo, N. Lütkenhaus and J. Preskill: "Security of quantum key distribution with imperfect devices", vol. 5, 325-360) (2004).

NOTE: Available at http://arxiv.org/abs/quant-ph/0212066.

- "White Paper on Quantum Key Distribution and Cryptography", Preprint arXiv:quant-ph/0701168, Alléaume R, Bouda J, Branciard C, Debuisschert T, Dianati M, Gisin N, Godfrey M, Grangier Ph, Länger T, Leverrier A, Lütkenhaus N, Painchault P, Peev M, Poppe A, Pornin Th, Rarity J, Renner R, Ribordy G, Riguidel M, Salvail L, Shields A, Weinfurter H, Zeilinge, A, 2006 SECOQC.

- UQC Report: "Updating Quantum Cryptography", Quantum Physics (quant-ph); Cyptography and Security. Donna Dodson, Mikio Fujiwara, Philippe Grangier, Masahito Hayashi, Kentaro Imafuku, Ken-ichi Kitayama, Prem Kumar, Christian Kurtsiefer, Gaby Lenhart, Norbert Luetkenhaus, Tsutomu Matsumoto, William J. Munro, Tsuyoshi Nishioka, Momtchil Peev, Masahide Sasaki, Yutaka Sata, Atsushi Takada, Masahiro Takeoka, Kiyoshi Tamaki, Hidema Tanaka, Yasuhiro Tokura, Akihisa Tomita, Morio Toyoshima, Rodney van Meter, Atsuhiro Yamagishi, Yoshihisa Yamamoto and Akihiro Yamamura, 2009.

NOTE: Available at http://arxiv.org/abs/0905.4325.

- "Introduction to Quantum Key distribution", V. Martin, J. Martínez-Mateo and M. Peev, Wiley Encyclopedia of Electrical and Electronics Engineering, 2017.

NOTE: Available at 10.1002/047134608X.W8354.

- Menezes A. J., van Oorschot P. C. and Vanstone S. A.: "Handbook of Applied Cryptography", (Boca Raton: CRC Press) 1997.

- Schneier B.: "Applied Cryptography", 1996, (New York: John Wiley).

- NIST Special Publication 800-53 Revision 4 "Security and Privacy Controls for Federal Information Systems and Organizations", Joint Task Force transformati:on initiative.

NOTE: Available at http://dx.doi.org/10.6028/NIST.SP.800-53r4 April 2013.

- OMG IDL Syntax and Semantics, Object Management Group Inc., formal/02-06-39 (CORBA 3.0 - OMG IDL Syntax and Semantics chapter).

NOTE: Available at http://www.omg.org/cgi-bin/doc?formal/02-06-39.

Annex E (informative): Change History

Date	Version	Information about changes
December 2010	1.1.1	Publication.
May 2018	1.2.0	Streamlined interface, QoS added, URI addressing, UUID identifiers for key streams.
May 2018	1.2.1	Minor corrections over 1.2.0 (updated version number, history description, contributors addressing and formatting).
January 2019	1.2.2	Additional minor corrections. Annex C clarification and deletion of old figure.
January 2019	1.2.3	Clarification of the terminology throughout the text, including in figures 1 and C.1. Revised annex B and annex D.
February 2019	1.2.4	Additional clarifications. Consistent naming convention of calls following Clean Code rules. Corrected bugs in the naming of figures. Discussed and rewritten Case 5, modified figure 6 and minor corrections.
September 2019	1.2.5	Correct mistakes in figures 2 and 3. Clarifications in the text. Define QoS as an inout argument, clarify its working and correct workflow figures to take this into account. Error codes added.
November 2019	1.2.9	Comments on accepted extensions beyond URIs. Structure to add metadata to keys.
January 2020	1.3.1	Reduced discussion on Key Management. Network case moved from appendix to main text. Minor mistakes corrected. Modified metadata types.
February 2020	1.3.3	Corrections and cleaning the metadata fields.
March 2020	1.3.4	Annex C.
May 2020	1.3.5	Collected discussions on Annex C. Definitions of Terms section revised.
May 2020	1.3.6	Added missing abbreviations. Corrected mistake in web address.

History

Document history		
V1.1.1	December 2010	Publication
V2.1.1	August 2020	Publication