Video Services Forum (VSF) Technical Recommendation TR-12

Client Device Discovery (CDD)

Bridging the First-Mile Gap: Modernizing Connectivity to the Cloud

June 1, 2026

VSF_TR-12_2026_06_01

INTELLECTUAL PROPERTY RIGHTS

THIS RECOMMENDATION IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NONINFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY USE OF THIS RECOMMENDATION SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE FORUM, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY MPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER, DIRECTLY OR INDIRECTLY, ARISING FROM THE USE OF THIS RECOMMENDATION.

LIMITATION OF LIABILITY

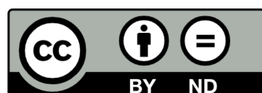
VSF SHALL NOT BE LIABLE FOR ANY AND ALL DAMAGES, DIRECT OR INDIRECT, ARISING FROM OR RELATING TO ANY USE OF THE CONTENTS CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION ANY AND ALL INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS, LOSS OF PROFITS, LITIGATION, OR THE LIKE), WHETHER BASED UPON BREACH OF CONTRACT, BREACH OF WARRANTY, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE FOREGOING NEGATION OF DAMAGES IS A FUNDAMENTAL ELEMENT OF THE USE OF THE

© 2025 Video Services Forum

This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit HTTPS://creativecommons.org/licenses/by-nd/4.0/
or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

http://www.videoservicesforum.org

CONTENTS HEREOF, AND THESE CONTENTS WOULD NOT BE PUBLISHED BY VSF WITHOUT SUCH LIMITATIONS.



BY-ND

Executive Summary

Effectively utilizing the public cloud is essential for media organizations seeking to build dynamic media architectures. However, operators frequently report that the most challenging aspect of cloud workflows is accessing distributed, often ground-based sources and destinations, and establishing the initial live signal connection. While robust transport protocols like SRT, RIST, and TR-07 solve the physical streaming challenge, they still require manual IP management and complex navigation of firewalls, Network Address Translation (NAT), and security groups on both the sender and receiver. The full benefit of these protocols is only realized when the cloud operator's 24/7 global access extends to ground-based or mobile endpoints, regardless of their location behind corporate firewalls, VPNs, or the open internet.

The Client Device Discovery (CDD) Technical Recommendation (TR-12) describes a method for device discovery, authentication, monitoring, and connection management via scalable, cloud-first techniques. Once a TR-12-enabled device is registered with a cloud service through a simplified pairing process, it remains persistently and globally accessible. This allows cloud operators or automated workflows to perform remote connection management without manual intervention at the edge.

TR-12 defines a client-server protocol and provides an open-source client SDK, an Application Reference Design (ARD), and a functional cloud service to serve as a remote endpoint. Recipients of this document are invited to download and test the SDK and submit technical comments.

The VSF also requests that recipients notify us of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the Recommendation set forth in this document, and to provide supporting documentation.

CC BY-ND

1 Table of Contents

1 Introduction (Informative).......... 5

1.1 Contributors.......... 6

1.2 About the Video Services Forum.......... 6

2 Conformance Notation.......... 7

3 References.......... 7

4 Overview (Informative).......... 7

5 Host Configuration.......... 8

6 Pairing.......... 9

6.1 The Pairing Process.......... 9

6.2 Throttling.......... 11

6.3 Pairing Models.......... 11

7 MQTT Communication.......... 12

7.1 Overview (Informative).......... 12

7.2 MQTT Settings.......... 12

7.3 Processing a Configuration Update.......... 13

7.4 Registration.......... 14

7.5 Reporting Status.......... 14

7.6 Certificate Rotation.......... 14

7.7 Deprovision.......... 15

8 Subscriptions.......... 16

8.1 Overview (Informative).......... 16

8.2 Thumbnail Subscription.......... 16

9 Nominal Operation (Informative).......... 17

9.1 Connect and Respond Loop.......... 17

9.2 Subscription Overview.......... 17

9.3 Deprovision Lifecycle.......... 18

10 The TR-12 Models (Informative).......... 18

10.1 Protocol Version.......... 18

10.2 Model Overview.......... 19

10.3 Registration Model.......... 19

CC BY-ND

10.4 Configuration Model ... 21
10.5 Status Model ... 23
Appendix A: Smithy Models ... 24

1 Introduction (Informative)

Operators of cloud video workflows benefit from global resource access, 24/7 monitoring, and consistent programmatic APIs that enable automation across large topologies. However, these capabilities generally do not extend to on-premises or mobile source/destination systems. While a handful of device manufacturers offer proprietary remote access, there is no industry standard that allows scaled cloud video services to interoperate with these devices across the internet without custom, system-by-system integration. For most manufacturers, developing and maintaining an enterprise-grade, secure, and reliable remote access platform is neither feasible nor easily monetizable through hardware sales alone.

The success of any improvement requires addressing the needs of three primary participants:

- **Device Manufacturers:** Require the freedom to expose unique device controls with low barriers to entry and high resilience.
- **Broadcasters and Operators:** Are unlikely to adopt solutions that require replacing existing device fleets. Solutions available via over-the-air (OTA) updates are preferred.
- **Cloud Infrastructure Providers:** Prefer investing in standards rather than device-specific customizations.

A viable solution must conform to cloud resource standards, including lifecycle patterns and host-provided, rotatable credentials. It must also utilize formal models based on modern API standards that offer built-in validation and inherent scalability, ensuring a consistent user experience across all device types with zero device-specific customization required by the host service.

This Technical Recommendation (TR-12) describes a methodology to achieve these goals by:

- **Providing Smithy and OpenAPI Models:** Utilizing models ubiquitous in enterprise cloud services to auto-generate classes for requests, responses, accessors, and validation across most programming languages.
- **Defining Three Primary API Models — Registration, Status, and Configuration:** These allow manufacturers to define device and channel-level settings unique to their hardware. While the models enforce structure, they do not dictate specific settings, allowing both client and host to consume shared definitions.
- **Enabling Dynamic UI Generation:** Allowing the host service to dynamically generate a console UI with type-specific widgets for each parameter based solely on the client-provided registration. Configuration updates can be validated by the host service before transmission to the client.

CC BY-ND

- **Standardizing HTTPS and MQTT Transport:** Utilizing an OAuth-based pairing process with a simple code to grant service-provided authentication and persistent credentials. These credentials allow the authorized client to establish a low-latency, bidirectional, and encrypted MQTT connection to the host.
- **Establishing MQTT Topic Structures:** Defining the specific topics used by the TR-12 client and host to manage status, configuration, and lifecycle events.

1.1 Contributors

Thank you to the following individuals who participated in the Video Services Forum TR-12 working group and helped shape this technical recommendation.

Mike Cronk: TVU Networks
Jeremy Gerdes: Osprey Video
Sydney Lovely: AWS Elemental
Drew Martin: Riedel
Michael Henry: AWS Elemental
Jesus Oliva: Haivision
Jarrod Nix: Osprey Video
Rob Porter: Sony
Wes Simpson: LearnIPvideo.com
Scott Whitcomb: Osprey Video
Joel Williams: Fox

1.2 About the Video Services Forum

The Video Services Forum, Inc. (www.vsf.tv) is an international association dedicated to video transport technologies, interoperability, quality metrics and education. The VSF is composed of service providers, users and manufacturers. The organization's activities include:

- Providing forums to identify issues involving the development, engineering, installation, testing and maintenance of audio and video services.
- Exchanging non-proprietary information to promote the development of video transport service technology and to foster resolution of issues common to the video services industry.
- Identification of video services applications and educational services utilizing video transport services.
- Promoting interoperability and encouraging technical standards for national and international standards bodies.

The VSF is an association incorporated under the Not For Profit Corporation Law of the State of New York. Membership is open to businesses, public sector organizations and individuals worldwide. For more information on the Video Services Forum or this document, please e-mail opsmgr@vsf.tv.

CC BY-ND

2 Conformance Notation

Normative text is text that describes elements of the design that are indispensable or contains the conformance language keywords: “shall”, “should”, or “may”. Informative text is text that is potentially helpful to the user, but not indispensable, and can be removed, changed, or added editorially without affecting interoperability. Informative text does not contain any conformance keywords.

All text in this document is, by default, normative, except the Introduction and any section explicitly labeled as “Informative” or individual paragraphs that start with “Note:”.
The keywords “shall” and “shall not” indicate requirements strictly to be followed in order to conform to the document and from which no deviation is permitted.
The keywords “should” and “should not” indicate that, among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.
The keywords “may” and “need not” indicate courses of action permissible within the limits of the document.
The keyword “reserved” indicates a provision that is not defined at this time, shall not be used, and may be defined in the future. The keyword “forbidden” indicates “reserved” and in addition indicates that the provision will never be defined in the future.
A conformant implementation according to this document is one that includes all mandatory provisions (“shall”) and, if implemented, all recommended provisions (“should”) as described. A conformant implementation need not implement optional provisions (“may”) and need not implement them as described.
Unless otherwise specified, the order of precedence of the types of normative information in this document shall be as follows: Normative prose shall be the authoritative definition; Tables shall be next; followed by formal languages; then figures; and then any other language forms.

3 References

OAUTH2: https://oauth.net/2/

MQTT: https://mqtt.org/
SMITHY: https://smithy.io/
SMITHY code generation: https://smithy.io/2.0/guides/using-code-generation/index.html
TR-12 Models: https://github.com/vsf-tv/TR-12-Models
TR-12 Client and Host SDK: https://github.com/vsf-tv/TR-12-Client-and-Host-Go
Any mention of references throughout the rest of this document refers to the versions described here, unless explicitly stated otherwise.

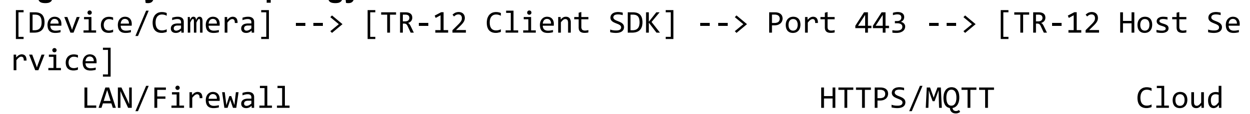
4 Overview (Informative)

The TR-12 protocol uses two distinct communication components: an **HTTPS**-based protocol for initial pairing and large payloads, and an **MQTT**-based protocol for ongoing device communication. Client devices initiate both HTTPS requests and MQTT connections. Both

CC BY-ND

require only outbound access via port 443 to reach a public TR-12 host endpoint over the internet. As a result, the client and host need not share a network, VPN, or require the host to have prior knowledge of the client's private or public IP address. This outbound-only, port 443 architecture bypasses the need for complex firewall rules or VPN overhead, making TR-12 an ideal solution for connecting devices behind restrictive corporate or field networks to a public cloud or privately hosted service.

Fig 1 — System topology:



```
[Device/Camera] --> [TR-12 Client SDK] --> Port 443 --> [TR-12 Host Se
rvice]
    LAN/Firewall                                        HTTPS/MQTT        Cloud
```

A video streaming device within a corporate LAN runs an instance of a TR-12 Client SDK. The Client SDK runs as a standalone process and exposes a REST API on localhost. The device makes requests on the client via APIs such as `connect()`, `get_configuration()`, and `disconnect()`. The Client SDK handles all communication with the host service.

Note: TR-12 does not require use of the open-source client SDK. Provided a client interacts with the host via the TR-12 protocol described in this document, any client-side internal architecture is permitted.

TR-12 host service internal implementation is out of scope. TR-12 imposes few requirements on the host service other than successful implementation of the TR-12 pairing and device-facing MQTT API transactions. The internal host service architecture is outside the scope of this document. In practice, host services provide a centralized registry for device accessibility, monitoring, and configuration. These services shall communicate with clients according to the TR-12 HTTP and MQTT models defined in the Smithy specifications in Appendix A.

The TR-12 protocol is defined through API request/response models. Important API names and Smithy model field names are referenced throughout this document in italics, such as *CreatePairingCode* and *AuthenticatePairingCodeResponse*.

---

5 Host Configuration

A TR-12 host service advertises itself via a **Host Configuration** document. The following fields are required by various TR-12 APIs.

Field	Description
`serviceId`	Unique identifier for the host service. The pairing API requires this value. A host service shall reject pairing requests that do not provide a matching value.
`serviceName`	Human-readable service name. May be used to inform the user which service is being connected.
`deviceTypes`	A list of one or more of: `SOURCE`, `DESTINATION`, `BOTH`. A host service shall

CC BY-ND

Field	Description
	reject pairing requests for unsupported device types.
`pairingUrl`	Complete public REST endpoint implementing the *CreatePairingCode* API.
`authenticatePairingCodeUrl`	Complete public REST endpoint implementing the *AuthenticatePairingCode* API.
`thumbnailMaximumSizeKB`	Maximum image size accepted for thumbnail subscriptions. See Section 8.
`logFileMaximumSizeKB`	Maximum log file size accepted for log upload subscriptions. See Section 8.

6 Pairing

Pairing is the process by which a client requests and receives the credentials necessary to authenticate with a TR-12 host. The client is responsible for securely persisting these credentials. Once successfully paired, a client can initiate and maintain MQTT connections until the credentials expire or are revoked. Paired clients can automatically reconnect after disconnections, power cycles, or network interruptions without requiring an operator to re-pair the device.

A TR-12 host shall provide the following HTTPS public REST APIs to handle pairing, at the URLs identified in the Host Configuration document:

- `POST /pair` — implements *CreatePairingCode*
- `POST /authenticate` — implements *AuthenticatePairingCode*

6.1 The Pairing Process

The client shall make *CreatePairingCode* and *AuthenticatePairingCode* requests when credentials are not available for the given host.

Fig 2 — Pairing sequence:

```
Client                          Host
  |--POST /pair (CSR, hostId) -->|
  |<-- 200 {pairingCode} --------|
  |                              |  (operator claims device in console
)
  |--POST /authenticate -------->|
  |<-- {status: STANDBY} --------|  (repeat until CLAIMED)
  |--POST /authenticate -------->|
  |<-- {status: CLAIMED,         |
  |     caCert, deviceCert,      |
  |     mqttUri, hostSettings} --|
```


CC BY-ND

```
|--MQTT CONNECT (TLS) -------->|
|--PUBLISH registration ------>|
```

CreatePairingCode: The client shall make a single *CreatePairingCode* request. The host shall return HTTP 200 with a *CreatePairingCodeResponse* on success, or HTTP 400 with a *CreatePairingCodeException* on failure.

The host shall reject *CreatePairingCode* requests that:

1. Provide an unsupported `deviceType`
2. Provide an incompatible `version` (see Section 10.1)
3. Provide a `hostId` that does not match the host's `serviceId`

The *CreatePairingCodeException* `reason` field shall be one of:

Reason	Meaning
`HOST_ID_MISMATCH`	The `hostId` in the request does not match the host's `serviceId`.
`VERSION_NOT_SUPPORTED`	The client's protocol version is not supported by this host.
`DEVICE_TYPE_NOT_SUPPORTED`	The `deviceType` is not in the host's supported `deviceTypes` list.

AuthenticatePairingCode: The client shall poll *AuthenticatePairingCode* at no more than the throttle rate (see Section 6.2) until the response `status` is `CLAIMED`.

On a successful *AuthenticatePairingCodeResponse* with `status: CLAIMED`, the client shall persist the response payload, which includes the X.509 credentials, *HostSettings*, and MQTT URI needed to connect to the host service. The client shall immediately establish an MQTT connection and publish its registration to inform the host that pairing is complete.

Pairing Security Features (Informative)

The pairing process includes several built-in security measures:

- **Rate limiting:** The host service should implement TPS limits to prevent denial-of-service and brute-force attempts.
- **Temporal validity:** Pairing codes are valid only for a fixed window of time (`pairingTimeoutSeconds`).
- **Access code entropy:** A 32-character access code is returned alongside the pairing code. The high entropy of this code ensures that the probability of spoofing within the expiration window is negligible.
- **Transport security:** HTTPS secures all pairing request and response payloads in transit.
- **Device certificate:** Generated by the TR-12 host from the client-provided X.509 Certificate Signing Request (CSR). MQTT connection requires the device private key and certificate.
- **Credential rotation:** X.509 credentials are periodically updated via the TR-12 certificate rotation mechanism once connected (see Section 7.6).

CC BY-ND

6.2 Throttling

- *CreatePairingCode:* The client shall not exceed 1 request per minute.
- *AuthenticatePairingCode:* The client shall not exceed 1 request every 3 seconds.

6.3 Pairing Models

The following fields appear in *CreatePairingCodeRequest*, *CreatePairingCodeResponse*, *AuthenticatePairingCodeRequest*, and *AuthenticatePairingCodeResponse*. See Appendix A for complete model definitions.

Field	Description
`deviceType`	One of `SOURCE`, `DESTINATION`, or `BOTH`.
`hostId`	The `serviceId` of the target host service.
`certificateSigningRequest`	PEM-encoded X.509 CSR generated by the client. The private key never leaves the device.
`version`	The client's *ProtocolVersion*. See Section 10.1.
`deviceId`	Assigned by the host on successful pairing. Used in subsequent *AuthenticatePairingCode* requests.
`pairingCode`	Short code displayed to the operator to claim the device.
`accessCode`	32-character high-entropy code paired with `pairingCode` for security.
`pairingTimeoutSeconds`	Time window in which the pairing code is valid.
`caCertificate`	Host CA certificate in PEM format, returned on `CLAIMED`.
`deviceCertificate`	X.509 device certificate signed by the host CA, returned on `CLAIMED`.
`mqttUri`	MQTT broker endpoint URI, returned on `CLAIMED`.
`hostSettings`	*HostSettings* structure containing MQTT topic names and connection parameters, returned on `CLAIMED`.

CC BY-ND

7 MQTT Communication

7.1 Overview (Informative)

MQTT is a lightweight, publish-subscribe messaging protocol designed for secure, client-initiated connections to a central broker (the host service). By utilizing TLS and host-provided credentials, MQTT excels in IoT environments, maintaining always-on bidirectional communication through a persistent TCP connection. This architecture allows devices behind restrictive firewalls to receive real-time commands and transmit updates instantly. Its minimal header overhead ensures that low-latency payloads are delivered efficiently, even over unreliable or bandwidth-constrained networks.

All MQTT messages published by the host to a device shall use the **retained** flag. This ensures that a device that was offline when a message was published receives it immediately upon reconnecting, without requiring the host to republish.

7.2 MQTT Settings

The client and host shall use **MQTT v5.0** encrypted via TLS, using the credentials obtained during the pairing process.

The *AuthenticatePairingCodeResponse* → *HostSettings* → `mqttAlpnProtocol` field specifies the ALPN protocol string. The client shall configure its TLS settings using this string identifier when connecting on port 443.
Note: ALPN (Application-Layer Protocol Negotiation) is a TLS extension that allows the client and server to negotiate the application protocol during the TLS handshake. Strict corporate firewalls often block port 8883 (standard MQTT TLS) but allow port 443 (HTTPS). ALPN allows MQTT to run over port 443 by negotiating the protocol at the TLS layer. For connections on port 8883, no ALPN extension is needed and the `mqttAlpnProtocol` field is ignored.
The *HostSettings* structure returned in *AuthenticatePairingCodeResponse* contains all MQTT connection parameters:

Field	Description
`mqttAlpnProtocol`	ALPN protocol string for TLS negotiation on port 443.
`mqttKeepaliveSeconds`	MQTT keepalive interval. The client shall use this value.
`minimumIntervalPublishSeconds`	Minimum interval between status and configuration publications. The client shall not publish more frequently than this value.
`pairingTimeoutSeconds`	Pairing code validity window (informational after pairing).
`deviceSubscribesToDesiredConfigurationTopic`	Topic on which the host publishes desired configuration.
`deviceSubscribesToThumbnailSubscriptionTopic`	Topic on which the host publishes thumbnail subscription requests.

CC BY-ND

Field	Description
`deviceSubscribesToCertificateRotationTopic`	Topic on which the host publishes new certificates.
`deviceSubscribesToDeprovisionTopic`	Topic on which the host publishes deprovision commands.
`deviceSubscribesToLogSubscriptionTopic`	Topic on which the host publishes log upload requests.
`devicePublishesRegistrationTopic`	Topic to which the device publishes its registration.
`devicePublishesStatusTopic`	Topic to which the device publishes its status.
`devicePublishesActualConfigurationTopic`	Topic to which the device publishes its actual configuration.
`devicePublishesDeprovisionAcknowledgementTopic`	Topic to which the device publishes deprovision acknowledgement.

7.3 Processing a Configuration Update

Upon receipt of a *DesiredDeviceConfiguration* message on `deviceSubscribesToDesiredConfigurationTopic`, the client shall apply the requested configuration to all relevant parameters.

Version field: *DesiredDeviceConfiguration* and each *DesiredChannelConfiguration* include a `version` field (epoch nanoseconds string). The host shall update this field whenever any parameter within that scope changes. The client shall apply device-level and channel-level configuration only when the associated `version` has changed from the last applied value. By maintaining independent `version` values at the device level and per channel, configuration changes are applied only where needed, avoiding unnecessary device restarts.
Desired/Actual configuration: The client shall report actual configuration by publishing an *ActualDeviceConfiguration* on `devicePublishesActualConfigurationTopic` immediately after applying any desired configuration. The client shall read back actual state from the underlying device — it shall not simply reflect the desired configuration. The client shall echo the `version` values from the desired configuration at the corresponding device-level and channel-level in the actual configuration.
Health: *ActualDeviceConfiguration* and *ActualChannelConfiguration* include an optional `health` field. The client should report `healthy` unless a problem exists. When a problem exists, the client shall set `health` to `degraded` or `critical` with a descriptive `message` and `timestamp`. This allows the host to surface device-side issues to cloud operators.
Processing flow:

1. Receive *DesiredDeviceConfiguration* on `deviceSubscribesToDesiredConfigurationTopic`
2. Walk the configuration model; check `version` at device level and per channel
3. Apply changed entities to the device
4. Read back actual device state

CC BY-ND

5. Publish *ActualDeviceConfiguration* on `devicePublishesActualConfigurationTopic`, reflecting applied `version` values and current health

7.4 Registration

On connect: The client shall publish a *DeviceRegistration* payload on `devicePublishesRegistrationTopic` immediately upon connecting to the MQTT broker, including on every reconnect.

Host behavior: The host service shall ignore all other MQTT publications from the client until the *DeviceRegistration* payload has been successfully received and processed.
Re-registration: The client may publish an updated *DeviceRegistration* at any time while connected, provided the only change is to `channelTemplates[].profiles`. This allows the host to be informed of profile changes made by the device user since the device connected. The host shall detect and apply only changes to `profiles`. The host may update the active channel configuration profile if needed to reflect changes to the registered profiles.
The *DeviceRegistration* payload uses a **template/assignment pattern** to efficiently represent multi-channel devices:

- `channelTemplates`: An array of up to 5 unique capability definitions. Each template defines the `channelType`, supported `settings`, `profiles`, and transport `protocols` for a class of channels.
- `channelAssignments`: An array of up to 50 entries, each mapping a `channelId` and human-readable name to a `templateId`. Channels sharing identical capabilities reference the same template.

This pattern keeps registration payloads compact for devices with many identical channels.

7.5 Reporting Status

Once connected and after publishing a registration payload, the client may publish telemetry to the host via a *DeviceStatus* payload on `devicePublishesStatusTopic`.

The client shall not publish status more frequently than `minimumIntervalPublishSeconds` from *HostSettings*.
Status shall reference only channels described in the registration.

7.6 Certificate Rotation

Rotation is the process by which a client replaces the credentials obtained during initial pairing or a previous rotation. This mechanism allows the host to maintain security standards without requiring manual operator intervention at the device.

Expiration policy: The host shall determine the validity period of credentials and issue rotation requests as appropriate to enable persistent client access. For example, given a 12-month expiration, a host may rotate credentials monthly so that a device offline for up to 11 months can still reconnect without re-pairing. Rotating too frequently creates unnecessary client processing burden and a transient offline state; the host should not rotate credentials more than necessary.

CC BY-ND

Client execution: Upon receiving a *CertificateRotationPayload* on `deviceSubscribesToCertificateRotationTopic`, the client shall:

1. Compare the new `deviceCertificate` with the currently stored certificate.
2. If different: persist the new certificate and `mqttUri`, disconnect from the MQTT broker, and reconnect using the updated credentials.
3. If identical: take no action (idempotent).

7.7 Deprovision

Deprovisioning ends a client's lifecycle. A subsequent reconnect requires re-pairing. Both the host and client may initiate a deprovision. The TR-12 deprovision process is designed to be transactional to avoid leaving a client in an ambiguous state when deprovisioned while offline.

Host-initiated deprovision:
Upon receiving a *DeprovisionPayload* on `deviceSubscribesToDeprovisionTopic`, the client shall:

1. Publish a *DeprovisionPayload* acknowledgement on `devicePublishesDeprovisionAcknowledgementTopic`.
2. Immediately disconnect from the MQTT broker.
3. Delete all associated persistent credentials from local storage.

Client-initiated deprovision:

The client shall:

1. Publish a *DeprovisionPayload* on `devicePublishesDeprovisionAcknowledgementTopic`.
2. Immediately disconnect from the MQTT broker.
3. Delete all associated persistent credentials from local storage.

Host finality: The host shall permanently delete the device's credentials and associated resources only after receiving the client's acknowledgement message. The host may delete credentials after a grace period equal to 3× the default credential expiration if no acknowledgement is received.

The *DeprovisionPayload* `reason` field shall be one of:

Reason	Meaning
`DEPROVISIONED`	Explicitly deprovisioned by the host operator.
`EXPIRED`	Device credentials expired and were not rotated.
`UNKNOWN`	Reason not specified.

(cc) BY-ND

8 Subscriptions

8.1 Overview (Informative)

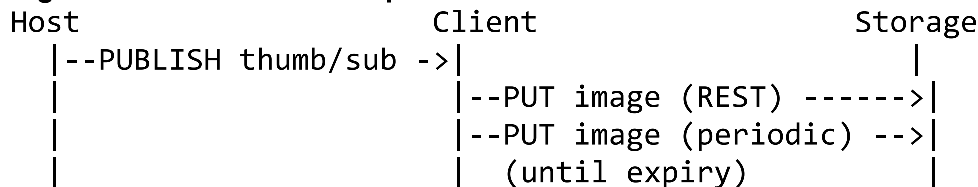
MQTT payload size and service operating cost constraints make it unsuitable for large, frequent payloads such as images and log files. TR-12 defines a **subscription** mechanism for these cases. A subscription is an MQTT request from the host that specifies a delivery frequency, maximum size, expiration time, and a REST PUT endpoint to which the client delivers the payload.

The host service will generally use a pre-signed URL to a persistent storage service, valid for 5 to 30 minutes. All major cloud services offer pre-signed URLs for secure temporary write access to cloud storage via a REST endpoint.
All subscription messages published by the host shall use the retained flag, so that a device reconnecting after an outage receives the current subscription state immediately.

8.2 Thumbnail Subscription

The host may request periodic thumbnail image delivery from the client by publishing a *ThumbnailSubscriptionPayload* on `deviceSubscribesToThumbnailSubscriptionTopic`.

Fig 4 — Thumbnail subscription:



Thumbnails are mapped to channels. The *ThumbnailSubscriptionPayload* contains a map of `channelId` → *ThumbnailSubscription*. The client resolves the local image path for each channel from the `thumbnailLocalPath` field in *ActualChannelConfiguration*.

Supported formats: The client shall ensure delivered image files use one of the following extensions: `.jpg`, `.jpeg`, `.JPG`, `.JPEG`, `.png`, `.PNG`.
Delivery: The client shall upload images to the `remotePath` URL via HTTP PUT at the interval specified by `periodSeconds`, until the subscription `expiresAt` time is reached.
Size limit: The client shall not upload images exceeding `maxSizeKB` kilobytes. The host's maximum accepted size is advertised in the Host Configuration `thumbnailMaximumSizeKB` field.
Subscription replacement: A new *ThumbnailSubscriptionPayload* replaces any existing subscription. The client shall not accumulate subscriptions.
The *ThumbnailSubscription* fields per channel:

Field	Description
`periodSeconds`	Upload interval in seconds.
`expiresAt`	Subscription expiry timestamp (ISO 8601).
`maxSizeKB`	Maximum accepted image size in kilobytes.
`remotePath`	REST PUT endpoint URL for image delivery.

cc BY-ND

Field	Description
headers	Optional HTTP headers to include in the PUT request (e.g. authorization).

9 Nominal Operation (Informative)

9.1 Connect and Respond Loop

Fig 3 — Normal operation loop:

```
Client                          Host
  |--PUBLISH registration ------>|
  |--PUBLISH status (periodic) ->|
  |<-- PUBLISH config/update ----|
  |  (apply config to device)    |
  |--PUBLISH actual config ------>|
  |  (repeat)                    |
```

After completing pairing, the client establishes an MQTT connection and immediately publishes its *DeviceRegistration*. While connected, the client periodically publishes *DeviceStatus*. When the host operator pushes a new desired configuration, the client receives it, applies the changes to the device, reads back the actual state, and publishes *ActualDeviceConfiguration*. This cycle continues until the device disconnects or is deprovisioned.

The client should implement an automatic reconnect loop with exponential backoff. On each reconnect, the client shall re-publish its *DeviceRegistration* before any other publications. Because all host-to-device MQTT messages are retained, a device that was offline when a configuration update, certificate rotation, thumbnail subscription, or deprovision command was published will receive it immediately upon reconnecting — without requiring the host to republish.

9.2 Subscription Overview

Fig 4 — Thumbnail subscription (repeated for context):

```
Host                    Client                   Storage
  |--PUBLISH thumb/sub ->|                         |
  |                      |--PUT image (REST) ------>|
  |                      |--PUT image (periodic) -->|
  |                      |  (until expiry)          |
```

The host requests thumbnail delivery by publishing a retained *ThumbnailSubscriptionPayload*. The client uploads images to the host-provided REST endpoint at the requested interval until the subscription expires. The client manages each channel's thumbnail upload independently.

CC BY-ND

9.3 Deprovision Lifecycle

Fig 5 — Deprovision:

```
Host                          Client
  |--PUBLISH deprovision ------->|
  |<-- PUBLISH deprovision/ack --|
  |  (host deletes credentials)  |
  |                              | (client deletes local credentials)
```

The host publishes a retained *DeprovisionPayload* on the device's deprovision topic. The client acknowledges by publishing a *DeprovisionPayload* on its acknowledgement topic, then disconnects and deletes local credentials. The host permanently removes the device record only after receiving the acknowledgement.

If the client initiates deprovision (e.g. user action on the device), it publishes the acknowledgement first without waiting for a host request. The host performs the same cleanup on receipt.
The retained deprovision message ensures that a device offline at the time of deprovisioning receives the command on its next reconnect and completes the lifecycle cleanly.

---

10 The TR-12 Models (Informative)

The TR-12 models define the client-host MQTT and HTTPS request/response contract. All client-host behaviors are built around these models. They provide common, versioned, and validatable message payloads and responses.

The models are defined using **Smithy**, an open-source interface definition language created by AWS and widely adopted across the industry. Smithy is protocol-agnostic — it describes the abstract API model, which can be compiled into HTTP REST, MQTT, gRPC, or other transports. Smithy's code generation toolchain produces programmatic class models and APIs in most languages from a single source of truth.
OpenAPI models are automatically generated from the Smithy definitions and are available in the TR-12 Models repository at: https://github.com/vsf-tv/TR-12-Models

10.1 Protocol Version

The *ProtocolVersion* structure is included in every *CreatePairingCodeRequest*. It allows the host to reject connections from devices with an incompatible protocol version.

```
structure ProtocolVersion {
    @default("6.0.0")
    version: String
}
```

The current protocol version is **6.0.0 (6/1/2026)**.

TR-12 protocol versioning follows semantic versioning (MAJOR.MINOR.PATCH):

(cc) BY-ND

Component	Meaning
MAJOR	Breaking change: removed or renamed field, type change, removed enum value, new required field, or handshake change. Old clients or hosts will fail. Not permitted after the TR-12 draft is ratified.
MINOR	Additive, backwards-compatible change: new optional field, new enum value, new operation, or new transport protocol variant. Old implementations safely ignore unknown fields.
PATCH	No wire format change: documentation or comment updates only.

The host shall reject pairing requests from a client whose MAJOR or MINOR version exceeds the host's supported version. The host should be backward compatible with all MINOR versions below its currently supported version.

10.2 Model Overview

TR-12 defines three primary protocol models:

Model	Direction	Purpose
DeviceRegistration	Device → Host	Advertises device capabilities: channel templates, assignments, settings, profiles, and supported transport protocols. Published on every MQTT connect.
DesiredDeviceConfiguration / *ActualDeviceConfiguration*	Host → Device / Device → Host	Carries the desired configuration from host to device, and the actual applied configuration from device to host.
DeviceStatus	Device → Host	Carries device-level and per-channel telemetry (bitrate, health, running state, etc.).

10.3 Registration Model

The *DeviceRegistration* payload is the primary mechanism by which the client defines its capabilities to the host. It uses a **template/assignment pattern**:

CC BY-ND

- **`channelTemplates`** (max 5): Unique capability definitions. Each *ChannelTemplate* has an `id`, `channelType` (`SOURCE` or `DESTINATION`), and optional `settings`, `profiles`, and `protocols` arrays.
- **`channelAssignments`** (max 50): Maps each `channelId` and human-readable `name` to a `templateId`. Multiple channels may reference the same template.
- **`settings`** (optional): Device-level settings not specific to any channel.

Note: *ChannelTemplate* has no `name` field. The channel name is carried on the *ChannelAssignment*.

Settings are field-value pairs with constraints. Each *Setting* has an `id`, `name`, `description`, and a `constraint` that is either:

- `enums`: A list of valid string values and a default.
- `ranges`: A numeric minimum, maximum, and default.

The host uses the registration to auto-generate a management console UI with type-appropriate widgets (toggles, sliders, dropdowns) without any device-specific front-end development.

Profiles are composite settings identified by an `id`, `name`, and `description`. A profile selection replaces individual setting values for a channel. Profiles and standard settings are mutually exclusive within a channel configuration.
Transport protocols advertised per template determine which *TransportProtocol* variants the host may include in a channel's desired configuration. Supported protocol names: `SRT_LISTENER`, `SRT_CALLER`, `ZIXI_PUSH`, `ZIXI_PULL`, `RIST_SIMPLE_CALLER`, `RIST_SIMPLE_LISTENER`, `RTP`
Example registration (abbreviated):

```
{
  "deviceRegistration": {
    "channelTemplates": [
      {
        "id": "tmpl-encoder",
        "channelType": "SOURCE",
        "settings": [
          {
            "id": "RS01",
            "name": "resolution",
            "description": "Output video dimensions.",
            "constraint": {
              "enums": {
                "values": ["1920x1080", "1280x720", "640x480"],
                "defaultValue": "1920x1080"
              }
            }
          },
          {
```


CC BY-ND

```
            "id": "MB01",
            "name": "max_bitrate",
            "description": "Maximum data rate limit (Kbps).",
            "constraint": {
              "ranges": {
                "minimum": 5000,
                "maximum": 50000,
                "defaultValue": 10000
              }
            }
          }
        ],
        "profiles": [
          { "id": "h265k", "name": "h265_4k", "description": "H.265, 6
0fps, 40Mbps, 4K, 3s GOP" }
        ],
        "protocols": ["SRT_CALLER", "SRT_LISTENER"]
      }
    ],
    "channelAssignments": [
      { "channelId": "CH01", "name": "Encoder Channel 1", "templateId"
: "tmpl-encoder" }
    ]
  }
}
```


10.4 Configuration Model

The configuration model has two variants:

- *DesiredDeviceConfiguration*: Sent from host to device. Contains only fields the host controls.
- *ActualDeviceConfiguration*: Reported from device to host. Extends the desired fields with device-reported fields (`health`, `thumbnailLocalPath`).

Both carry a `version` string at the device level and per channel. The host shall stamp a new `version` (epoch nanoseconds) on any entity whose content changes. The client shall apply only entities whose `version` has changed.

Each *DesiredChannelConfiguration* / *ActualChannelConfiguration* contains:

Field	Description
`id`	Channel identifier. Must match a `channelId` in the registration.
`version`	Epoch nanoseconds string. Stamped independently per channel.

CC BY-ND

Field	Description
state	ACTIVE or IDLE.
channelSettings	Either standardSettings (list of {id, value} pairs) or profile ({id}). Mutually exclusive.
protocol	Optional *TransportProtocol* union. One of: srtListener, srtCaller, ristSimpleListener, ristSimpleCaller, zixiPush, zixiPull, rtp.
health	(Actual only) *Health* union: healthy, degraded, or critical.
thumbnailLocalPath	(Actual only) Local filesystem path to the channel's thumbnail image. Used by the client to service thumbnail subscriptions. The host may ignore this field.

The host shall validate configuration payloads against the device's registration before publishing:

- Each channel id must match a channelId in the registration.
- Each standardSettings entry id must match a Setting.id in the channel's template.
- Each profile id must match a ProfileDefinition.id in the channel's template.
- If a protocol variant is specified, it must correspond to a protocol name in the channel template's protocols list.

Example desired configuration (abbreviated):

```
{
  "desiredDeviceConfiguration": {
    "version": "1748000000000000000",
    "channels": [
      {
        "id": "CH01",
        "version": "1748000000000000001",
        "state": "ACTIVE",
        "channelSettings": {
          "standardSettings": [
            { "id": "RS01", "value": "1920x1080" },
            { "id": "MB01", "value": "15000" }
          ]
        },
        "protocol": {
          "srtCaller": {
```


CC BY-ND

```
            "address": "ingest.example.com",
            "port": 9000,
            "minimumLatencyMilliseconds": 120
          }
        }
      }
    ]
  }
}
```


10.5 Status Model

The *DeviceStatus* payload carries device-level and per-channel telemetry. It is published on `devicePublishesStatusTopic` at no more than `minimumIntervalPublishSeconds`.

DeviceStatus contains:

- `status`: A list of *StatusValue* entries for device-level metrics (e.g. CPU load, memory usage, connectivity).
- `channels`: A list of *ChannelStatus* entries, one per active channel.

Each *ChannelStatus* contains:

- `id`: Channel identifier. Must match a `channelId` in the registration.
- `state`: Current channel state (`ACTIVE` or `IDLE`).
- `status`: A list of *StatusValue* entries for channel-level metrics (e.g. bitrate, packet loss).

Each *StatusValue* contains:

- `name`: Metric name (e.g. `"bitrate-bps"`).
- `description`: Human-readable description (e.g. `"Actual bitrate."`).
- `value`: Current value as a string (e.g. `"15234000"`).

Example status (abbreviated):


```
{
  "deviceStatus": {
    "status": [
      { "name": "cpu", "description": "Current CPU utilization.", "val
ue": "41%" }
    ],
    "channels": [
      {
        "id": "CH01",
        "state": "ACTIVE",
        "status": [
          { "name": "bitrate-bps", "description": "Actual bitrate.", "
value": "15234000" }
```


(cc) BY-ND

```
        ]
      }
    ]
  }
}
```

Status shall reference only channels described in the registration.

Appendix A: Smithy Models

The complete, authoritative Smithy model definitions for the TR-12 protocol are maintained at:

https://github.com/vsf-tv/TR-12-Models
The repository contains the following source files:

File	Contents
`model/shapes/common.smithy`	*ProtocolVersion*, pairing and auth models, *HostSettings*, *Health*, *CertificateRotationPayload*, *DeprovisionPayload*, *ThumbnailSubscriptionPayload*, *LogSubscriptionPayload*
`model/shapes/registration.smithy`	*DeviceRegistration*, *ChannelTemplate*, *ChannelAssignment*, *Setting*, *ProfileDefinition*, *ProtocolList*
`model/shapes/configuration.smithy`	*DesiredDeviceConfiguration*, *ActualDeviceConfiguration*, *DesiredChannelConfiguration*, *ActualChannelConfiguration*, *ChannelSettings*, *TransportProtocol* union and all transport protocol structures
`model/shapes/status.smithy`	*DeviceStatus*, *ChannelStatus*, *StatusValue*
`model/service.smithy`	*HostServiceApi* — all HTTP and MQTT operation definitions

OpenAPI models are automatically generated from the Smithy source and are available in the repository under `build/smithy/source/openapi/`.
Code generation for Go, TypeScript, Python, C++, and other languages is supported via the `generate-tr12-models.sh` script in the repository.

CC BY-ND