Video Services Forum (VSF)
Technical Recommendation TR-12

Client Device Discovery (CDD)

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

March 4, 2026
VSF_TR-12_2026_03_04

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

http://www.videoservicesforum.org

CONTENTS HEREOF, AND THESE CONTENTS WOULD NOT BE PUBLISHED BY VSF WITHOUT SUCH LIMITATIONS.

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 API 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.

Table of Contents

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.
- **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 participated in the Video Services Forum TR-12 working group that help to shape this technical recommendation and provide feedback.

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.

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

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 is comprised of two distinct communication components: an **HTTPS**-based protocol for initial pairing and large payloads, and an **MQTT**-based protocol for message communication. Client devices initiate both HTTPS requests and MQTT connections. Both require only outbound access via Port 443 to reach a public TR-12 service endpoint **over the internet** and to complete all necessary transactions. As a result, the client and host need not share a network, VPN, or require the host have prior knowledge of the client’s private or public IP address. This "outbound-only", Port 443 architecture effectively bypasses the need for complex firewall rules or VPN overhead, making TR-12 an ideal solution for connecting devices located behind restrictive corporate or field networks to a public cloud (or privately hosted) service.

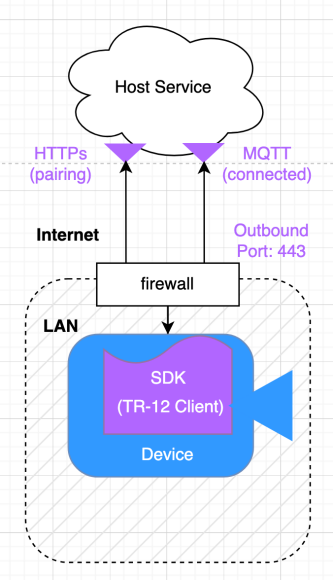


Fig1 shows a typical TR-12 system topology. A video streaming device within a corporate LAN/Firewall is running an instance of a **TR-12 Client SDK**. (https://github.com/vsf-tv/gccg-cdd). Here, the Client SDK runs as a stand-alone process and exposes a REST API on localhost. The device (a camera in this example) simply makes requests on the client via convenient APIs like *connect(), get configuration(),* disconnect(), etc. The Client SDK, in turn, handles all communication with the Host Service. *Note: TR-12 doesn't not require use of the open-source client SDK: Provided a client interacts with the host via the TR-12 protocol describe in this document, any client-side internal architecture is permitted*.

TR-12 host service's internal implementation is out of scope.
TR-12 imposes few requirements on the *host service* other than the successful implementation of the TR-12 pairing and device-facing MQTT API transactions. The internal host service architecture—specifically how a provider implements backend access to registered clients—is outside the scope of this document. In practice, host services will provide a centralized registry for device accessibility, monitoring, and configuration. These services must communicate with clients according to the TR-12 HTTP and MQTT models defined in the SMITHY specifications found in the Appendices.

Models:
The TR-12 protocol is defined primarily through API request/response models defined in the Appendix: SMITHY definition. Important API names and SMITHY model field names are referenced throughout this document, highlighted via italics such as *GetPairingCode* API and *AuthResponse Model.*

5 Host Configuration

TR-12 host services can advertise themselves via a simple **Host Configuration**. The following fields are needed by various TR-12 APIs.

serviceId: The pairing API requires this param. A host service shall reject pairing requests that do not provide a matching value.

serviceName: Describes the service name (friendly name) and may be used to inform the user to which service is being connected.

deviceTypes: A list of one or more the following: "SOURCE", "DESTINATION", "BOTH". A service may reject a pairing request for unsupported device types. A TR-12 service must support “SOURCE”, “DESTINATIN” or “BOTH”.

pairingUrl: A complete, public REST endpoint that implements the *GetPairingCode* API.

authUrl: A complete, public REST endpoint that implements the *AuthenticatePairingCode* API.

thumbnailMaxSizeKB: Maximum size accepted for supporting thumbnail subscriptions: See Subscriptions

logFileMaxSizeKB Maximum size accepted for supporting log upload subscriptions: See Subscriptions

6 Pairing

Pairing is the process by which a client requests and receives the necessary credentials 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 by the host or the client. Paired clients can automatically reconnect after disconnections, power cycles or network interruptions without requiring an operator intervention to re-pair the device.

A TR-12 host shall provide the following HTTPS public REST APIs to handle pairing, the URLs for which are identified in the Host Configuration document described above. See Appendix *HostServiceApi: Pair and Authenticate Request.*

6.1 The pairing process

The client shall make Pair and Auth requests when credentials are not available for the given host. The client shall make a single *Pair* request and only proceed to make Auth requests if the PairResponse returns success via the *PairResponse:PairResult:PairSuccessData*.

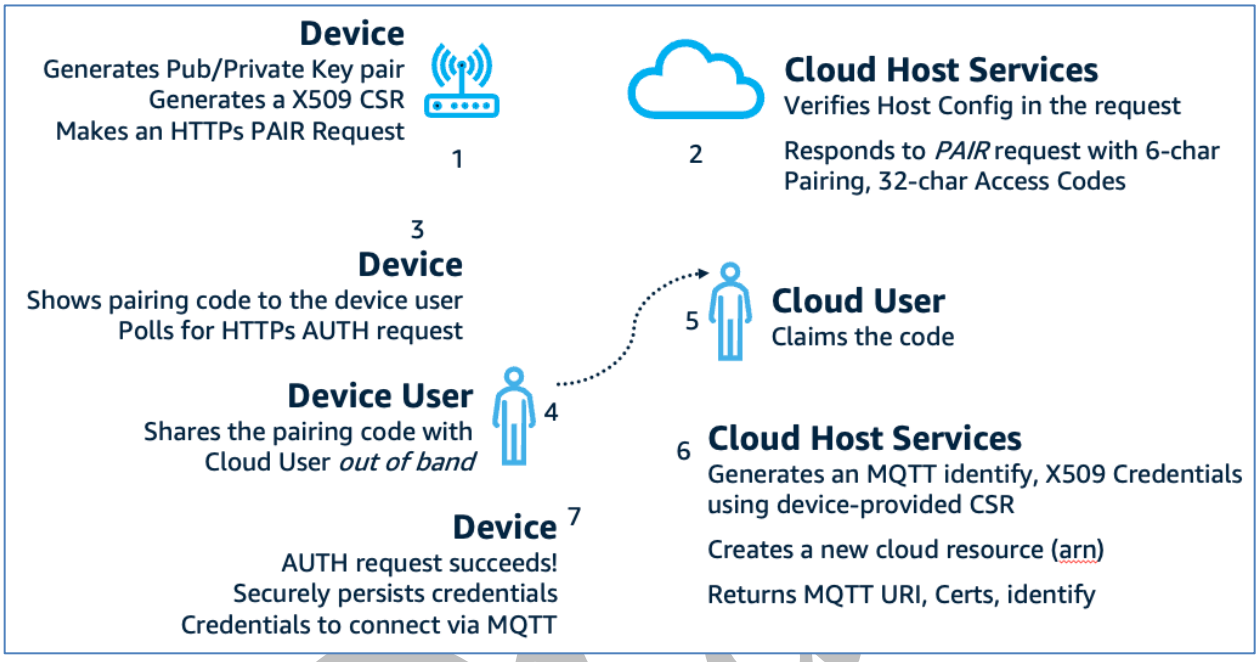
The host shall reject Pair API request whose *PairRequest*: 1) Provides an unsupported *PairRequest:deviceType* 2) Not a compatible PairRequest:*ProtocolVersion*

The client shall make Authenticate requests at less than the Throttle Rate (see below) until the *AuthResponse:AuthStatus* == ‘CLAIMED’.

On successful *AuthResponse*, the client shall persist the *AuthResponse* payload that includes among other parameters the X509 credentials, HostSettings, and MQTT URI (endpoint) that are

needed to connect to the host service. The shall immediately connect to inform the host service that the pairing process is complete.

Fig 2 shows the pairing process steps initiated by the client Device on initial connection if no credentials corresponding to the given host are available. After successful pairing, the device persists the credentials. The credentials are used to connect to the host service via MQTT.



Pairing Security Features (Informative)

The pairing process includes several built-in security measures to protect against unauthorized device claims and brute-force attacks:

- **Rate Limiting (TPS):** The host service will implement **Transactions Per Second (TPS)** to prevent denial-of-service (DoS) and brute-force attempts.
- **Temporal Validity:** Pairing codes are only valid for a fixed window of time.
- **Access Code Entropy:** a **32-character access code**. The high entropy of this code ensures that the probability of spoofing within the expiration window is low.
- **Mitigation of Malicious Claims:** TPS limits "Host-side claim spoofing"
- **Transport Security: HTTPS** secures pairing request/responses payloads in transit.
- **Device Cert:** is generated by the TR-12 host using the client-provided **X509 Certificate Signing Request** (CSR). Client MQTT connection requires the device private key and CSR.
- **Rotation:** x509 credentials are periodically updated via the TR-12 API once connected.

6.2 Throttling

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

6.3 Pairing Models

The following params described here appear in the Appendix*: PairRequest* and *PairResult: PairSuccessData* models.

- **deviceType:** must be one of “SOURCE”, “DESTINATION” or “BOTH”
- **hostsId:** The client provides a string describing the device type or unique instance
- **csr:** The client provides an X509 Certificate Signing Request to the *PairRequest*
- **caCert:** The host Certificate Authority CA Certificate in PEM format <string>.
- **deviceCert: :** X509 Device Certificate generated by the host using the client provided CSR. On authorization, the host shall use the client-provided CSR and the hosts chosen Certificate Authority (CA) to generate an X509 device certificate.
- **version:** The client must identify the TR-12 SMITHY: *ProtocolVersion*: A host service shall reject pairing requests for unsupported versions. A host service must be backward compatible with all SMITHY model versions below the host’s currently supported version.

7 MQTT Communication

7.1 Overview (Informative)

Protocol Overview MQTT is a lightweight, publish-subscribe (pub-sub) 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.

7.2 MQTT Client: Settings

Both the client/host MQTT setting shall use the following the **MQTT v5.0** protocol encrypted via TLS using the credentials obtained during the pairing process.

The *AuthResponse:HostSettings:iotProtocolName* parameter describes the ALPN protocol string. The client shall configure the IOT Client TSL settings using this string identifier.

> *Note: **ALPN** stands for Application-Layer Protocol Negotiation. It is a TLS (Transport Layer Security) extension that allows the client and server to negotiate which application*

protocol they are going to use during the initial TLS handshake, before any actual data is sent. Typically, secure MQTT runs over port 8883. However, strict corporate firewalls often block port 8883, only allowing standard web traffic through ports 80 (HTTP) and 443 (HTTPS). ALPN solves this problem. It allows your MQTT client to connect to the broker on port 443

7.3 MQTT Client: Processing a Configuration change.

Configuration Updates: Upon receipt of a configuration update, the client shall immediately apply the requested configuration to all relevant parameters. Following processing the update, the client shall publish its current state (*DeviceConfiguration*) to the *pubReportActualConfigurationTopic*. This creates the "Request → Apply → Acknowledge" loop, ensuring the host has a complete and confirmed record of the device's actual state.

When the device can not comply with a desired configuration (Informative)

There are a few reasons a client may not be able to comply with a *desired configuration request* including an unsupported combination of *DeviceConfiguration: SimpleSetting*, a bug or transient error state on the device. The host is informed via the actual configuration message on the *pubReportActualConfigurationTopic* and can thus inform the host user, raise and alert if desired.

***Note**: **A Proposed Update being considered to DeviceConfiguration model**: to includes a (required?) field (type string) for the client to provide a reason the actual configuration differs from the requested. The host service could forward this information to the host operator.*

7.4 MQTT Client: Registration

Client Registration: Registration payload is the primary mechanism by which the client defines its capabilities to the host.

- **On Connect:** The client shall publish the Registration payload immediately upon connecting to the MQTT broker on the *pubReportRegistrationTopic*
- **Host Logic:** The host service shall ignore all other MQTT requests from the client until this *Registration* payload is successfully received and processed.

7.5 MQTT Client: Reporting Status

Once connected and the client has published a registration payload, the client may provide telemetry to the host via a valid *DeviceStatus* payload model on the *pubReportStatusTopic. See Appendix: DeviceStatus.*

7.6 MQTT Client: Certificate Rotation

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

Expiration Policy: The host shall determine the validity period of credentials and make rotation requests to the client they deem appropriate to enable persistent client access. ***Note:*** *Given an example expiration of 12 moths, a host may elect to rotate credentials every month to ensure that a device can be offline up to 11 months and still be able to reconnect without re-pairing. A device must of course be online to process credential rotation. Rotating too often creates added client processing burden as well as a transient offline state, so while possible to rotate frequently, it's not recommended to rotate credentials more than necessary.*

Client Execution: Upon receiving a *CertRotate* message, the client shall immediately replace its local credentials, perform an MQTT disconnect, and re-establish the connection using the updated certificates. The client shall ignore *CertRotate* requests containing a duplicate payload.

7.7 MQTT Client: Deprovision

Deprovisioning is the process by which a client's lifecycle ends and subsequent reconnect requires (re)pairing. Both the host and client may initiate a deprovision. To avoid leaving a client in an ambiguous state when it has been deprovisioned while offline (not connected to the MQTT host)—where it cannot distinguish between a revoked credential and a network outage—the TR-12 deprovisioning process is designed to be transactional.

Client Requirements:

Service-Side deprovision: This is a deprovision initiated by the host service. The client, upon receiving a message on the *subDeprovisionTopic*, shall:

Client-Side Deprovision: This is a deprovision initiated by the device or device user. The client shall:

1. Publish an acknowledgment message to the *pubDeprovisionTopic*.
2. Immediately disconnect from the MQTT broker.
3. Delete all associated persistent credentials from the local file system.

Host Requirements:

Finality: Only after the host receives the client's **acknowledgment** message, shall the host permanently delete the credentials and associated resources.

Credential Grace Period. The host may delete the credentials after a period equal to 3x the default credential expiration if there was no client's **acknowledgment** message received.

8 Subscriptions

TR-12 provides a mechanism where a host service may request delivery of large payloads via a subscription. A subscription is a request to provide a given client payload that include a frequency, max-size and expiration to a given REST Put endpoint. TR-12 describes subscriptions for Thumbnails and Logs.

8.1 Overview (informative)

MQTT payload size and service operating cost limitations may not be suited for large payloads. As such, host-service operators will want control over when and how these large payloads are transmitted and paid for.

TR-12 defines a process for requesting clients send large, frequent payloads called **Subscriptions**. Subscriptions are an MQTT request, with delivery accomplished via the client making a REST PUT request to a host-provided, temporary secure endpoint defined by the: *ThumbnailRequest:remotePath*

The host service will generally use a pre-signed URL to a persistent host storage service valid from 5 to 30 minutes. The host service may even elect to require the user requesting the subscribed file to provide the destination resource URL and necessary access permissions to defer handling and storage costs to the user. *Note: all major cloud services offer pre-signed URLs for secure temporary write access to secure cloud storage via a REST endpoint.*

8.2 Thumbnail Subscription

- **Registration:** The client may advertise the availability of Thumbnail images via the *Registration:ThumbnailList. The client may omit this setting if the following conditions can be satisfied.*
- **Image Type:** *Registration:ThumbnailList: Thumbnail*:***localPath*** describes the complete path where the device's generated images are written.
- **JPEG and PNG are supported.** The client shall ensure that images files include the following suffix: .jpg .JPEG .JPG .png .PNG.

- **Request:** the host may request thumbnails via the subThumbnailSubcriptionTopic with the *ThumbnailRequest* model.
- **Delivery:** The client shall deliver images according to the subscription request until the request has expired.

9 Nominal Operation Overview (informative)

9.1 Connect and Respond Loop (Informative)

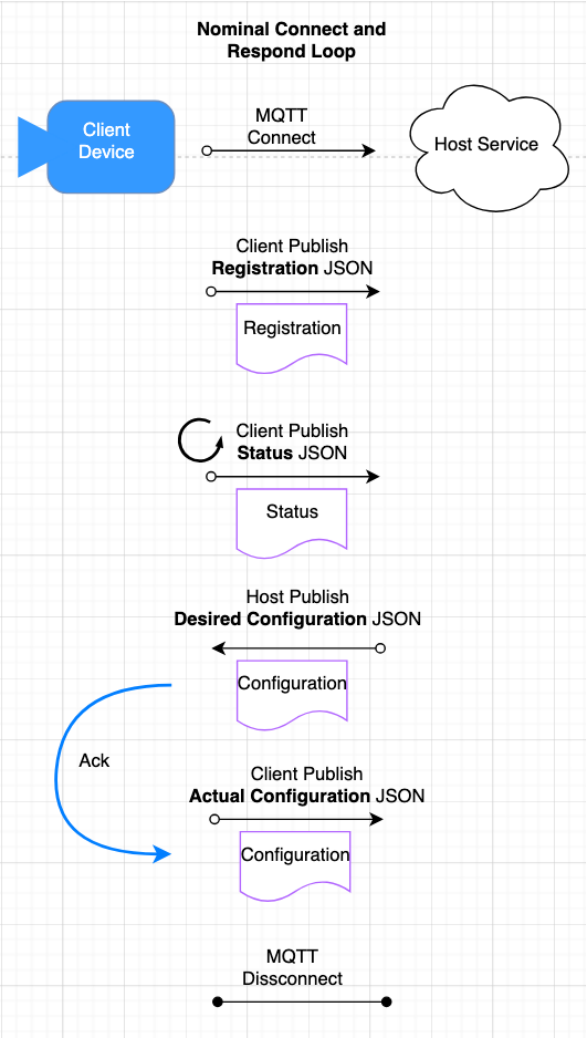


Fig3 shows a client connecting and then immediately publishing the **Registration** JSON. While connected, the client periodically updates status by publishing a **Status** JSON. The host service publishes a desired **Configuration** JSON request. The client device immediately updates its internal settings to comply then immediately publishes the actual **Configuration** JSON. This cycle continues until the device disconnects.

9.2 Subscription Overview (Informative)

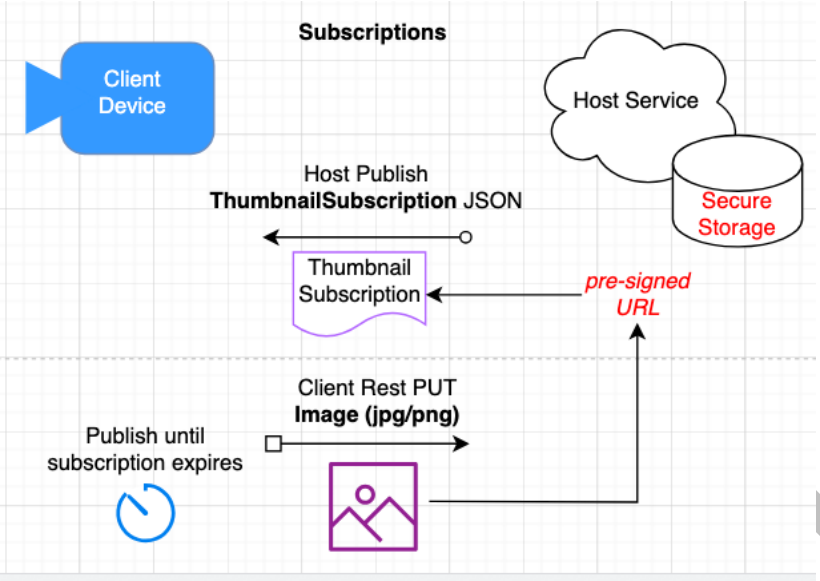


Fig4 shows a host requesting the client send thumbnail images periodically until the subscription expires to the given pre-signed, secure, temporary URL. The client repeatedly will upload images to the pre-signed URL via REST PUT.

9.3 Lifecycle Deprovision Overview (Informative)

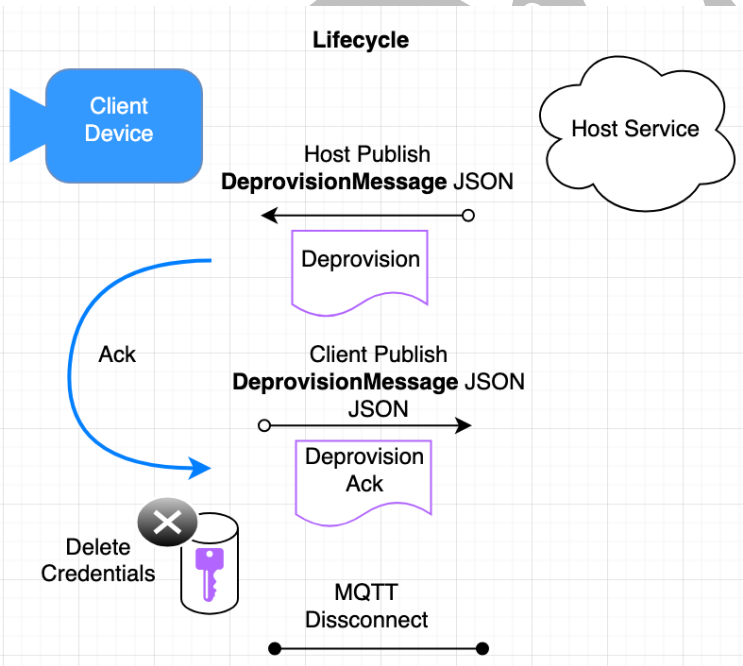


Fig5 Shows a host deprovisioning the client (Host Publish) and the Client acknowledging the request. Alternatively, the client can initiate a deprovision by publishing the same acknowledge message without a request from the host service. The client finally deletes credentials for this host.

10 Models (informative)

Why SMITHY: While SMITHY was created by AWS to manage its own massive ecosystem, it was released as an open-source project and has seen significant adoption by other large tech companies and open-source frameworks. Because SMITHY is protocol-agnostic (it doesn't force you into REST/HTTP like OpenAPI does), it is highly attractive to teams building complex microservices, RPC frameworks, or multi-language environments. SMITHY also provides a code generation mechanism that also translates into OpenAPI models, then creates programmatic class models and APIs in most languages. OpenAPI is fundamentally designed to describe HTTP REST APIs, often functioning as a documentation tool first and a code-generation tool second. Smithy was built from the ground up for code generation first. It treats the API as an abstract, mathematical model that can be compiled down into an HTTP REST server, a gRPC service, an MQTT stream, or a local CLI tool without having to rewrite the core business definitions.

Note: OpenAPI models are automatically generated *and are available in the open-source TR-12 SDK at: ../build/smithy/source/openapi.*

Integrating the TR-12 model into a client program requires building the desired model in the desired language and calling a handful of HTTP and MQTT APIs. The generated models will handle serialization and validation.

Registration, Configuration, and Status models are formally defined in the Appendices via **SMITHY** specifications. These models act as a contract between the client and the host, ensuring that both sides have a shared understanding of the device's capabilities and data structures.

The Registration Process Upon initial connection, the client informs the host of its specific system capabilities. This registration payload includes:

> **System Capabilities:** Specific configuration parameters on the local system.
>
> **Channel Architecture:** The number of available media channels.
>
> **Per-Channel Settings:** Specific configuration parameters available for each individual channel.

10.1 Registration Model

Dynamic UI Generation: The host can auto-generate a web-based management console. Using the types defined in the registration (e.g. integer, and ENUM string), the host renders "type-

appropriate" UI widgets (such as toggles, sliders, entry fields, and pull downs widgets) without requiring device-specific front-end development.

Simple Settings: field-value pairs whose values can be **enums** or **ranges**. These are used to precisely configure device parameters.

Profiles: A profile is an ID, Friendly Name and description. These are used to select a 'composite' setting that may affect any number of actual channel settings.

Registration: The following is a partial Registration payload where *simpleSettings* are defined as *enums* and *range*s as well as *Profiles*. See Appendix: *Registration* for the complete model.

simpleSettings: Example.

```
            "name": "max_bitrate",
            "info": "Maximum data rate limit (Kbps).",
            "id": "MB01",
            "ranges": {
              "min": 5000,
              "max": 50000,
              "defaultValue": 10000
            }
Or

            "name": "resolution",
            "info": "Output video dimensions.",
            "id": "RS01",
            "enums": {
              "values": [
                "1920x1080",
                "1280x720",
                "640x480"
              ],
              "defaultValue": "1920x1080"
            }
```

Profiles: Example:

```
          {
            "name": "h265_4k",
            "id": "h265k",
            "info": "H265, 60fps, 40mps, 4k, 3s gop"
          },
```

The host service shall validate Configuration payloads range and enum values against the Registration payload. **For example,** *a Configuration payload where the MB01=50001 is invalid due to the range setting maximum of 50000 for that param.*

10.2 Configuration Model

Configuration is formally defined in the Appendices. The Configuration payload is dependent on the **Registration** payload; the host service shall use the registration elements to validate configuration requests. The host shall ensure the *Configuration* payload contains only params defined in the *Registration* and conforms to defined *enums* and *ranges* values and or Profiles.

10.3 Status Model

System Status: Global health metrics (e.g., CPU load, memory usage, or connectivity strength).

Per-Channel Status: A structured list containing the **ID**, **Info**, and **Value** for each active media or data path.

System Status Example:

```
{
  "name": "cpu",
  "value": "41%",
  "info": "Current CPU utilization."
}
```

Channel Status Example

```
{
 "name": "bitrate-bps",
 "value": "0",
 "info": "Actual bitrate."
}
```

Status shall refer only to channels described in the Registration.

Appendix A SMITHY PAIR API Model

Pairing HTTPS REST API Definition.

```
$version: "2"

namespace com.example.cdd.internal

use aws.protocols#restJson1
use com.example.cdd.common#HostSettings
```

```
use com.example.cdd.common#PairRequest
use com.example.cdd.common#PairResponse
use com.example.cdd.common#AuthRequest
use com.example.cdd.common#AuthResponse
use com.example.cdd.common#CertRotate
use com.example.cdd.common#DeprovisionMessage
use com.example.cdd.common#ThumbnailSubscription
use com.example.cdd.common#LogRequest
use com.example.cdd.common#HostConfig
use com.example.cdd.common#VersionResponse

@restJson1
service HostServiceApi {
   version: "1.0"
   operations: [
      Pair,
      Authenticate,
   ]
}

@http(method: "POST", uri: "/pair")
operation Pair {
   input: PairRequest
   output: PairResponse
}

@http(method: "POST", uri: "/authenticate")
operation Authenticate {
   input: AuthRequest
   output: AuthResponse
}
```

Appendix B SMITHY Registration Model

```
namespace com.example.cdd.registration
use com.example.cdd.common#ChannelType
use com.example.cdd.common#SupportedProtocol
use com.example.cdd.common#StringList

structure DeviceRegistration {
   @required
   channels: ChannelList,
   simpleSettings: SettingsList,
   thumbnails: ThumbnailList
}

list ChannelList {
```

```
    member: Channel
}

structure Channel {
    @required
    name: String,
    @required
    id: String,
    channelType: ChannelType,
    simpleSettings: SettingsList,
    profiles: ProfileList,
    connectionProtocols: ProtocolList
}

list SettingsList {
    member: Setting
}

structure Setting {
    @required
    id: String,
    @required
    name: String,
    @required
    info: String,
    enums: EnumValues,
    ranges: RangeValues
}

structure EnumValues {
    @required
    values: StringList,
    @required
    defaultValue: String
}

structure RangeValues {
    @required
    min: Double,
    @required
    max: Double,
    @required
    defaultValue: Double
}

list ProfileList {
    member: ProfileDefinition
}

structure ProfileDefinition {
    @required
    name: String,
    @required
```

```
    id: String,
    @required
    info: String
}

list ProtocolList {
    member: SupportedProtocol
}

list ThumbnailList {
    member: Thumbnail
}

structure Thumbnail {
    @required
    name: String,
    @required
    id: String,
    @required
    localPath: String
}
```

Appendix C TR-12 SMITHY Configuration Model

```
$version: "2"

namespace com.example.cdd.configuration
use com.example.cdd.common#ChannelState
use com.example.cdd.common#IdAndValueList

structure DeviceConfiguration {
    @required
    channels: ChannelConfigurationList
    simpleSettings: IdAndValueList,
}

list ChannelConfigurationList {
    member: ChannelConfiguration
}

structure ChannelConfiguration {
    @required
    id: String,
    @required
    state: ChannelState,
    settings: SettingsChoice,
    connection: Connection
```

```
}

union SettingsChoice {
   simpleSettings: IdAndValueList,
   profile: SettingProfile
}

union RistStreamIdentifier {
   synchronizationSource: Integer,
   streamId: String
}

structure SettingProfile {
   @required
   id: String
}

structure Connection {
   transportProtocol: TransportProtocol
}

@length(min: 32, max: 32)
@pattern("^[a-fA-F0-9]+$")
@documentation("A 32-character hexadecimal string.")
string Hex32

@length(min: 64, max: 64)
@pattern("^[a-fA-F0-9]+$")
@documentation("A 64-character hexadecimal string.")
string Hex64

structure EncryptionAes128 {
   @required
   passcode: Hex32
}

structure EncryptionAes256 {
   @required
   passcode: Hex64
}

union Encryption {
   aes128: EncryptionAes128
   aes256: EncryptionAes256
}

union TransportProtocol {
   srtListener: SrtListenerTransportProtocol,
   srtCaller: SrtCallerTransportProtocol,
   ristListener: RistListenerTransportProtocol,
   ristCaller: RistCallerTransportProtocol,
   zixiListener: ZixiListenerTransportProtocol,
   zixiCaller: ZixiCallerTransportProtocol
```

```
}

structure SrtListenerTransportProtocol {
   streamId: String,
   @required
   @range(min: 1024, max: 65535)
   port: Integer,
   @required
   @default(3000)
   minimumLatencyMilliseconds: Integer,
   encryption: Encryption,
   interface: String
}

structure SrtCallerTransportProtocol {
   streamId: String,
   @required
   ip: String,
   @required
   @range(min: 1, max: 65535)
   port: Integer,
   @required
   @default(3000)
   minimumLatencyMilliseconds: Integer,
   encryption: Encryption
}

structure RistListenerTransportProtocol {
   streamId: RistStreamIdentifier,
   @required
   port: Integer,
   @required
   @default(3000)
   minimumLatencyMilliseconds: Integer,
   encryption: Encryption
   interface: String
}

structure RistCallerTransportProtocol {
   streamId: RistStreamIdentifier,
   @required
   ip: String,
   @required
   port: Integer,
   @required
   @default(3000)
   minimumLatencyMilliseconds: Integer,
   encryption: Encryption
}

structure ZixiListenerTransportProtocol {
   @required
   streamId: String,
```

```
    @required
    port: Integer,
    @required
    @default(3000)
    minimumLatencyMilliseconds: Integer,
    encryption: Encryption
    interface: String
}

structure ZixiCallerTransportProtocol {
    @required
    streamId: String,
    @required
    ip: String,
    @required
    port: Integer,
    @required
    @default(3000)
    minimumLatencyMilliseconds: Integer,
    encryption: Encryption
}
```

Appendix D TR-12 SMITHY Status Model

```
$version: "2"

namespace com.example.cdd.status
use com.example.cdd.common#ChannelState

structure DeviceStatus {
    @required
    status: StatusValueList
    channels: ChannelStatusList
}

list ChannelStatusList {
    member: ChannelStatus
}

structure ChannelStatus {
    @required
    id: String,
    @required
    state: ChannelState,
    @required
    status: StatusValueList
}
```

```
list StatusValueList {
    member: StatusValue
}

structure StatusValue {
    @required
    name: String,
    @required
    info: String,
    @required
    value: String
}
```

Appendix E TR-12 SMITHY Common Model

```
$version: "2"

namespace com.example.cdd.common

structure ProtocolVersion {
    @default("1.0.0")
    version: String
}

enum ChannelState {
    ACTIVE
    IDLE
}

enum ChannelType {
    SOURCE
    DESTINATION
}

enum AuthStatus {
    STANDBY
    CLAIMED
}

enum SupportedProtocol {
    SRT_LISTENER
    SRT_CALLER
    ZIXI_LISTENER
    ZIXI_CALLER
    RIST_CALLER
    RIST_LISTENER
}
```

```
structure IdAndValue {
   @required
   key: String
   @required
   value: String
}

//TODO: make this a map
list IdAndValueList {
   member: IdAndValue
}

list StringList {
   member: String
}

structure HostSettings {
   @required
   // TODO: should this be an enum?
   // TODO: Do we want to expose iot or keep that as an implementation detail?
   iotProtocolName: String
   @required
   pairingTimeoutSeconds: Integer
   @required
   minIntervalPubSeconds: Integer
   @required
   mqttKeepaliveSeconds: Integer
   @required
   subUpdateTopic: String
   @required
   subUpdateThumbnailSubscriptionTopic: String
   @required
   pubReportSchemaTopic: String
   @required
   pubReportRegistrationTopic: String
   @required
   pubReportStatusTopic: String
   @required
   pubReportActualConfigurationTopic: String
   @required
   subUpdateCertsTopic: String
   @required
   pubDeprovisionTopic: String
   @required
   subDeprovisionTopic: String
   @required
   subUpdateLogSubscriptionTopic: String
}

# hostId:  the ID string provided by the target host's HostSettings
# deviceTyp: the SMITHY TR-12 version supplied by the TR-12 SMITHY public SDK.
# csr:  is a PEM formatted X509 Certificate Signing Request
```

```
# version: the SMITHY TR-12 version supplied by the TR-12 SMITHY public SDK.
structure PairRequest {
    @required
    deviceType: String
    @required
    hostId: String
    @required
    csr: String
    @required
    version: String
}

enum PairFailureReason {
    HOST_ID_MISMATCH
    VERSION_NOT_SUPPORTED
    DEVICE_TYPE_NOT_SUPPORTED
}

union PairResult {
    success: PairSuccessData
    failure: PairFailureData
}

structure PairSuccessData {
    @required
    deviceId: String
    @required
    pairingCode: String
    @required
    accessCode: String
    @required
    pairingTimeoutSeconds: Integer
}

structure PairFailureData {
    @required
    reason: PairFailureReason
}

structure PairResponse {
    @required
    result: PairResult
}

# Use the param values obtained vai the GetPairingCode response
structure AuthRequest {
    @required
    deviceId: String
    @required
    pairingCode: String
    @required
    accessCode: String
}
```

```
# deviceCert: X509 Device Certificate generated by the host using the client provided CSR
# caCert: host Certificate Authority CA Certificate in PEM format <string>
structure AuthResponse {
   @required
   status: AuthStatus
   caCert: String
   deviceCert: String
   mqttUri: String
   region: String
   hostSettings: HostSettings
}

structure ConnectionSettings {
   @required
   deviceId: String
   @required
   uri: String
   @required
   region: String
}

structure CertRotate {
   @required
   mqttUri: String
   @required
   deviceCert: String
   @required
   region: String
}

enum DeprovisionReason {
   DEPROVISIONED
   EXPIRED
   UNKNOWN
}

structure DeprovisionMessage {
   reason: DeprovisionReason
   //TODO: use timestamp instead?
   time: Integer
}

structure ThumbnailRequest {
   period: Integer
   expires: Integer
   maxSizeKilobyte: Integer
   localPath: String
   remotePath: String
}

structure ThumbnailSubscription {
   @required
```

```
    requests: ThumbnailRequestMap
}

map ThumbnailRequestMap {
    key: String
    value: ThumbnailRequest
}

structure LogRequest {
    // TODO: use timestamp?
    expires: Integer
    remotePath: String
}

structure ReportMessage {
    @required
    message: Document
}

structure HostConfig {
    @required
    serviceId: String
    @required
    serviceName: String
    @required
    deviceTypes: StringList
    @required
    thumbnailMaxSizeKB: Integer
    @required
    logFileMaxSizeKB: Integer
    @required
    pairingUrl: String
    @required
    authUrl: String
}

structure VersionResponse {
    @required
    version: ProtocolVersion
}

1
```

Appendix F CSR Certificate Signing Request Example

The TR-12 client must include an X509 Certificate Signing Request (CSR) in the GetPairingCode request model. The following is an example python code snippet that uses the cryptography library to generate a compliant CSR. CSR generation is well described in the cryptographic standards and widely available in various libraries.

```
# Example CSR generated via python cryptography: import x509
 ...
 csr = (
     x509.CertificateSigningRequestBuilder()
     .subject_name(
         x509.Name(
             [
                 x509.NameAttribute(NameOID.ORGANIZATION_NAME, "VSF-CDD"),
                 x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
             ]
         )
     )
     .sign(private_key_b, hashes.SHA256())
 )

# Response is a string, not binary PEM
 return csr.public_bytes(serialization.Encoding.PEM).decode('utf-8')
```