

IoT Hub

Developer Manual

Product Documentation



Copyright Notice

©2013-2024 Tencent Cloud. All rights reserved.

Copyright in this document is exclusively owned by Tencent Cloud. You must not reproduce, modify, copy or distribute in any way, in whole or in part, the contents of this document without Tencent Cloud's the prior written consent.

Trademark Notice



All trademarks associated with Tencent Cloud and its services are owned by Tencent Cloud Computing (Beijing) Company Limited and its affiliated companies. Trademarks of third parties referred to in this document are owned by their respective proprietors.

Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.

Contents

Developer Manual

- Feature Components

- Signature Algorithm

- Device Authentication

 - Overview

 - Device-Level Key Authentication

 - Product-Level Key Authentication

 - Dynamic Registration API Description

- Device Connection Protocol

 - MQTT-Based Device Connection

 - MQTT-Based Device Connection over TCP

 - MQTT-Based Device Connection over WebSocket

 - MQTT Persistent Session

 - CoAP-Based Device Connection

 - HTTP-Based Device Connection

 - Device Connection Regions

- Gateway Subdevice

 - Feature Overview

 - Topological Relationship Management

 - Proxied Subdevice Connection and Disconnection

 - Proxied Subdevice Publishing and Subscribing

 - Subdevice Firmware Update

- Message Communication

 - Broadcast Communication

 - RRPC Communication

- Device Shadow

 - Device Shadow Details

 - Device Shadow Data Flow

- Device Firmware Upgrade

- Remote Device Configuration

- Resource Management

- Device Log Reporting

- NTP Service

Developer Manual

Feature Components

Last updated : 2024-12-27 15:44:26

1. SDK

For more information, please see the [SDK documentation](#). Currently, IoT Hub supports device SDKs for Linux and Android and supports porting to different hardware platforms.

2. Device Connection

Devices can be connected to the IoT Hub platform through the SDK:

The application layer is based on MQTT and CoAP protocols.

The transport layer is based on TCP and UDP protocols, and on this basis, secure network transfer protocols TLS and DTLS are introduced for two-way authentication and encrypted data transfer between clients and servers.

The SDK supports RTOS portability for cross-platform porting and detachment of framework from hardware abstraction layer, enabling quick and easy connection to IoT Hub from different platforms.

The device SDK supports TLS (for MQTT) and DTLS (for CoAP) for asymmetric and symmetric encryption to protect device communication security:

Asymmetric encryption

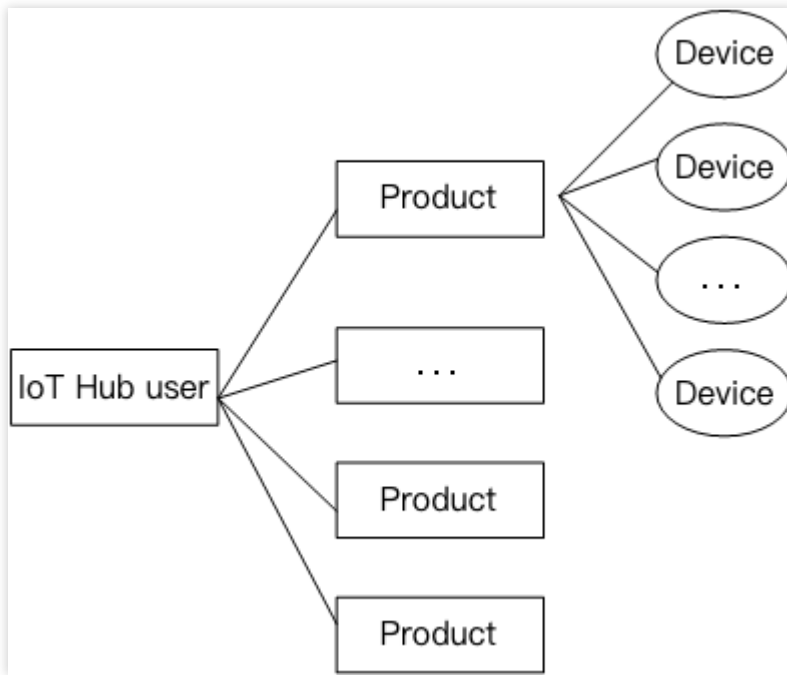
This is based on the certificate and asymmetric encryption algorithm for a high security level and suitable for devices with high hardware specifications and low sensitivity to power consumption. It relies on device certificates, private keys, and root certificates, and relevant information will be returned when a device is created in IoT Hub.

Symmetric encryption

This is based on the key and symmetric encryption algorithm for a general security level and suitable for resource-constrained devices sensitive to power consumption. It relies on the device `psKey`, and relevant information will be returned when a device is created in IoT Hub.

In addition to connection through the device SDK, IoT Hub also supports HTTP connection, which has low requirements for connection and is suitable for low-power data reporting scenarios over non-persistent connections.

3. Device Management



Up to 2,000 products can be created under one Tencent Cloud account, and up to 1 million devices can be created for one product. A device can only belong to one product. Product names and device names must be unique under the same Tencent Cloud account.

Devices can be enabled/disabled. After a device is disabled, it cannot be connected to the IoT Hub platform, and device-related operations cannot be performed, but the information associated with it will be retained, so device information can still be queried.

4. Permission Management

In IoT Hub, the topics that devices can publish and subscribe to are strictly managed. All devices under the same product have the same topic class permissions, including the following by default:

Topic	Description
<code>\${productId}/\${deviceName}/event</code>	Publishing permission for the device to report data
<code>\${productId}/\${deviceName}/control</code>	Subscribing permission for the device to get the data sent by the backend

For a specific device, the `productId` and `deviceName` marked with the `$` symbol above should be mapped to the specific product ID and device name. For example, if a product named `pro` (with the product ID `pro_id`) has 2 devices (named `dev_1` and `dev_2` respectively), then the topics that `dev_1` can publish include `pro_id/dev_1/event` but not `pro_id/dev_2/event`, and the topics that it can subscribe to include `pro_id/dev_1/control` but not `pro_id/dev_2/control`.

You can edit and modify the topic permissions in the console and add or remove the topic class permissions of products.

To facilitate batch subscription to topics by the device SDK, wildcards can be used to indicate multiple matching topics when a device subscribes to or unsubscribes from topics:

Wildcard	Description
#	This wildcard can only appear at the end of the topic, representing the topics at the current level and all sub-levels; for example, if the wildcard topic is <code>pro_id/dev_1/#</code> , it can represent not only <code>pro_id/dev_1/event</code> but also <code>pro_id/dev_1/event/subeventA</code>
+	This wildcard can only appear after <code>deviceName</code> , representing all the topics at the current level; for example, if the wildcard topic is <code>pro_id/dev_1/event/+</code> , it can represent <code>pro_id/dev_1/event/subeventA</code> and <code>pro_id/dev_1/event/subeventB</code> but not <code>pro_id/dev_1/event/subeventA/close</code> . This wildcard can appear multiple times, such as <code>pro_id/dev_1/event/+subeventA/+</code>

A wildcard must be used as a complete level; for example, both `${productId}/${deviceName}/e#` and `${productId}/${deviceName}/e+` are invalid.

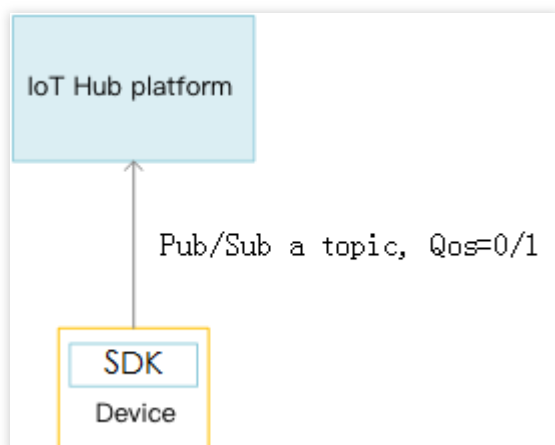
The system topics defined by IoT Hub (`$shadow`, `$ota`, and `$sys`) don't support wildcards.

Effect of the wildcard when subscribing to topics: all the topics with the subscribing permission under the product matching the wildcard topic are subscribed to, and a success message will be returned even if the matched topic list is empty.

Effect of the wildcard when unsubscribing from topics: all the subscribed topics matching the wildcard topic are unsubscribed from, and a success message will be returned even if the matched topic list is empty.

`${productId}/${deviceName}/#` can be used to unsubscribe from all topics.

5. Message Management



For MQTT data transfer, IoT Hub supports QoS 0 or 1 but not QoS 2. Device messages can be stored offline.

If QoS is 0, the message will be sent to the device at most once

For scenarios where the requirement for data transfer reliability is not high, please select this QoS level for publishing and subscribing.

If QoS is 1, the device should receive the message at least once

For scenarios where the requirement for data transfer reliability is high, please select this QoS level for publishing and subscribing.

Other parameters are as follows:

Parameter	Description
Topic name length	Up to 64 bytes
MQTT protocol packet size	Up to 16 KB
QoS 1 message storage period (if the recipient is offline or online sending fails)	24 hours
Number of QoS 1 messages not confirmed by the device	Up to 150

6. Device Shadow

Device shadow is essentially a copy of device data in JSON format cached on the server and is mainly used to save:

Current device configurations

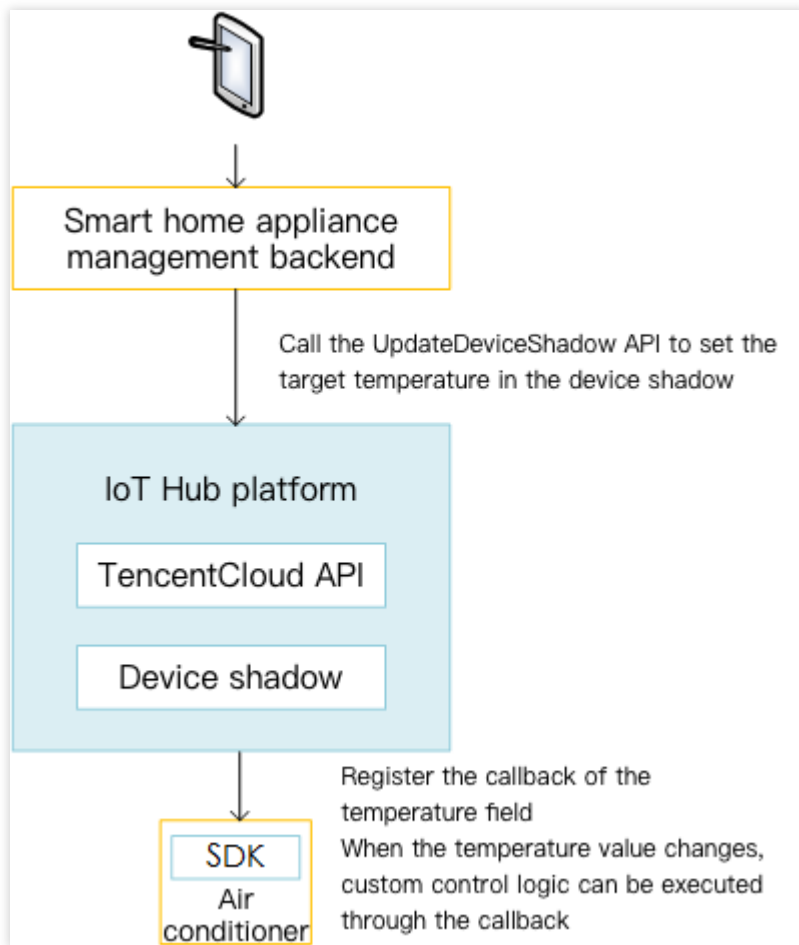
Current device status

As an intermediary, device shadow can effectively implement two-way data sync between device and user application:

For device configuration, the user application does not need to directly modify the device; instead, it can modify the device shadow on the server, which will sync modifications to the device. In this way, if the device is offline at the time of modification, it will receive the latest configuration from the shadow once coming back online.

For device status, the device reports the status to the device shadow, and when users initiate queries, they can simply query the shadow. This can effectively reduce the network interactions between the device and the server, especially for low-power devices.

The figure below is a sample use case of the device shadow in "Getting Started":

**Note:**

Device shadow and device message have different applicable scenarios. In terms of implementation mechanism, the server-side device shadow always saves the last copy of data, and multiple messages that successively arrive do not overwrite one another.

For scenarios where the device reports data, the device shadow is more suitable for reporting the device's own information (such as energy consumption), while the device message is more suitable for reporting the data collected by the device (such as the measured temperature).

For scenarios where the device receives data, the device shadow is more suitable for notifying the device of updated configuration (such as changing the target temperature), while the device message is more suitable for real-time control of the device (such as turning the device to the left by 45 degrees).

For more information, please see [Device Shadow Details](#).

7. Rule Engine

Based on the rule engine, you can configure rules to do the following:

Syntax rules

IoT Hub supports SQL-like syntax and basic semantic operations. The contents of device messages can be parsed, filtered, extracted, and reintegrated through simple syntax, with the results forwarded to Tencent Cloud's backend services such as storage, function, and TBDS for seamless data connection.

Device-to-Device connection

In order to isolate device data, devices can only publish and subscribe to messages in their own topics (for more information, please see [Permission Management](#)). To implement message connection, the repub feature of the rule engine is needed.

Device-to-Server connection

The rule engine provides a simple forwarding feature that can copy messages to your server through HTTP requests. This can implement fast connection between device messages and your services.

Device-to-Cloud connection

Tencent Cloud offers corresponding services (such as TencentDB and TBDS) for scenarios where users require further processing of device data (such as persistent storage and big data analysis).

For more information, please see [Rule Engine Details](#).

8. Message Queue

As devices are connected only to IoT Hub, IoT Hub can write specified device messages to Tencent Cloud CMQ or CKafka queues. From there, third-party services can get the device messages through the SDK APIs of CMQ or CKafka, enabling async message communication between devices and third-party services. Based on this, data storage, computational analysis, and device control logic can be implemented on the backend.

9. Console

The [IoT Hub console](#) provides visual management UIs where you can manage products, devices, and permissions, configure the rule engine, and perform other operations. You can try it out easily.

10. TencentCloud API

For the device management flow in IoT scenarios, IoT Hub provides various APIs in Python, PHP, Java, Go, Node.js, and .NET for fast and batch operations on the backend. Currently, it offers APIs related to product, device, task, message, rule engine, and device shadow. For more information, please see [TencentCloud API Overview](#).

11. Firmware Update

When firmware has security risks or functional problems, IoT Hub servers can perform OTA updates to eliminate dangers and reduce security risks.

12. Collaboration Management

IoT Hub supports secure access, use, and management of cloud account resources through [CAM](#). Isolation and collaboration of IoT Hub resources are implemented through identity and policy management of sub-accounts and collaborators.

Signature Algorithm

Last updated : 2024-12-27 15:44:26

Overview

When a device initiates an HTTP/HTTPS request to the platform, the request message should contain the signature information (X-TC-Signature) for requester identity verification.

Signing Steps

Sample device request message:

```
curl -X POST https://ap-guangzhou.gateway.tencentdevices.com/device/register \\  
-H "Content-Type: application/json; charset=utf-8" \\  
-H "X-TC-Algorithm: hmacsha256" \\  
-H "X-TC-Timestamp: 155***065" \\  
-H "X-TC-Nonce: 5456" \\  
-H "X-TC-Signature:  
2230eefd229f582d8b1b891af***b91597240707d778ab3738f756258d7652c" \\  
-d '{"ProductId":"ASJ***GX","DeviceName":"xyz"}'
```

1. Concatenate the string to sign

```
StringToSign =  
    HTTPRequestMethod + \n +  
    CanonicalHost + \n +  
    CanonicalURI + \n +  
    CanonicalQueryString + \n +  
    Algorithm + \n +  
    RequestTimestamp + \n +  
    Nonce + \n +  
    HashedCanonicalRequest
```

Parameter	Description
HTTPRequestMethod	HTTP request method. POST is supported
CanonicalHost	Host address of the HTTP request
CanonicalURI	URI of the HTTP request; for example, the URI of <code>https://ap-</code>

	<code>guangzhou.gateway.tencentdevices.com/device/register</code> is <code>/device/register</code>
CanonicalQueryString	Query string in the URL of the initiated HTTP request, which is always an empty string <code>" "</code> for POST requests
Algorithm	Signature algorithm. Currently, HMACSHA256 and HMACSHA1 are supported
RequestTimestamp	Request timestamp
Nonce	Random number
HashedCanonicalRequest	Hash value of the request body, which is calculated by SHA256 hashing the HTTP request body, performing hexadecimal encoding, and then converting the encoded string to lowercase letters

According to the above rules, the canonical signature string obtained in the sample is as follows:

```
POST
ap-guangzhou.gateway.tencentdevices.com
/device/register

hmacsha256
155****065
5456
35e9c5b0e3ae67532d3c9f17ead6c902226****b1ff7f6e89887f1398934f064
```

2. Calculate the signature

The pseudo code for using key signatures, including product-level keys and device-level keys, is as follows:

```
Signature = Base64_Encode(HMAC_SHA256(SignSecret, StringToSign))
```

Parameter	Description
SignSecret	Signature key. `ProductSecret` is used for dynamic registration, and `psk` is used for devices to publish messages or report logs
StringToSign	String to sign

The pseudo code for using certificate signatures is as follows:

```
Signature = Base64_Encode(RSA_SHA256(PrivateKey, StringToSign))
```

Parameter	Description
PrivateKey	Certificate private key. Device X.509 private key certificate is used for devices to publish messages or report logs
StringToSign	String to sign

3. Assemble the request message

Based on the signature string obtained above, the final complete request is as follows:

```
POST https://ap-guangzhou.gateway.tencentdevices.com/devregister
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 155****065
X-TC-Nonce: 5456
X-TC-Signature:
2230eefd229f582d8b1b891af71***1597240707d778ab3738f756258d7652c

{"ProductId":"ASJ****GX","DeviceName":"xyz"}
```

Sample Code

Below is the sample code in Python 3:

```
import hashlib
import random
import time
import hmac
import base64

if __name__ == '__main__':
    sign_format = '%s\\n%s\\n%s\\n%s\\n%s\\n%d\\n%d\\n%s'
    url_format = '%s://ap-guangzhou.gateway.tencentdevices.com/device/register'
    request_format = "{\\"ProductId\\" : \\"%s\\" , \\"DeviceName\\" : \\"%s\\"}"
    device_name = 'dev***'
    product_id = 'JCZ****KXS'
    product_secret = 'X42fPqw*****94cY5sQ1Y'

    request_text = request_format % (product_id, device_name)
    request_hash = hashlib.sha256(request_text.encode("utf-8")).hexdigest()
```

```
nonce = random.randrange(2147483647)
timestamp = int(time.time())
sign_content = sign_format % (
    "POST", "ap-guangzhou.gateway.tencentdevices.com",
    "/device/register", "", "hmacsha256", timestamp,
    nonce, request_hash)
print("\nsign_content: \n" + sign_content)

sign_base64 = base64.b64encode(hmac.new(product_secret.encode("utf-8"),
    sign_content.encode("utf-8"), hashlib.sha256).digest())

print("sign_base64: " + str(sign_base64))
```

Device Authentication

Overview

Last updated : 2024-12-27 15:44:26

IoT Hub assigns a unique product ID to each created product. You can customize the `Devicename` to identify devices and use the product ID + device ID + device certificate/key to authenticate devices. You need to select the device authentication method when creating a product. During connection, a device needs to report the information of the product, device, and corresponding key according to the specified method and can be connected to IoT Hub only after successful authentication. As different users have different requirements for device resources and security levels, IoT Hub provides multiple authentication schemes to meet the needs in different use cases.

IoT Hub provides the following three authentication schemes:

Certificate authentication (device-level): it assigns a certificate + private key to each device and uses asymmetric encryption to authenticate the access. You need to burn different configuration information for each device.

Key authentication (device-level): it assigns a device key to each device and uses symmetric encryption to authenticate the access. You need to burn different configuration information for each device.

Dynamic registration authentication (product-level): it assigns a unified key to all devices under the same product, and a device gets a device certificate/key through a registration request for authentication. You can burn the same configuration information for the same batch of devices.

The three schemes have their own pros and cons in terms of ease of use, security, and device resource requirement. You can comprehensively evaluate them and choose the most appropriate one according to your own business scenarios. They are as compared below:

Feature	Certificate Authentication	Key Authentication	Dynamic Registration Authentication
Burned device information	<code>ProductId</code> , <code>Devicename</code> , device certificate, and device private key	<code>ProductId</code> , <code>Devicename</code> , and device key	<code>ProductId</code> , <code>Devicename</code> , and <code>ProductSecret</code>
Whether device creation is required	Yes	Yes	Devices can be automatically created according to the <code>Devicename</code> carried in the registration request.
Security	High	Average	Average
Use limit	Up to 1 million devices can be created under one product.	Up to 1 million devices can be	Up to 1 million devices can be created under one product. You can customize the maximum number of devices

		created under one product.	automatically created through registration requests.
Device resource requirement	High, with TLS support required	Low	Low, with only AES support required.

Device-Level Key Authentication

Last updated : 2024-12-27 15:44:26

Overview

The IoT Hub platform supports device-level key authentication. In this mode, you need to burn different configuration firmware for each device. The platform will perform certificate authentication or key authentication according to your selection. After successful authentication, devices can establish a connection with the platform for data communication.

Flowchart

Device-level key authentication requires you to burn different firmware for each device. It incurs certain implementation costs in production applications, but it has higher security and is thus recommended.


Directions

1. Log in to the [IoT Hub console](#) and create a product and a device as instructed in [Device Connection Preparations](#).
2. Get the product information on the product details page and get the device name and device certificate/key on the device details page.

Product information

Product Settings Devices Permissions Message Queues

Basic Information

Product Name	prod2
Product Type	General
Product ID	
Authentication Method	Certificate
Data Format	JSON
Product Description	Not entered
CA Certificate	Tencent Cloud certificate Download
Enabled/Disabled ⓘ	<input checked="" type="checkbox"/> Enabled

Device information

Basic Information

Device Name	door1
Remarks	door-test
Online Status	Inactive Reset
Tag	No tag information. Add
Device Certificate	Download
Device Private Key	Download
Enabled/Disabled ⓘ	<input checked="" type="checkbox"/> Enabled
Firmware Version	Not reported

Log Configuration

Device Log	Disabled
Log Level	None

3. Burn the device firmware in the following steps:

3.1 Download the [device SDK](#).

3.2 Implement the HAL layer functions in the SDK for reading and writing product and device information, including `ProductID` , `Devicename` , and device certificate or key. For more information, please see [SDK for C Connection Description](#).

3.3 Develop the device firmware based on the SDK according to your actual business needs so as to implement various features such as unique device ID reading, dynamic device registration, connection authentication, communication, and OTA.

3.4 In the production process, batch burn the developed and tested device firmware into the device.

4. The device uses the burned device-level certificate/key to establish a connection with the platform. After successful authentication, it will be activated and connected. At this time, it can exchange data with the cloud to implement business requirements.

Product-Level Key Authentication

Last updated : 2024-12-27 15:44:26

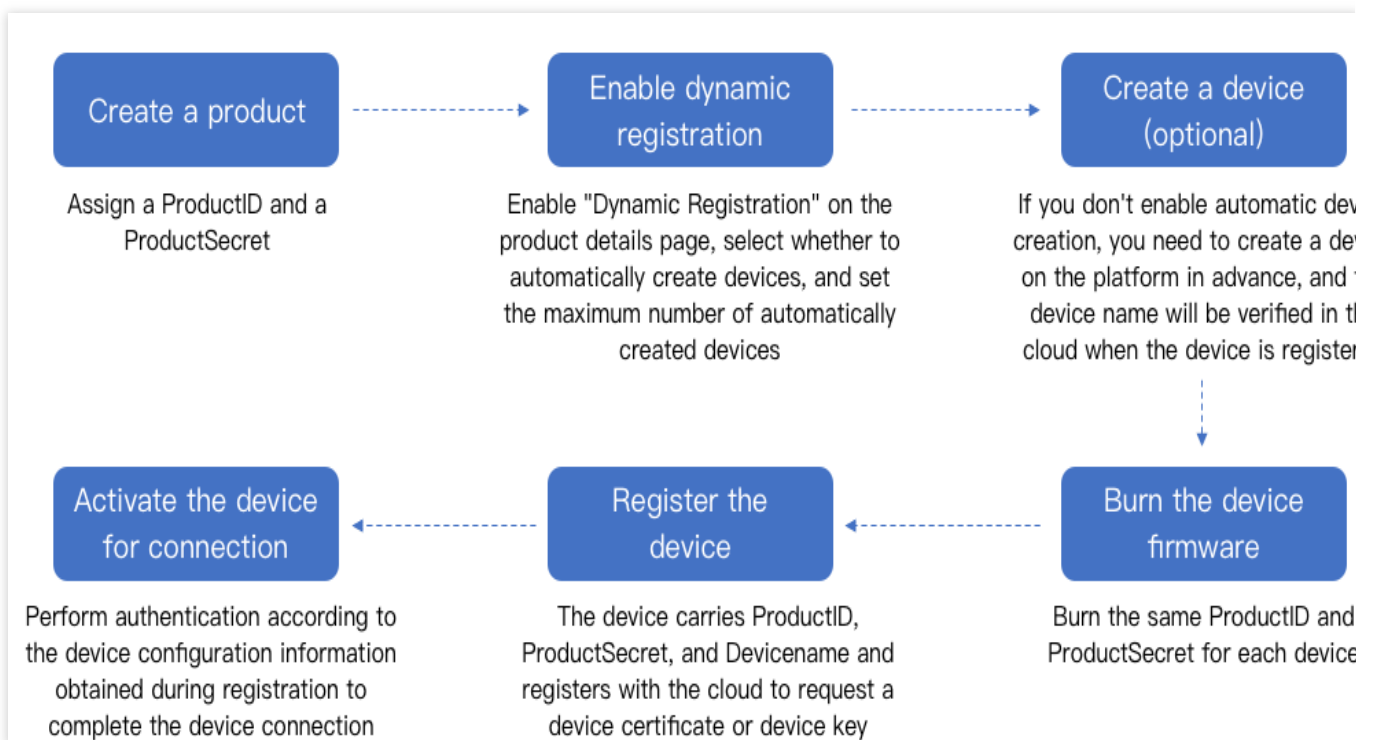
Overview

The IoT Hub platform supports product-level key authentication. In this mode, you only need to enable dynamic device registration and then burn the same configuration firmware (ProductID + ProductSecret) for all devices under the same product. In this way, the devices can get device certificates or keys through registration requests and then communicate with the platform.

Flowchart

Note:

If you want to use the dynamic registration feature, you need to manually enable dynamic registration for the product on the product details page in the console.

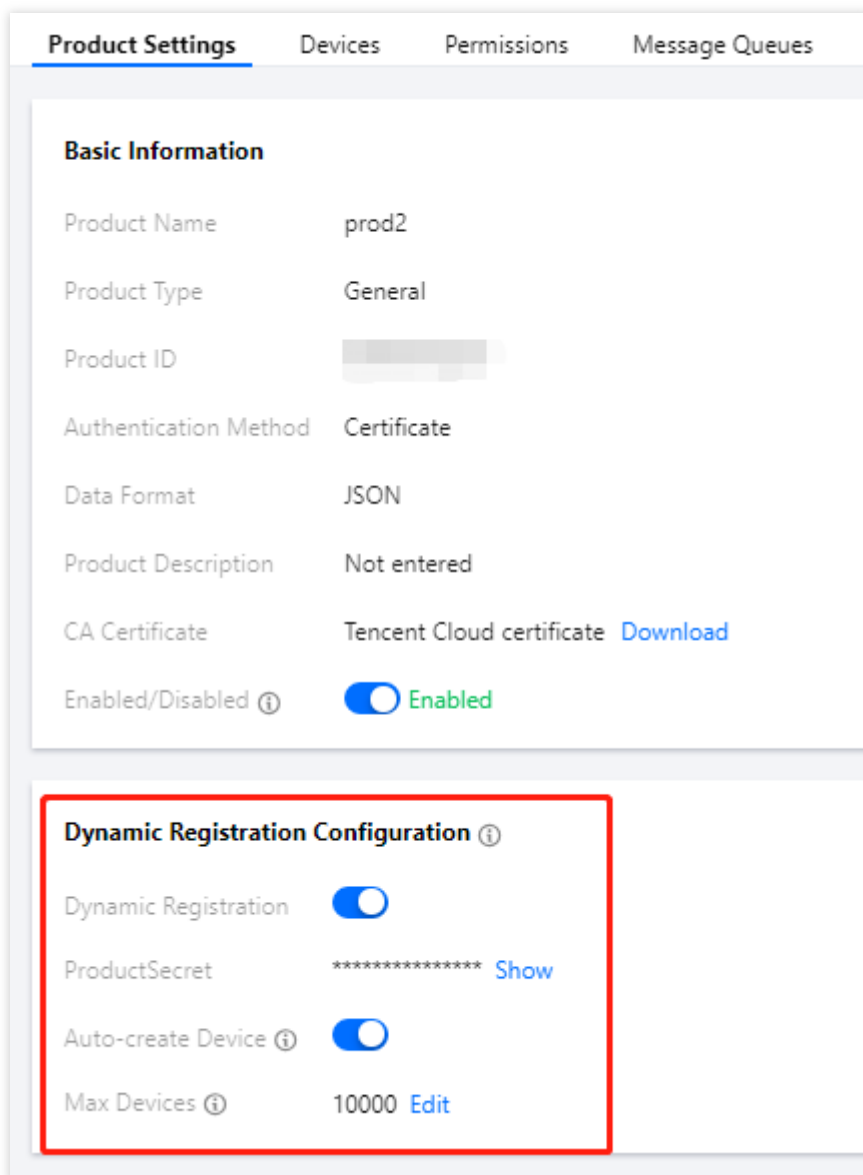


Directions

1. Log in to the [IoT Hub console](#) and create a product as instructed in [Device Connection Preparations](#).
2. Enable **Dynamic Registration** on the product details page, select whether to automatically create devices, and set the maximum number of automatically created devices.

Note:

To prevent too many devices from being created in unpredictable situations (such as device firmware bugs and product key theft), if you select automatic device creation, we recommend you set an appropriate device quantity upper limit.

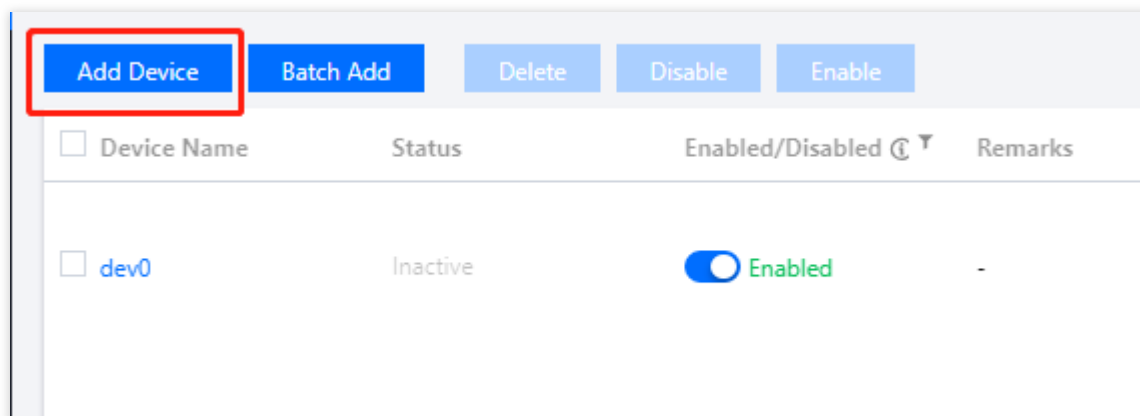


3. Create a device under the product (optional).

You can add devices to the device list in the console or create devices through TencentCloud API.

If you don't enable automatic device creation, IoT Hub will verify whether each requested device name has been created in the cloud during device registration. We recommend you use a unique identifier that can be read by the

device as the `Devicename` , such as IMEI, SN, or MAC address, which facilitates the smooth completion of the entire process.



Burn the device firmware in the following steps:

1. Download the [device SDK](#).
2. Implement the HAL layer functions in the SDK for reading and writing product and device information, including `ProductID` , `ProductSecret` , and `Devicename` , and enable the dynamic registration feature in the SDK. For more information, please see [SDK for C Connection Description](#).
3. Develop the device firmware based on the SDK according to your actual business needs so as to implement various features such as unique device ID reading, dynamic device registration, connection authentication, communication, and OTA.
4. In the production process, batch burn the developed and tested device firmware into the device.
5. After successful device registration, power-on, and connection, initiate a registration request to get the device certificate or key.
6. The device uses the obtained device-level certificate/key to establish a connection with the platform. After successful authentication, it will be activated and connected. At this time, it can exchange data with the cloud to implement business requirements.

Dynamic Registration API Description

Last updated : 2024-12-27 15:44:26

Parameter Description

When a device is dynamically registered, it needs to carry `ProductId` and `DeviceName` to initiate an `http/https` request to the platform. The request API and parameters are as detailed below:

Requested URL:

```
https://ap-guangzhou.gateway.tencentdevices.com/device/register
```

```
http://ap-guangzhou.gateway.tencentdevices.com/device/register
```

Request method: POST

Request parameters

Parameter	Required	Type	Description
ProductId	Yes	string	Product ID
DeviceName	Yes	string	Device name

Note:

The API only supports the `application/json` format.

Signature generation

Use the HMACSHA256 algorithm to sign the request message. For more information, please see [Signature Algorithm](#).

Platform response parameters

Parameter	Type	Description
RequestId	String	Request ID
Len	Int64	Length of the returned payload
Payload	String	Returned device registration information, which is encrypted and needs to be decrypted and processed by the device itself

Note:

The encryption process is to convert the raw payload in JSON format into a string, perform AES encryption on it, and then perform Base64 encryption on it. The AES encryption algorithm is CBC mode, where the key length is 128 bits,

the first 16 bits of `productSecret` are taken, and the offset is the character "0" with a length of 16 bits.

Raw payload content description:

Key	Value	Description
encryptionType	1	Encryption type. 1: certificate authentication 2: key authentication
psk	1239466501	Device key. This parameter is available when the product authentication type is key authentication.
clientCert	-	String format of device certificate file. This parameter is available when the product authentication type is certificate authentication.
clientKey	-	String format of device private key file. This parameter is available when the product authentication type is certificate authentication.

Sample Code

Request packet

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/register
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 1551****65
X-TC-Nonce: 5456
X-TC-Signature:
2230eefd229f582d8b1b891af7107b91597****07d778ab3738f756258d7652c
{"ProductId":"ASJ****GX","DeviceName":"xyz"}
```

Response packet

```
{
  "Response": {
    "Len": 53,
    "Payload":
"031T01DWAoqFePDt71VuZXuLzkUzbIhGOnvMzpaFtNgOjagyFNHVSostN19ztvhOuRx0dMM/DMoWAX
QCfL7jyA==",
    "RequestId": "f4da4f1f-d72e-40f1-****-349fc0072ba0"
  }
}
```

Payload data parsing sample

Note:

The following data is for test only. When you use it formally, please ensure that your information is not leaked.

1. The raw payload content is:

```
s6FB3a1BA/YYbcmSE12XpeDvmQNDcf1QgVD141RRbmmAnFwQfp1ECAu50016mCOvY1JJ6V59yM4OqQS  
iWphfTg==
```

2. After Base64-decoding:

```
b3a141ddad4103f6186dc992135d97a5e0d599034371fd508150f5e354516e69809c5c107e9d440  
80bb93b4d7a9823af625249e95e7dc8ce0ea904a25a985f4e
```

3. AES decryption:

Product key: `hzvf5LF9S0isvBhDSauWMaIk`

Data after decryption: `{"encryptionType":2,"psk":"1DZ6Uqt+I9E0wW7rvDUs7Q=="}`

Device Connection Protocol

MQTT-Based Device Connection

MQTT-Based Device Connection over TCP

Last updated : 2024-12-27 15:44:27

MQTT Protocol Description

Currently, IoT Hub supports MQTT standard protocol access (compatible with v3.1.1). For more information, please see [MQTT Version 3.1.1](#).

Differences from standard MQTT

1. PUB, SUB, PING, PONG, CONNECT, DISCONNECT, and UNSUB messages of MQTT are supported.
2. `cleanSession` is supported.
3. `will` and `retain msg` are not supported.
4. QoS 2 is not supported.

Security level of MQTT channel

TLSv1, TLSv1.1, and TLSv1.2 protocols are supported to establish secure connections, delivering a high security level.

Topic specification

By default, after a product is created, all devices under it will have the permissions of the following topic classes:

1. `${productId}/${deviceName}/control` for subscribing
2. `${productId}/${deviceName}/event` for publishing
3. `${productId}/${deviceName}/data` for subscribing and publishing
4. `$shadow/operation/${productId}/${deviceName}` for publishing. It is distinguished by the internal type of the packet (`update` or `get` , corresponding to updating or pulling the device shadow document).
5. `$shadow/operation/result/${productId}/${deviceName}` for subscribing. It is distinguished by the internal type of the packet (`update` , `get` , or `delta`). `update` and `get` correspond to updating and pulling the device shadow document respectively. After you modify the device shadow document through the RESTful API, the server will publish messages through this topic, whose type will be `delta` at this time.
6. `$ota/report/${productId}/${deviceName}` for publishing, through which the device reports the version number and the download/upgrade progress to the cloud.

7. `ota/update/${productID}/${deviceName}` for subscribing, through which the device receives the upgrade message from the cloud.

MQTT Connection

The MQTT protocol supports connection to the IoT Hub platform through two methods: device certificate and key signature. You can choose a method according to your own business scenario. The connection parameters are as follows:

Connection Authentication Method	Connection Domain Name and Port	Connect Message Parameter
Certificate	<p>MQTT server connection address. For devices in the Guangzhou region, enter <code>\${productID}.iotcloud.tencentdevices.com</code> , where <code>\${productID}</code> is a variable parameter, and you need to enter the product ID automatically generated when you create the product, such as <code>1A17RZR3XX.iotcloud.tencentdevices.com</code> .</p> <p>Port: 8883</p>	<p>KeepAlive: the time to keep the c IoT Hub does not receive the cli KeepAlive value, it will disco ClientId: <code>\${productID}\${dev</code> the product ID and device name. UserName: <code>\${productID}\${deviceNam</code> For more information, please see connecting key-authenticated dev PassWord: password, which can</p>
Key	<p>The MQTT server connection address is the same as that for certificate authentication. Port: 1883</p>	<p>KeepAlive: the time to keep the c ClientId: <code>\${productID}\${dev</code> UserName: <code>\${productID}\${deviceNam</code> For more information, please see connecting key-authenticated dev PassWord: password. For more ii the "Guide to connecting key-autf</p>

Note:

The `PassWord` field will not be verified when a certificate-authenticated device is connected, so you can enter any value for it during certificate authentication.

Guide to connecting certificate-authenticated device

IoT Hub uses TLS encryption to ensure the security of devices when transferring data. When a certificate-authenticated device is connected, after getting the certificate, key, and CA certificate files of the device, set the values of `KeepAlive` , `ClientId` , `UserName` , `PassWord` , etc. (this step is not required for devices

connected through the Tencent Cloud device SDK, as the SDK can automatically generate the parameters based on the device information). Then, the device uploads the authentication files to the URL (connection domain name and port) corresponding to certificate authentication, and sends an `MqttConnect` message after successful authentication to complete the TCP-based MQTT connection.

Guide to connecting key-authenticated device

IoT Hub supports HMACSHA256 and HMACSHA1 algorithms to generate digest signatures based on device keys. The process of connecting to IoT Hub through signature is as follows:

1. Log in to the [IoT Hub console](#). You can create products, add devices, and get device keys in the console.
2. Generate the `username` field according to the requirements of IoT Hub in the following format:

```
The format of the `username` field is as follows:  
{productId}{deviceName};{sdkappid};{connid};{expiry}  
Note: `{}` indicates a variable and is not a concatenating symbol.
```

The descriptions of each field are as follows:

`productId`: product ID

`deviceName`: device name

`sdkappid`: fixed at `12010126`

`connid`: a random string

`expiry`: signature validity period, which is a UTF-8 string of the number of seconds since 00:00:00 UTC on January 1, 1970.

3. Base64-decode the device key to get the raw key `raw_key`.
4. Use the `raw_key` generated in step 3 to generate a digest string for the `username` with the HMACSHA1 or HMACSHA256 algorithm, which is referred to as a token.
5. Generate the `password` field according to the requirements of IoT Hub in the following format:

```
The format of the `password` field is as follows:  
{token};hmac signature algorithm  
Enter the digest algorithm used in step 3 in the `hmac signature algorithm` field.
```

As a comparison, the code samples for generating signatures in Python, Java, Node.js, JavaScript, and C are as follows:

The code in Python is as follows:

```
#!/usr/bin/python  
# -*- coding: UTF-8 -*-  
import base64  
import hashlib  
import hmac  
import random  
import string
```

```
import time
import sys
# Generate a random string of the specified length
def RandomConnid(length):
    return ''.join(random.choice(string.ascii_uppercase + string.digits) for _
in range(length))
# Generate the parameters required for connection to IoT Hub
def IotHmac(productID, devicename, devicePsk):
    # 1. Generate `connid` as a random string to facilitate troubleshooting on
the backend
    connid = RandomConnid(5)
    # 2. Generate the expiration time of the signature, which is a UTF-8
string of the number of seconds since 00:00:00 UTC on January 1, 1970
    expiry = int(time.time()) + 60 * 60
    # 3. Generate the `clientid` part of MQTT in the format of
`${productid}${devicename}`
    clientid = "{}{}".format(productID, devicename)
    # 4. Generate the `username` part of MQTT in the format of
`${clientid};${sdkappid};${connid};${expiry}`
    username = "{};12010126;{};{}".format(clientid, connid, expiry)
    # 5. Sign the `username` to generate a token
    secret_key = devicePsk.encode('utf-8') # convert to bytes
    data_to_sign = username.encode('utf-8') # convert to bytes
    secret_key = base64.b64decode(secret_key) # this is still bytes
    token = hmac.new(secret_key, data_to_sign,
digestmod=hashlib.sha256).hexdigest()
    # 6. Generate the `password` field according to the rules of IoT Hub
platform
    password = "{};{}".format(token, "hmacsha256")
    return {
        "clientid" : clientid,
        "username" : username,
        "password" : password
    }
if __name__ == '__main__':
    print(IotHmac(sys.argv[1], sys.argv[2], sys.argv[3]))
```

Save the above code in `IotHmac.py` and run the following command. Here, replace `YOUR_PRODUCTID` , `YOUR_DEVICENAME` , and `YOUR_PSK` with the product ID, device name, and device key of the device you actually created.

```
python3 IotHmac.py "YOUR_PRODUCTID" "YOUR_DEVICENAME" "YOUR_PSK"
```

The code in Java is as follows:

```
import javax.crypto.Mac;
```

```
import javax.crypto.spec.SecretKeySpec;
import java.util.*;
public class IotHmac {
    public static void main(String[] args) throws Exception {
        System.out.println(IotHmac("YOUR_PRODUCTID","YOUR_DEVICENAME","YOUR_PSK"));
    }
    public static Map<String, String> IotHmac(String productID, String devicename,
        String devicePsk) throws Exception {
        final Base64.Decoder decoder = Base64.getDecoder();
        //1. Generate `connid` as a random string to facilitate
        troubleshooting on the backend
        String connid = HMACSHA256.getRandomString2(5);
        //2. Generate the expiration time of the signature, which is a UTF-
        8 string of the number of seconds since 00:00:00 UTC on January 1, 1970
        Long expiry = Calendar.getInstance().getTimeInMillis()/1000 +600;
        //3. Generate the `clientid` part of MQTT in the format of
        `${productid}${devicename}`
        String clientid = productID+devicename;
        //4. Generate the `username` part of MQTT in the format of
        `${clientid};${sdkappid};${connid};${expiry}`
        String username = clientid+";"+"12010126;" +connid+";"+expiry;
        //5. Sign the `username` to generate a token. Then, generate the
        `password` field according to the rules of IoT Hub platform
        String password = HMACSHA256.getSignature(username.getBytes(),
        decoder.decode(devicePsk)) + ";hmacsha256";
        Map<String,String> map = new HashMap<>();
        map.put("clientid",clientid);
        map.put("username",username);
        map.put("password",password);
        return map;
    }
    public static class HMACSHA256 {
        private static final String HMAC_SHA256 = "HmacSHA256";
        /**
         * Generate the signature data
         *
         * @param data The data to be encrypted
         * @param key The key used for encryption
         * @return The generated hexadecimal string
         */
        public static String getSignature(byte[] data, byte[] key) {
            try {
                SecretKeySpec signingKey = new SecretKeySpec(key,
                HMAC_SHA256);
                Mac mac = Mac.getInstance(HMAC_SHA256);
                mac.init(signingKey);
```

```
        byte[] rawHmac = mac.doFinal(data);
        return bytesToHexString(rawHmac);
    }catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
/**
 * Convert the `byte[]` array into a hexadecimal string
 *
 * @param bytes The byte array to be converted
 * @return Converted result
 */
private static String bytesToHexString(byte[] bytes) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < bytes.length; i++) {
        String hex = Integer.toHexString(0xFF & bytes[i]);
        if (hex.length() == 1) {
            sb.append('0');
        }
        sb.append(hex);
    }
    return sb.toString();
}
public static String getRandomString2(int length) {
    Random random = new Random();
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < length; i++) {
        int number = random.nextInt(3);
        long result = 0;
        switch (number) {
            case 0:
                result = Math.round(Math.random() * 25 + 65);
                sb.append(String.valueOf((char) result));
                break;
            case 1:
                result = Math.round(Math.random() * 25 + 97);
                sb.append(String.valueOf((char) result));
                break;
            case 2:
                sb.append(String.valueOf(new Random().nextInt(10)));
                break;
        }
    }
    return sb.toString();
}
}
```

```
}
```

The code in Node.js and JavaScript is as follows:

```
// The following is the way to import the node. If a browser is used, use the
// corresponding way to import the `crypto-js` library
const crypto = require('crypto-js')

// Function for generating random numbers
const randomString = (len) => {
  len = len || 32;
  var chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
  var maxPos = chars.length;
  var pwd = '';
  for (let i = 0; i < len; i++) {
    pwd += chars.charAt(Math.floor(Math.random() * maxPos));
  }
  return pwd;
}

// The product ID, device name, and device key are required
const productId = 'YOUR_PRODUCTID';
const deviceName = 'YOUR_DEVICENAME';
const devicePsk = 'YOUR_PSK';

// 1. Generate `connid` as a random string to facilitate troubleshooting on the
// backend
const connid = randomString(5);
// 2. Generate the expiration time of the signature, which is a UTF-8 string of
// the number of seconds since 00:00:00 UTC on January 1, 1970
const expiry = Math.round(new Date().getTime() / 1000) + 3600 * 24;
// 3. Generate the `clientid` part of MQTT in the format of
// `${productId}${devicename}`
const clientId = productId + deviceName;
// 4. Generate the `username` part of MQTT in the format of
// `${clientId};${sdkappid};${connid};${expiry}`
const userName = `${clientId};12010126;${connid};${expiry}`;
//5. Sign the `username` to generate a token. Then, generate the `password`
// field according to the rules of IoT Hub platform
const rawKey = crypto.enc.Base64.parse(devicePsk); // Base64-decode the
// device key
const token = crypto.HmacSHA256(userName, rawKey);
const password = token.toString(crypto.enc.Hex) + ";hmacsha256";
console.log(`userName:${userName}\\npassword:${password}`);
```

The code in C is as follows:

Note:

For more information on the code in C, please see [here](#).

```
#include "limits.h"
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include "HAL_Platform.h"
#include "utils_base64.h"
#include "utils_hmac.h"

/* Max size of base64 encoded PSK = 64, after decode: 64/4*3 = 48*/
#define DECODE_PSK_LENGTH 48

/* MAX valid time when connect to MQTT server. 0: always valid */
/* Use this only if the device has accurate UTC time. Otherwise, set to 0 */
#define MAX_ACCESS_EXPIRE_TIMEOUT (0)

/* Max size of conn Id */
#define MAX_CONN_ID_LEN (6)

/* IoT C-SDK APPID */
#define QCLOUD_IOT_DEVICE_SDK_APPID "21***06"
#define QCLOUD_IOT_DEVICE_SDK_APPID_LEN (sizeof(QCLOUD_IOT_DEVICE_SDK_APPID) - 1)

static void HexDump(char *pData, uint16_t len)
{
    int i;

    for (i = 0; i < len; i++) {
        HAL_Printf("0x%02.2x ", (unsigned char)pData[i]);
    }
    HAL_Printf("\n");
}

static void get_next_conn_id(char *conn_id)
{
    int i;
    srand((unsigned)HAL_GetTimeMs());
    for (i = 0; i < MAX_CONN_ID_LEN - 1; i++) {
        int flag = rand() % 3;
        switch (flag) {
            case 0:
                conn_id[i] = (rand() % 26) + 'a';
                break;
        }
    }
}
```



```
        case 1:
            conn_id[i] = (rand() % 26) + 'A';
            break;
        case 2:
            conn_id[i] = (rand() % 10) + '0';
            break;
    }
}

conn_id[MAX_CONN_ID_LEN - 1] = '\\0';
}

int main(int argc, char **argv)
{
    char *product_id    = NULL;
    char *device_name   = NULL;
    char *device_secret = NULL;

    char *username      = NULL;
    int   username_len  = 0;
    char  conn_id[MAX_CONN_ID_LEN];

    char password[51]   = {0};
    char username_sign[41] = {0};

    char  psk_base64decode[DECODE_PSK_LENGTH];
    size_t psk_base64decode_len = 0;

    long cur_timestamp = 0;

    if (argc != 4) {
        HAL_Printf("please ./qcloud-mqtt-sign product_id device_name
device_secret\\r\\n");
        return -1;
    }

    product_id    = argv[1];
    device_name   = argv[2];
    device_secret = argv[3];

    /* first device_secret base64 decode */
    qcloud_iot_utils_base64decode((unsigned char *)psk_base64decode,
DECODE_PSK_LENGTH, &psk_base64decode_len,
                                (unsigned char *)device_secret,
strlen(device_secret));
    HAL_Printf("device_secret base64 decode:");
    HexDump(psk_base64decode, psk_base64decode_len);
}
```

```
/* second create mqtt username
 * [productdevicename;appid;randomconnid;timestamp] */
cur_timestamp = HAL_Timer_current_sec() + MAX_ACCESS_EXPIRE_TIMEOUT / 1000;
if (cur_timestamp <= 0 || MAX_ACCESS_EXPIRE_TIMEOUT <= 0) {
    cur_timestamp = LONG_MAX;
}

// 20 for timestamp length & delimiter
username_len = strlen(product_id) + strlen(device_name) +
QCLOUD_IOT_DEVICE_SDK_APPID_LEN + MAX_CONN_ID_LEN + 20;
username      = (char *)HAL_Malloc(username_len);
if (username == NULL) {
    HAL_Printf("malloc username failed!\r\n");
    return -1;
}

get_next_conn_id(conn_id);
HAL_Snprintf(username, username_len, "%s%s;%s;%s;%ld", product_id,
device_name, QCLOUD_IOT_DEVICE_SDK_APPID,
            conn_id, cur_timestamp);

/* third use psk_base64decode hmac_sha1 calc mqtt username sign crate mqtt
 * password */
utils_hmac_sha1(username, strlen(username), username_sign,
psk_base64decode, psk_base64decode_len);
HAL_Printf("username sign: %s\r\n", username_sign);
HAL_Snprintf(password, 51, "%s;hmacsha1", username_sign);

HAL_Printf("Client ID: %s\r\n", product_id, device_name);
HAL_Printf("username : %s\r\n", username);
HAL_Printf("password : %s\r\n", password);

HAL_Free(username);

return 0;
}
```

6. Finally, enter the parameters generated above in the corresponding MQTT `connect` message.

7. Enter the `clientid` value in the `clientid` field of the MQTT protocol.

8. Enter the `username` value in the `username` field of the MQTT protocol.

9. Enter the `password` value in the `password` field of the MQTT protocol and send a `MqttConnect` message to the domain name and port of key authentication to connect to IoT Hub.

MQTT-Based Device Connection over WebSocket

Last updated : 2024-12-27 15:44:26

MQTT-WebSocket Overview

The IoT Hub platform supports MQTT communication based on WebSocket, so that devices can use the MQTT protocol for message transfer on the basis of the WebSocket protocol. In this way, browser-based applications can implement data communication with the platform and devices connected to the platform. In addition, WebSocket uses ports 443/80, which means that messages can pass through most firewalls during transfer.

MQTT-WebSocket Connection

As both the MQTT-WebSocket and MQTT-TCP protocols ultimately transfer messages based on MQTT, they have the same parameters for MQTT connection. The main difference lies in the protocol and port of the MQTT connection to the platform. Key-authenticated devices use WS for connection, while certificate-authenticated devices use WSS, i.e., WS+TLS.

Guide to connecting certificate-authenticated device

1. Download files such as the certificate and device private key.
2. Connect to the domain name. For devices in the Guangzhou region, connect to `${ProductId}.ap-guangzhou.iothub.tencentdevices.com:443`, where `${ProductId}` is the product ID.
3. Set the MQTT connection parameters:

The connection parameter settings are the same as those for MQTT-TCP connection. For more information, please see the MQTT connection section in [MQTT-Based Device Connection over TCP](#).

```
UserName:${productid}${devicename};${sdkappid};${connid};${expiry}
Password: password (you can set any value)
ClientId:${ProductId}${DeviceName}
KeepAlive: time to keep the connection alive. Value range: 0-900s
```

Guide to connecting key-authenticated device

1. Get the device key.
2. Connect to the domain name. For devices in the Guangzhou region, connect to `${ProductId}.ap-guangzhou.iothub.tencentdevices.com:80`, where `${ProductId}` is the product ID.

3. Set the MQTT connection parameters:

The connection parameter settings are the same as those for MQTT-TCP connection. For more information, please see the "Guide to connecting key-authenticated device" section in [MQTT-Based Device Connection over TCP](#).

```
UserName:${productid}${devicename};${sdkappid};${connid};${expiry}
PassWord:${token};hmac signature algorithm
ClientId:${ProductId}${DeviceName}
KeepAlive: time to keep the connection alive. Value range: 0-900s
```

MQTT Persistent Session

Last updated : 2024-12-27 15:44:26

IoT Hub supports the MQTT v3.1.1 protocol and supports the quality of service levels of QoS 0 and QoS 1 (but not QoS 2). Using an MQTT persistent session can save the subscription status of devices and subscribed messages that devices have not received. When a device goes offline and goes online again, it can restore the previous session to receive subscribed messages sent when it was offline.

Creating MQTT Persistent Session on Device

When a device is connected to IoT Hub, the `CleanSession` flag in the variable header part of the `Connect` message can be set to 0. IoT Hub will determine the session status of the device according to the `ClientId` when the device is connected. If there is no session currently, it will create a persistent session. If there is an existing session, it will communicate with the device based on the session process.

IoT Hub Response Description

After the device sends a `Connect` message, IoT Hub will return a `Connack` message, in which the connection confirmation flag `SessionPresent` indicates whether IoT Hub includes the session status corresponding to the `ClientId` when the device is connected. If `SessionPresent` is 0, no persistent session is created, and the device needs to establish the session status again. If `SessionPresent` is 1, a persistent session has been created.

After the device is successfully connected, if it enters an existing persistent session, IoT Hub will send the stored QoS 1 messages and unacknowledged QoS 1 messages to the device.

After the device is successfully connected, if a new persistent session is created, IoT Hub will save the subscription status of the device and store the QoS 1 (excluding QoS 0) messages that the device has subscribed to when it is offline. When it goes online again, IoT Hub will send the stored QoS 1 messages and unacknowledged QoS 1 messages to it.

Note:

IoT Hub sends the stored QoS 1 messages sequentially at 500 ms intervals.

Only QoS 1 messages can be stored in a persistent session. Up to 150 messages can be stored for a maximum of 24 hours for each device.

Closing MQTT Persistent Session

The MQTT persistent session can be closed in the following two ways:

When connecting the device to IoT Hub, set the `CleanSession` flag in the variable header part of the `Connect` message to 1.

When the device is disconnected for **more than 24 hours**, the persistent session will be closed automatically.

Note:

Device disconnections include disconnection caused by the device sending the `disconnect` message and disconnection caused by the device communication timeout.

CoAP-Based Device Connection

Last updated : 2024-12-27 15:44:26

Currently, IoT Hub supports connection over the standard CoAP protocol. For more information, please see [RFC 7252](#).

Differences from Standard CoAP

1. Currently, only message reporting is supported, i.e., reporting SDK messages to IoT Hub.
2. The POST method is supported, while GET/PUT/DELETE methods are not.

Security Level of CoAP Channel

1. DTLS protocol is supported to establish a secure connection.
2. Asymmetric encryption is supported.

Connection Parameters

1. Server address. For devices in the Guangzhou region, enter `${ProductId}.iotcloud.tencentdevices.com`. Here, `${ProductId}` is a variable parameter, and you should replace it with the product ID automatically generated when you create the product.
2. The connection port is 5684.

URI Specification

The CoAP message is sent to the URI in the format of `/${productId}/${deviceName}/xxx`, where `productId` is the product ID registered in the console, and `deviceName` is the name of the device under the `productId`.

By default, after a product is created, all devices under it will have the permissions of the following topic classes:

1. `${productId}/${deviceName}/event` for publishing
2. `${productId}/${deviceName}/control` for subscribing
3. `${productId}/${deviceName}/data` for publishing and subscribing

In other words, the URI corresponds to the MQTT topic.

HTTP-Based Device Connection

Last updated : 2024-12-27 15:44:26

Parameter Description

When a device reports a message, it needs to carry `ProductId` , `DeviceName` , and `TopicName` to initiate an `http/https` request to the platform. The request API and parameters are as detailed below:

Requested URL:

```
https://ap-guangzhou.gateway.tencentdevices.com/device/publish
```

```
http://ap-guangzhou.gateway.tencentdevices.com/device/publish
```

Request method: POST

Request parameters

Parameter	Required	Type	Description
ProductId	Yes	String	Product ID
DeviceName	Yes	String	Device name
TopicName	Yes	String	Name of the topic for publishing the message
Payload	Yes	String	Content of the published message
PayloadEncoding	No	String	Encoding for the published message. Currently, only Base64-encoding is supported. If this parameter is left empty, the original message content will be sent.
Qos	Yes	Integer	Message QoS level

Note:

The API only supports the `application/json` format.

Signature generation

There are two types of signatures for request messages. Key authentication uses the HMACSHA256 algorithm, and certificate authentication uses the RSA_SHA256 algorithm. For more information, please see [Signature Algorithm](#).

Platform response parameters

Parameter	Type	Description

RequestId	String	Request ID
-----------	--------	------------

Sample Code

Request packet

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/publish
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 155****065
X-TC-Nonce: 5456
X-TC-Signature:
2230eefd229f582d8b1b891af7107b915972407****78ab3738f756258d7652c
{"DeviceName":"AAAAAA","Payload":"123","ProductId":"G8N****AHB","Qos":1,"TopicName":"G8N****AHB/AAAAAA/data"}
```

Response packet

```
{
  "Response": {
    "RequestId": "f4da4f1f-d72e-40f1-****-349fc0072ba0"
  }
}
```

Device Connection Regions

Last updated : 2024-12-27 15:44:26

Currently, IoT Hub can be connected from the following regions:

Chinese Site	International Site
Chinese mainland	Chinese mainland
East US (Virginia)	East US (Virginia)
Central Europe (Frankfurt)	Central Europe (Frankfurt)
Southeast Asia (Bangkok)	Southeast Asia (Bangkok)

Domains for MQTT Connection

When connecting a device, you can use one of the following server domains based on your needs:

Region	Domain
Chinese mainland	<code>\${productid}.iotcloud.tencentdevices.com</code>
East US (Virginia)	<code>\${productid}.us-east.iotcloud.tencentdevices.com</code>
Central Europe (Frankfurt)	<code>\${productid}.europe.iothub.tencentdevices.com</code>
Southeast Asia (Bangkok)	<code>\${productid}.ap-bangkok.iothub.tencentdevices.com</code>

Domains for CoAP Connection

When connecting a device, you can use one of the following server domains based on your needs:

Region	Domain
Chinese mainland	<code>\${productid}.iotcloud.tencentdevices.com</code>
East US (Virginia)	<code>\${productid}.us-east.iotcloud.tencentdevices.com</code>
Central Europe (Frankfurt)	<code>\${productid}.europe.iothub.tencentdevices.com</code>
Southeast Asia (Bangkok)	<code>\${productid}.ap-bangkok.iothub.tencentdevices.com</code>

Domains for HTTP Connection

When connecting a device, you can use one of the following server domains based on your needs:

Region	Domain
Chinese mainland	ap-guangzhou.gateway.tencentdevices.com
East US (Virginia)	us-east.gateway.tencentdevices.com
Central Europe (Frankfurt)	europa.gateway.tencentdevices.com
Southeast Asia (Bangkok)	ap-bangkok.gateway.tencentdevices.com

Gateway Subdevice Feature Overview

Last updated : 2024-12-27 15:54:13

Device Classification

The IoT Hub platform divides devices into the following two categories (i.e., node categories) according to their functionality:

General device: this device category is further divided into two types: devices that have the ability to connect to the platform, and devices that can connect to the platform through gateway devices.

Gateway device: this category of device can directly connect to the platform and can accept subdevices for them to join the LAN.

Overview

Devices that don't have direct access to the Ethernet can be connected to the network of the local gateway device first and then connected to the IoT Hub platform through the communication feature of the gateway device. For the subdevices that join or leave the LAN, the gateway device can bind or unbind them on the platform and report the topological relationships with them, so as to implement the real-time monitoring of all devices in the LAN by the platform.

Connection Method

Gateway devices can be connected to the IoT Hub platform in the same way as general devices. For more information, please see [Device Connection](#). After a gateway device is connected, it can connect/disconnect subdevices in the same LAN to/from the platform and manage the topological relationships between them. Subdevices can be connected to the platform through a gateway device after successful authentication in the following two ways:

Device-level key authentication

The gateway device gets the device certificate or key of the subdevice, generates the subdevice binding signature string, reports it to the platform, and completes the identity verification on behalf of the subdevice.

Product-level key authentication

The gateway device gets the `ProductKey` (product key) of the subdevice, generates a signature, and sends a dynamic registration request to the platform. After successful authentication, the platform will return the

`DeviceCert` or `DeviceSecret` of the subdevice, based on which the gateway will generate the subdevice binding signature string and report it to the platform. Then, after successful verification, the subdevice can be connected.

Topological Relationship Management

Last updated : 2024-12-27 15:54:13

Feature overview

A gateway device can bind and unbind subdevices under it through data communication with the cloud. To implement this feature, the following two topics will be used:

Data upstream topic (for publishing): `$gateway/operation/${productid}/${devicename}`

Data downstream topic (for subscribing): `$gateway/operation/result/${productid}/${devicename}`

Binding device

The gateway device can request to add its topological relationship with the subdevice through the data upstream topic so as to bind the subdevice. After the request succeeds, the platform will return the binding information of the subdevice through the data downstream topic.

Data format of the subdevice binding request:

```
{
  "type": "bind",
  "payload": {
    "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev",
        "signature": "signature",
        "random": 121213,
        "timestamp": 1589786839,
        "signmethod": "hmacsha256",
        "authtype": "psk"
      }
    ]
  }
}
```

Request parameter description:

Parameter	Type	Description
type	String	Gateway message type. For subdevice binding, the value is <code>bind</code>
payload.devices	Array	List of subdevices to be bound
product_id	String	Subdevice product ID

device_name	String	Subdevice name
signature	String	Signature string for subdevice binding. Signature algorithm: 1. Concatenate the product ID, device name, random number, and timestamp into the string to sign: <code>text=\${product_id}\${device_name};\${random};\${expiration_time}</code> 2. Use the PSK of the device or the SHA1 digest of the certificate to sign: <code>sign = hmac_sha1(device_secret, text)</code>
random	Int	Random number
timestamp	Int	Timestamp in seconds
signmethod	String	Signature algorithm. <code>hmacsha1</code> and <code>hmacsha256</code> are supported
authtype	String	Signature type. psk: uses device PSK to sign certificate: uses device public key certificate to sign

Data format of the subdevice binding response:

```
{
  "type": "bind",
  "payload": {
    "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev",
        "result": -1
      }
    ]
  }
}
```

Response parameter description:

Parameter	Type	Description
type	String	Gateway message type. For subdevice binding, the value is <code>bind</code>
payload.devices	Array	List of subdevices to be bound
product_id	String	Subdevice product ID
device_name	String	Subdevice name
result	Int	Subdevice binding result. For specific error codes, please see the table below

Unbinding device

The gateway device can request to unbind its topological relationship with the subdevice through the data upstream topic. After the request succeeds, the platform will return the unbinding information of the subdevice through the data downstream topic.

Data format of the subdevice unbinding request:

```
{
  "type": "unbind",
  "payload": {
    "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev"
      }
    ]
  }
}
```

Request parameter description:

Parameter	Type	Description
type	String	Gateway message type. For subdevice unbinding, the value is <code>unbind</code>
payload.devices	Array	List of subdevices to be unbound
product_id	String	Subdevice product ID
device_name	String	Subdevice name

Data format of the subdevice unbinding response:

```
{
  "type": "bind",
  "payload": {
    "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev",
        "result": -1
      }
    ]
  }
}
```



```
}

```

Response parameter description:

Parameter	Type	Description
type	String	Gateway message type. For subdevice unbinding, the value is <code>unbind</code>
payload.devices	Array	List of subdevices to be unbound
product_id	String	Subdevice product ID
device_name	String	Subdevice name
result	Int	Subdevice binding result. For more information, please see Error codes

Querying topological relationship

The gateway device can upstream a request to query the topological relationship of the subdevice through this topic.

Data upstream topic: `$gateway/operation/${productid}/${devicename}`

Data downstream topic: `$gateway/operation/result/${productid}/${devicename}`

Data format of the subdevice topological relationship query request:

```
{
  "type": "describe_sub_devices"
}
```

Request parameter description:

Parameter	Type	Description
type	String	Gateway message type. For subdevice query, the value is <code>describe_sub_devices</code>

Data format of the subdevice topological relationship query response:

```
{
  "type": "describe_sub_devices",
  "payload": {
    "devices": [
      {
        "product_id": "XKFA****LX",
        "device_name": "2OGDy7Ws8mG****YUe"
      },
      {

```

```

    "product_id": "XKFA****LX",
    "device_name": "5gcEHg3Yuvm****2p8"
  },
  {
    "product_id": "XKFA****LX",
    "device_name": "hmIjq0gEFcf****F5X"
  },
  {
    "product_id": "XKFA****LX",
    "device_name": "x9pVpmdRmET****mkM"
  },
  {
    "product_id": "XKFA****LX",
    "device_name": "zmHv6o6n4G3****Bgh"
  }
]
}
}

```

Response parameter description:

Parameter	Type	Description
type	String	Gateway message type. For subdevice query, the value is <code>describe_sub_devices</code>
payload.devices	Array	List of subdevices bound to the gateway
product_id	String	Subdevice product ID
device_name	String	Subdevice name

Changing topological relationship

The gateway device can subscribe to the topological relationship changes of the subdevice on the platform through this topic.

Data downstream topic: `$gateway/operation/result/${productid}/${devicename}`

When the subdevice is bound or unbound, the gateway will receive the change in the topological relationship of the subdevice. The data format is as follows:

```

{
  "type": "change",
  "payload": {
    "status": 0, // 0: unbound, 1: bound
    "devices": [
      {

```

```

    "product_id": "CFCS***G7",
    "device_name": "***ev",
  }
]
}
}

```

Request parameter description:

Parameter	Type	Description
type	String	Gateway message type. For topological relationship change, the value is <code>change</code>
status	Int	Topological relationship change status. 0: unbound 1: bound
payload.devices	Array	List of subdevices bound to the gateway
product_id	String	Subdevice product ID
device_name	String	Subdevice name

Data format of the gateway response:

```

{
  "type": "change",
  "result": 0
}

```

Response parameter description:

Parameter	Type	Description
type	String	Gateway message type. For topological relationship change, the value is <code>change</code>
result	Int	Gateway response processing result

Error codes

Error Code	Description
0	Success

-1	The gateway device is not bound to the subdevice
-2	System error. Subdevice connection or disconnection failed
801	Request parameter error
802	The device name is invalid, or the device does not exist
803	Signature verification failed
804	The signature algorithm is not supported
805	The signature request has expired
806	This device has already been bound
807	Non-general devices cannot be bound
808	Forbidden operation
809	Duplicate binding
810	Unsupported subdevice

Proxied Subdevice Connection and Disconnection

Last updated : 2024-12-27 15:54:13

Feature Overview

A gateway device can connect and disconnect subdevices under it through the data communication with the cloud.

This feature uses the same topics as those used in gateway subdevice topology management:

Data upstream topic (for publishing): `$gateway/operation/${productid}/${devicename}`

Data downstream topic (for subscribing): `$gateway/operation/result/${productid}/${devicename}`

Proxied subdevice connection

The gateway device can connect the subdevice to the platform through the data upstream topic. After the request succeeds, the platform will return the connection information of the subdevice through the data downstream topic.

Data format of the proxied subdevice connection request:

```
{
  "type": "online",
  "payload": {
    "devices": [
      {
        "product_id": "CFCSQ5EAG7",
        "device_name": "onlinedev00"
      }
    ]
  }
}
```

Data format of the proxied subdevice connection response:

```
{
  "type": "online",
  "payload": {
    "devices": [
      {
        "product_id": "CFCSQ5EAG7",
        "device_name": "onlinedev00",
        "result": 0
      }
    ]
  }
}
```

Request parameter description:

Parameter	Type	Description
type	String	Gateway message type. For proxied subdevice connection, the value is <code>online</code>
payload.devices	Array	List of subdevices to be connected
product_id	String	Subdevice product ID
device_name	String	Subdevice name

Response parameter description:

Parameter	Type	Description
type	String	Gateway message type. For proxied subdevice connection, the value is <code>online</code>
payload.devices	Array	List of subdevices to be connected
product_id	String	Subdevice product ID
device_name	String	Subdevice name
result	Int	Subdevice connection result. For specific error codes, please see the table below

Proxied subdevice disconnection

The gateway device can disconnect the subdevice from the platform through the data upstream topic. After the request succeeds, the platform will return the disconnection success information of the subdevice through the data downstream topic.

Data format of the proxied subdevice disconnection request:

```
{
  "type": "offline",
  "payload": {
    "devices": [
      {
        "product_id": "CFCSQ5EAG7",
        "device_name": "offlinedev00"
      }
    ]
  }
}
```

Data format of the proxied subdevice disconnection response:

```
{
  "type": "offline",
  "payload": {
    "devices": [
      {
        "product_id": "CFCSQ5EAG7",
        "device_name": "offlinedev00",
        "result": -1
      }
    ]
  }
}
```

Request parameter description:

Parameter	Type	Description
type	String	Gateway message type. For proxied subdevice disconnection, the value is <code>offline</code>
payload.devices	Array	List of proxied subdevices to be disconnected
product_id	String	Subdevice product ID
device_name	String	Subdevice name

Response parameter description:

Parameter	Type	Description
type	String	Gateway message type. For proxied subdevice disconnection, the value is <code>offline</code>
payload.devices	Array	List of proxied subdevices to be disconnected
product_id	String	Subdevice product ID
device_name	String	Subdevice name
result	Int	Subdevice disconnection result. For specific error codes, please see the table below

Error codes

Error Code	Description

0	Success
-1	The gateway device is not bound to the subdevice
-2	System error. Subdevice connection or disconnection failed
801	Request parameter error
802	The device name is invalid, or the device does not exist
810	Unsupported subdevice

Proxied Subdevice Publishing and Subscribing

Last updated : 2024-12-27 15:54:13

Overview

A gateway device can publish and subscribe to messages on behalf of subdevices under it through data communication with the cloud.

Prerequisites

Before publishing and subscribing to messages, please connect gateway devices and subdevices as instructed in [Gateway Product Connection](#) and [Proxied Subdevice Connection and Disconnection](#).

Publishing and Subscribing to Messages

Gateway devices can use the topic permissions of subdevices and send/receive messages on behalf of subdevices.

Subdevice Firmware Update

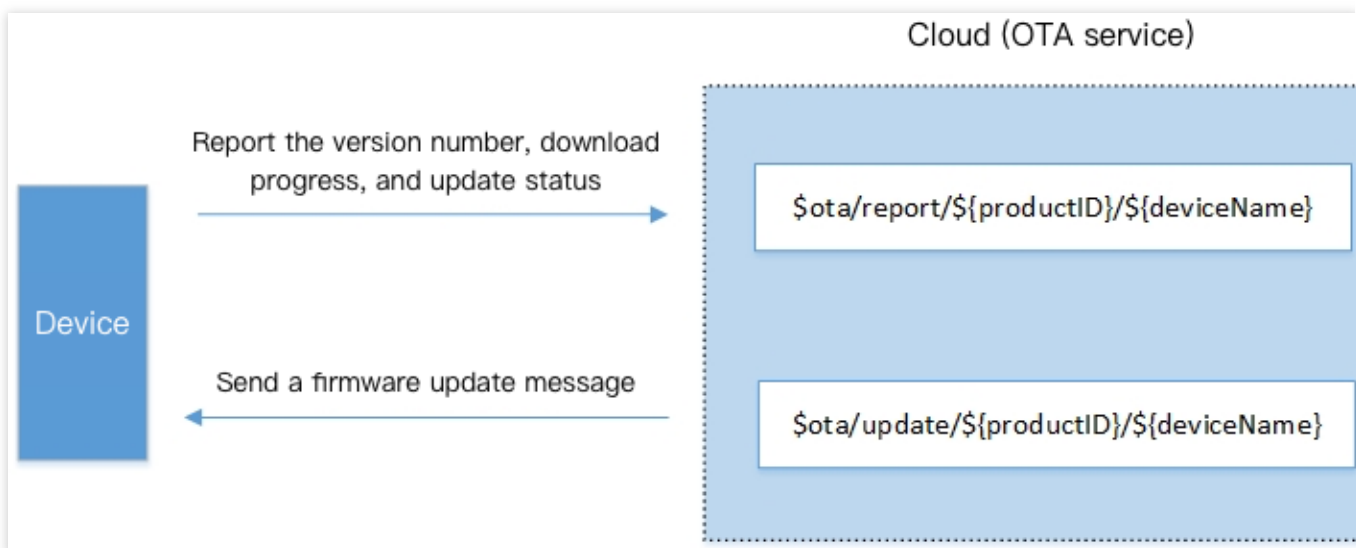
Last updated : 2024-12-27 15:54:14

Overview

When a subdevice under a gateway has new features available or vulnerabilities that need to be fixed, firmware update can be quickly performed for it through the device firmware update service.

How It Works

During the firmware update process, the gateway needs to use the following two topics to communicate with the cloud on behalf of the subdevice:



Below is the sample code:

```
$ota/report/${productID}/${deviceName}
```

This topic is used to publish (upstream) messages, through which the device reports `$ota/update/${productID}/${deviceName}`

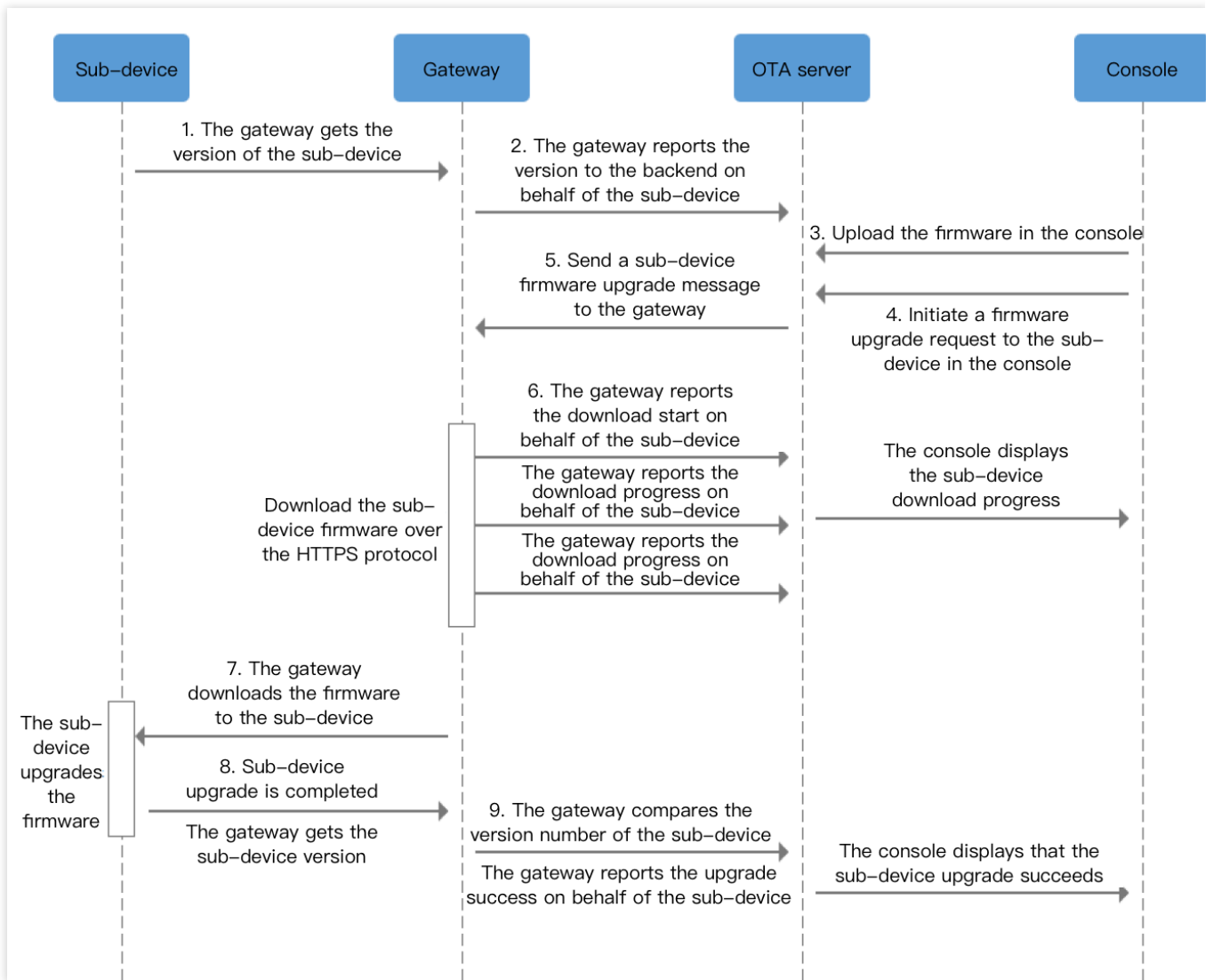
This topic is used to subscribe to (downstream) messages, through which the device

Process

Taking MQTT as an example, the update process of the subdevice is as follows:

Note:

For the specific directions of firmware update, please see [Device Firmware Update](#).



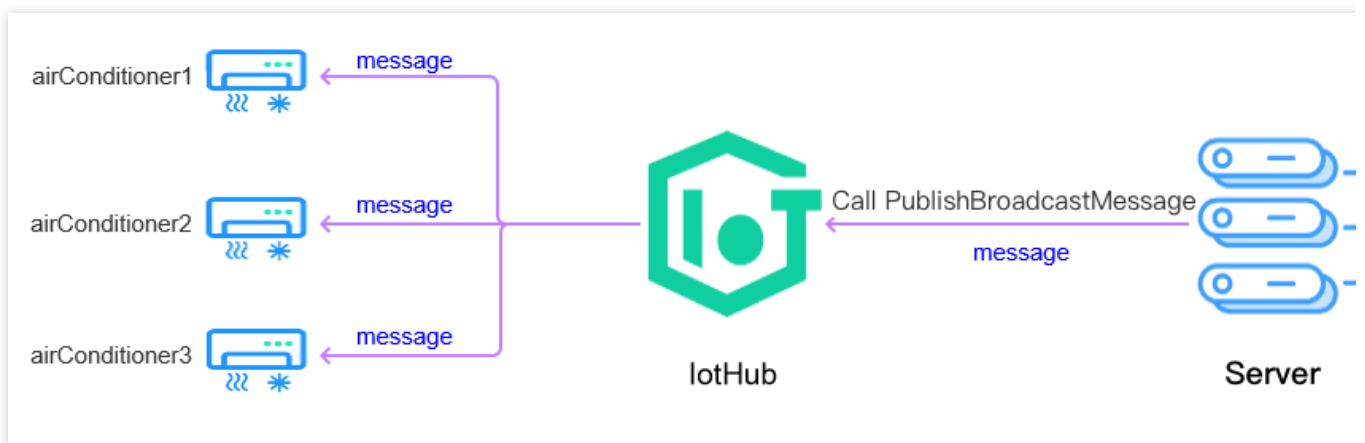
Message Communication

Broadcast Communication

Last updated : 2024-12-27 15:54:13

Feature Overview

The IoT Hub platform provides a broadcast communication topic. The server can publish a broadcast message by calling the broadcast communication API, which can be received by online devices that have subscribed to the broadcast topic under the same product.



Broadcast Topic

The broadcast communication topic is ``$broadcast/rxd/${ProductId}/${DeviceName}``, where ``${ProductId}`` and ``${DeviceName}`` represent the specific product ID and device name.

Broadcast Communication Sample

The sample completes device connection through the device-side [C-SDK](#) on Linux and calls APIs together with Tencent Cloud [API Explorer](#). The specific steps are as follows:

Creating device in console

Scenario description

If you have multiple air conditioner devices connected to the IoT Hub platform, then the server can send the same instruction to all of them for turning off.

The server calls the `PublishBroadcastMessage` API and specifies the `ProductId` and the `Payload` of the broadcast message. Then, all online devices that have subscribed to the broadcast topic under the product will receive the broadcast message with the `Payload`.

Creating product and device

Create an air conditioner product and `airConditioner1`, `airConditioner2`, and other devices as instructed in [Device Interconnection](#).

Compiling and running demo (with key-authenticated device as example)

1. Compile the SDK

Modify `CMakeLists.txt` to ensure that the following options exist:

```
set (BUILD_TYPE "release")
set (COMPILE_TOOLS "gcc")
set (PLATFORM "linux")
set (FEATURE_MQTT_COMM_ENABLED ON)
set (FEATURE_BROADCAST_ENABLED ON)
set (FEATURE_AUTH_MODE "KEY")
set (FEATURE_AUTH_WITH_NOTLS OFF)
set (FEATURE_DEBUG_DEV_INFO_USED OFF)
```

Run the following script for compilation:

```
./cmake_build.sh
```

The demo output `broadcast_sample` is in the `output/release/bin` folder.

2. Enter the device information

Enter the information of the `airConditioner1` device created above in a JSON file

`aircond_device_info1.json` :

```
{
  "auth_mode": "KEY",
  "productId": "KL4J2****8",
  "deviceName": "airConditioner1",
  "key_deviceinfo": {
    "deviceSecret": "zOZXUaycuwlePt****8dBA=="
  }
}
```

Enter the information of the `airConditioner2` device in another JSON file

`aircond_device_info2.json` :

```
{
  "auth_mode": "KEY",
  "productId": "KL4J2****8",
  "deviceName": "airConditioner2",
  "key_deviceinfo": {
    "deviceSecret": "+IiVNsYKRh0AW****IE07A=="
  }
}
```

Enter the information of other devices in corresponding JSON files in turn.

3. Run the `broadcast_sample` demo

As this scenario involves multiple demos running simultaneously, you can open multiple terminals to run the

`broadcast_sample` demo, and you will see that all the demos subscribe to the

`$broadcast/rxd/${productID}/${deviceName}` topic and are in the waiting status.

The output of the `airConditioner1` device is as follows:

```
./broadcast_sample -c ./aircond_device_info1.json -l 100
INF|2020-08-03 22:50:28|qcloud_iot_device.c|iot_device_info_set(50): SDK_Ver:
3.2.0, Product_ID: KL4J2****8, Device_Name: airConditioner1
DBG|2020-08-03 22:50:28|HAL_TLS_mbedtls.c|HAL_TLS_Connect(200): Setting up the
SSL/TLS structure...
DBG|2020-08-03 22:50:28|HAL_TLS_mbedtls.c|HAL_TLS_Connect(242): Performing the
SSL/TLS handshake...
DBG|2020-08-03 22:50:28|HAL_TLS_mbedtls.c|HAL_TLS_Connect(243): Connecting to
/KL4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 22:50:28|HAL_TLS_mbedtls.c|HAL_TLS_Connect(265): connected with
/KL4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 22:50:28|mqtt_client.c|IOT_MQTT_Construct(113): mqtt connect
with id: 9**** success
INF|2020-08-03 22:50:28|broadcast_sample.c|main(197): Cloud Device Construct
Success
DBG|2020-08-03 22:50:28|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(142):
topicName=$broadcast/rxd/KL4J2****8/airConditioner1|packet_id=*****
INF|2020-08-03 22:50:28|broadcast_sample.c|mqtt_event_handler(49): subscribe
success, packet-id=*****
DBG|2020-08-03 22:50:28|broadcast.c|_broadcast_event_callback(37): broadcast
topic subscribe success
```

The output of the `airConditioner2` device is as follows:

```
./broadcast_sample -c ./aircond_device_info2.json -l 100
```

```
INF|2020-08-03 22:51:24|qcloud_iot_device.c|iot_device_info_set(50): SDK_Ver:
3.2.0, Product_ID: KL4J2****8, Device_Name: airConditioner2
DBG|2020-08-03 22:51:24|HAL_TLS_mbedtls.c|HAL_TLS_Connect(200): Setting up the
SSL/TLS structure...
DBG|2020-08-03 22:51:24|HAL_TLS_mbedtls.c|HAL_TLS_Connect(242): Performing the
SSL/TLS handshake...
DBG|2020-08-03 22:51:24|HAL_TLS_mbedtls.c|HAL_TLS_Connect(243): Connecting to
/KL4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 22:51:24|HAL_TLS_mbedtls.c|HAL_TLS_Connect(265): connected with
/KL4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 22:51:25|mqtt_client.c|IOT_MQTT_Construct(113): mqtt connect
with id: f**** success
INF|2020-08-03 22:51:25|broadcast_sample.c|main(197): Cloud Device Construct
Success
DBG|2020-08-03 22:51:25|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(142):
topicName=$broadcast/rxd/KL4J2****8/airConditioner2|packet_id=*****
INF|2020-08-03 22:51:25|broadcast_sample.c|_mqtt_event_handler(49): subscribe
success, packet-id=*****
DBG|2020-08-03 22:51:25|broadcast.c|_broadcast_event_callback(37): broadcast
topic subscribe success
```

4. Call TencentCloud API `PublishBroadcastMessage` to send a broadcast message

Go to [API Explorer](#), enter the personal key and device parameter information, select **Online Call**, and send the request.

5. Observe the message reception of the air conditioners

Observe the printout of the `airConditioner1` device, and you can see that the message sent by the server has been received.

```
DBG|2020-08-03 22:55:32|broadcast.c|_broadcast_message_cb(25): topic=$broadcast/rxd
INF|2020-08-03 22:55:32|broadcast.c|_broadcast_message_cb(26): len=6, topic_msg=clo
INF|2020-08-03 22:55:32|broadcast_sample.c|_broadcast_message_handler(134): broadca
```

Observe the printout of the `airConditioner2` device, and you can see that the message sent by the server has also been received.

```
DBG|2020-08-03 22:55:32|broadcast.c|_broadcast_message_cb(25): topic=$broadcast/rxd
INF|2020-08-03 22:55:32|broadcast.c|_broadcast_message_cb(26): len=6, topic_msg=clo
INF|2020-08-03 22:55:32|broadcast_sample.c|_broadcast_message_handler(134): broadca
```

6. Turn off air conditioner devices

The devices that have received the instruction will parse the instruction for processing.

RRPC Communication

Last updated : 2024-12-27 15:54:13

Feature Overview

Because of the async communication mode of the MQTT protocol based on the publish/subscribe pattern, after the server controls a device, it cannot synchronously get the result returned by the device. To solve this problem, IoT Hub uses the Revert RPC (RRPC) technology to implement a sync communication mechanism.

How Communication Works

Communication topics

The subscription message topic `$rrpc/rxd/${productID}/${deviceName}/+` is used to subscribe to RRPC request messages sent by the cloud (downstream).

The request message topic `$rrpc/rxd/${productID}/${deviceName}/${processID}` is used for the cloud to publish (downstream) RRPC request messages.

The response message topic `$rrpc/txd/${productID}/${deviceName}/${processID}` is used to publish (upstream) RRPC response messages.

Note:

`${productID}` : product ID

`${deviceName}` : device name

`${processID}` : unique message ID generated by the server to identify different RRPC messages. The corresponding RRPC request message can be found through the `processID` carried in the RRPC response message.

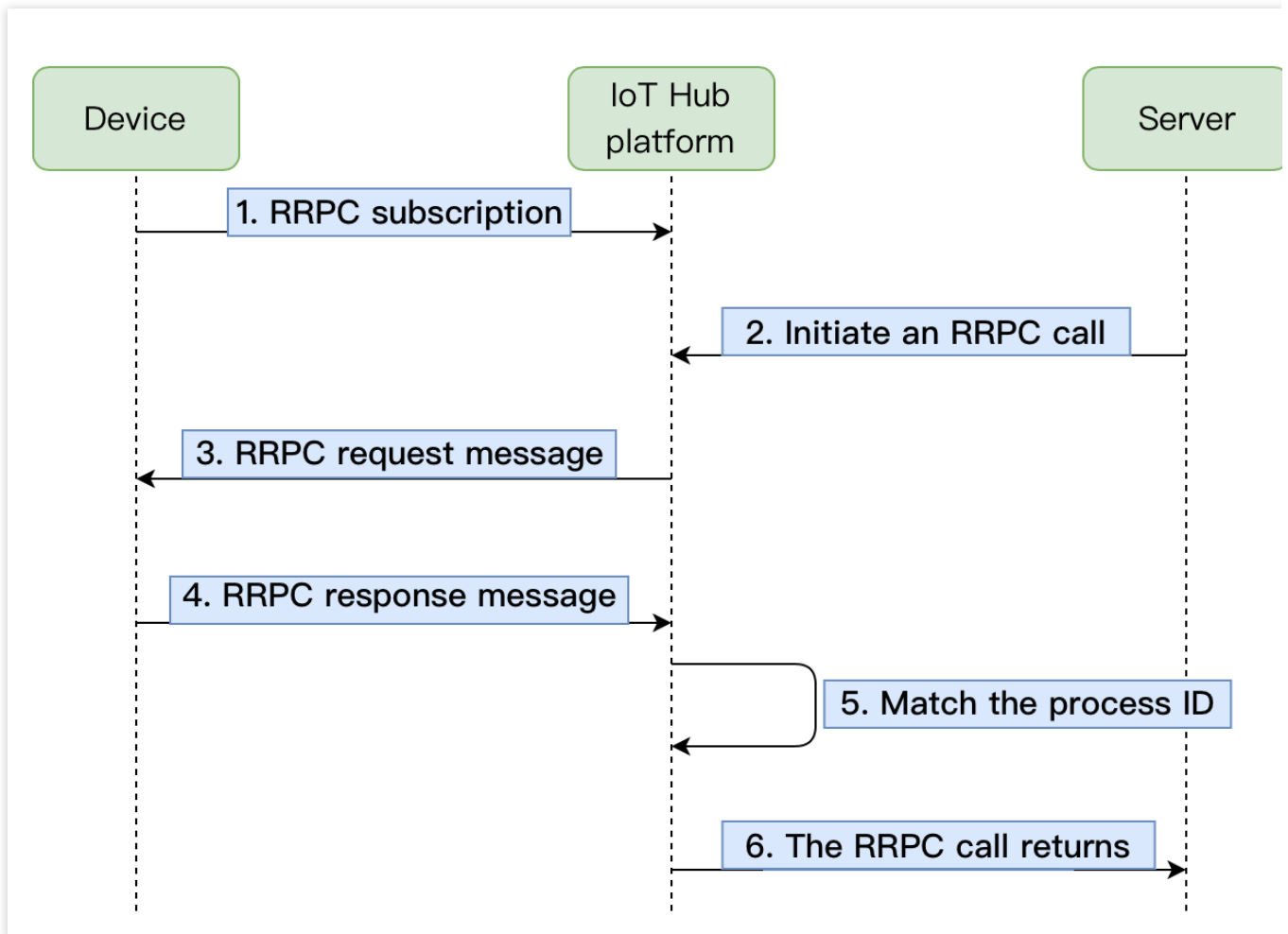
Communication process

1. The device subscribes to the RRPC subscription message topic.
2. The server publishes an RRPC request message by calling the PublishRRPCMessage API.
3. After receiving the message, the device extracts the `processID` distributed by the cloud in the request message topic, sets it as the `processID` of the response message topic, and publishes a return message of the device to the response message topic.
4. After IoT Hub receives the return message from the device, it matches the message according to the `processID` and sends the return message to the server.

Note:

RRPC requests time out in 4s, that is, if the device doesn't respond within 4s, the request will be considered to have timed out.

The flowchart is as follows:



RRPC communication sample

The sample completes device connection through the device-side [C-SDK](#) on Linux and calls APIs together with Tencent Cloud API Explorer. The specific steps are as follows:

Creating device in console

Creating product and device

Create an air conditioner product and an `airConditioner1` device as instructed in [Device Interconnection](#).

Compiling and running demo (with key-authenticated device as example)

1. Compile the SDK

Modify `CMakeLists.txt` and make sure that the following options exist:

```
set (BUILD_TYPE "release")
set (COMPILE_TOOLS "gcc")
set (PLATFORM "linux")
set (FEATURE_MQTT_COMM_ENABLED ON)
set (FEATURE_RRPC_ENABLED ON)
set (FEATURE_AUTH_MODE "KEY")
set (FEATURE_AUTH_WITH_NOTLS OFF)
set (FEATURE_DEBUG_DEV_INFO_USED OFF)
```

Run the following script for compilation:

```
./cmake_build.sh
```

The demo output `rrpc_sample` is in the `output/release/bin` folder.

2. Enter the device information

Enter the information of the `airConditioner1` device created above in the JSON file

`aircond_device_info1.json`.

```
{
  "auth_mode": "KEY",
  "productId": "KL4J2****8",
  "deviceName": "airConditioner1",
  "key_deviceinfo": {
    "deviceSecret": "zOZXUaycuwleP****78dBA=="
  }
}
```

3. Run the `rrpc_sample` demo

You can see that the `airConditioner1` device has subscribed to the RRPC message and then entered the waiting status.

```
./rrpc_sample -c ./aircond_device_info1.json -l 1000
INF|2020-08-03 23:57:55|qcloud_iot_device.c|iot_device_info_set(50): SDK_Ver:
3.2.0, Product_ID: KL4J2****8, Device_Name: airConditioner1
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(200): Setting up the
SSL/TLS structure...
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(242): Performing the
SSL/TLS handshake...
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(243): Connecting to
/KL4J2****8.iotcloud.tencentdevices.com/8883...
```

```
INF|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(265): connected with
/KL4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 23:57:56|mqtt_client.c|IOT_MQTT_Construct(113): mqtt connect
with id: 2**** success
INF|2020-08-03 23:57:56|rrpc_sample.c|main(206): Cloud Device Construct Success
DBG|2020-08-03 23:57:56|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(142):
topicName=$rrpc/rxd/KL4J2****8/airConditioner1/|packet_id=****
INF|2020-08-03 23:57:56|rrpc_sample.c|_mqtt_event_handler(49): subscribe
success, packet-id=*****
DBG|2020-08-03 23:57:56|rrpc_client.c|_rrpc_event_callback(104): rrpc topic
subscribe success
```

4. Call TencentCloud API `PublishRRPCMessage` to send an RRPC request message

Go to [API Explorer](#), enter the personal key and device parameter information, select **Online Call**, and send the request.

5. Observe the RRPC request message

Observe the printout of the `airConditioner1` device, and you can see that the RRPC request message has been received, and the `processID` is `***`.

```
DBG|2020-08-04 00:07:36|rrpc_client.c|_rrpc_message_cb(85):
topic=$rrpc/rxd/KL4J2****8/airConditioner1/***
INF|2020-08-04 00:07:36|rrpc_client.c|_rrpc_message_cb(86): len=6,
topic_msg=closed
INF|2020-08-04 00:07:36|rrpc_client.c|_rrpc_get_process_id(76): len=3, process
id=***
INF|2020-08-04 00:07:36|rrpc_sample.c|_rrpc_message_handler(137): rrpc
message=closed
```

6. Observe the RRPC response message

Observe the printout of the `airConditioner1` device, and you can see that the RRPC request message has been processed, the RRPC response message has been replied, and the `processID` is `***`.

```
DBG|2020-08-04 00:07:36|mqtt_client_publish.c|qcloud_iot_mqtt_publish(340):
publish
packetID=0|topicName=$rrpc/txd/KL4J2****8/airConditioner1/***|payload=ok
```

7. Observe the server response result

Observe the response result of the server, and you can see that the RRPC response message has been received. `MessageId` is `***`, and `Payload` is `****` after being `Base64-encoded`, which is the same as the actual

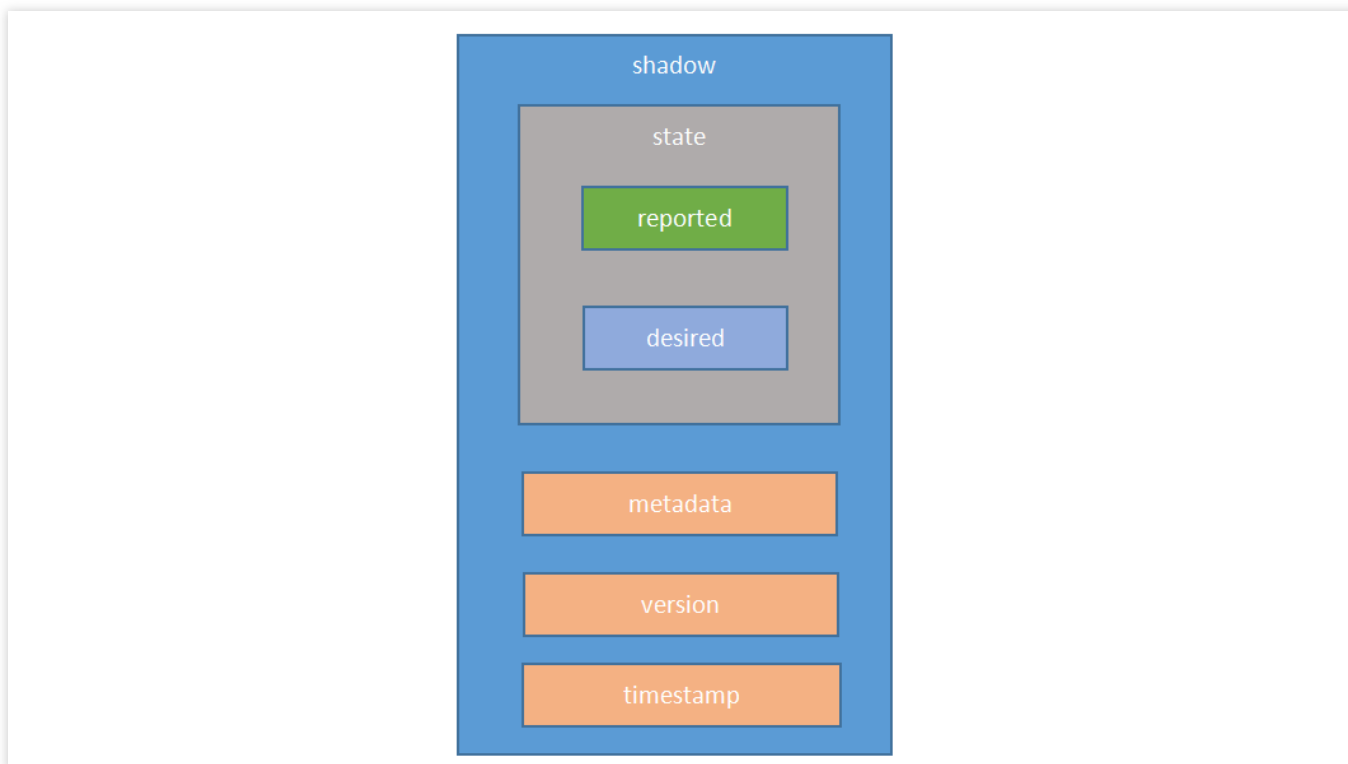
response message of the client after being `Base64-encoded` . At this point, it can be confirmed that the response message has been received.

Device Shadow

Device Shadow Details

Last updated : 2024-12-27 15:54:13

A device shadow document is a file of status and configuration data cached by the server for a device, which is stored as JSON text and consists of the following parts:



state

reported

This is the status reported by the device itself. The device can write data to this part of the document to report its new status, and the application can read this part to get the status of the device.

desired

This is the desired status of the device. The application writes data to this part of the document through the HTTP RESTful API to update the device status. The device SDK syncs the shadow data to the device by registering relevant attributes and callback through the device shadow service.

metadata

This is the metadata information of the device shadow, including the last updated time of each attribute in the `state` section.

version

This is the version number of the device shadow document, which is increased each time the document is updated. The version number is maintained by Tencent Cloud on the backend, ensuring that the data of the device is consistent with that of the device shadow.

timestamp

This is the last updated time of the device shadow document. Below is a sample document:

```
{
  "state": {
    "reported": {
      "attr_name1": "value1"
    },
    "desired": {
      "attr_name2": "value2"
    }
  },
  "metadata": {
    "reported": {
      "attr_name1": {
        "timestamp": 123456789
      }
    },
    "desired": {
      "attr_name2": {
        "timestamp": 123456789
      }
    }
  },
  "version": 1,
  "timestamp": 123456789
}
```

Blank part

Below is a sample device shadow document that is blank:

```
{
  "state": {},
  "metadata": {},
  "version": 0
}
```

Only when the device shadow document has the desired status will there be a `desired` part, and the `reported` part can be empty; for example:

```
{
  "state": {
    "desired": {
      "attr_name2": "value2"
    }
  },
  "metadata": {
    "desired": {
      "attr_name2": {
        "timestamp": 123456789
      }
    }
  },
  "version": 1,
  "timestamp": 123456789
}
```

After the device status is successfully updated, the latest status needs to be reported, and the `desired` part needs to be removed from the document. To remove this part, you need to set it to `null`; for example:

```
{
  "state": {
    "reported": {
      "attr_name1": "new_value1",
      "attr_name2": "new_value2"
    },
    "desired": null
  },
  "version": 1
}
```

Array

The device shadow document supports arrays. Only an entire array but not an element in the array can be updated, and none of the elements can be null.

Device Shadow Data Flow

Last updated : 2024-12-27 15:54:13

Device Shadow Topic

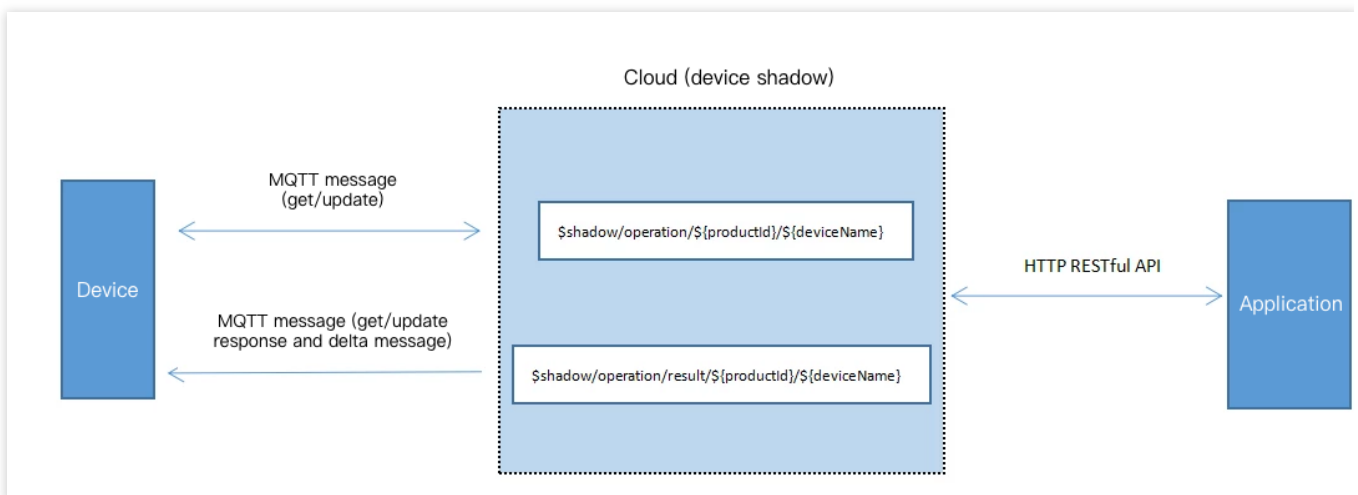
A device shadow acts as an intermediary, allowing the device and your application to view and update the device status. Communication between the device, application, and device shadow is implemented through two special topics:

`$shadow/operation/${productId}/${deviceName}` is used to publish (upstream) messages, through which `GET/UPDATE` operations can be implemented on device shadow data.

`$shadow/operation/result/${productId}/${deviceName}` is used to subscribe to (downstream) messages, through which the shadow server sends responses and push messages.

Note:

The above topics are created by the system by default when the device is created and will be automatically subscribed to within the device SDK.



Getting Shadow Status by Device

If the device wants to get the latest status of the device shadow, it needs to publish a GET message to the

`$shadow/operation/${productId}/${deviceName}` topic. The SDK provides an API that sends the GET message in a specific JSON string format:

```
{
  "type": "get",
```



```
"clientToken": "clientToken"
}
```

Note:

`clientToken` is the token used to uniquely identify the session, which is generated by the requester and returned as is by the responder.

For example, an air conditioner device can send a GET message to

`$shadow/operation/ABC1234567/AirConditioner` through the API provided by the SDK to get the latest device parameters.

The device shadow server responds by sending a message to the

`$shadow/operation/result/${productId}/${deviceName}` topic to return all data content of the device shadow through the JSON data, and the SDK will notify the business layer through the corresponding callback function.

The shadow server responds to the air conditioner device's GET request by sending the following data to

`$shadow/operation/result/ABC1234567/AirConditioner` . Below is the sample code:

```
{
  "type": "get",
  "result": 0,
  "timestamp": 1514967088,
  "clientToken": "clientToken",
  "payload": {
    "state": {
      "reported": {
        "temperature": 27
      },
      "desired": {
        "temperature": 25
      },
      "delta": {
        "temperature": 25
      }
    },
    "metadata": {
      "reported": {
        "temperature": {
          "timestamp": 1514967066
        }
      },
      "desired": {
        "temperature": {
          "timestamp": 1514967076
        }
      }
    }
  }
}
```

```
"delta": {
  "temperature": {
    "timestamp": 1514967076
  }
},
"version": 1,
"timestamp": 1514967076
}
```

If there is a `desired` part in the device shadow document, the device shadow service will automatically generate the corresponding `delta` part; otherwise, no content will be present in the `desired` and `delta` parts.

Note:

The device shadow service **does not store** the `delta` messages.

Updating Shadow by Device

The device tells the device shadow server its current status by sending an UPDATE message to the `$shadow/operation/${productId}/${deviceName}` topic. The SDK provides a corresponding API to send UPDATE messages, and the business layer only needs to specify the content of the `reported` field. The message content is in a specific JSON string format.

The air conditioner device sends an UPDATE message to

`$shadow/operation/ABC1234567/AirConditioner` to report its current device status. Below is the sample code:

```
{
  "type": "update",
  "state": {
    "reported": {
      "temperature": 27
    }
  },
  "version": 1,
  "clientToken": "clientToken"
}
```

When the device shadow server receives this message, it first determines whether the version in the message matches the version stored on it, and if so, it will perform the device shadow update process.

The shadow server responds to the air conditioner device with a message. Below is the sample code:

```
{
  "type": "update",
  "result": 0,
  "timestamp": 1514967066,
  "clientToken": "clientToken",
  "payload": {
    "state": {
      "reported": {
        "temperature": 27
      }
    }
  },
  "metadata": {
    "reported": {
      "temperature": {
        "timestamp": 1514967066
      }
    }
  },
  "version": 2,
  "timestamp": 1514967066
}
```

If the version in the message does not match the version stored on the device shadow server, the device shadow service will respond by sending the following message to

```
$shadow/operation/result/ABC1234567/AirConditioner :
```

```
{
  "type": "update",
  "result": 5005,
  "timestamp": 1514967066,
  "clientToken": "clientToken",
  "payload": {
    "state": {
      "reported": {
        "temperature": 27,
        "mode": "cool"
      }
    }
  },
  "metadata": {
    "reported": {
      "temperature": {
        "timestamp": 1514967066
      },
      "mode": {
```

```
    "timestamp": 1514967050
  }
},
"version": 2,
"timestamp": 1514967066
}
```

At this point, the full content of the device shadow document will be returned in the payload.

Updating Shadow by Application

The application modifies the device shadow's `desired` field through the HTTP RESTful API.

The application modifies the air conditioner's operating parameters through the HTTP RESTful API. Below is the sample code:

```
{
  "type": "update",
  "state": {
    "desired": {
      "temperature": 25
    }
  },
  "version": 2,
  "clientToken": "clientToken"
}
```

When the device shadow server receives this message, it first determines whether the version in the message matches the version stored on it, and if so, it will perform the device shadow update process and respond to the application with a JSON message through the HTTP RESTful API.

```
{
  "type": "update",
  "result": 0,
  "timestamp": 1514967076,
  "clientToken": "clientToken",
  "payload": {
    "state": {
      "desired": {
        "temperature": 25
      }
    }
  },
  "metadata": {
```

```
"desired": {
  "temperature": {
    "timestamp": 1514967076
  }
},
"version": 3,
"timestamp": 1514967076
}
```

In addition, the shadow server sends a `delta` message to

```
$shadow/operation/result/ABC1234567/AirConditioner .
```

```
{
  "type": "delta",
  "timestamp": 1514967076,
  "payload": {
    "state": {
      "temperature": 25
    },
    "metadata": {
      "temperature": {
        "timestamp": 1514967076
      }
    }
  },
  "version": 3,
  "timestamp": 1514967076
}
```

The SDK notifies the business layer that the message has been received through the corresponding callback function.

Responding to `delta` Message by Device

After the device receives the `delta` message, the business layer can empty the content of the `desired` field and send it to the device shadow server by sending the message to the

`$shadow/operation/${productId}/${deviceName}` topic, indicating that the device has **responded to** this `delta` message.

For example, after the air conditioner adjusts the temperature, it will send a message to

```
$shadow/operation/ABC1234567/AirConditioner :
```

```
{
  "type": "update",
  "state": {
    "desired": null
  },
  "version": 3,
  "clientToken": "clientToken"
}
```

The SDK provides a corresponding API to send the above message. When the device shadow server receives this message, it clears the content of the `desired` field to prevent repeated sending due to the differences between the parameter value in the `reported` field and that in the `desired` field.

After receiving the message, the shadow server sends a response message to

```
$shadow/operation/result/${productId}/${deviceName} .
```

For example, after receiving the `"desired":null` message from the air conditioner, the shadow server will send a device shadow update success message to

```
$shadow/operation/result/ABC1234567/AirConditioner .
```

```
{
  "type": "update",
  "result": 0,
  "timestamp": 1514967086,
  "clientToken": "clientToken",
  "payload": {
    "state": {
      "reported": {
        "temperature": 25
      },
      "desired": null
    },
    "metadata": {
      "reported": {
        "temperature": {
          "timestamp": 1514967086
        }
      },
      "desired": {
        "temperature": {
          "timestamp": 1514967086
        }
      }
    }
  },
}
```

```
"version": 4,  
"timestamp": 1514967086  
}  
}
```

If some fields reported by the device are null, the corresponding fields in the device shadow will be deleted. After the update succeeds, the fields in the returned payload will contain only the content related to the updated fields.

If the version value carried during the device update is below that stored on the server, the data on the device is old. At this time, the server will send a failure message, where the error code (the `result` field) will clearly tell the SDK that the update failed and the reason is that the version is too low. In addition, the server will also send the latest content in the payload to the device.

Device Firmware Upgrade

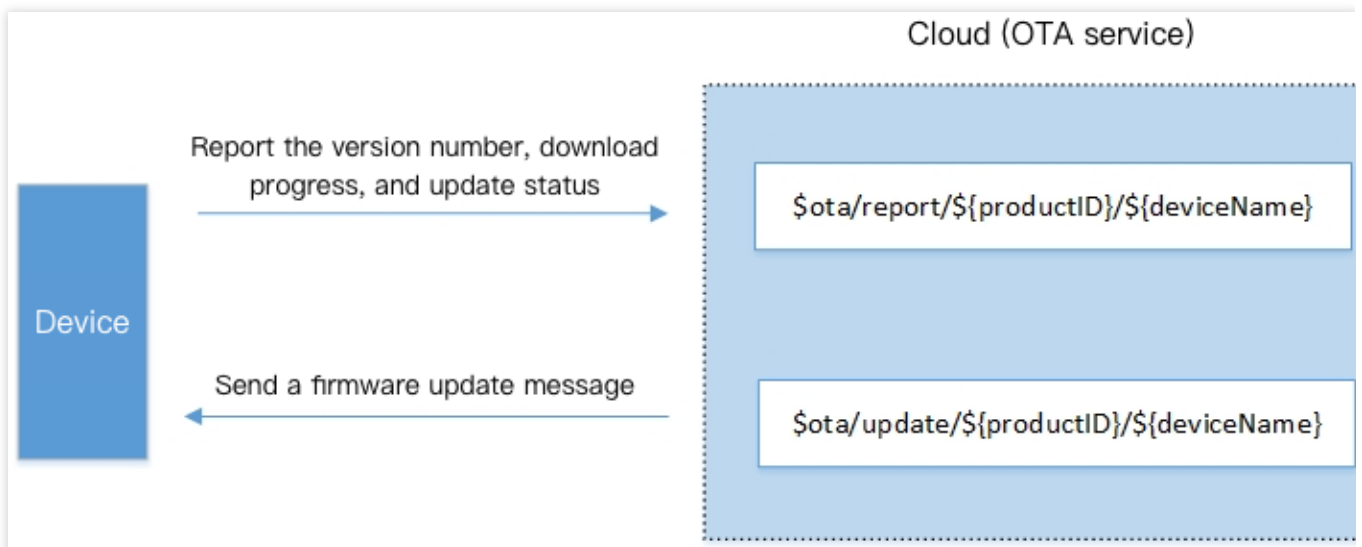
Last updated : 2024-12-27 15:54:13

Overview

Device firmware update is an important part of the IoT Hub service. When a device has new features available or vulnerabilities that need to be fixed, firmware update can be quickly performed for it through the device firmware update feature.

How It Works

During the firmware update process, the device needs to subscribe to the following two topics to communicate with the cloud:



Below is a sample:

```
$ota/report/${productID}/${deviceName}
```

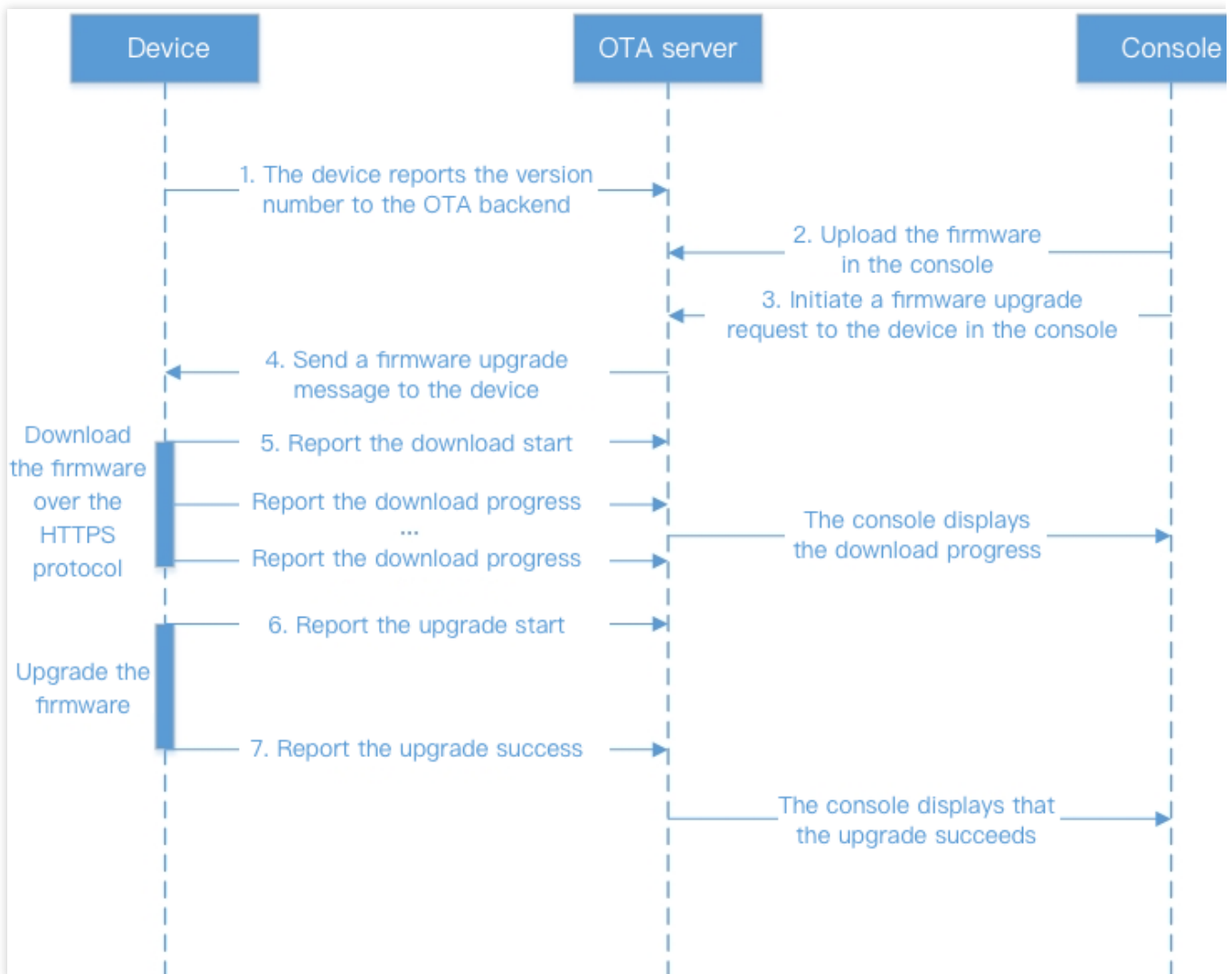
This topic is used to publish (upstream) messages, through which the device reports

```
$ota/update/${productID}/${deviceName}
```

This topic is used to subscribe to (downstream) messages, through which the device

Process

Taking MQTT as an example, the update process of the device is as follows:



1. The device reports its current version number. It publishes a message in JSON format with the following content to the `$ota/report/${productID}/${deviceName}` topic over the MQTT protocol to report its version number:

```

{
  "type": "report_version",
  "report": {
    "version": "0.1"
  }
}
// type: message type
// version: the reported version number
  
```

2. Log in to the [IoT Hub console](#), upload the firmware, and update the specified device to the specified version.

3. After the firmware update operation is triggered, the device will receive a firmware update message with the following content through the subscribed `$ota/update/${productID}/${deviceName}` topic:

```
{
  "file_size": 708482,
  "md5sum": "36eb5951179db14a63****a37a9322a2",
  "type": "update_firmware",
  "url": "https://ota-1255858890.cos.ap-guangzhou.myqcloud.com",
  "version": "0.2"
}
// type: the message type is `update_firmware`
// version: updated version
// url: URL of the downloaded firmware
// md5asum: MD5 value of the firmware
// file_size: firmware size in bytes
```

4. After receiving the firmware update message, the device will download the firmware from the URL. During the download process, the device SDK keeps reporting the download progress with the following content through the

`$ota/report/${productID}/${deviceName}` topic:

```
{
  "type": "report_progress",
  "report":{
    "progress":{
      "state":"downloading",
      "percent":"10",
      "result_code":"0",
      "result_msg":""
    },
    "version": "0.2"
  }
}
// type: message type
// state: the status is "downloading"
// percent: the current download progress in percentages
```

5. After downloading the firmware, the device needs to report an update start message with the following content through the `$ota/report/${productID}/${deviceName}` topic:

```
{
  "type": "report_progress",
  "report":{
    "progress":{
      "state":"burning",
      "result_code":"0",
      "result_msg":""
    },
    "version": "0.2"
  }
}
```

```
}  
// type: message type  
// state: the status is "burning"
```

6. After the device firmware update is completed, the device will report an update success message with the following content to the `$ota/report/${productID}/${deviceName}` topic:

```
{  
  "type": "report_progress",  
  "report": {  
    "progress": {  
      "state": "done",  
      "result_code": "0",  
      "result_msg": ""  
    },  
    "version": "0.2"  
  }  
}  
// type: message type  
// state: the status is "completed"
```

In the process of downloading or updating the firmware, if a failure occurs, an update failure message with the following content will be reported through the `$ota/report/${productID}/${deviceName}` topic:

```
{  
  "type": "report_progress",  
  "report": {  
    "progress": {  
      "state": "fail",  
      "result_code": "-1",  
      "result_msg": "time_out"  
    },  
    "version": "0.2"  
  }  
}  
// state: the status is "failed"  
// result_code: error code. -1: download timed out; -2: the file does not exist; -3  
// result_msg: error message
```

Checkpoint Restart of OTA

IoT devices sometimes may be in weak network environments. In this case, the connection may be unstable, firmware download may be interrupted, and if the firmware is downloaded from offset 0 every time, it may never complete.

Therefore, the checkpoint restart feature of firmware download is particularly necessary as detailed below:

Checkpoint restart refers to resuming file download or upload from where interrupted. To implement this feature, the device needs to record the checkpoint where firmware download is interrupted as well as the MD5, file size, and version information of the firmware.

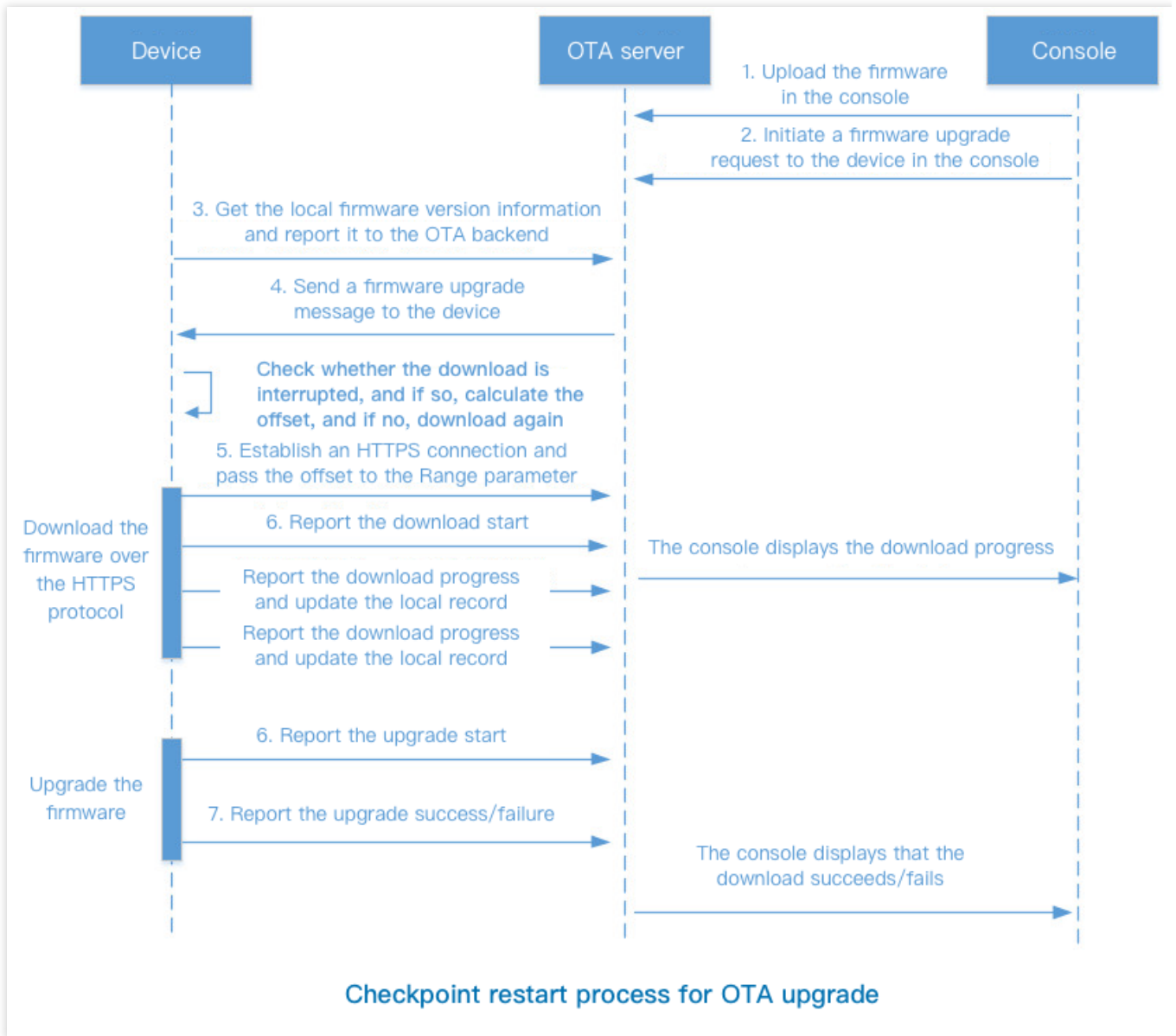
In case of OTA interruption, the device will report its version number to the IoT Hub platform, and if the reported version number is different from the target version number to be updated to, the platform will distribute a firmware update message again, and the device will get the target firmware information and compare it with the locally recorded interrupted firmware information. After determining that they are the same firmware, the device will restart download from the checkpoint.

The checkpoint restart process for OTA update is as follows:

Note:

Steps 3–6 may be performed multiple times in a weak network environment, and step 7 will be performed only after they succeed.

After step 3 is performed, the device will receive the message that step 4 needs to be performed.



Remote Device Configuration

Last updated : 2024-12-27 15:54:13

Feature Overview

In device use cases, if system parameters need to be updated for devices, such as device IP, port number, and serial port parameter, the remote configuration feature can be used to this end.

Feature Details

Remote device configuration supports two configuration update methods: active distribution by IoT Hub and active request by device. For scenarios where all devices under the same product need to update their configurations, the former can be used to distribute the configuration information to all devices through the remote configuration topic. For scenarios where certain devices need to update their configurations, the latter can be used.

Remote configuration request topic: `$config/operation/${productid}/${devicename}`

Remote configuration subscription topic: `$config/operation/result/${productid}/${devicename}`

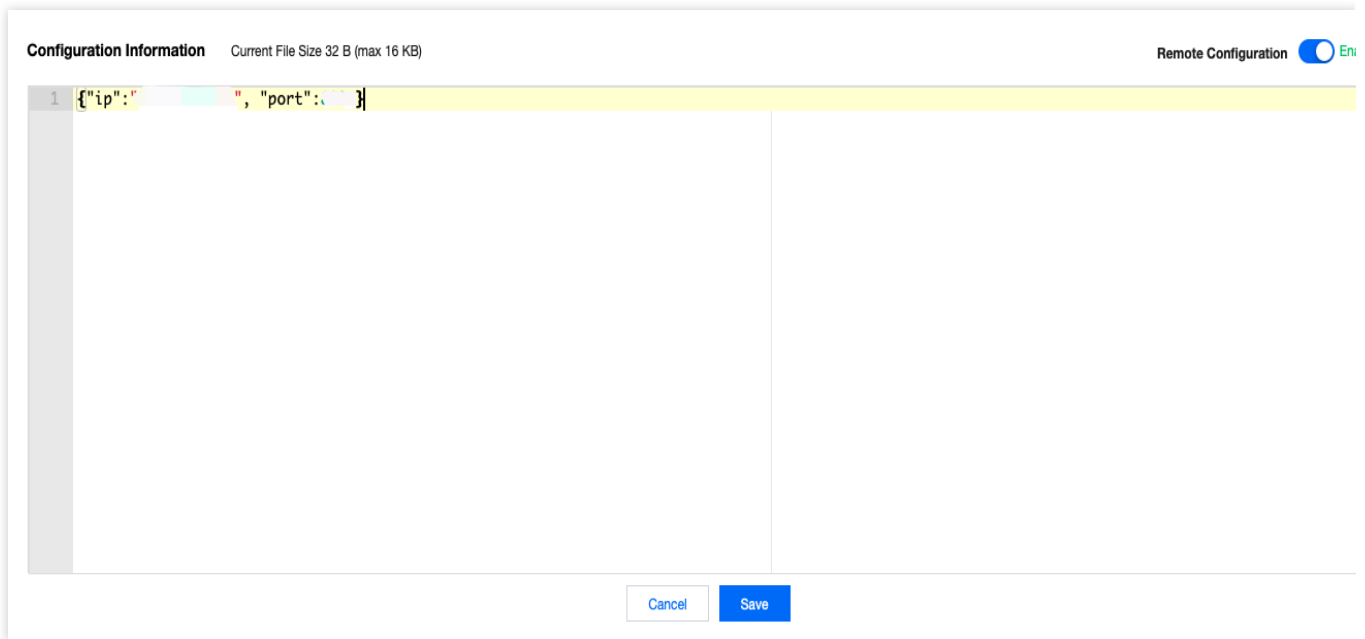
Note:

`${productID}` : product ID

`${deviceName}` : device name

Active distribution by IoT Hub

1. The device subscribes to the remote configuration topic.
2. On the configuration page in the [IoT Hub console](#), enable remote configuration and enter the configuration information in JSON format.



3. Click **Batch Distribute** to distribute the configuration information to all devices under the product in batches through the remote configuration subscription topic.

4. The format of the message distributed by the cloud through the remote configuration subscription topic is as follows:

```
{\"type\": \"push\",
  \"result\": 0,
  \"payload\": {yourConfigurationMessage}
}
```

Parameter description:

Parameter	Type	Description
type	string	The value is `push` for active distribution by IoT Hub. push: active distribution by IoT Hub reply: active request by device
result	int	Error code. 0: success 1001: the configuration is disabled
payload	string	Configuration information details

After the device successfully receives the configuration information distributed by IoT Hub, it will get the configuration information by calling the callback function provided in the SDK and update the information into its system parameters. The logic of updating configuration parameters in this part should be customized by yourself.

Active request by device

1. On the configuration page in the console, enable remote configuration and enter the configuration information in JSON format.
2. The device subscribes to the remote configuration topic and sends a remote configuration request through the topic.
3. After the cloud successfully receives the device's request for remote configuration information, it will send the device configuration information on the configuration page to the device through the remote configuration subscription topic.

The content of the configuration request message sent by the device is fixed as follows:

```
{"type": "get" }
```

Parameter description:

Parameter	Type	Description
type	string	The value is `get` for active request by device

The format of the message distributed by the cloud through the remote configuration subscription topic is as follows:

```
{"type": "reply",
  "result": 0,
  "payload": {yourConfigurationMessage}
}
```

Parameter description:

Parameter	Type	Description
type	string	The value is `reply` for active request by device. push: active distribution by IoT Hub reply: active request by device
result	int	Error code. 0: success 1001: the configuration is disabled
payload	string	Configuration information details

4. The steps after the device receives the data are the same as those of active distribution by the cloud.

Configuration information management

IoT Hub provides the configuration information management feature, and you can query the last five configuration information records in the console. After you edit and save the configuration information again, the last configuration information will be displayed in the configuration information record. You can view the number, update time, and configuration content for easy management.

Configuration Records

No.

Update Time

Operation

No data yet

Resource Management

Last updated : 2024-12-27 15:54:13

Feature Overview

The resource management feature is mainly used to transfer resources between devices and the platform. The following two topics are required for this feature:

Data upstream topic (for publishing): `$resource/up/service/${productid}/${devicename}`

Data downstream topic (for subscribing): `$resource/down/service/${productid}/${devicename}`

Device Resource Upload

Step 1. Create a resource upload task on the device

1. The device sends a message in JSON format with the following content to

`$resource/up/service/${productid}/${devicename}` to create a device resource upload task:

```
{
  "type": "create_upload_task",
  "size": 100,
  "name": "zxc",
  "md5sum": "*****",
}
```

2. After successful creation, the backend returns the resource upload URL through

`$resource/down/service/${productid}/${devicename}` with a message in JSON format with the following content:

```
{
  "type": "create_upload_task_rsp",
  "size": 100,
  "name": "zxc",
  "md5sum": "*****",
  "url": "https://iothub.cos.ap-guangzhou.myqcloud.com/*****"
}
```

Step 2. Report the resource upload progress

1. Resource upload uses HTTP PUT requests, so the Base64-encoded MD5 value needs to be added to the header. During the resource upload process, the device reports the resource upload progress through

`resource/up/service/{productid}/{devicename}` with a message in JSON format with the following content:

```
{
  "type": "report_upload_progress",
  "name": "zxc",
  "progress": {
    "state": "uploading",
    "percent": 89,
    "result_code": 0,
    "result_msg": ""
  }
}
```

2. The response to progress reporting is sent to the device through

`resource/down/service/{productid}/{devicename}` with a message in JSON format with the following content:

```
{
  "type": "report_upload_progress_rsp",
  "result_code": 0,
  "result_msg": "ok"
}
```

Platform Resource Delivery

Step 1. Query the resource download URL

1. The device sends a message in JSON format with the following content through

`resource/up/service/{productid}/{devicename}` to query the download task:

```
{
  "type": "get_download_task"
}
```

2. If there is a download task, the result will be delivered through

`resource/down/service/{productid}/{devicename}` with a message in JSON format with the following content:

```
{
  "type": "get_download_task_rsp",
  "size": 372338,
  "name": "AAAA",
  "md5sum": "a567907174*****3bb9a2bb20716fd97",
}
```

```
"url":"https://iothub.cos.ap-guangzhou.myqcloud.com/*****"  
}
```

Step 2. Report the resource download progress

1. The resource download progress is reported through

`$resource/up/service/${productid}/${devicename}` with a message in JSON format with the following content:

```
{  
  "type":"report_download_progress",  
  "name":"zxc",  
  "progress":{  
    "state":"downloading",  
    "percent":89,  
    "result_code":0,  
    "result_msg":""  
  }  
}
```

2. The response to progress reporting is sent to the device through

`$resource/down/service/${productid}/${devicename}` with a message in JSON format with the following content:

```
{  
  "type":"report_download_progress_rsp",  
  "result_code":0,  
  "result_msg":"ok"  
}
```

Device Log Reporting

Last updated : 2024-12-27 15:54:13

Feature Overview

The device log feature is mainly used for the platform to remotely view the device operation logs. The platform can ask a device to report logs by sending a message. Log levels include ERROR, WARN, INFO, and DEBUG. The following two topics are needed for this feature:

Data upstream topic (for publishing): `$log/operation/${productid}/${devicename}`

Data downstream topic (for subscribing): `$log/operation/result/${productid}/${devicename}`

Querying Log Level

1. The device sends a message in JSON format with the following content to the

`$log/operation/${productid}/${devicename}` topic to query whether it should upload logs and the required log level:

```
{
  "type": "get_log_level",
  "clientToken": "PPXLSKBUPZ-**"
}
```

2. The device actively queries whether it needs to report logs, or the platform remotely asks the device to enable log reporting. Specifically, the backend sends a message in JSON format with the following content to require log reporting and indicate the log level:

```
{
  "type": "get_log_level",
  "clientToken": "PPXLSKBUPZ-**",
  "log_level": 4,
  "result": 0,
  "timestamp": 1619599073
}
//log_level: 0: do not report logs; 1: ERROR; 2: WARN; 3: INFO; 4: DEBUG
```

Log Upload

Parameter description

When a device uploads logs, it needs to carry `ProductId` and `DeviceName` to initiate an `http/https` request to the platform. The request API and parameters are as detailed below:

Requested URL:

`http://ap-guangzhou.gateway.tencentdevices.com/device/reportlog` `https://ap-guangzhou.gateway.tencentdevices.com/device/reportlog`

Request method: POST

Request parameters

Parameter	Required	Type	Description
ProductId	Yes	String	Product ID
DeviceName	Yes	String	Device name
Message	Yes	Array	Reported log content, which is a string array. The log level needs to be added before each log entry. Currently, DBG, INF, ERR, and WRN are supported

Note:

The API only supports the `application/json` format.

Signature generation

There are two types of signatures for request messages. Key authentication uses the HMACSHA256 algorithm, and certificate authentication uses the RSA_SHA256 algorithm. For more information, please see [Signature Algorithm](#).

Platform response parameters

Parameter	Type	Description
RequestId	String	Request ID

Sample Code

Request packet

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/reportlog
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
```

```
X-TC-Timestamp: 1551****65
X-TC-Nonce: 5456
X-TC-Signature:
2230eefd229f582d8b1b891af7107b91597240707d7****3738f756258d7652c
{"DeviceName":"AAAAAA","Message":["INFMqtt connect
success."],"ProductId":"G8N9****HB"}
```

Response packet

```
{
  "Response": {
    "RequestId": "f4da4f1f-d72e-40f1-****-349fc0072ba0"
  }
}
```

NTP Service

Last updated : 2024-12-27 15:54:13

Feature Overview

The NTP feature is mainly used to solve the problem with resource-constrained devices where they don't have the NTP service and thus have no accurate timestamps. The following two topics are required for this feature:

Request topic (for publishing): `$sys/operation/${ProductId}/${DeviceName}` .

Response topic (for subscribing): `$sys/operation/result/${ProductId}/${DeviceName}` .

How It Works

The IoT Hub platform draws on the principles of the NTP protocol and uses the platform itself as an NTP server. After a device requests the platform, the platform will return the NTP time. After the device receives the response, it will calculate the current accurate time based on the request time and receipt time.

Directions

1. The device sends a message in JSON format with the following content to

`$sys/operation/${ProductId}/${DeviceName}` to request the platform for the NTP time and records the request time `deviceSendtime` :

```
{
  "type": "get",
  "resource": [
    "time"
  ]
}
```

2. The platform returns the NTP time through `$sys/operation/result/${ProductId}/${DeviceName}` with a message in JSON format with the following content, and the device records the receipt time

`deviceRecvtime` :

```
{
  "type": "get",
  "time": 1621562342,
  "ntptime1": 1621562342773,
  "ntptime2": 1621562342773
}
```


3. The accurate time is calculated through the NTP time ($\text{\${ntptime1}} + \text{\${ntptime2}}$) received by the device, receipt time ($\text{\${deviceRecvtime}}$), and request time ($\text{\${deviceSendtime}}$) as follows:

$$\text{Accurate time} = (\text{\${ntptime1}} + \text{\${ntptime2}} + \text{\${deviceRecvtime}} - \text{\${deviceSendtime}}) / 2$$