



Nabto Push

NABTO/001/TEN/050

1 Contents

2	A	Abstract	2
3	E	Bibliography	2
4	ſ	Nabto Platform Basics	3
	4.1	Nabto Communication Patterns	4
5	ι	Using Nabto Push Services	4
	5.1	Service Configuration	5
	5.2	Performance notes	6
6	I	Integrating Nabto Push in the uNabto Device	7
	6.1	Push using the push_service module	8
	6.2	Push without the push_service module1	0

2 Abstract

This document describes how to implement solution that use the Nabto Push communication pattern to send information from a uNabto device to an external service (such as a mobile push notification service or a Big Data analysis solution) on initiative of the uNabto device using the secure channel already established between a uNabto device and the Nabto basestation.

This basically enables embedded devices to send push notifications or communicate with a Big Data solution securely without the hassle of embedding an HTTPS client or e.g. using MQTT + TLS.

3 Bibliography

TEN023	NABTO/001/TEN/023: uNabto SDK - Writing a uNabto device application
TEN025	NABTO/001/TEN/025: Writing a Nabto API client application
TEN036	NABTO/001/TEN/036: Security in Nabto Solutions

nabto

4 Nabto Platform Basics



The Nabto platform consists of 3 components:

- Nabto client: Libraries supplied by Nabto, used by the customer's application
- Nabto **device**: The uNabto SDK an open source framework supplied by Nabto, integrated with the customer's device application
- Nabto **basestation**: Services supplied by Nabto (Nabto- or self-hosted) that mediates connections between Nabto clients and devices.

The Nabto client initiates a direct, encrypted connection to the Nabto enabled device – the Nabto basestation mediates this direct connection: The device's unique name, e.g. <serial>.vendor.net, is mapped to the IP address of the Nabto basestation – this is where devices register when online and where clients look for available devices. After connection establishment, the client and device communicates directly with each other, the basestation is out of the loop – no data is stored on the basestation, it only knows about currently available Nabto enabled devices.

The client can also discover the device if located on the same LAN and communicate directly without the basestation – useful for bootstrap scenarios or for offline use.

Integrating Nabto on the customer's device is the topic of [TEN023].

Nabto client applications are developed using the Nabto Client SDK described in [TEN025]. The Nabto Client SDK is the lowest level way of developing a Nabto application - several wrappers exist on top of this lowest level SDK to provide a more abstract experience, for instance for developing Cordova/Ionic or Xamarin hybrid apps or just simplify native Android and iOS app development.

4.1 Nabto Communication Patterns

The Nabto platform supports 3 communication patterns that will be referenced throughout this document:

RPC: The Nabto RPC communication mechanism allows a client to securely invoke a remote function on a Nabto device. The device implements an interface definition shared between client and device, the client works with normal JSON documents, exchanged in a compact representation with the device.

Streaming: Nabto Streaming can be used for retrieving larger amounts of data from a device or sending e.g. a firmware update. With sufficient resources available on the device, Nabto Streaming can be used for high performance streaming suitable for video scenarios.

Push: Nabto Push is used for communication initiated by the device, for instance to implement mobile push notifications or to support big data scenarios where data is collected centrally for further analysis. Nabto Push can also trigger an M2M scenario using RPC or Streaming - e.g. when a certain condition is triggered, the device sends a Nabto Push message and a server function invokes an RPC function or streams data.

5 Using Nabto Push Services

The Nabto Push communication pattern allows Nabto enabled devices to send information on their own initiative to services configured by the vendor (denoted *endpoints*). Currently two different types of endpoints exist: Firebase Cloud Messaging (for mobile push to iOS, Android and web apps) and Generic HTTP (denoted webhook - for vendor specific services):



Native support for additional push service providers (such as Urban Airship) will be added in the future.

The webhook endpoints can be used for sending data to a data warehouse / big data analysis solution. Or it can be used for triggering control functionality: The IoT device may trigger a notification as a threshold condition is met, the configured webhook then uses Nabto RPC to invoke the IoT device based on a central decision, e.g. involving other services or a rule engine:



The IoT device likely already exposes a Nabto RPC interface used with regular client interaction. Also, Nabto Streaming can be used from the webhook to send or receive larger amounts of data in response to a push notification.

The webhook endpoints can also be used for adding currently unsupported mobile Push service providers through a simple adapter.

The push messages from device to basestation are sent on the secure channel established when the device registers with the basestation as outlined in [TEN036]. The shared secret based approach makes secure push notifications feasible on even very resource constrained devices.

5.1 Service Configuration

The actual endpoints are configured in the Nabto Cloud Console at https://console.cloud.nabto.com:

Console					
Home Domains	Firebase Setting Configure firebase here, fireb devices through Firebase Clo firebase console Firebase Server Key	S base is used for sending push no bud Messaging. The firebase ser	tifications to iOS and Android ver key can be found on the		
	Push Notification Webhooks (BETA) Configure a webhook for push notifications. The configured webhook will be invoked when a push notification webhook is sent.				
	Webhook URL	Webhook User	Webhook Password		
	Connect Webhooks (BETA) Configure a webhook for connects, each time a client or be invoked with the client id and the device id, the answ decides if a connect is allowed to the specific device.				
	Connect Webhook URL	Connect webhook user	Connect webhook password		
	Allow Self Signed	d Certificates			
	If this option is checked self signed certificates is allowed. S used in conjunction with fingerprint based authentication. If this option probably has to be checked.				
	Allow Self Signed Certification	ites			
	Domain Master S	Secret			

To use Firebase Cloud Messaging for sending push notifications, a Firebase Server Key must be specified, obtained through the Firebase Console: Click the settings icon for your project > Project Settings > Cloud Messaging > Server key.

For generic webhooks, a URL and an optional HTTP basic auth username and password can be specified. The push data is then send as HTTP post data payload exactly as transmitted from the uNabto device.

5.2 Performance notes

It is important to note that the performance of push notifications to phones will depend on the Firebase service, and for Apple devices also on the Apple push service. Neither Google nor Apple makes any promises on delivery times or even that notifications are ever delivered. This means that though push notifications for phones can be useful to alert users of critical system events, it should not be the only means of conveying critical information to users.

For example, the Apple developer documentation states that if multiple notifications are sent simultaneously only the latest notification is forwarded to the phone, the rest is simply discarded.

(https://developer.apple.com/library/content/technotes/tn2265/_index.html#//apple_ref/doc/uid/DTS40010376 -CH1-TNTAG23)

Using webhooks, the basestation will send the notification to an endpoint configured by the customer, assuming the provided endpoint is reachable by the basestation.

It is important to note that a positive status indication on the uNabto side as elaborated below means the basestation has successfully received the notification, *not* that the notification has successfully been delivered to the endpoint. This means that if the basestation is not able to reach the endpoint, the notification delivery will currently fail silently to the uNabto device and with no ability to trace this from a central log. This will be improved in a near future release.

6 Integrating Nabto Push in the uNabto Device

Push notification support is provided at two levels in the uNabto SDK: The core uNabto SDK provides the low level capabilities to send push notifications to the basestation and the higher level push_service module provides an easy to use "fire and forget" API that maintains necessary state during communication with the basestation at the cost of a bit higher resource requirements.

Push notifications in uNabto are configured using the following definitions in unabto_config.h:

- NABTO_ENABLE_PUSH enables push notifications, default is 0 (off)
- NABTO_PUSH_QUEUE_LENGTH determines the amount of push notifications uNabto can store at a given time, default is 10. Each entry requires approximately 20 bytes of memory. This is also used to determine buffer length in the push_service module.
- NABTO_PUSH_BUFFER_ELEMENT_SIZE Determines the maximum size of a push notification to be stored by the push_service module. If this module is used, it will use in the excess of NABTO_PUSH_QUEUE_LENGTH * NABTO_PUSH_BUFFER_ELEMENT_SIZE bytes of memory to store notifications.

6.1 Push using the push_service module

The easiest way to enable a uNabto device to send push notifications is to use the push_service module in uNabto. This module handles the proper construction of payloads for the push notification packet sent to the basestation, as well as storing notifications while waiting for a resolution of the notification.

Using this module, it is as simple as follows to send a push notification from a device:

```
void callback_function(void* context, const unabto_push_hint* hint) {
    printf("got notification callback with status indication %d", (int)hint);
}
push_message mp;
init_push_message(&pm, UNABTO_PUSH_PNS_ID_FIREBASE, "\"to\": \"<Firebase_Token>\"");
add_title(&pm, "High temperature");
add_body(&pm, "Temperature is 42 degrees");
send_push_message(&pm, &callback_function, null);
```

First a variable of type push message is defined and initialized with the function init_push_message:

init_push_message(push_message* msg, uint16_t pnsid, const char* staticData)

The pnsid parameter must currently be either UNABTO_PUSH_PNS_ID_FIREBASE or UNABTO_PUSH_PNS_ID_WEBHOOK.

The staticData parameter must be a zero-terminated JSON formatted string which will be used directly in the JSON document forwarded by the basestation. For webhooks, this document can be anything, however, for Firebase it must follow the Firebase HTTP Server protocol (<u>https://firebase.google.com/docs/cloud-messaging/http-server-ref</u>).

Using the Firebase HTTP Server protocol, the staticData document must contain a "to" field defining which mobile device (or other supported app) the notification should be sent to, any other parameters of this protocol can also be included.

Since the staticData "to" field contains the Firebase token for the receiving phone, it is recommended that this string is built on the client side, and then transferred to the uNabto device, and that all event specific data is added using the helper functions described below.

Once the message is Initialized, additional fields can be added using the following helper functions:

- add_title() adds a title field to the notification
- add_body() adds a body field to the notification
- add_title_loc_key() adds a title localization key to the notification
- add_title_loc_string_arg() adds a title localization string argument to the notification
- add_body_loc_key() adds a body localization key to the notification

• add_body_loc_string_arg() - adds a body localization string argument to the notification

add_title(), add_body(), add_title_loc_key() and add_body_loc_key() should maximally be called once. The string_arg functions to add string arguments to the notifications can be added multiple times.

Once the notification is constructed, it can be sent using the <code>send_push_message()</code> function, which in addition to the constructed message takes a callback function and a pointer to some user defined context to be added to the callback.

The best practice is to just reference templates that are bundled in the client app instead of hardcoding strings in the device application. Parameters are then supplied to the template - for greater flexibility, simpler maintenance and to support localization. To do that, use the "loc" functions:

```
push_message mp;
init_push_message(&pm, UNABTO_PUSH_PNS_ID_FIREBASE, "\"to\": \"<Firebase_Token>\"");
add_title_loc_key(&pm, "title_high_temperature");
add_body_loc_key(&pm, "body_temperature_message");
add_body_loc_string_arg(&pm, "42");
send_push_message(&pm, &callback_function, null);
```

In the latter example, the construction of the string shown to the user is postponed to the individual target client to allow its localization framework to look up the localized string associated with the given key and patches it with the data.

When the message is sent using send_push_message(), the push_service module will then interact with the uNabto core to transmit the notification to the basestation. Once the basestation has successfully received the notification, UNABTO_PUSH_HINT_OK is returned by the core to the push_service module, which will forward this status to the application using the previously provided callback function.

It is important to note that a return status of OK means the basestation has successfully received the notification, *not* that the notification has successfully been delivered to the endpoint. This means that if the basestation is not able to reach the endpoint, the notification delivery will currently fail silently to the uNabto device and with no ability to trace this from a central log. This will be improved in a near future release.

Any other returned status means the notification was not sent to the basestation, and the notification will not arrive at its destination. A more detailed description of return statuses is provided in the following section.

6.2 Push without the push_service module

The push_service module described above reserves enough memory to store entire push notifications. As many push notifications may reuse most of the data in a notification, this may be inefficient. Hence, for memory critical systems, it may be beneficial to handle the memory allocation in the application and not use the push_service module and instead use the uNabto core functionalities directly.

Push notifications in the uNabto core works by first calling the unabto_send_push_notification function with the PNS ID and the address of an integer where the core can assign a sequence number to the notification. This function will return immediately, and if the return status was UNABTO_PUSH_HINT_OK, the core will start preparing to send the notification asynchronously. Once the core is ready to send the notification, it will request the data to be put into the notification by calling the function:

```
uint8_t* unabto_push_notification_get_data(uint8_t* bufStart, const uint8_t* bufEnd, uint32_t
seq)
```

This function should be implemented in the application. bufStart is a pointer to where the data should be copied, bufEnd is a pointer to the last address available for the data, and seq is the sequence number of the notification for which the data is requested as returned by unabto_send_push_notification().

The function must return a pointer to the last data element copied. The data provided with the unabto_push_notification_get_data function must be in nabto payloads of type NP PAYLOAD TYPE PUSH DATA. This payload is defined as:

The purpose can be either NP_PAYLOAD_PUSH_DATA_PURPOSE_STATIC or

NP_PAYLOAD_PUSH_DATA_PURPOSE_DYNAMIC. A push notification should not contain more than 1 payload of each purpose. This means the notification can contain up to 2 payloads. Each payload type can use 2 different encodings, NP_PAYLOAD_PUSH_DATA_ENCODING_JSON and NP_PAYLOAD_PUSH_DATA_ENCODING_TLV.

JSON encoding means the data field contains a JSON formatted string, and TLV encoding means the data follows a TLV format described in the following section. The TLV encoding can contain multiple TLV values in sequence.

Using Firebase to send push notifications requires a payload with purpose NP_PAYLOAD_PUSH_DATA_PURPOSE_STATIC which follows the same restrictions as described in the previous section. Once the uNabto core has received the notification data, it will start encrypting the data, and send the notification to the basestation. If any of the notification preparation steps fails, or when the transmission to the basestation has been resolved, the uNabto core will call the function:

void unabto_push_notification_callback(uint32_t seq, unabto_push_hint* hint)

This function must also be implemented in the application. The hint reference will reflect if notification preparation step failed or the notification transmission status using the following status codes:

UNABTO_PUSH_HINT_OK UNABTO_PUSH_HINT_QUEUE_FULL UNABTO_PUSH_HINT_INVALID_DATA_PROVIDED UNABTO_PUSH_HINT_NO_CRYPTO_CONTEXT UNABTO_PUSH_HINT_ENCRYPTION_FAILED UNABTO_PUSH_HINT_FAILED UNABTO_PUSH_HINT_QUOTA_EXCEEDED UNABTO_PUSH_HINT_QUOTA_EXCEEDED

OK means the notification was successfully received by the basestation. As described in the previous section, this is not an acknowledgement that it was send to the final endpoint.

QUEUE_FULL means the unabto_send_push_notification was not able to enqueue another notification as NABTO_PUSH_QUEUE_LENGTH notifications is already in the queue.

INVALID_DATA_PROVIDED means that the data provided in the unabto_push_get_data call did not fit properly within the provided buffer.

NO_CRYPTO_CONTEXT means the core was not able to encrypt the notification as the unabto_main_context did not contain a crypto context.

ENCRYPTION_FAILED is reserved for encryption errors but currently not used.

QUOTA_EXCEEDED means the basestation returned this status, this results in the uNabto core refusing to transmit new notifications for the next UNABTO_PUSH_QUOTA_EXCEEDED_BACKOFF milliseconds which defaults to 1 minute.

After this backoff period, notifications will be sent normally. This is meant as a way to guard the basestation against faulty devices, but is currently not implemented in the basestation.

Similarly, UNABTO_PUSH_HINT_QUOTA_EXCEEDED_REATTACH means the uNabto core will refuse to send notifications until the device has reattached. This is meant as a way to guard the basestation against extremely faulty devices, but has also yet to be implemented in the basestation.

In conclusion, to use the uNabto core push functionality, the application must implement the functions:

```
uint8_t* unabto_push_notification_get_data(uint8_t* bufStart, const uint8_t* bufEnd, uint32_t
seq)
```

void unabto_push_notification_callback(uint32_t seq, unabto_push_hint* hint)

Notifications are then sent by calling unabto_send_push_notification(). This will call the unabto_push_notification_get_data() function to retrieve the notification data. And once the notification has been resolved, the core will call unabto_push_notification_callback() function which provides a hint to the status of the notification. For examples on how to implement this, see the uNabto push_service module.

6.2.1 The TLV format

A TLV value is defined as:

++		+
+0	Туре	
+1	Length	+
+2	Data	+
+		+

Where the currently supported values for the Type field is:

NP PAYLOAD PUSH DATA VALUE TITLE

NP PAYLOAD PUSH DATA VALUE BODY

NP PAYLOAD PUSH DATA VALUE BODY LOC KEY

NP PAYLOAD PUSH DATA VALUE BODY LOC STRING ARG

NP PAYLOAD PUSH DATA VALUE TITLE LOC KEY

NP PAYLOAD PUSH DATA VALUE TITLE LOC STRING ARG

The Length field is the length of the Type field, the Length field, and the Data field combined, and the Data field is a string.