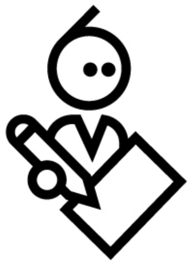
Nabto SDK

How to write a Nabto API client application

NABTO/001/TEN/025



Contents

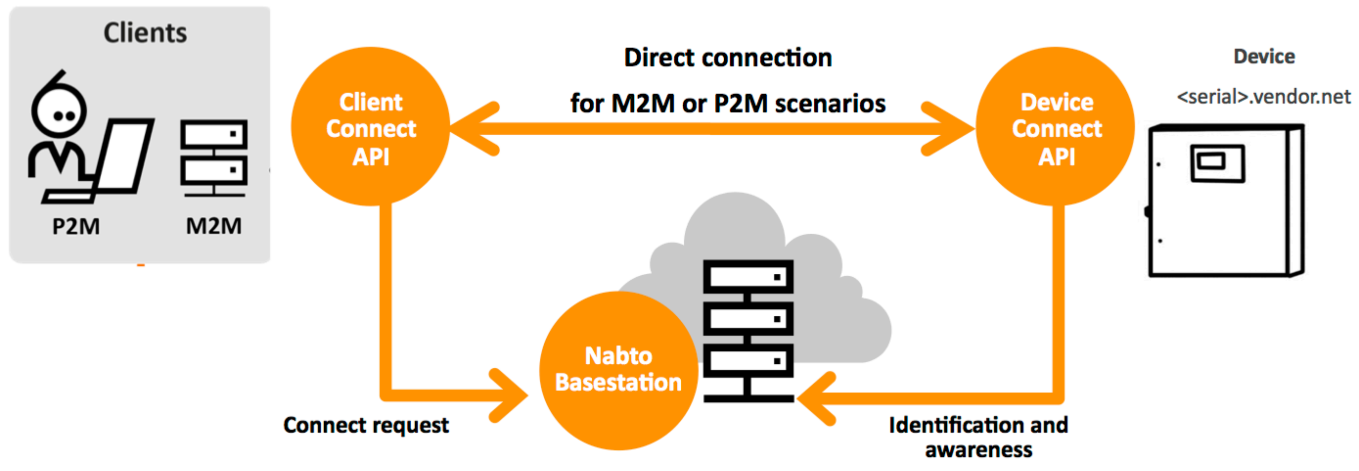
1 Abstract

This document describes how to write a native Nabto Client using the Nabto Client API library.

2 Bibliography

TEN023	NABTO/001/TEN/023: uNabto SDK - Writing a uNabto device application
TEN024	NABTO/001/TEN/024: uNabto SDK - Writing a uNabto HTML client application
TEN025	NABTO/001/TEN/025: uNabto SDK - Writing a Nabto API client application

4 Nabto Platform Basics



The Nabto platform consists of 3 components:

- Nabto **client**: Binaries supplied by Nabto, used by the customer's HTML or native 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. Also supplies the user interface to Nabto HTML clients.

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 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].

The customer's client application may use the Nabto client in different ways: The customer application can be an HTML application that uses the Nabto client to retrieve JSON data in a web application – in this scenario the Nabto client is typically a web browser plugin or mobile app, hosting the customer application. The latter is distributed from the basestation to the client and is denoted an HTML device driver bundle. Writing such an application is the topic of [TEN024].

The customer's client application can also be a native (non-HTML) application, linked with a Nabto client API library. The native client application can use the same request/response mechanism to invoke the device as HTML applications do. Additionally, the native client can establish streaming data connections with the device – this is a popular way of adding seamless, secure remote access capabilities to legacy client and device applications. Native client applications are the topic of this document.

5 The Nabto Client API

The Nabto platform provides several pre-built clients: Apps for smartphones and tablets, browser plugins, standalone desktop applications and a hosted HTTP-Nabto bridge for browsers without plugins. All of these clients are built around the same common library – the Nabto Client API Library. This library is freely available to enable custom applications built around the Nabto platform.

This document describes in detail how to write the different types of API library based applications and using tunneling.

The Nabto Client API is available as a basic C library with access to all functionality on the platform. Additionally, an object oriented .NET library is provided, wrapping the lower level API in the typical abstractions used on the .NET platform – e.g., it can replace traditional NetworkStream objects in applications upgrading from a proprietary client/server implementation to using Nabto.

The Nabto streaming data capabilities (e.g. video streaming) are only available through the Nabto Client API (not available for HTML clients).

Microsoft Windows (32/64-bit)	C library, .NET 4.0 abstraction
Mac OS X	C library, .NET 4.0 abstraction (requires Mono)
Linux (32/64-bit)	C library, .NET 4.0 abstraction (requires Mono)
Android 3.x and newer	C library with JNI wrapper
iOS 4.x and newer	C library
Windows Phone 8.0 and newer	No library support due to platform limitations

6 Nabto Client Scenarios

Each of the different Nabto usage scenarios (request/response JSON requests, tunneling, streaming) are described below. Some steps are common for all scenarios, described in *Starting a Nabto Client API Library Session*. The sections assume a Nabto Client user profile exists on the client – this can either be pre-installed, setup through an existing pre-built client or the profile can be prepared programmatically as described in *User Profile Management*.

6.1 Starting a Nabto Client API Library Session

When using the Nabto Client API Library, the library must first be initialized by invoking **nabtoStartup()**. Once everything is done and the library is about to be unloaded, **nabtoShutdown()** is to be invoked .

Once the library is initialized, a user session must be created. It provides the context in which all the actual use of the library takes place. For typical library scenarios, the **nabtoOpenSession(**email, password**)** variant is used: The credentials specified unlocks an existing private key in the "users" subdirectory of the home directory. See section "User Profile Management" for details on creating a private key. A special account "guest" with an empty password creates a guest session, if the special guest profile is available on the client platform (per default it is installed next to the client library, e.g. in `/usr/share/nabto/users` on Unix systems).

Once all requests are done in a session, **nabtoCloseSession()** is invoked to close the current session.

Successfully opening a session only means that the user's local private key could successfully be opened – it is encrypted with the password specified. The private key is associated with a certificate (a signed public key) that contains the user's email address. This certificate is later used for authentication when trying to communicate with a remote peer. Hence, it is still possible to get an "Access Denied" error, even with a fully valid session – if the remote peer has not granted the user in question access to the device or service requested.

The simple initialization sequence hence looks as follows:

```
nabto_status_t st = nabtoStartup(NULL);
// if st != NABTO_OK, fail
nabto_session_t session;
st = nabtoOpenSession(&session, ”user@example.org”, ”secret”);
// if st == NABTO_OK, use the session as described in next sections
nabtoCloseSession(session);
nabtoShutdown();
```

A session can also be created using **nabtoOpenSessionBare()**, without supplying credentials. This is useful in the special use case of creating a web browser component where user credentials are supplied through a form served as HTML content by the library.

6.2 uNabto Request/Response Communication

The services provided by a uNabto device are defined in the **unabto_queries.xml** file. It is a simple XML format describing the Nabto interface of the device (more details are provided in [TEN023]). As an example, consider the following query model file for a weather station device that supports a single query, **house_temperature.json**:

```
<?xml version="1.0"?>
<unabto_queries
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:noNamespaceSchemaLocation="http://www.nabto.com/unabto/query_model.xsd">
  <query name="house_temperature.json" id="1">
    <request>
      <parameter name="sensor_id" type="uint32"/>
    </request>
    <response format="json">
      <parameter name="temperature" type="int32"/>
    </response>
  </query>
</unabto_queries>
```

Two methods exist to retrieve the temperature reading from a device that supports this interface: A synchronous and an asynchronous approach. The latter is described in the next section. To synchronously retrieve the result, **nabtoFetchUrl()** is invoked – the response from the device is returned as a JSON object represented in a string, allocated by the library.

The full sequence looks as follows:

```
int main(int argc, char* argv[])
{
    nabto_status_t st = nabtoStartup(NULL);
    if (st != NABTO_OK) { /* handle error and stop */ }
    nabto_session_t session;
    st = nabtoOpenSession(&session, ”user@example.org”, ”secret”);
    if (st != NABTO_OK) { /* handle error and stop */ }
    char* resultBuffer;
    size_t resultSize;
    char* mimeType;
    const char* nabtoUrl = ”nabto://weatherdemo.nabto.net/house_temperature?sensor_id=3”;
    st = nabtoFetchUrl(session, nabtoUrl, &resultBuffer, &resultSize, &mimeType);
    if (st != NABTO_OK) { /* handle error and stop */ }
    // use data residing in resultBuffer
    nabtoFree(resultBuffer);
    nabtoFree(mimeType);
    nabtoCloseSession(session);
    nabtoShutdown();
}
```

The JSON string in **resultBuffer** has the following format:

```
{"request": {"sensor_id": 1}, "response": {"temperature": 1753} }
```

The client application has ownership of the memory used for resultBuffer and mimeType and must free it when no longer needed. The **nabtoFree()** function must be used for this.

Error handling may the current generation of the API[1]: Most errors are communicated through the JSON response, meaning that the application will have to parse the JSON response in order to figure out the true status of the request. The only errors communicated as return codes are related to invalid API usage.

An example error response looks as follows:

```
{ "error": {"event": 2000030, "header": "Parameter missing (2000030)","body":
""} }
```

The status code of **nabtoFetchUrl()** is still **NABTO_OK**, indicating the basic interaction with the API was succesfully: **nabtoStartup()** and **nabtoOpenSession()** was invoked successfully prior to the invocation of **nabtoFetchUrl()**, but something went wrong with the execution of the request (in this case, a parameter is missing).

6.3 Asynchronous Request/Response Communication

The requests defined in **unabto_queries.xml** can also be executed asynchronously to allow execution of lengthy requests in the background. Several different API functions are involved in asynchronous request execution, the flow is as follows:

1. Each asynchronous request is initialized with **nabtoAsyncInit()** that accepts the request string and returns a handle to use for subsequent invocations of the API.
2. The background retrieval is started with **nabtoAsyncFetch()**. This function accepts a user callback function (**NabtoAsyncStatusCallbackFunc**) invoked by the API when (some) data is ready or an error occurs.
3. The user callback **NabtoAsyncStatusCallbackFunc** function is invoked by the library and if the status indicates data is ready, the user callback function can invoke **nabtoGetAsyncData()** to retrieve the ready chunk of data.
4. After data is retrieved, **nabtoAsyncClose()** releases the asynchronous handle.

The following shows the overall structure – a full example is provided in Appendix A:

[1] the API was designed to be used in Nabto's HTML client applications (apps and browser plugins) with the end user application being implemented in HTML

```
void NABTOAPI callback(nabto_async_status_t status, void* arg, void* userData) {
    context_t* context = (context_t*)userData;
    char chunk[CHUNK_SIZE];
    size_t actualSize;
    do {
       nabtoGetAsyncData(context->resource_ , chunk, CHUNK_SIZE, &actualSize);
         memcpy(context->data_ + context->length_, chunk, actualSize);
         context->length_ += actualSize;
    } while (actualSize > 0);
}

int main(void) {
    nabto_status_t st = nabtoStartup(NULL);
    nabto_session_t session;
    st = nabtoOpenSession(&session, "user@example.org", "12345678");
    context_t context;
    const char* nabtoUrl =
       "nabto://weatherdemo.nabto.net/house_temperature?sensor_id=3";
    st = nabtoAsyncInit(session, &(context.ressource_), nabtoUrl);
    if (st != NABTO_OK) { /* handle error and stop */ }

    st = nabtoAsyncFetch(resource, &callback, &context);
    if (st != NABTO_OK) { /* handle error and stop */ }

    // do other stuff, data is retrieved in the background,
    // callback() is invoked when ready

    while (!context.done_) {
       sleep(1);
    }

    // use data retrieved, see nabtoFetchUrl() example

    nabtoAsyncClose(resource);
    nabtoCloseSession(session);
    nabtoShutdown();
}
```

A note on words: "Asynchronous" in this client context has nothing to do with an asynchronous uNabto application event handler on the device described in [TEN023]: A synchronous client request can invoke an asynchronous uNabto application implementation and vice versa.

6.4 TCP tunneling

The Nabto TCP tunneling API is a high level wrapper for the underlying uNabto streaming implementation allowing applications to tunnel traffic through Nabto by integrating through a simple TCP socket, just like e.g. SSH tunnels. The underlying streaming API is described in the next section and can be used for closer integration in applications.

To setup a tunnel, the API must be initialized as described in the previous sections – that is, **nabtoStartup**() must be invoked and a valid session must be opened with **nabtoOpenSession**().

When the session is established, a tunnel is established with **nabtoTunnelOpenTcp**(). 4 tunnel specific parameters must be specified:

- **localPort**: The local TCP port on which the API will listen – this is what the legacy application connects to.
- **nabtoHost**: The remote Nabto host to which a Nabto connection will be established and through which the TCP traffic will be tunneled
- **remoteHost**: The regular TCP host to which the remote tunnel endpoint will establish a connection (typically this is localhost – and many Nabto tunnel servers have a restriction to not allow "jumping" to remote TCP hosts).
- **remotePort**: The TCP port of target service to connect to.

The parameters are indicated in below overview:

```
          +---------+         +-------------+               +---------------+
          | nabto   |   nabto |   nabto     |   tcp/ip      |  remote       |
  |-------+ client +----------+   device    +------------|--+ TCP server    |
localPort | API     |         | "nabtoHost" |    remotePort | “remoteHost”  |
          +---------+         +-------------+               +---------------+
```

A sample scenario could be a web server running on a remote nabto host "`streamdemo.nabto.net`". The **localPort** is chosen to 8080. The **remoteHost** parameter is `localhost` as the web server is co-located with the Nabto tunnel endpoint, the **remotePort** is 80. Hence, after successfully invoking **nabtoTunnelOpenTcp**() with these parameters, applications may connect to TCP port 8080 on the machine running the Nabto Client API application.

If the return value of **nabtoTunnelOpenTcp**() indicates success, the opaque tunnel handle supplied to the function is initialized. It can be used for subsequent invocations of **nabtoTunnelClose**() when a tunnel is no longer needed and **nabtoTunnelInfo**() to retrieve information about the current state of the tunnel: Is the tunnel being established, is it closed or is it successfully established as either local, remote (peer-to-peer) or relayed.

The following few lines sets up a tunnel:

```
nabto_status_t st = nabtoStartupNULL);
nabto_session_t session;
st = nabtoOpenSession(&session, "user@example.org", "12345678");

nabto_tunnel_t tunnel;
st = nabtoTunnelOpenTcp(&tunnel, session, 8080, "streamdemo.nabto.net", "localhost", 80);
while (st == NABTO_OK) {
    nabto_tunnel_state_t status;
    st = nabtoTunnelInfo(tunnel, NTI_STATUS, sizeof(status), &status);
    // ignore error handling / status updates for simplicity
    sleep(1);
}

// if status is ok, use the TCP tunnel

nabtoTunnelClose(tunnel);
nabtoCloseSession(session);
nabtoShutdown();
```

6.5 uNabto streaming

The Nabto stream abstraction provides reliable, TCP-like means of communication directly to Nabto Client API applications (vs. the indirect TCP tunneling approach described above). The stream implementation supports even very resource limited devices, meaning that reliable streaming is available even without TCP and with very little memory available. A typical use case is to securely push files (e.g., a firmware update) directly to a device, deployed behind a firewall.

To setup a tunnel, the API must be initialized as described in the previous sections – that is, **nabtoStartup**() must be invoked and a valid session must be opened with **nabtoOpenSession**().

When the session is established, a stream connection to a listening server is established with **nabtoStreamOpen**(). Only the target Nabto hostname is needed to specify as input. If the return value of **nabtoStreamOpen**() indicates success, the opaque stream handle supplied to the function is initialized, consider it similar to an opened TCP socket handle.

The application can now send and receive data on the stream using **nabtoStreamWrite**() and **nabtoStreamRead**(). To close a stream, **nabtoStreamClose**() is invoked.

A full example looks as follows:

```
nabto_status_t st;
/* in all of the below: handle error and abort if st != NABTO_OK */
st = nabtoStartup(NULL);

nabto_session_t session;
nabto_status_t st = nabtoOpenSession(&session, "user@example.org", "12345678");

nabto_stream_t stream;
st = nabtoStreamOpen(&stream, session, query);

const char* message = "Hello, world!";
st = nabtoStreamWrite(stream, message, strlen(message));

char* response;
size_t actual;
st = nabtoStreamRead(stream, &response, &actual);

/* use response for something */

nabtoFree(response);
nabtoStreamClose(stream);
nabtoCloseSession(session);
nabtoShutdown();
```

6.6 User Profile Management

When invoking **nabtoOpenSession**() to establish the initial context, the private key of a user profile is unlocked and used. This private key and an associated signed certificate must somehow be available to the Nabto Client API Library at this point – either the pair can be pre-installed (e.g., distributed with an application) or it can be generated.

The API function **nabtoCreateProfile**() can be used to create the key pair – when invoked, the Nabto Client API Library creates a new private key and an associated public key. It connects to the currently associated Nabto portal service to get the public key signed. This invocation requires an existing, verified user account on the portal. The portal to be used can be set in the Nabto configuration file as the urlPortalDomain variable[2].

If the user does not yet have a portal account, it is possible to signup for a new account using **nabtoSignup**(). When invoked with an email address and password for the account, the portal is contacted to start the signup flow: An email is sent to the user who must in turn click a verification link. Once done, **nabtoCreateProfile**() can be invoked.

[2] It can also be set through the API urlPortalDomain configuration option: invoke **nabtoSetOption**() prior to invoking **nabtoStartup**()

If the user has forgotten the account password, **nabtoResetAccountPassword**() initiates the account password reset flow by sending a new verification email to the user's email address.

A list of profiles available for use with **nabtoOpenSession**() can be obtained with **nabtoGetCertificates**().

Profiles are stored in the Nabto home directory – the directory specified to **nabtoStartup**() (see above).

7 Appendix A: Full Async Example

```
#define CHUNK_SIZE   8192
#define BUF_SIZE     16384

typedef struct {
    char data_[BUF_SIZE];
    size_t length_;
    nabto_async_resource_t ressource_;
    bool done_;
} context_t;

void NABTOAPI callback(nabto_async_status_t status, void* arg, void* userData) {
    context_t* context = (context_t*)userData;
    if (status == NAS_CHUNK_READY) {
       char chunk[CHUNK_SIZE];
        size_t actualSize;
        do {
           nabtoGetAsyncData(context->resource_ , chunk, CHUNK_SIZE, &actualSize);
           assert(context->length_ + actualSize < BUF_SIZE);
         memcpy(context->data_ + context->length_, chunk, actualSize);
         context->length_ += actualSize;
        } while (actualSize > 0);

    } else if (status == NAS_CLOSED) {
       context->done_ = true;
    }
}

int main(void) {
    nabto_status_t st = nabtoStartup(NULL);
    if (st != NABTO_OK) { /* handle error and stop */ }
    nabto_session_t session;
    st = nabtoOpenSession(&session, "user@example.org", "12345678");
    if (st != NABTO_OK) { /* handle error and stop */ }

    context_t* context = (context_t*)malloc(sizeof(context_t));
    memset(context, 0, sizeof(context_t));
    const char* nabtoUrl =
       "nabto://weatherdemo.nabto.net/house_temperature?sensor_id=3";
    st = nabtoAsyncInit(session, &(context->ressource_), nabtoUrl);
    if (st != NABTO_OK) { /* handle error and stop */ }

    st = nabtoAsyncFetch(resource, &callback, context);
    if (st != NABTO_OK) { /* handle error and stop */ }

    /* do other stuff, data is retrieved in the background and callback() is invoked when
ready */

    while (!context->done_) {
       sleep(1);
    }

    /* use data retrieved, see nabtoFetchUrl() example */
    nabtoAsyncClose(resource);
    nabtoCloseSession(session);
    nabtoShutdown();
}
```