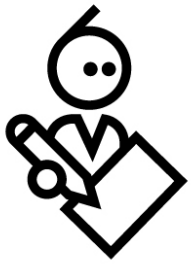
Nabto SDK

Writing a Nabto API Client Application

NABTO/001/TEN/025



Contents

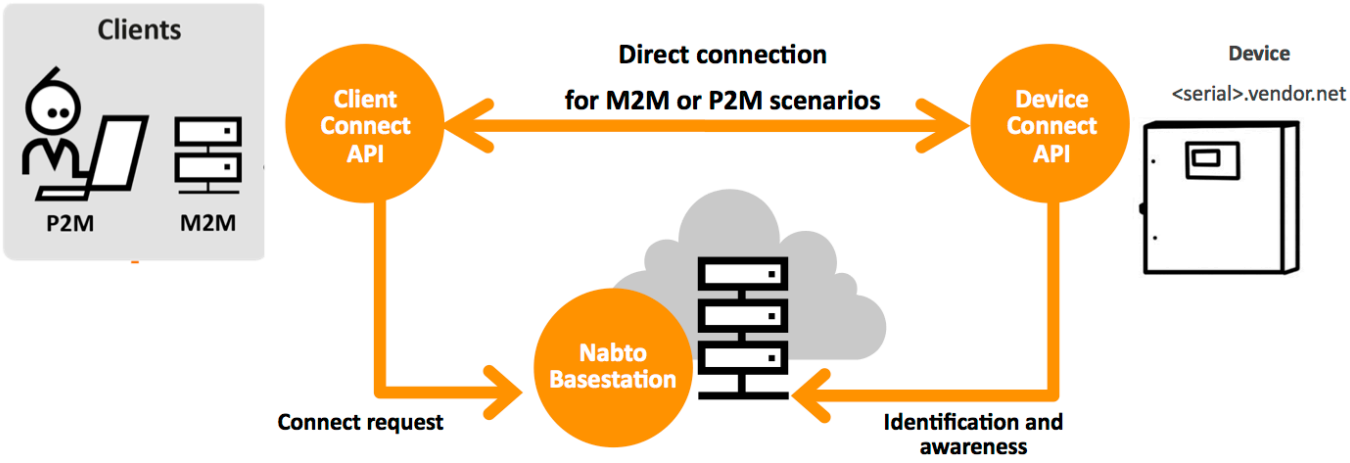
1 Abstract

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
TEN025	NABTO/001/TEN/025: Writing a Nabto API client application
TEN036	NABTO/001/TEN/036: Security in Nabto Solutions

3 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.

3.1 Nabto Communication Patterns

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


RPC: The Nabto P2P-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 P2P-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 P2P-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.

4 The Nabto Client API

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).

Wrappers exist to simplify development on the most popular platforms. All libraries and wrappers are downloadable directly from https://www.nabto.com/downloads.html or from repositories linked to from there.

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, JNI wrapper
iOS 4.x and newer	C library, Objective C wrapper
Apache Cordova	Cordova plugin supporting iOS and Android (only RPC communciation)

Ionic2	Starter app (including TypeScript wrapper for the Cordova plugin)
Xamarin	Xamarin wrapper for iOS and Android

5 Nabto Client Scenarios

Each of the different Nabto usage scenarios (RPC 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* (recommended).

5.1 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 associated public key (either CA signed or self-signed, depending on the security model chosen) is later used for authentication and authorization 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 or the certificate cannot be validated.

The simple initialization sequence hence looks as follows:

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

5.2 Nabto RPC Communication

5.2.1 Interface Definition

The services provided by a uNabto device are defined in an XML based interface definition, describing requests and responses. The format is described in detail in Appendix: RPC Interface Definition Details. Section 5.6 of [TEN023] describes the implementation of the interface as seen from the device.

The interface definition is very simple - 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>
      <parameter name="temperature" type="int32"/>
    </response>
  </query>
</unabto_queries>
```

All parameters accepted as input by the device is described in the **request** section of each named query and the output parameters described in the **response** section. A successful RPC invocation result will always be a JSON document with fields corresponding to these parameters.

The client automatically encodes request parameter and decodes responses according to this definition. The device must be implemented accordingly by the developer as specified in [TEN023].

To invoke a device from the client, the interface definition must first be supplied to the SDK as a string. Two functions exist for this - either the interface can be set per individual device using **nabtoRpcSetInterface** or a default can be specified using **nabtoRpcSetDefaultInterface**.

A full example is provided in the next section.

5.2.2 RPC Invocation

Once the interface has been supplied as outlined in the previous section, the remote device can be invoked using **nabtoRpcInvoke**. Request parameters as specified in the RPC interface must be specified in the URL, similar to an HTTP GET request.

If invocation succeeds, a JSON response document (null-terminated string) corresponding to the interface definition holds the response. See next section for handling errors.

The resulting document must be freed by the caller by invoking **nabtoFree**.

A full example showing setting the interface and invoking the device is seen below.

Note that all invocation is synchronous at the basic API level (higher level abstractions (Cordova) provide an asynchronous API).

```
int main(int argc, char* argv[])
{
    nabto_status_t st = nabtoStartup(NULL);
    if (st != NABTO_OK) { /* handle error and stop */ }
    nabto_handle_t session;
    st = nabtoOpenSession(&session, ”user@example.org”, ”secret”);
    if (st != NABTO_OK) { /* handle error and stop */ }

    const char* modelXml_ = readModelXml();
    char* err;
    nabto_status_t status = nabtoRpcSetDefaultInterface(session, modelXml_, &err);
    if (status == NABTO_FAILED_WITH_JSON_MESSAGE) {
        std::cerr << "Could not set RPC interface: " << err << std::endl;
        exit(1);
    } else if (status != NABTO_OK) {
        // handle error and return
    }

    char* json;
    const char* nabtoUrl = ”nabto://weatherdemo.nabto.net/house_temperature?sensor_id=3”;
    st = nabtoRpcInvoke(session, nabtoUrl, &json);
    if (st != NABTO_OK) { /* handle error and stop */ }
    // use data residing in resultBuffer
    nabtoFree(json);
    nabtoCloseSession(session);
    nabtoShutdown();
}
```

The resulting JSON document has the following format:

```
{
   "request" : {
      "sensor_id" : 1
   },
   "response" : {
      "temperature" : 21
   }
}
```

5.2.3 Error Handling

Most errors are communicated through the JSON response, the only errors communicated as return codes are related to invalid API usage or bad installation - see the API status enum in **nabto_client_api.h** for details.

If a JSON error is returned from the RPC functions, the API return code is **NABTO_FAILED_WITH_JSON_MESSAGE**. The **error** output parameter for **nabtoRpcSetInterface/nabtoRpcSetDefaultInterface** or the json output parameter for **nabtoRpcInvoke** is then set to a JSON document:

```
{
   "error" : {
      "body" : "The following error(s) occurred: \n   Model parse error in document :
Mismatched close tag </xquery> under parent <query>!\n",
      "detail" : "",
      "event" : 2000023,
      "header" : "Could not parse interface definition"
   }
}
```

The error document must be freed by the caller with **nabtoFree**.

Nabto error codes in th **event** field are divided in two scopes; 1.000.000 for fundamental Nabto errors and 2.000.000 for general application errors.

The special error code 2000065 (UNABTO_APPLICATION_EXCEPTION) allows the device application to return a specific error code from the application_event handler (the function invoked in the device by the uNabto framework to handle an incoming client request) - meaning that all communication between the client and device was ok, but some exceptional condition occurred at the application level in the device.

The possible error codes to use in the device and their mapping to what the client sees can be found in the uNabto SDK (`unabto/src/unabto/unabto_app.h`) . See [TEN023] section "The client query handler" for details on the return errors in the device application.

The error is supplied to the application in the "detail" field:

```
{
    "error": {
        "event": "2000065",
        "header": "Error in device application (2000065)",
        "detail": "NP_E_INV_QUERY_ID",
        "body": "Communication with the device succeeded, but the application on the device
returned error code NP_E_INV_QUERY_ID"
}
```

An overview of all Nabto event codes see section "Appendix: Nabto Error Codes". The Nabto Cordova plugin[1] demonstrates a simplified error handling approach that will be incorporated in the core Nabto Client SDK if successful. It aggregates the different error categories (API, Nabto events, device exceptions) into a single abstraction.

[1] https://github.com/nabto/cordova-plugin-nabto

If available, the Nabto Client Log can be inspected for more details (normally located in <homedir>/logs/nabto-api-log.txt, where <homedir> is the directory supplied to nabtoStartup() or the platform default, e.g. ~/.nabto on macOS/Linux). Remote syslog can be enabled from the basestation for both devices and clients, contact Nabto support for more information in this regard.

5.2.4 Migrating from nabtoFetchUrl() to nabtoRpcInvoke()

The Nabto HTML DD based applications where a full HTML application is served in a .zip file (the HTML Device Driver bundle) is now deprecated as browser support for plugins is being abandoned or amputated by all major browser vendors. Moreover, most new Nabto applications just used the HTML DD bundle as a way to specify an RPC interface used by nabtoFetchUrl(), i.e. not any content in the bundle but a nabto/unabto_queries.xml file.

To migrate the latter applications, i.e. applications that only use the HTML DD bundle for the interface specfication, the only change is that the client must be bundled with the interface specification instead of this residing centrally in the HTML DD repo. So basically, the unabto_queries.xml exactly as was deployed in the HTML DD repository in the nabto subdir of the zip should just be moved to the client application, either as a resource file or as a string in the source code.

The contents of unabto_queries.xml is then passed to one of the functions **nabtoRpcSetInterface** or **nabtoRpcSetDefaultInterface** prior to invocation of **nabtoRpcInvoke.**

5.3 TCP tunneling

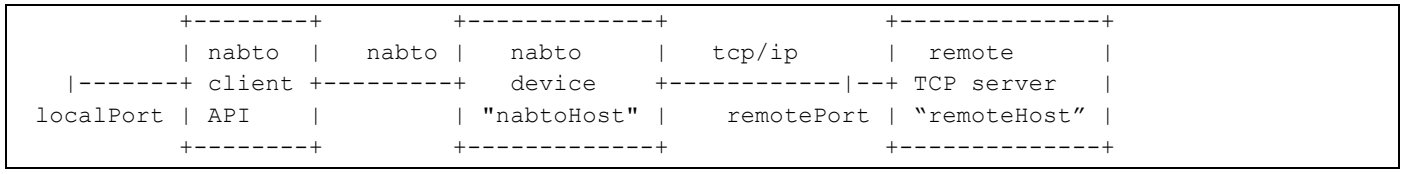
The Nabto TCP tunneling API is a high level wrapper for the underlying uNabto streaming implementation allowing applications to tunnel traffic through Nabto by integrating through a simple TCP socket, just like e.g. SSH tunnels. The underlying streaming API is described in the next section and can be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



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.

NOTE: If using localhost as remote host instead of 127.0.0.1 make sure that localhost is defined on the target.

The following few lines sets up a tunnel:

```
nabto_status_t st = nabtoStartupNULL);
nabto_handle_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();
```

5.4 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_handle_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();
```

5.5 User Profile Management

When invoking **nabtoOpenSession**() to establish the initial context, the private key of a user profile is unlocked and used. 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).

This private key and an associated public key (CA or self-signed certificate based on the authentication approach) must somehow be available to the Nabto Client API Library when invoking **nabtoOpenSession**– either the pair can be pre-installed (e.g., distributed with an application) or it can be generated. To decide if you should use CA or self-signed certificates in your solution, please consult section 8 of [TEN036]. Regardless of the approach chosen to creating the profile, a session is opened and the device invoked in the same way.

5.5.1 CA Signed Certificates

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 webservice host to get the public key signed. This invocation requires an existing, verified user account has been created through the webservice. The host to be used can be set in the Nabto configuration file as the urlPortalDomain variable[2].

The drawback of this approach is the added complexity in the sense that a central user database must be present and maintained. Up until Nabto platform release 3.0.14, Nabto provided such a central service and client API functions to sign up and reset password (the services still run for legacy clients). Issuing signed certificates is still supported for new scenarios that need this but you must provide the user management services yourself, contact Nabto for details. For new projects the new self-signed cert approach is recommended instead, see next section.

5.5.2 Self-Signed Certificates

To create a private key + self-signed certificate, invoke **nabtoCreateSelfSignedProfile.** This is the recommended approach to keep solution simple, yet secure; instead of involving a central CA, the device authenticates the client based on an initial pairing in a trusted setting involving exchange of the public key for later verification of the client in an untrusted setting. Please see section 8 in [TEN036] for details.

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

6 Appendix: RPC Interface Definition Details

6.1 RPC Interface Definition

The RPC Interface Definition[3] describes in detail the requests supported by the device and the responses sent to the client. The Nabto Plugin and Client uses this model directly to parse requests received from the device and encode requests sent to the device.

The query model schema is available from http://www.nabto.com/unabto/query_model.xsd . See *uNabto SDK – How to write a uNabto application* for implementation details on the uNabto device side.

A simple query model example could look like this:

```
<?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="light_read.json" description="Read light status" id="2">
      <request>
        <parameter name="light_id" type="uint16"/>
      </request>
      <response>
        <parameter name="light_state" type="uint8"/>
      </response>
</query>
</unabto_queries>
```

This model describes a query named `light_read.json`, with a single input (request) parameter named `light_id` and a single output (response) parameter named **light_state**. The mandatory query tag is used for identification in applications on requests and responses. The mandatory query id attribute is a compact identification of the query, used as opcode when sending requests to the device. The optional description attribute is a user-friendly name for the query.

[3] Formerly known as the "query model" in the legacy HTML DD style applications, residing in the unabto_queries.xml file in the HTML DD bundle

6.2 Parameter Types

The following types are supported in the interface definition:

Types	Description
uint8, uint16, uint32	Common type unsigned integers.
int8, int16, int32	Common type signed integers.
raw	An arbitrary length blob of data.
list	JSON formatted array or dictionary.

The common types **float** and **double** should be sent using raw or list as floating point types vary in representation on different platforms.

A Nabto message must not exceed the maximum UDP packet size limit, which is approximately 1300 bytes when you subtract the Nabto headers (see [TEN023] for a complete description of the Nabto packet sizes).

All types received by the client application are represented as strings in JSON. The client automatically converts requests to the type defined in the interface definition, and vice versa with the responses. The type is therefore only really important for the uNabto implementation, while the client side only needs to handle nicely formatted JSON requests / responses.

6.2.1 Raw

The raw type is used for arbitrary length data, e.g. strings, buffers and custom encoded data. When using raw it is left up to the developer to format the request as wanted. Nabto also implements a representation formatter, which can be used to translate raw parameters into hexadecimal format. It is used by supplying the parameter with a `representation="hex"` attribute as seen below:

```
<query name="read_data.json" description="Read and write raw data" id="5">
      <request>
        <parameter name="data_read" type="raw"/>
      </request>
      <response format="json">
        <parameter name="data_write" type="raw" representation="hex"/>
      </response>
</query>
```

This changes the JSON output received by the application to a hexadecimal representation. E.g. the string "nabto" sent from the uNabto application is received by the application as "6E6162746F" (contrived example as the hex encoding is mostly useful for binary, non-printable data). The hex attribute can also be used on the request sent to the uNabto device by appending the hex argument to the request parameter. This method can become handy

when dealing with an embedded application that uses a low level communication protocol like modbus commands.

When receiving JSON formatted raw responses with complex data inside, the Nabto Client automatically escapes characters, which would otherwise make it an invalid JSON object.

6.2.2 Lists

List elements give the application developer the ability to create more dynamic requests and responses. It is the equivalent of the JavaScript object array received in JSON format. A list can contain an arbitrary number of parameters and even other lists.

Here we are expanding on the previous example by making a query that receives a list of ids and returns a list of statuses:

```
<query name="multiple_light_read.json" description="Read light status from given ids" id="3">
      <request>
        <list name="light_ids">
          <parameter name="id" type="uint16"/>
        </list>
      </request>
      <response format="json">
        <list name="light_states>
          <parameter name="state" type="uint8"/>
        </list>
      </response>
    </query>
</unabto_queries>
```

When specifiying lists as a request parameter, the special JSON encoding of requests must be used - instead of specifying each parameter in the URL, a single "json=" parameter is specified in the URL when calling nabtoInvokeRpc.

An example JSON request adhering to this interface definition looks as follows:

```
{
   "request" : {
      "light_ids" : [
         { "id" : 0 },
         { "id" : 7 },
         { "id" : 42 }
      ]
   }
}
```

Meaning that the API invocation would look like:

```
...
const char* jsonRequest = "{\"request\":"
     "{{\"light_ids\":[{\"id\":0},{\"id\":7},{\"id\":42}]}}";
snprintf(urlBuf, MAX_URL_SIZE, "nabto://demodevice.nabto.net?json=%s", jsonRequest);
st = nabtoRpcInvoke(session, urlBuf, &json);
...
```

7 Appendix: Nabto Error Codes

```
UNSPECIFIED_ERROR                   =    1000000
PROGRAM_VERSION_CONFLICT            =    1000001
PROTOCOL_VERSION_CONFLICT           =    1000002
UDP_SOCKET_CREATION_ERROR           =    1000003
RESOLVER_ERROR                      =    1000004
STUN_ERROR                          =    1000005
UDT_SOCKET_CREATION_ERROR           =    1000006
UDT_CONNECTION_ERROR                =    1000007
FALLBACK_CONNECTION_ERROR           =    1000008
CONNECTION_ERROR                    =    1000009
UNKNOWN_SERVER                      =    1000010
ACCESS_DENIED                       =    1000011
CANNOT_VERIFY_CLIENT_CERTIFICATE    =    1000012
BAD_ID_IN_SERVER_CERTIFICATE        =    1000013
CANNOT_VERIFY_SERVER_CERTIFICATE    =    1000014
MICROSERVER_NOT_KNOWN               =    1000015
CONNECTION_PROBLEM                  =    1000016
ATTACH_LOST                         =    1000017
RESSOURCE_PROBLEM                   =    1000018
SYSTEM_ERROR                        =    1000019
ENCRYPTION_MISMATCH                 =    1000020
MICROSERVER_BUSY                    =    1000021
MICROSERVER_ADDR_MISMATCH           =    1000022
NO_RSP_FROM_CONTROLLER              =    1000023
MICROSERVER_REATTACHING             =    1000024
NO_RSP_FROM_SERVERPEER              =    1000025
CONNECT_TIMEOUT                     =    1000026
SELF_SIGNED_NOT_ALLOWED             =    1000027
CERT_ID_NOT_FOUND                   =    1000100
CERT_FILE_NOT_FOUND                 =    1000101
CERT_FILE_INVALID                   =    1000102
CERT_INVALID_PSW                    =    1000103
CERT_KEY_FILE_MISSING               =    1000104
```

```
UNSPECIFIED_APL_ERROR                =    2000000
HTTP_SERVER_UNAVAILABLE              =    2000001
MISSING_PARAMETERS                   =    2000002
PASSWORD_TOO_SHORT                   =    2000003
DATA_DIR_ACCESS_ERROR                =    2000004
CERT_CREATION_ERROR                  =    2000005
CERT_SIGNING_ERROR                   =    2000006
CERT_SAVING_ERROR                    =    2000007
HTML_TEMPLATE_RENDERING_ERROR        =    2000008
PORTAL_LOGIN_FAILURE                 =    2000009
PROXY_COULD_NOT_START                =    2000010
NETWORK_PROBED                       =    2000011
POST_DATA_RETRIEVAL_FAILED           =    2000012
POST_DATA_SUBMISSION_FAILED          =    2000013
NOT_LOGGED_IN                        =    2000014
TIME_OUT                             =    2000015
INVALID_PORT_NUMBER                  =    2000016
TUNNEL_COULD_NOT_START               =    2000017
PORT_ALREADY_IN_USE                  =    2000018
TUNNEL_DESTRUCTION_ERROR             =    2000019
DEVICE_REGISTRATION_FAILED           =    2000020
COULD_NOT_UNLOCK_DEVICE_KEY          =    2000021
COULD_NOT_STOP_CLIENT                =    2000022
QUERY_MODEL_PARSE_ERROR              =    2000023
INITIALIZATION_ERROR                 =    2000024
INTERNAL_ERROR                       =    2000025
QUERY_MODEL_INVALID_ID               =    2000026
QUERY_MODEL_NO_SUCH_REQUEST          =    2000027
QUERY_MODEL_NO_SUCH_PARAMETER        =    2000028
QUERY_MODEL_PARAMETER_PARSE_ERROR    =    2000029
QUERY_MODEL_MISSING_PARAMETER        =    2000030
GUIREP_DOWNLOAD_FAIL                 =    2000031
QUERY_SEND_FAILURE                   =    2000032
QUERY_RESPONSE_RECV_FAILURE          =    2000033
QUERY_RESPONSE_DECODE_FAILURE        =    2000034
NO_ACTIVE_REQUEST                    =    2000035
UNEXPECTED_RESPONSE_SIZE             =    2000036
GUIREP_INSTALL_FAILED                =    2000037
GUIREP_BAD_STRUCTURE                 =    2000038
FILE_NOT_FOUND                       =    2000039
INSTALLATION_FILE_MISSING            =    2000040
PORTAL_AND_KEY_PASSWORD_MISMATCH     =    2000041
INVALID_URL                          =    2000042
OTHER_TASK_ACTIVE                    =    2000043
UNSUPPORTED_OBJECT_TYPE              =    2000044
```

```
TCP_SOCKET_PROBLEM                  =    2000045
ROOT_CERT_MISSING                   =    2000046
CONNECTION_RESET_BY_PEER            =    2000047
NO_NETWORK                          =    2000048
NO_INTERNET_ACCESS                  =    2000049
LOCAL_MICRO_CONNECT_FAILED          =    2000050
INVALID_EMAIL_ADDRESS               =    2000051
EMAIL_ADDRESS_IN_USE                =    2000052
DEVICE_ERR_UNKNOWN_QUERY_ID         =    2000053
BUFFER_TOO_SMALL                    =    2000054
INVALID_BUFFER                      =    2000055
INVALID_ENDPOINT                    =    2000056
NO_DATA_AVAILABLE                   =    2000057
DATA_TRANSMISSION_PROBLEM           =    2000058
SESSION_KEY_MISSING                 =    2000059
SESSION_KEY_INVALID                 =    2000060
MANIFEST_PARSE_ERROR                =    2000061
GENERIC_LOGIN_FAIL                  =    2000062
QUERY_JSON_PARSE_ERROR              =    2000063
EMPTY_PARAMETER                     =    2000064
UNABTO_APPLICATION_EXCEPTION        =    2000065
QUERY_MODEL_MISSING                 =    2000066
RPC_INTERFACE_NOT_SET               =    2000067
MISSING_PREPARE                     =    2000068
```