Nabto SDK

Nabto API 客户端应用手册

NABTO/001/TEN/025

目录

1 摘要 3

2 参考文档 3

4 Nabto 平台基础知识 4

5 Nabto 客户端 API 5

6 Nabto 客户端方案 5

6.1 启动 Nabto 客户端 API 库会话 6
6.2 uNabto 请求/响应通信 6
6.3 异步请求/响应通信 9
6.4 TCP 隧道协议 11
6.5 uNabto 数据流 12
6.6 用户配置文件管理 13

7 附录 A ： 完整异步示例 15

1 摘要

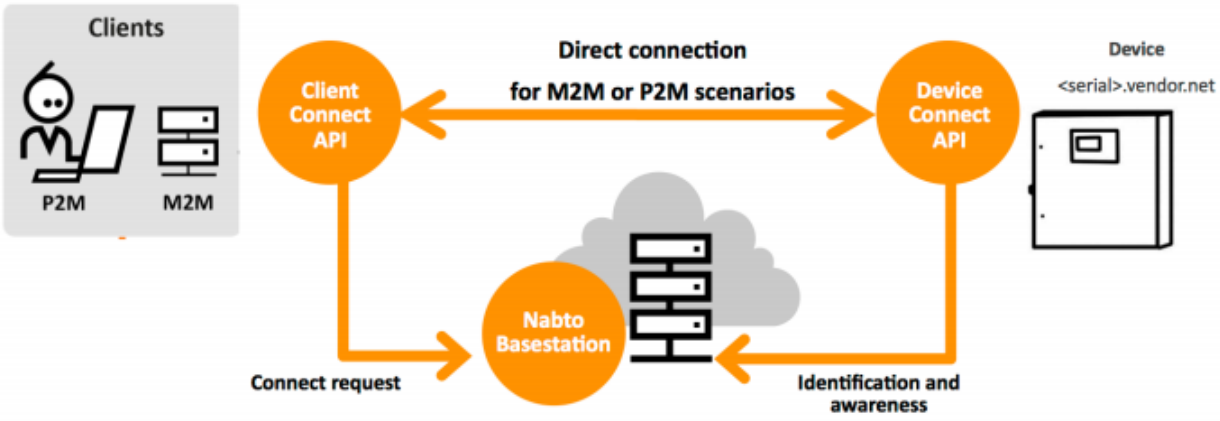
本文主要介绍如何使用 Nabto Client API 库编写本地 Nabto Client。

2 参考文档

TEN023	NABTO/001/TEN/023:uNabto SDK – uNabto 设备应用程序的编写
TEN024	NABTO/001/TEN/024:uNabto SDK – uNabto HTML 客户端应用程序的编写
TEN025	NABTO/001/TEN/025:uNabto SDK – Nabto API 客户端应用程序的编写

3

4 Nabto 平台基础知识



Nabto 平台包括三个部分：

- Nabto 客户端：由 Nabto 提供的二进制文件，用于客户的 HTML 或本地应用程序
- Nabto 装置：由 Nabto 提供的一个开源 SDK ，整合客户的设备应用程序
- Nabto 基站：由 Nabto （ Nabto 或自托管）提供的传达连接 Nabto 客户端和设备的服务，同时也向 Nabto HTML 客户端提供用户界面。

Nabto 客户端向 Nabto 的有效设备发起一个加密直连 - 并由 Nabto 基站调节这种直连：该设备唯一的名称，例如<SERIAL>.vendor.net 被映射到该 Nabto 基站的 IP 地址——这可以用来监测设备在线注册并帮助客户端寻找可用的设备。连接建立后，该基站会跳出循环——没有数据会被存储在基站，它仅显示当前有效的 Nabto 功能设备。

如果位于同一 LAN 上，客户端也可以发现设备，并且不通过基站进行直接通信——用于引导程序脚本或离线应用。

在客户设备上整合 Nabto 请详见[TEN023]。

用户的客户端应用程序可以以不同的方式使用该 Nabto 客户端：在 Web 应用中，客户应用程序可以作为一个 HTML 应用程序，使用该 Nabto 客户端检索 JSON 数据——在这种情况下 Nabto 客户端就是一个典型的 Web 浏览器插件或移动应用程序，代管客户应用程序。后者是由基站分配到用户端，并用来表示一个 HTML 的设备驱动程序软件包。如何写这样的应用程序请详见[TEN024]。

用户的客户端应用程序也可以作为一个本地（非 HTML）应用程序，和 Nabto 客户端 API 库进行连接。本地客户端应用程序可以使用相同的请求/响应机制，如同 HTML 应用程序一样

调用设备。此外，本机客户端可以建立数据流同设备进行连接——加入无缝、安全的远程访问功能，因而使它相较于传统客户端更为大众所接受。本地客户端应用程序详见本文档。

5 Nabto 客户端 API

Nabto 平台提供了一些预构置客户端：智能手机和平板电脑 APP 程序，浏览器插件，单机应用程序以及一个无插件 HTTP-Nabto 桥浏览器。所有这些客户端都围绕同一个公用库建立——Nabto 客户端 API 库。该库是免费提供的，用于围绕 Nabto 平台构建自定义应用程序。

本文详细介绍了如何编写不同类型的基于应用程序以及使用隧道技术的 API 库。

Nabto 客户端 API 库可用作能够访问平台上所有功能的基本 C 语言库。此外，也可作为面向对象的.NET 库提供，用于包装在.NET 平台上的典型抽象应用中较低级别的 API- 例如，它可以通过使用 Nabto 在应用程序升级为专有的客户端/服务器的过程中取代传统的 NetworkStream 对象。

Nabto 流媒体数据的功能（如视频流）仅在 Nabto 客户端 API 下有效（不适用于 HTML 客户端）。

Microsoft Windows (32/64- bit)	C 语言库，.NET4.0 抽象层
Mac OS X	C 语言库，.NET4.0 抽象层（单独申请）
Linux (32/64- bit)	C 语言库，.NET4.0 抽象层（单独申请）
Android 3.x and newer	带有 JNI 的 C 语言库
iOS 4.x and newer	C 语言库
Windows Phone 8.0 and newer	由于平台的限制没有库的支持

6 Nabto 客户端方案

具体不同 Nabto 使用场景（请求/响应 JSON 请求，隧道，流媒体）的描述详见下文。有些步骤适用于所有情况，在 *Starting a Nabto Client API Library Session* 已经有所说明。该章节假定一个 Nabto 客户端用户配置文件存在于客户端上——它既可以是预安装的，通过一个现有的预构建客户端对其进行设置，也可以是以编程方式拟定的，详见 *User Profile Management* 。

6.1 启动 Nabto 客户端 API 库会话

当使用 Nabto 客户端 API 库，该库必须首先通过调用 nabtoStartup()函数进行初始化。一旦一切准备就绪，API 库将被卸载， nabtoShutdown()函数将被调用。

一旦此函数库被初始化，一个用户会话必须被创建。它提供了这个函数库在所有实际应用中的情形。对于典型的库方案，nabtoOpenSession （email,password）变量将被启用：认证信息指定解锁一个主目录中的“users”子目录中的现有的密钥。创建一个私有密钥详见“User Profile Management”章节。一个特殊空密码账户“guest”可以创建一个访客会话，如果特别来宾配置文件是可用于客户端平台（安装客户端库的同时会创建一个默认的账户，比如在 Unix 系统中的路径是 in/usr/share/nabto/users）。

一旦所有的请求都在一个会话中完成， nabtoCloseSession()函数会被调用来关闭当前会话。

成功打开一个会话意味着用户的本地密钥可以被成功打开——它是用指定的密码进行加密的。密钥与包含用户的电子邮件地址的证书（一个签名的公钥）关联。此证书后用于试图与远程对等体进行沟通时的身份验证。因此，如果远程对等方不具有授予这个用户访问的设备或服务请求，它仍然有可能得到一个“ Access Denied”错误，即使有一个完全有效的会话。

简单初始化序列如下：

```
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();
```

一个会话可以也可以使用 nabtoOpenSessionBare()函数创建，而无需提供认证信息。这在特殊应用的情况下，如建立一个可以由库通过有如 HTML 页面形式进行提供用户认证信息的 Web 浏览器组件，是非常有用的。

6.2 uNabto 请求/响应通信

由 uNabto 设备提供的服务在 unabto_queries.xml 文件中定义。这是一个简单的 XML 格式文件用于描述设备的 Nabto 接口（详情请见[TEN023]）。例如，假设下面的查询模型

文件是针对一个支持单个查询的气象站设备，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>
```

有两种方法可以从支持这个接口的设备中获取温度读数：同步和异步方式。后者将在接下来的章节中描述。若要同步检索结果，调用 nabtoFetchUrl（）函数——从设备返回的响应信号作为一个 JSON 对象表示在一个字符串中，由库进行分配

完整序列如下所示：

```
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();
}
```

JSON 串在 **resultBuffer** 中为如下格式:

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

客户端应用程序拥有 ResultBuffer 和 mime 类型的内存所有权，当它不再需要时必须将其释放。**nabtoFree()**函数必须在此调用。

目前这一代 API[1]的可能错误处理：大多数错误都将通过 JSON 响应传达，这意味着应用程序将不得不解析 JSON 响应，以弄清楚请求的真实状况。返回码传达的少部分误差与无效的 API 的用法有关。

一个典型的错误响应如下：

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

函数 nabtoFetchUrl()的状态代码仍然是 NABTO_OK，说明与 API 的基本交互是成功的：nabtoStartup()和 nabtoOpenSession()函数已在 nabtoFetchUrl()函数调用之前被成功唤醒，但请求的执行出现了错误（在这种情况下，一个参数丢失）。

[1] 该 API 和执行于 HTML 的最终用户应用程序被设计在 Nabto 的 HTML 客户端应用程序(APPs 和浏览器插件）中

6.3 异步请求/响应通信

在 unabto_queries.xml 文件中定义的请求也可以异步执行，以允许在后台执行很长的请求。几个不同的 API 函数都参与了异步请求的执行，如下：

1. 每个异步请求都要先执行初始化函数 nabtoAsyncInit()，它接受请求字符串，并返回一个用于 API 的后续调用的操作。

2. 背景检索开始于 nabtoAsyncFetch()函数。当数据准备好或发生错误时，这个函数接受一个被 API 唤醒的用户回调函数（ NabtoAsyncStatusCallbackFunc ）。

3. 用户回调函数 NabtoAsyncStatusCallbackFunc 被库调用，如果状态指示数据已准备就绪，用户回调函数可以调用 nabtoGetAsyncData()函数来检索就绪数据块。

4. 数据检索完成后， nabtoAsyncClose()函数将释放异步处理。

下面显示了整体结构——附录 A 中提供了一个完整的示例：

```
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 ,
        // cal lback() is invoked when ready

        whil e (!context.done_) {
   sleep(1);
        }

        //    use data retrieved, see nabtoFetchUrl() example

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

说明："Asynchronous"在本客户端环境下与在[TEN023]中描述的设备上的异步 uNabto 应用处理程序无关：一个同步客户端请求可以实现调用一个异步 uNabto 应用，反之亦然。

6.4 TCP 隧道协议

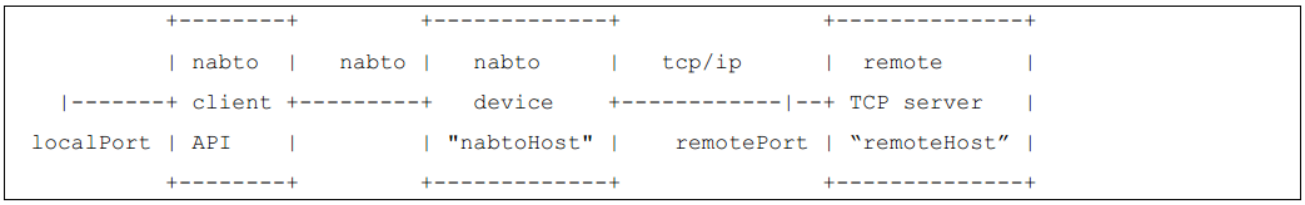
Nabto TCP隧道API是一个针对基础uNabto流实现的高层次封装，它允许应用通过Nabto进行隧道传输并通过一个简单的 TCP 套接字的整合，例如 SSH 隧道。基础流 API 将在下一节中进行描述，并且可以被用于在更紧密应用程序的结合。

要设置一个隧道协议，如前面章节中描述，首先必须初始化 API——也就是说，nabtoStartup()函数必须被调用，并且一个有效的会话必须通过 nabtoOpenSession()函数打开。

当会话已经建立，一个隧道协议也会通过 nabtoTunnelOpenTcp()函数建立。 同时 4 个隧道协议特有的参数也必须指定：

- localPort：API 将要监听的本地 TCP 端口——本地应用程序将连接到此端口。
- nabtoHost：Nabto 连接的远程 Nabto 主机将被建立，TCP 传输也将会通过此参数打开
- remoteHost：远程隧道终端的常规 TCP 主机将建立一个连接(通常这是一个本地主机—— 许多 Nabto 隧道服务器被限制不允许跳转到远程 TCP 主机)。
- remotePort:准备连接的目标服务器 TCP 端口。

这些参数表述如下图所示：



举例如一个远程 nabto 主机上运行的 Web 服务器“ streamdemo.nabto.net ”。将localPort设置为8080，当Web服务器同位于Nabto隧道终端时，RemoteHost就是localhost，将 remotePort 设置为 80 。因此，在成功调用 nabtoTunnelOpenTcp()函数与这些参数之后，应用程序就可以连接到运行着 Nabto 客户端 API 应用程序的计算机 TCP 端口 8080 上。

如果 nabto TunnelOpenTcp()函数的返回值表示成功，不透明隧道处理函数初始化完成。当隧道协议时不再需要时，它可以用于后续调用 nabtotunnelclose()函数，也可以用于nabtotunnelinfo()函数来检索隧道当前状态信息：隧道是否建立，它是被关闭了还是顺利地作为本地、远程(点-点)或中继建立起来了。

如下几行语句是对隧道进行设置：

```
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 数据流

Nabto 数据流抽象概念提供了可靠的直属 Nabto 客户端 API 应用程序的类 TCP 通讯方式（与上面描述的间接的 TCP 隧道协议接近）。此数据流实现甚至可以支持资源非常有限的设备，这意味着即使没有 TCP 和可用内存很少的情况下，这种可靠的数据流是依然有效的。一个典型的用例安全地将入栈数据（如固件更新）直接加载到一个设备，部署在防火墙后面。

要建立一个隧道，如前面章节中描述的 API 必须被初始化——也就是说，nabtoStartup()函数必须被调用，而且一个有效的会话必须通过 nabtoOpenSession()函数打开。

当会话被建立时，侦听服务器的数据流连接通过 nabtoStreamOpen()函数建立。只有目标 Nabto 主机名是需要指定输入。如果 nabto StreamOpen()函数的返回值显示成功，说明提供给函数的不透明数据流操作初始化完成，并把它当做类似于一个打开的 TCP 套接字操作。

该应用程序现在可以通过 nabtoStreamWrite()和 nabtoStreamRead()函数在数据流中发送和接收数据。想要关闭一个数据流，则需调用 nabtoStreamClose()函数。

一个完整示例如下所示：

```
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 用户配置文件管理

当调用 nabtoOpenSession()函数建立初步会话时，用户配置文件的密钥被解锁和使用。与此同时这个密钥和一个相关的签名证书必须以某种方式提供给Nabto客户端API库——无论它们是预安装的（例如，应用程序分配的）还是生成的。

API 函数 nabtoCreateProfile()被调用时可用于创建密钥对，该 Nabto 客户端 API 库会创建一个新的密钥和一个关联公钥。它连接到当前关联的 Nabto 门户服务从而得到的公钥签名。这个调用需要一个在门户网站上现有的，经过验证的用户账户。要使用该门户可以在 Nabto 配置文件中设置 urlPortalDomain[2] 变量。

[2] 此项也可以通过 API urlPortalDomain 配置项进行设置：需在调用 **nabtoStartup ()**函数之前调用 **nabtoSetOption ()**函数

如果用户还没有一个门户网站账户，可以通过 nabtoSignup()函数来注册一个新的账户。当一个电子邮件地址和账户密码被调用，门户网站连接并开始注册流程：一个电子邮件将会发送用户，用户必须依次点击验证链接。以上操作完成后，nabtoCreateProfile()才能被调用。

如果用户忘记了账号密码，nabtoResetAccountPassword()函数可以通过发送一个新的验证邮件到用户的电子邮件地址启动该账户的密码复位流程。

通过调用 nabtoGetCertificates()函数可以获得一个可与 nabtoOpenSession()函数配合使用的配置文件列表。

配置文件存储在 Nabto 主目录中—— 此目录在 nabtoStartup()函数中说明（见上文）。

7 附录 A ： 完整异步示例

```
#define CHUNK_SIZE      8192
#define BUF_SIZE        16384

typedef struct {
        char da ta_[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 invok ed when
ready */

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

        /* use data retrieved, see nabtoFetchUrl() example */

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