Nabto Tunnels

NABTO/001/TEN/030

1 Contents

1 Abstract

This document describes how to use the Nabto Tunnel facilities to build e.g. P2P RTSP streaming solutions.

2 Bibliography

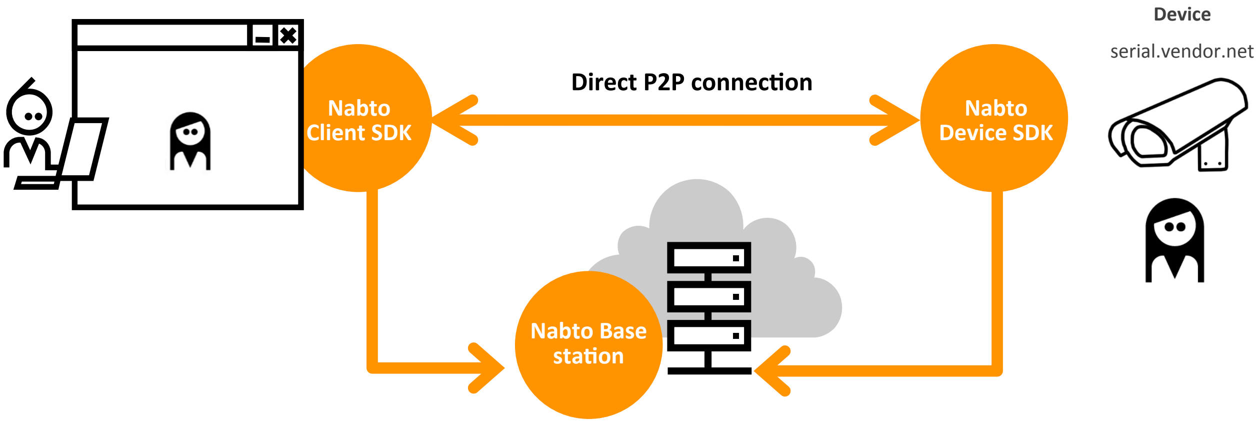
All documents are available for download from https://nabto.com.

TEN029	NABTO/001/TEN/029: Nabto Platform Specifications (general overview of Nabto)
TEN023	NABTO/001/TEN/023: uNabto SDK - Writing a uNabto device application
TEN025	NABTO/001/TEN/025: uNabto SDK - Writing a Nabto API client application
TEN036	NABTO/001/TEN/036: Security in Nabto Solutions

3 Introduction

This document describes in detail how to build and use the Nabto Tunnel client and server software, often used as RTSP proxy client and server in video streaming applications, allowing extremely simple integration with existing applications.

A Nabto Tunnel solution is built on top of the uNabto SDK (for devices) and the Nabto Client SDK (for mobile, desktop and m2m client applications):



On the device, the Nabto Tunnel application is installed in front of an existing TCP server (e.g., an existing RTSP server), essentially acting as a reverse proxy (or Nabto-TCP gateway). On the client, the Nabto Tunnel client end-point serves a TCP endpoint to existing applications and transparently tunnels traffic to the target service on the device - essentially acting as a device proxy (or TCP-Nabto gateway).

An introduction to the two individual SDKs is given in [TEN023] and [TEN025], respectively.

Nabto provides an open source tunnel application built with the uNabto SDK - this is often installed as is by video solution providers on the target system (e.g., camera, DVR or NVR) or with slight modifications. This application is described in section 4 below.

Vendors may then integrate their existing video player component with the Nabto Client SDK. Nabto provides demo applications that integrate with the SDK, downloadable from Apples App Store and Google Play (search for Nabto VIdeo). Desktop demos are also available, downloadable from https://nabto.com. Client applications are described in section 5 below.

The tunnelling facilities can be used for any TCP data transport that must happen securely and with no hassle for the user to configure firewalls. For other uses of the Nabto platform, see the general introduction in [TEN029].

It is of crucial importance to understand how to ensure security of the overall remote access solution: Please carefully read and understand the main sections of the document "Security in Nabto Solutions" [TEN036] on how to use the facilities in the Nabto platform to build a secure solution.

4 The uNabto Tunnel Application

The application is supplied as part of the uNabto SDK as an example of using the Nabto streaming abstraction. For further information on uNabto streaming on the device side, see section 7 in [TEN023].

4.1 Prerequisites

The uNabto SDK is downloaded from https://nabto.com > Downloads > uNabto Server > uNabto SDK.

To build the tunnel application, a C toolchain is needed as well as the cmake build tool (downloadable from http://www.cmake.org).

4.2 Compiling the uNabto Tunnel Application

Once downloaded, unpack the source. The tunnel application is located in unabto_sdk/unabto/apps/tunnel:

```
mkdir build_unabto_tunnel
cd build_unabto_tunnel
export CC=<path to c compiler, only necessary if this is a crosscompilation.>
cmake -DCMAKE_BUILD_TYPE=Release <path to unabto tunnel folder, e.g.
~/Downloads/unabto_sdk/unabto/apps/tunnel>
make
```

Note the Release build type – if not specifying on the commandline, optimization is disabled per default resulting in poor video streaming performance.

4.3 Compiling With OpenSSL

The generic encryption module as part of the uNabto SDK is not optmized for all platforms. If high performance streaming is needed, the encryption module has to be replaced by a faster module. For systems, which can run the OpenSSL library, it is recommended to use this library.

If OpenSSL is available for the target platform, the next step can be skipped - it describes how to compile the openssl libcrypto.a library for a specifc platform (but also see sections 4.5.3 and 4.6.4 below for fine-tuning).

Note that if using an ARM platform, Nabto comes bundled with the necessary software so this separate build step can be skipped. See section 4.5.3 instead for how to enable the bundled, highly optimized ARM openssl.

4.3.1 Compile OpenSSL for a Specific Target

Download latest stable OpenSSL source bundle from https://www.openssl.org.

Configure the uNabto SDK to use it and build the libraries as follows:

```
mkdir unabtotunnel
cd unabtotunnel
export TUNNEL_DIR=`pwd`
# The openssl target should be a valid target for the platform in use.
export OPENSSL_TARGET= # chose a valid openssl target like linux-armv4 linux-generic32
wget <openssl 1.x>
tar xf <openssl 1.x>
cd <openssl 1.x>
export CC=<path to c compiler, only necessary if this is a cross compilation>
./Configure $OPENSSL_TARGET -prefix=$TUNNEL_DIR/external
make
make install_sw
```

Now the install directory should contain a libssl.a and a libcrypto.a file.

To be able to compile the unabto_tunnel with the newly compiled OpenSSL library, the `UNABTO_EXTERNAL_BUILD_ROOT` option should be added to the CMake command:

```
cmake  -DCMAKE_BUILD_TYPE=Release -DUNABTO_EXTERNAL_BUILD_ROOT=$TUNNEL_DIR/external <path to
unabto tunnel folder>
```

4.3.2 Code Size with OpenSSL

Compiled for the Raspberry PI arm linux platform the tunnel has the following executable sizes:

Without libcrypto.a the unabto_tunnel executable is 137kB.

With libcrypto.a the unabto_tunnel executable is 1.3MB.

This size is likely optimizable if OpenSSL is tweaked for the platform. For ARM v4 linux platforms we have extracted the relevant OpenSSL assembler code into the openssl_armv4 crypto module and the openssl_armv4 random module. This gives a total size for the Linux ARM platforms around 200kB.

4.3.3 Compiling uNabto Tunnel for Windows CE

The unabto_tunnel application can be compiled for Windows CE out of the box, by using the CMake project.

```
mkdir tunnel_wince
cd tunnel_wince
cmake -G "Visual Studio 8 2005 <SDK> (<Architecture>)" <path to unabto/apps/tunnel>
cmake --build . --config Release
```

4.3.3.1 Compiling uNabto Tunnel for Beckhoff Windows CE 6.0 PLCs

A specific example for a Beckhoff PLC using the SDK from ftp://ftp.beckhoff.com/Software/embPC-Control/CE/Solutions/SDK/Beckhoff_HMI_600_V2.2_SDK.msi

```
cmake -G "Visual Studio 8 2005 Beckhoff_HMI_600 (x86)" <path to unabto/apps/tunnel>
cmake --build . --config Release
```

4.4 Running uNabto Tunnel on a Device

To start the tunnel application, a device id and corresponding cryptographic key is needed, see [TEN036], specifically section 9 "Installing uNabto Device Crypto Keys".

For development and test purposes, a device id and key can be obtained from the Nabto developer portal, https://console.cloud.nabto.com

The unabto_tunnel binary can then be run like this:

```
./unabto_tunnel -d <Device ID> -k <Key> --allow-port=554 --allow-port=80 --no-access-control
```

How exactly the key and device id is supplied to the executable is of course up to the integrator - either use an appropriate wrapper script that e.g. reads the key from persistent storage somewhere on the device. Or pass it through some other means, e.g. through bluetooth. Or customize the uNabto tunnel executable to securely retrieve from some central service.

4.4.1 Access control

Note the "--no-access-control" option in the above example - it means it is up to the application that uses the tunnel to authorize the user, e.g. through HTTP or RTSP basic auth (the model described in section 8.3 of TEN036 "Security in Nabto Solutions").

If not specifying this option, the PPKA approach in section 8.2 of TEN036 applies, meaning that the user's public key must have been added to the device's Access Control List prior to allowing a tunnel to be established - this must then be done in a trusted setting, e.g. on a local network.

The tunnel application supports such adding and removing of public keys (public key fingerprints, that is) through the "fp_acl_ae_dispatch" function that accepts Nabto RPC function invocations from the client to change the ACL, the RPC interface definition is located in unabto/src/modules/fingerprint_acl/unabto_queries-fp-acl-snippet.xml. Also see the README file in unabto/src/modules/fingerprint_acl and section 5.5.2 "Self-Signed Certificates" in TEN025 "Nabto Client SDK".

If using signed certificates (section 8.1 in TEN036), a custom solution that authorizes the client based on the certificate's userid must be implemented. The unabto/src/modules/acl can be used for maintaining Access Control Lists for this.

4.4.2 Performance

It can be necessary to increase the system priority of the tunnel process if the load on the system is high to ensure a stable throughput. Care must of course be taken with respect to how this impacts other parts of the system, so remember to always evaluate the full system performance when taking these measures (e.g. does live audio still work as expected?).

To increase the process priority, run with a very low *nice* value. Adjust the value to the lowest possible priority that yields the desired performance of the overall system. For instance, start with a value of "-19" by running the tunnel as follows:

```
nice -n -19 ./unabto_tunnel -d <Device ID> -k <Key> --allow-port=554 --no-access-control
```

Experiments with such low nice value should be run with care in a setting that can be easily undone e.g. by re-booting the camera.

4.5 Compiling a High-Performance Tunnel for a Resource Constrained Linux Based IP Camera

4.5.1 CPU Tuning

For most processors, some performance can be gained by specifying various compiler options. For example, the following ARM processor would benefit from the CFLAGS option `–mcpu=arm1136j-s`.

```
~ # cat /proc/cpuinfo
Processor       : ARMv6-compatible processor rev 5 (v6l)
BogoMIPS        : 384.20
Features        : swp half thumb fastmult edsp java
CPU implementer : 0x41
CPU architecture: 6TEJ
CPU variant     : 0x1
CPU part        : 0xb36
CPU revision    : 5
```

So to benefit from specific processor optimizations, use e.g. export CFLAGS="-mcpu=arm1136j-s" when building the uNabto tunnel application.

4.5.2 Buffer Tuning

The Linux device in this example is an HD video camera with a single h264 source. The video source will be 1Mbit/s so the streaming buffers should be configured to handle a fair amount of unacknowledged data on the network path.

Say, the maximum latency for a link is 1.5s - then 1Mbit/s * 1.5s gives 1.5Mbit(187kB) buffer needed. The default segment size is 1311 bytes per segment. The `NABTO_STREAM_*_WINDOW_SIZE` should then be 187/1.311 = 142. Please note that for relay connections, you must consider the full roundtrip latency including the gateway (device->gateway + gateway->client + client->gateway + gateway->device).

Since it is an HD video camera with limited memory, the number of concurrent streams needs to be optimized for the target to save memory. A good value will be 16 then the camera will have good video streaming performance and it can handle a few simple http requests.

The unabto_config.h is as follows:

```
// unabto_config.h
#ifndef UNABTO_CONFIG_H
#define UNABTO_CONFIG_H

#include <modules/log/dynamic/unabto_dynamic_log.h>

#define NABTO_ENABLE_STREAM 1
#define NABTO_STREAM_MAX_STREAMS 16

#define NABTO_STREAM_RECEIVE_WINDOW_SIZE 142

#define NABTO_SET_TIME_FROM_ALIVE 0

#define NABTO_APPLICATION_EVENT_MODEL_ASYNC 1
#define NABTO_ENABLE_EXTENDED_RENDEZVOUS_MULTIPLE_SOCKETS 1

#define NABTO_APPREQ_QUEUE_SIZE NABTO_CONNECTIONS_SIZE
#define NABTO_ENABLE_DEBUG_PACKETS 1
#define NABTO_ENABLE_DEBUG_SYSLOG_CONFIG 1

#ifdef LOG_ALL
#define NABTO_LOG_ALL 1
#endif

#endif
```

4.5.3 OpenSSL For ARMv4+

uNabto comes with a highly optimized OpenSSL targeted for ARM cpu's without hardware aes and sha256 support. You can hence use this without the full OpenSSL build described earlier. This implementation has been measured to perform much better (on ARM Linux) than hardware implementations due to the overhead involved in the latter and the small buffers in use in Nabto.

To use this special version, you need to call CMake with the following two options -DUNABTO_CRYPTO_MODULE=openssl_armv4 and -DUNABTO_RANDOM_MODULE=openssl_armv4. E.g.

```
cmake -DCMAKE_BUILD_TYPE=Release -DUNABTO_CRYPTO_MODULE=openssl_armv4 \
   -DUNABTO_RANDOM_MODULE=openssl_armv4 ...
```

It's also likely that OpenSSL can be compiled more optimal for a specific platform. Look in the OpenSSL documentation to get more information about platform optimizations.

4.6 FAQ

4.6.1 Problem: System says "Killed"

When running the unabto_tunnel executable the system says `Killed`.

Solution: This is probably due to too much memory usage try adjusting the buffer sizes.

4.6.2 Problem: Executable not found

When running the unabto_tunnel executable the system says

```
sh ./unabto_tunnel: not found
```

Solution: the executable is not compliant with the system try to compile a small test program that works and use the same options for the tunnel compilation.

4.6.3 Problem: gethostbyname warning

When compiling the tunnel, the compiler says:

```
Warning: Using ‘gethostbyname’ in statically linked applications requires at runtime the
shared libraries from the glibc version used for linking
```

Solution: for testing you can use the –A option to the tunnel. The problem can be that you are building an executable for an uclibc platform with a glibc based compiler.

4.6.4 Problem: Poor video stream performance and/or uNabto tunnel uses too much CPU

Please see section 4.5 on how to build a performance tuned library. Especially pay attention to building an optimized version of OpenSSL as outlined in section 4.5.3 - and how to make sure the correct version of the library is used. Also, double check you are setting the cmake build type to Release as shown in all examples above.

If the video stream performance is poor even with low tunnel CPU consumption, please check the network quality - do you have sufficient bandwidth available for the stream in question? Has the tunnel been configured with a sufficiently large window for the latency in question as described in section 4.5.2? Did you consider the larger latency for relayed connections? What is the network quality - do you have a large packet loss on either peer's network (e.g. a poor wifi connection)?

Often the standard ping tool proves very useful in diagnosing problems like this - try to measure latency between peers (the opposite peer's WAN address) and between each peer and the basestation. The latter can be measured by simply pinging the device's DNS name on each peer, which always resolves to the associated basesetation's IP address. Also, the ping tool reports the packet loss on the network in question.

If contacting Nabto support about performance problems please provide answers to the questions in this section - what is the CPU usage during streaming and the results of the ping tests described above. And if you observe peer-to-peer or relay connections in the poor performing scenario.

4.6.5 Problem: Poor live video stream performance during e.g. cloud recording

Make sure the device has sufficient upstream bandwidth available for all concurrent activities. Also ensure that the tunnel binary does get enough time on the CPU; often it makes sense to run e.g. cloud recording at a lower system priority (higher *nice* value) than the tunnel for live feeds. See section 4.4.2 on how to set a higher priority (lower nice value) for the Nabto tunnel process.

5 Nabto Tunnel Clients

Nabto tunnel clients are developed as Nabto Client SDK applications as outlined in section 5.4 of [TEN025]. Several ready-to-use examples exist for use during development and testing.

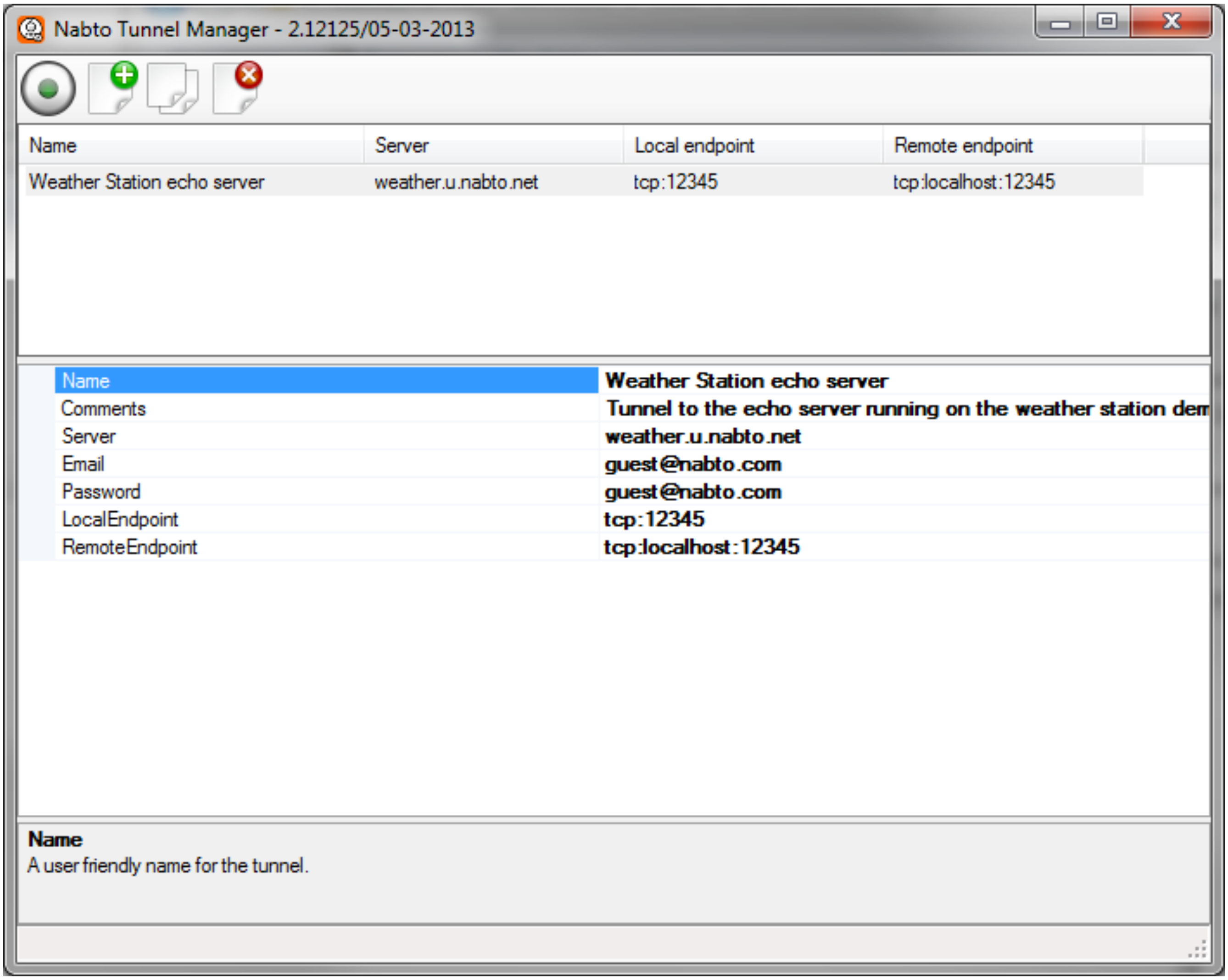
5.1 Demo Apps for iOS and Android

Search Apple's App Store and/or Google Play for "Nabto Video". The apps are very simple demonstrations of the streaming capabilities of the platform. You basically enter the device id - either manually, by scanning a QR code or by discovering it on the local network. Then the app sets up a tunnel and shows a video feed.

Please note that Nabto has several apps in the app stores - only the one(s) mentioning "video" are capable of establishing TCP tunnels and streaming video.

5.2 Desktop Demos for win32/.NET

An application for managing tunnels can be downloaded from https://nabto.com - Nabto Tunnel Manager. It sets up the TCP-Nabto tunnels - the actual client application runs external to the application. For instance, this could be the VLC video player.



Nabto Tunnel Manager - 2.12125/05-03-2013

Name	Server	Local endpoint	Remote endpoint
Weather Station echo server	weather.u.nabto.net	tcp:12345	tcp:localhost:12345

Name	Weather Station echo server
Comments	Tunnel to the echo server running on the weather station dem
Server	weather.u.nabto.net
Email	guest@nabto.com
Password	guest@nabto.com
LocalEndpoint	tcp:12345
RemoteEndpoint	tcp:localhost:12345

Name
A user friendly name for the tunnel.

5.3 Commandline Demos

Various simple commandline tools built with the Nabto Client SDK are also available for download from https://nabto.com: The .NET demos bundle and the simple test utility "Simple Client", downloadable under the "Nabto Client SDK" subpage.

The latter is used as follows:

```
$ simpleclient_app -q streamdemo.nabto.net --tunnel 12345:127.0.0.1:80
No username specified, defaulting to guest.
Connecting to streamdemo.nabto.net . connected
connected tunnelVer: 1  tunnelStatus REMOTE_P2P (4)
state has changed for tunnel 0xedbb1773 status REMOTE_P2P (4)
```

5.4 Building Custom Tunnel Clients

The following snippet from section 5.4 in [TEN025] demonstrates all it takes to open a Nabto tunnel endpoint from a client application using the Nabto Client SDK

```
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();
```

Please see [TEN025] for details on how to interact with the client API.