nDPI - Quick Start Guide

Open and Extensible LGPLv3 Deep Packet Inspection Library

Version 1.6

July 2015

© 2011-15

|  |  |
| --- | --- |
| nDPI web | <http://www.ntop.org/products/ndpi/> |
| ntop site | www.ntop.org |
| nDPI License | LGPLv3 |

Table of Contents

1. Introduction 4

1.1 Download Source 4

2. nDPI Library 5

2.1 Compiling nDPI Source Code 5

2.2 Compiling the demo pcapReader Source Code 5

2.3 pcapReader Command Line Options 5

2.4 Protocol File 6

3. Examples 8

3.1 Live Capture Mode 8

3.2 pcap Capture Mode 8

3.3 Protocol File 9

4. API nDPI 10

5. Developing nDPI custom protocol 13

5.1 Introduction 13

5.2 Creating new protocol 13

5.3 Add your protocol to nDPI 15

# 1. Introduction

nDPI is a DPI library based on [OpenDPI](http://code.google.com/p/opendpi/) and currently maintained by ntop.

In order to provide you a cross-platform DPI experience, we also support Windows, in addition to Unix/Linux. Furthermore, we have modified nDPI to make it more suitable for traffic monitoring applications.We have achieved this by disabling specific features that are unnecessary for network traffic monitoring, thereby speeding up the DPI engine.

nDPI allows application-layer detection of protocols, regardless of the port being used. This means that it is possible to both detect known protocols on non-standard ports (e.g. detect HTTP on ports other than 80), and also the opposite (e.g. detect Skype traffic on port 80). This is because nowadays the concept of port = application no longer holds.

Over the past few months we have added several features including:

* Enhancement of the demo [pcapReader](https://svn.ntop.org/svn/ntop/trunk/nDPI/example/pcapReader.c) application both in terms of speed/features and encapsulations supported (for instance you can now analyse GTP tunnelled traffic).
* Ability to compile nDPI inside the Linux kernel so that you can use it for developing efficient kernel-based modules.
* Various speed enhancements so that nDPI is now faster than its predecessor.
* Added many protocols (to date we support ~170 protocols) ranging from “business” protocols such as SAP and Citrix, as well as “desktop” protocols such as Dropbox and Spotify.
* Ability to define port (and port range)-based protocol detection, so that you can complement protocol detection with classic port-based detection.
* In order to let nDPI support encrypted connections, we have added a decoder for SSL (both client and server) certificates, thus we can figure out the protocol using the encryption certificate. This allows us to identify protocols such as Citrix Online and Apple iCloud that otherwise would be undetected.
* Ability to support sub-protocols using string-based matching

## 1.1 Download Source

nDPI is automatically downloaded when you build ntop and nProbe. However nothing prevents you from using it as a standalone DPI library. The source code can be downloaded from https://github.com/ntop/nDPI.

# 2. nDPI Library

## 2.1 Compiling nDPI Source Code

Start using nDPI Library is very simple. In order to compile this library you must meet certain prerequisites such as:

GNU autotools/libtool

gawk

gcc

To do this you need to install them using the following commands:

Fedora yum groupinstall Development tools

yum install automake libpcap-devel gcc-c++ libtool

Debian apt-get install build-essential libpcap-dev

Mac OSX port install XXX (Please install macports)

Once done that, you can compile the nDPI source code as follows:

cd <nDPI source code directory>

./autogen.sh

./configure

make

## 2.2 Compiling the demo ndpiReader Source Code

Starting using ndpiReader demo is also simple. In order to compile this you must use the following command:

cd <nDPI source code directory>/example

make

## 2.3 ndpiReader Command Line Options

The demo ndpiReader application can be used both in terms of speed/features analysis and encapsulations support. In particular, it is possible to specify a lot of command line options.

The available options and a minimal explanation of each option are listed below:

ndpiReader -i <file|device> [-f <filter>][-s <duration>]

[-p <protos>][-l <loops> [-q][-d][-h][-t][-v <level>]

[-n <threads>] [-w <file>] [-j <file>]

Usage:

-i <file.pcap|device> | Specify a pcap file/playlist to read packets from or a device for live capture (comma-separated list)

-f <BPF filter> | Specify a BPF filter for filtering selected traffic

-s <duration> | Maximum capture duration in seconds (live traffic capture only)

-p <file>.protos | Specify a protocol file (eg. protos.txt)

-l <num loops> | Number of detection loops (test only)

-n <num threads> | Number of threads. Default: number of interfaces in -i. Ignored with pcap files.

-j <file.json> | Specify a file to write the content of packets in .json format

-d | Disable protocol guess and use only DPI

-q | Quiet mode

-t | Dissect GTP tunnels

-r | Print nDPI version and git revision

-w <path> | Write test output on the specified file. This is useful for

| testing purposes in order to compare results across runs

-h | This help

-v <1|2> | Verbose 'unknown protocol' packet print. 1=verbose, 2=very verbose

* i <file.pcap|device>

This specifies a pcap file to read packets from or a device for live capture. Only one of these two can be specified.

* f <BPF filter>

It specifies a BPF filter for filtering selected traffic. It allows nDPI to take only those packets that match the filter (if specified).

* s <duration>

It defines the capture duration in seconds, only for live traffic capture.

* p <file>.protos

It specifies a protocol file (e.g. protos.txt) to expand the support of sub-protocols and port-based protocol detection. Be careful, the protocol defined to protos file overwrites the existing protocol.

* l <num loops>

Number of detection loops (test only).

* d

This flag disables the nDPI protocol guess and uses only DPI.

* t

It dissects GTP tunnels

* h

It prints the ndpiReader help.

* v <1|2>

Using this flag, ndpiReader generates verbose output that can be used to tune its performance. Number one is the lowest level that displays the packets with ‘unknown protocol', number two is more verbose.

UDP 62.101.93.101:53 > 192.168.1.132:56130 [proto: 5/DNS][2 pkts/260 bytes][chat.stackoverflow.com]

TCP 192.168.1.132:59323 > 62.161.94.220:80 [proto: 7/HTTP][8 pkts/1925 bytes][]

UDP 62.101.93.101:53 > 192.168.1.132:56682 [proto: 5/DNS][2 pkts/258 bytes][diy.stackexchange.com]

UDP 62.101.93.101:53 > 192.168.1.132:56916 [proto: 5/DNS][2 pkts/524 bytes][conjugator.reverso.net]

.....................................................................................

.....................................................................................

TCP 192.168.1.132:59323 > 62.161.94.220:80 [proto: 7/HTTP][8 pkts/1925 bytes][]

Undetected flows:

TCP 192.168.1.132:57995 > 157.55.133.142:12350 [proto: 0/Unknown][3 pkts/208 bytes][]

TCP 65.55.223.47:33033 > 192.168.1.132:57997 [proto: 0/Unknown][8 pkts/547 bytes][]

TCP 127.23.238.168:16384 > 240.199.103.219:0 [proto: 0/Unknown][11 pkts/1611 bytes][]

TCP 127.23.238.168:0 > 240.199.103.219:16384 [proto: 0/Unknown][11 pkts/2734 bytes][]

## 2.4 Protocol File

nDPI has the ability to support sub-protocols using string-based matching. This is because many new sub-protocols such as Apple iCloud/iMessage, WhatsApp and many others use HTTP(S) that can be detected by decoding the SSL certificate host or the HTTP “Host:”. Thus we have decided to embed in nDPI an [efficient string-matching library](http://multifast.sourceforge.net/) based on the popular [Aho-Corasick](http://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_string_matching_algorithm) algorithm for matching hundred of thousand sub-strings efficiently (i.e. fast enough to sustain 10 Gbit traffic on commodity hardware).

You can specify sub-protocols at runtime using a protocol file with the following format:

# Subprotocols

# Format:

# host:"<value>",host:"<value>",.....@<subproto>

host:"googlesyndacation.com"@Google

host:"venere.com"@Veneer

in addition you can specify a port-based protocol detection using the following format:

# Format:

# <tcp|udp>:,<tcp|udp>:,.....@

tcp:81,tcp:8181@HTTP

udp:5061-5062@SIP

tcp:860,udp:860,tcp:3260,udp:3260@iSCSI

tcp:3000@ntop

You can test your custom configuration using the ndpiReader (use -p option) application or enhance your application using the ndpi\_load\_protocols\_file() nDPI API call.

# 3. Examples

In this section we show some ndpiReader use cases.

## 3.1 Live Capture Mode

The following example shows the ndpiReader live capture mode by using the parameter -i to specify the device and the parameter -s to specify the live capture duration.

$ ./ndpiReader -i eth0 -s 20

-----------------------------------------------------------

\* NOTE: This is demo app to show \*some\* nDPI features.

\* In this demo we have implemented only some basic features

\* just to show you what you can do with the library. Feel

\* free to extend it and send us the patches for inclusion

------------------------------------------------------------

Using nDPI nDPI ($Revision: #### $)

Capturing live traffic from device eth0...

Capturing traffic up to 20 seconds

pcap file contains

IP packets: 2390 of 2391 packets total

IP bytes: 1775743

Unique flows: 78

nDPI throughout: 122.30 pps / 709.92 Kb/sec

Guessed flow protocols: 0

Detected protocols:

DNS packets: 57 bytes: 7904 flows: 28

SSL\_No\_Cert packets: 483 bytes: 229203 flows: 6

FaceBook packets: 136 bytes: 74702 flows: 4

DropBox packets: 9 bytes: 668 flows: 3

Skype packets: 5 bytes: 339 flows: 3

Google packets: 1700 bytes: 619135 flows: 34

## 3.2 pcap Capture Mode

The most simple way to create a pcap file is to use tcpdump command as in the following example:

ntop$ tcpdump -ni eth0 -s0 -w /var/tmp/capture.pcap -v

tcpdump: listening on en1, link-type EN10MB (Ethernet), capture size 65535 bytes

Got 0

Got 64

Got 75

Got 76

^C122 packets captured

122 packets received by filter

0 packets dropped by kernel

Once the pcap file has been created you will be able to launch the demo ndpiReader with the parameter -i:

$ ./ndpiReader -i /var/tmp/capture.pcap

-----------------------------------------------------------

\* NOTE: This is demo app to show \*some\* nDPI features.

\* In this demo we have implemented only some basic features

\* just to show you what you can do with the library. Feel

\* free to extend it and send us the patches for inclusion

------------------------------------------------------------

Using nDPI nDPI ($Revision: #### $)

Reading packets from pcap file /var/tmp/capture.pcap...

pcap file contains

IP packets: 4911 of 4911 packets total

IP bytes: 3321544

Unique flows: 145

nDPI throughout: 612.80 K pps / 3.09 Gb/sec

Guessed flow protocols: 11

Detected protocols:

Unknown packets: 6 bytes: 764 flows: 1

DNS packets: 50 bytes: 6158 flows: 25

HTTP packets: 2537 bytes: 841638 flows: 73

SSL\_No\_Cert packets: 1522 bytes: 303380 flows: 10

SSL packets: 24 bytes: 1648 flows: 8

FaceBook packets: 644 bytes: 201216 flows: 17

Skype packets: 11 bytes: 872 flows: 6

## 3.3 Protocol File

In order to clarify the features of the protocol file we are now going to explain how you can identify the flow of ntop.org.

For instance it is possible to do it by editing protos.txt.

ntop$ echo 'host:"ntop.org"@nTop'> protos.txt

Once the protocol file has been modified you will be able to launch the demo ndpiReader with the parameter -p:

$ ./ndpiReader -i en1 -s 30 -p protos.txt

-----------------------------------------------------------

\* NOTE: This is demo app to show \*some\* nDPI features.

\* In this demo we have implemented only some basic features

\* just to show you what you can do with the library. Feel

\* free to extend it and send us the patches for inclusion

------------------------------------------------------------

Using nDPI nDPI ($Revision: #### $)

Capturing live traffic from device en1...

Capturing traffic up to 30 seconds

WARNING: only IPv4/IPv6 packets are supported in this demo (nDPI supports both IPv4 and IPv6), all other packets will be discarded

pcap file contains

IP packets: 4755 of 4757 packets total

IP bytes: 1766370

Unique flows: 245

Guessed flow protocols: 16

Detected protocols:

Unknown packets: 1 bytes: 94 flows: 1

DNS packets: 38 bytes: 5160 flows: 19

HTTP packets: 265 bytes: 59831 flows: 20

SSDP packets: 20 bytes: 9564 flows: 14

SSL packets: 33 bytes: 2572 flows: 13

DropBox packets: 17 bytes: 2481 flows: 6

Skype packets: 12 bytes: 944 flows: 2

Google packets: 2544 bytes: 612765 flows: 94

**nTop packets: 407 bytes: 66765 flows: 32**

# 4. API nDPI

# 

In this section the nDPI API is highlighted.

The demo ndpiReader will be now taken has a basic example to show how to initialize the library. It is required to have a compiled library and a properly configured Makefile (i.e the demo Makefile).

To start to using the API of nDPI within your application - in addition to your includes - you must also add the following include file:

#include "ndpi\_main.h"

The library can be initialized as follows:

1. Declare the protocol bitmask and initialise the detection module

NDPI\_PROTOCOL\_BITMASK all;

ndpi\_struct = ndpi\_init\_detection\_module(

detection\_ticks\_resolution,

malloc\_wrapper,

free\_wrapper,

debug\_printf);

This function will allow you to initialise the detection module. The fields have the following meanings:

* u\_int32\_t ticks\_per\_second;  
   The timestamp resolution per second (like 1000 for millisecond resolution).
* void\* (\*\_\_ndpi\_malloc)(unsigned long size);  
   Function pointer to a memory allocator.
* void\* (\*\_\_ndpi\_free)(void\* prt);  
   Function pointer to a debug output function, use NULL in productive environments.

1. Enable all protocols (note that you can enable a subset of the protocols if you ) via the appropriate macro and set them within the detection module.

// enable all protocols

NDPI\_BITMASK\_SET\_ALL(all);

ndpi\_set\_protocol\_detection\_bitmask2(ndpi\_struct, &all);

This function will allow you to set the protocol bitmask already defined within the detection module.

1. In order to load an existing protocol file you must use the following function:

dpi\_load\_protocols\_file(ndpi\_struct, \_protoFilePath);

1. Once captured the flows from your pcap file or ingress device, they can be analyzed by using the following function:

protocol = (const u\_int32\_t)ndpi\_detection\_process\_packet(

ndpi\_struct,

ndpi\_flow,iph ? (uint8\_t \*)iph : (uint8\_t \*)if,

ipsize,

time,

src,

dst);

The fields have the following meanings:

* struct ndpi\_detection\_module\_struct \*ndpi\_struct;  
   The detection module.
* struct ndpi\_flow\_struct \*flow;  
   Flow void pointer to the connection state machine.
* const unsigned char \*packet;  
   The packet as unsigned char pointer with the length of packetlen. The pointer  
   must point to the Layer 3 (IP header).
* const unsigned short packetlen;  
   Packetlen the length of the packet.
* const u\_int32\_t current\_tickt;  
   The current timestamp for the packet.
* struct ndpi\_id\_struct \*src;  
   Void pointer to the source subscriber state machine.
* struct ndpi\_id\_struct \*dst;  
   Void pointer to the destination subscriber state machine.

1. Once the flows have been analysed, it is necessary to destroy the detection module via the use of the following function:

ndpi\_exit\_detection\_module(ndpi\_struct, free\_wrapper);

The fields have the following meanings:

* struct ndpi\_detection\_module\_struct\* ndpi\_struct;  
   The detection module to be cleared.
* void (\*ndpi\_free) (void \*ptr);  
   Function pointer to a memory free function.

For further information we suggest to read the files

nDPI/example/ndpiReader.c,

nDPI/src/include/ndpi\_structs.h,

nDPI/src/include/ndpi\_public\_functions.h

nDPI/src/ndpi\_main.c.

The protocol dissector files are contained in the nDPI/src/protocols directory.

# 5. Developing nDPI custom protocol

In this section we show the way to include your protocol inside nDPI.

## 5.1 Introduction

Each nDPI protocol is implemented as an entry function to be used at runtime by nDPI. The nDPI comes with several protocols that can be used as example for this activity. Below, we list the main concepts you need to know if you plan to develop an nDPI protocol.

## 5.2 Creating new protocol

Each protocol has to have a corresponding #define inside the following include file

<nDPI source code directory>/src/include/ndpi\_protocols\_osdpi.h

as follow

#define NDPI\_PROTOCOL\_MY\_PROTOCOL 171

where NDPI\_PROTOCOL\_MY\_PROTOCOL is the protocol “name” and 171 is the protocol ID that must be unique.

Once the protocol has been defined, you must create a new protocol source file like

<nDPI source code directory>/src/lib/protocols/my\_protocol.c

with the following content

#include "ndpi\_utils.h"

#ifdef NDPI\_PROTOCOL\_MY\_PROTOCOLS

.....

#endif

where it will be necessary to define an entry function as a follow

void ndpi\_search\_my\_protocol(

struct ndpi\_detection\_module\_struct \*ndpi\_struct,

struct ndpi\_flow\_struct \*flow)

{

struct ndpi\_packet\_struct \*packet = &flow->packet;

NDPI\_LOG(NDPI\_PROTOCOL\_MY\_PROTOCOL, ndpi\_struct, NDPI\_LOG\_DEBUG, "my protocol detection...\n");

/\* skip marked packets by checking if the detection protocol stack \*/

if (packet->detected\_protocol\_stack[0] != NDPI\_PROTOCOL\_MY\_PROTOCOL) {

ndpi\_check\_my\_protocol(ndpi\_struct, flow);

}

}

and a detection core function to process a packet of a flow with the following content

static void ndpi\_check\_my\_protocol(

struct ndpi\_detection\_module\_struct \*ndpi\_struct,

struct ndpi\_flow\_struct \*flow)

{

struct ndpi\_packet\_struct \*packet = &flow->packet;

u\_int32\_t payload\_len = packet->payload\_packet\_len;

.....

.....

if(“Found Protocol”) {

NDPI\_LOG(NDPI\_PROTOCOL\_MY\_PROTOCOL, ndpi\_struct, NDPI\_LOG\_DEBUG, "Found my protocol.\n");

ndpi\_int\_my\_protocol\_add\_connection(ndpi\_struct, flow);

return;

}

/\*Exclude Protocol\*/

NDPI\_LOG(NDPI\_PROTOCOL\_MY\_PROTOCOL, ndpi\_struct, NDPI\_LOG\_DEBUG,

“exclude my protocol.\n");

NDPI\_ADD\_PROTOCOL\_TO\_BITMASK(

flow->excluded\_protocol\_bitmask,

NDPI\_PROTOCOL\_MY\_PROTOCOL);

}

}

and a specific function to report the correct identification of the protocol as follow

static void ndpi\_int\_my\_protocol\_add\_connection(

struct ndpi\_detection\_module\_struct \*ndpi\_struct,

struct ndpi\_flow\_struct \*flow,

u\_int8\_t due\_to\_correlation)

{

ndpi\_int\_add\_connection(ndpi\_struct, flow,

NDPI\_PROTOCOL\_MY\_PROTOCOL,

/\*Choose the type of your protocol\*/

NDPI\_CORRELATED\_PROTOCOL or NDPI\_REAL\_PROTOCOL);

}

## 5.3 Add your protocol to nDPI

Once the protocol has been created, you must declare your entry function in the following include file

<nDPI source code directory>/src/include/ndpi\_protocols.h

with the following content

/\* my protocol entry \*/

void ndpi\_search\_my\_protocol(

struct ndpi\_detection\_module\_struct \*ndpi\_struct,

struct ndpi\_flow\_struct \*flow);

Each protocol must be associated with a NDPI\_SELECTION\_BITMASK. The full list of NDPI\_SELECTION\_BITMASK is contained in the file

<nDPI source code directory>/src/include/ndpi\_define.h

After choosing the selection bitmask for your protocol, you must inform nDPI of the new protocol by editing the file

<nDPI source code directory>/src/lib/ndpi\_main.c

it is necessary to add your protocol to the the function as follow

void ndpi\_set\_protocol\_detection\_bitmask2(

struct ndpi\_detection\_module\_struct \*ndpi\_struct,

const NDPI\_PROTOCOL\_BITMASK \* dbm)

.....

.....

.....

#ifdef NDPI\_PROTOCOL\_MY\_PROTOCOL

ndpi\_set\_bitmask\_protocol\_detection(ndpi\_struct,detection\_bitmask,a,

NDPI\_PROTOCOL\_MY\_PROTOCOL,

ndpi\_search\_my\_protocol,

NDPI\_SELECTION\_BITMASK\_MY\_PROTOCOL,

SAVE\_DETECTION\_BITMASK\_AS\_UNKNOW,

ADD\_TO\_DETECTION\_BITMASK);

/\* Update callback\_buffer index \*/

a++;

#endif

.....

.....

.....

ndpi\_struct->callback\_buffer\_size = a;

NDPI\_LOG(NDPI\_PROTOCOL\_UNKNOWN, ndpi\_struct, NDPI\_LOG\_DEBUG,

"callback\_buffer\_size is %u\n", ndpi\_struct->callback\_buffer\_size);