NDN Over UDP using Arduino

Antonio Cardace*, Davide Aguiari*,
* DISI, University of Bologna, Italy

Emails: antonio.cardace2@studio.unibo.it, davide.aguiari@studio.unibo.it

***Abstract*—Named Data Networking (NDN) is a promising paradigm for the future Internet architecture which opens up new possibilities for the data exchange among routers.**
In order to learn NDN principles, a simpler NDN protocol has been developed in a mobile environment by the means of different boards.
This paper is a mobile system project overview and It wants to explore the potentialities of the NDN paradigm applying to a IoT (Internet of Things) context.

I. Introduction

Nowadays, research - whether academic or not - is actively proceeding with the development of new and effective network paradigms, in order to cover the increasingly stringent requirements of computer networks.
The Internet is based on the IP network layer, which allows its functionalities to work globally. Every IP address identifies an end-point that can produce data or forward a request to the correct destination.
Sustained growth in e-commerce, digital media, social networking, IoT and smartphone applications has led to dominant use of the Internet as a distribution network and they intensified the discover of new and efficient data transfer protocols.
One of the most promising initiatives in this context is NDN (Name Data Networking): founded in September 2010, is one of the five "Future Internet Architecture Program" projects.
NDN changes the semantics of network service from delivering the packet to a given destination address to fetching data identified by a given name.
The name in an NDN packet can name anything an endpoint, a data chunk in a movie or a book, a command to turn on some lights, etc.
The project in this paper aims to test the NDN protocol on different single-board computers like Arduino UNO and Intel Galileo, developing a simpler routing protocol than the NDN official ones. NDN over UDP is a C++ library that is able to serve different kind of data from different sensors plugged to the boards.
The following document consists of this introduction and six sections: [II] the related works, used as starting point, [III] an overview of the boards architecture and of the software, [IV] a detailed description of the code, the Arduino API included and its functions, [V] a performance and correctness evaluation [VI] the conclusion.

II. Related works

In order to understand the NDN protocol dynamics, several academic papers have been studied:

- *Named Data Networking*
  by Zhang, Crowley, Papadopoulos is the masterpiece about NDN protocol in which they descrive Interest and Data packets, the routing logic, the reasons behind this new network paradigm, the architecture and the researchers' approaches.
- *Named Data Networking: a Natural Design for Data Collection in Wireless Sensor Networks*
  by Amadeo, Campolo, Molinaro and Mitton, instead, shows how NDN can be used in a IoT (Internet of Things) context, in a wireless sensors netowrk. Here NDNOverIP/NDNOverUDP, the procol we developed in our project, can be found.
- "*Named Data Networking of Things (Invited Paper)*"
  by Wentao Shang, Adeola Bannis, Teng Liang, Zhehao Wang, Yingdi Yu, Alexander Afanasyev, Jeff Thompson, Jeff Burke, Beichuan Zhang, and Lixia Zhang, yet another survey about NDN and its applications in the Internet of Things.

Finally, some open-source projects hosted on Github, like ndn-js or ndn-cpp, have been taken as example.

III. Architecture

It has been used two Intel Galileo connected to a network switch via Ethernet and one Arduino in order to check the Interest packets and Data packets forwarding behavior; a DHT-22 temperature/humidity module was plugged on every board.
Galileo is a X86 computer, it ships with an Intel Quark SoC X1000 Application Processor, a 32-bit Intel Pentium-class system on a chip, while Arduino UNO is a atmega328 microcontroller with 32K of RAM memory.
The Galileo board is also SW compatible with the Arduino SW Development Environment, which makes usability and introduction a snap.
In addition to Arduino HW and SW compatibility, the Galileo board has several PC industry standard I/O ports and features to expand native usage and capabilities beyond the Arduino shield ecosystem. A full sized mini-PCI Express slot, 100Mb Ethernet port, Micro-SD slot, RS-232 serial port, USB Host port, USB Client port, and 8MByte NOR flash come standard

on the board [1].
All the boards are connected to a Local Area Network and they belong to the same subnet even if the Intel Galileo boards have a serious UDP bug when trying to send UDP packets. [2] The developed library uses two new data structures, these are the NDN data packet (NDNDataPacket) and the NDN Interest packet (NDNInterestPacket), suitably simplified: from the original NDN packets, signatures and cache delays have been removed.
Here are the two packets in details:

```
typedef struct NDNInterestPacket {
#ifdef __ARDUINO_X86__
  unsigned long ip;
#endif
  byte type;
  unsigned long nonce;
  unsigned short nameLength;
  char *name;
} NDNInterestPacket;

typedef struct NDNDataPacket {
#ifdef __ARDUINO_X86__
  unsigned long ip;
#endif
  byte type;
  unsigned short nameLength;
  unsigned long contentLength;
  char *name;
  char *content;
} NDNDataPacket;
```

A third data structure NDNRouteENtry has been developed: it represents a FIB (Forwarding Information Base) entry and it's defined as follows:

```
typedef struct NDNRouteEntry {
  boolean freeBlock;
  unsigned long nonce;
  uint8_t interestHash[10];
  IPAddress ip;
  unsigned long timestamp;
} NDNRouteEntry;
```

The library follows the Arduino API Style Guide and it contains the begin() method to initialize a library instance; stop() to delete the utilized data structures and the startDaemon() routine which has the core role of correctly routing Interest and Data packet.
PublishInterests() must be used by the producers to inform the libray on how they plan to produce a particular interest, dumpRoutingTable() shows the FIB in the Serial Monitor while addNDNNodes() (only available for the Intel platform) identifies the Intel Galileo boards connected to the subnet in order to do UDP multicast.
The library constants are stored in the header file:

- NDN UDP port: 8888
- UDP Buffer size: 1K (256)
- FIB entries: 10
- Entry hash size: 10
- FBI entry TimeToLive (TTL): 5000ms

[1]Intel Galileo Datasheet: http://www.intel.com/newsroom/kits/quark/galileo/pdfs/Intel_Galileo_Datasheet.pdf

[2]More details here: http://www.inspirel.com/yami4/intel_galileo.html

IV. Implementation

Since the project has been developed for the Arduino platform the programming environment used was Arduino-IDE and the programming language of the library is C.
Moreover for testing purposes two other pieces of software have been developed:

- **ndn-client**, written in Go;
- **NDNProducerSim**, a producer simulator written in C;

The NDN library following the style guides proposed by Arduino is composed of a single directory **NDNOverUDP**, inside this directory there are the following files:

- **examples**, example sketches for the Arduino-IDE
- **utility**
- **NDNOverUDP.h**, library header
- **NDNOverUDP.cpp**, library source code

A. ndn-client

It has been developed to be used as a swiss-knife for the NDN protocol, it has multiple features which will be described below.
When invoked from the command line it requires one mandatory argument, which is the interest name:

```
$ ndn-client [OPTION]... INTEREST
```

Given this argument the client proceeds to broadcast the interest packet as a UDP datagram on the local network on the port 8888 (which has been chosen as the port of reference for the NDN protocol), then it listens on the same port for any incoming Data packet and prints on standard output the content of the just arrived packet.
As shown by the usage string above the **ndn-client** has many other features which can be used through command line options, the following is a short description of them:

- **-sd**, send a Data packet instead of a Interest one;
- **-c "string"**, content of the Data packet to be sent;
- **-dd**, Print dump of the received Data packet;
- **-di**, Print dump of the sent Interest packet;
- **-x**, Print a hex dump;
- **-nl**, Do not wait for a response Data packet;
- **-gw**, Instead of broadcasting the packet, use a NDN Gateway;
- **-intel**, Supply this option if using a network composed of Intel Galileo;

B. NDNProducerSim

This is a simple piece of software which simulates the presence of a NDN producer on the local network. It can be invoked from the command line as shown below:

```
$ NDNProducerSim INTEREST CONTENT
```

The inner workings are pretty simple, the program listens on any interface for a UDP datagram, when one arrives it parses the packet, if it is an Interest packet and the name in it matches **INTEREST**, which has been supplied to the simulator, then it replies with a Data packet with content **CONTENT**.
The application ignores any datagram which is not an Interest packet and does not match the name it can produce.

C. NDNOverUDP

This is the core library, it's been designed to be as developers-friendly as possible, the aim was to let programmers easily declare the interests they are able to produce and how they can produce them.
Following this simple schema instantiating a NDN producer on a local network can be done in few lines.
Here is an example of the only lines of code required to make an Arduino become a NDN-Forwarder/Router:

```
#include <NDNOverUDP.h>
#include <Ethernet.h>

NDNOverUDP ndn;
byte mac[] = { A MAC address };

void setup() {
ndn.begin(mac);
}
void loop() {
 ndn.startDaemon();
}
```

Here is another example where we would like the Arduino to become a NDN-Forwarder/Router and a producer as well.
This can be done publishing the interest we are able to produce.

```
#include <NDNOverUDP.h>
#include <Ethernet.h>

NDNOverUDP ndn;
byte mac[] = { A MAC address };

int homeTemp(char **buf) {
*buf = new char[20];
return sprintf(*buf,"%d", readSensor())+1;
}

char *names[20] = { "/home/temp"};
dataProducer producerFunctions[] = {
        homeTemp };

void setup() {
ndn.begin(mac);
ndn.publishInterests(names,
        producerFunctions, 1);
}

void loop() {
ndn.startDaemon();
}
```

1) FIB: The core component of the library is the FIB [1] (Routing table) and its associated routing algorithm.
The FIB has a fixed size in memory, this has been done due to the operating platform since the Arduino devices are usually higly constrained on resources, for example during the tests one of the device used was an Arduino UNO which only has 32K of memory, a good trade-off seemed to be a FIB which could contain up to 10 outstanding interest routes (interest requests which are yet to be fulfilled) at a given time. This parameter of the FIB can be changed modifying the compile-time definition **NDN_ROUTING_TABLE_SIZE**.
The most relevant fields of each entry the routing table are:

- **nonce**, stores the nonce of the interest request;
- **interestHash**, hash of the interest name;
- **ip**, ip address of the interest packet sender;
- **timestamp**, useful for an aging algorithm;

In order to save space instead of storing the interest name (which can be of arbitrary size) we store its hash instead.
The hash function used is the one offered from the library **ArduinoSpritzCipher**, the length in bytes of the hash representation can be defined at compile time modifying the definition **NDN_ROUTING_HASH_SIZE**, currently it is set at 16 bytes.

2) Routing Algorithm: The routing algorithm used in NDNOverUDP has been designed to be simple, efficient and fast.
As can be seen in the algorithm 1 the FIB stores in memory the backward path (in this implementation the IP address) for a Data packet to reach the consumer who has requested that same packet.

Algorithm 1: NDN Routing

```
Data: packet
while packet ← receivePacket() do
    parsePacket(packet);
    if isInterestPacket(packet) then
        if amIProducer(packet.name) then
            content ← produceContent(packet.name);
            toIP ← FIB.getRoute(packet.name);
            sendData(packet.name, content, toIP);
        else
            if notAlreadyForwarded(packet) then
                FIB.setRoute(packet);
                forwardInterest(packet);
            end
        end
    end
    if isDataPacket(packet) then
        toIP ← FIB.getRoute(packet.name);
        sendData(packet.name, content, toIP);
        FIB.deleteRoute(packet);
    end
end
```

The FIB is not a standard routing table which permanently stores (once discovered) the next hop for a given destination, it instead temporarily stores the hop back to the interest sender, as a matter of fact every interest packet which the NDN node cannot fulfill itself gets broadcasted to every other node in the local network.
In this implementation due to memory constraints imposed by the Arduino devices there is an aging algorithm which periodically checks the FIB and deletes any outstanding interest request which has not been fulfilled in a given time.

3) Arduino vs Galileo: Due to some bugs in the UDP stack there is a compile-time flag __ARDUINO_x86__ which is supplied to the compiler only when compiling for the Intel Galileo platform.
The bug makes impossible for an Intel Galileo to send broadcast UDP datagrams and to correctly parse the IP address of the sender of a datagram, for these reasons the compile-time flag is used throughout the code to distinguish between the standard Arduino and the one from Intel.
The differences in the code due to the Galileo bugs are a different packet structure and a multicast (through unicast) approach instead of the broadcast one used in the standard Arduino, of course the latter makes the Galileo implementation less flexible due to the requirement of the developer to instruct the library of all the NDN nodes present in the network.

D. utility

This directory contains a single file **util.h** in which there are the classic networking functions useful for translating the binary representation of a number from the host endianness to the network byte order and vice versa.
This header was necessary due to the lack of these functions in the standard libraries included by Arduino-IDE (although on the Galileo platform they are included by default).

V. Performance evaluation

To evaluate the performance and the correctness of the implementation several experiments have been conducted, here is a list with the scenarios we deployed to test the system:

1) Single Arduino
2) Arduino and NDNProducerSim
3) Network of Intel Galileo

There are no relevant network metrics to collect since the system is made and has been thus tested on a local network, therefore there's no significant latency and no packet loss.
Nevertheless it is important to mark that thanks to the small packet size (both Data and Interest) and simple and fast routing algorithm even the Arduino platform is able to process and respond to an interest request almost immediately, even when there is a routing process in the middle, as a matter of fact the response time to an Interest packet is indistinguishable from any ordinary output of a command line application.
Due to the available resources, which were three Galileo at hand, the most complex test we performed consisted of a scenario where one Galileo was directly connected to the ndn-client and the other two were accessible only by routing through the first device, this was achieved using the *–gw* flag of the ndn-client application which transmits a unicast Interest packet instead of broadcasting it.
In this last test we recorded no significant latency in getting a response from the client, and no packet loss.

VI. Conclusions

The scope of the project, which was implementing the NDN protocol over UDP for small and local IoT applications, imposes a limitation, the library as it is now cannot be used over the Internet, in fact it deals only within a local network, moreover it has been tested only with very few devices, therefore there's no claim that this system is scalable with more than 10-20 devices although the whole project has been designed and developed with scalability in mind.
NDNOverUDP has got very much room for improvements, here are few ideas for further development:

- implement security and encryption [2];
- extend the implementation to support IPv6;
- implement NAT traversal;
- add optional-fields to Interest and Data packets;
- implement Directed-Diffusion routing [3];

Although there is plenty of room for improvements, since this is a proof of concept, the library is ready for local contexts and easy to use for developers wishing to integrate NDN protocol in their local networks.

REFERENCES

[1] P. C. C. P. L. W. Lixia Zhang, kc claffy and B. Zhang, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, July 2014.

[2] IEEE First International Conference on Internet-of-Things Design and Implementation, *Named Data Networking of Things (Invited Paper)*, April 2016.

[3] IEEE Wireless Days (WD), *Named Data Networking: A natural design for data collection in Wireless Sensor Networks*, November 2013.