

NDN Beacon Project Report

Quanjie (Nero) Geng

Overview

This project is an experimentation on creating Eddystone style beaconing over NDN. The Beacon iOS app is used as a reference when designing the functionalities of the NDN beacon. The NDN beacon prototype is an RFduino. It is essentially an Arduino with Bluetooth Low Energy (BLE) built in. An RFduino producer app is written with NDN-CPP-LITE. A sample consumer app is written with NDN-JS and can be run on macOS with Node.js and noble installed (more details to be provided).

Background

Named Data Networking (NDN) <https://named-data.net/>

Eddystone <https://developers.google.com/beacons/edystone>

RFduino <http://www.rfduino.com/>

Bluetooth Low Energy (BLE) https://en.wikipedia.org/wiki/Bluetooth_Low_Energy

Protocol Design

The protocol design is split into two parts: 1) NDN-level protocol, and 2) BLE-level protocol. This split is due to the fact that the two protocols live in two different layers in the OSI model.

NDN-Level Protocol

Namespace

NDN beacons use the namespace prefix “/ndnbeacon/<device name>”. The “<device name>” component can be extended into multiple components should the need arise. For example, some NDN beacons are deployed at Sam’s house as part of some smart home system. It is natural to include “/samshouse” in the device name to distinguish these beacons from the ones in Sam’s neighbor Joe’s house. To distinguish between individual NDN beacons at Sam’s house, product serial number can be added to the device name field. The resulting namespace for each individual NDN beacon will look like “/ndnbeacon/<user’s home>/<serial number>”.

Method Types

The following set of methods are drawn from the Beacon iOS app.

1. LED - controls the LED on the NDN beacon. This is an example of interacting with physical components associated with the NDN beacon. It can be easily extended to

controlling servo, sensors, and other interesting components attached to the RFduino. To use this method, send interest packet with suffix “/LED/<r><g>”.

2. SETURL - set the URL for the NDN beacon. This changes the content to be remembered by the NDN beacon. To use this method, send interest packet with suffix “/SETURL/<URL>”.
3. GETURL - get the URL from the NDN beacon. This retrieves the content of the NDN beacon. To use this method, send interest packet with suffix “/GETURL”.
4. HELP - request info on available methods from the NDN beacon. The NDN beacon will respond to this method by sending back all available methods in comma separated format (e.g. “LED, SETURL, GETURL, HELP”). To use this method, send interest packet with suffix “/HELP”.

The most important feature of NDN beacon is setting and getting an URL. This is also the core of Eddystone in terms of user interaction. Eddystone aims to ease users' end by having beacons broadcast only URL to actual content. In this way, users would not need to download specific mobile apps to interact with Eddystone style beacons. If the business owner really wants his/her customers to download some app, he/she can broadcast the download URL over the beacon. With Eddystone beacon, the consumer end is in place in the Android phones.

Since NDN is not built-in to phones, I imagine that some mobile app will still be needed to do NDN communication, thus comes the addition of the “HELP” method. An NDN beacon mobile app should first send a “HELP” interest to any newly known NDN beacon to determine all its functionalities before proceeding to other use cases.

BLE-Level Protocol

BLE only defines server-client relationship and data transmission in terms of bytes. Each BLE data packet is 20 bytes max, so an additional BLE-level protocol is needed to ensure NDN transmission can happen.

NDN packets goes from tens of bytes to hundreds of bytes. In order to transfer these amount with BLE's tiny little 20 byte window, I adopted a protocol from Susmit's ndn-btle repo. Basically, NDN packets is cut into chunks of 18 bytes. Each BLE packet has a one byte index, followed by a one byte number-of-chunks, followed by the 18 byte content. If there isn't enough content to fill the 18 bytes in the last chunk, I will just send 2 + length of remainder content over BLE.

Development Details

Producer App

This refers to the NDN beacon app running on RFduino.

Environment

Microcontroller: RFD22102
IDE: Arduino IDE 1.8.5 on macOS 10.13.2
Language: C++
Bluetooth Library: RFduinoBLE
NDN Library: NDN-CPP-LITE

Setup

1. Follow this RFduino [guide](#) to get the RFduino driver/library and Arduino IDE installed. Try out a few examples provided in the IDE to make sure you have the correct setup.
2. Follow this NDN-CPP [guide](#) to install the NDN-CPP library.
3. Partially follow this [guide](#) to include the NDN-CPP-LITE library in your Arduino sketch. By partially, I mean the following:
 - a. Instead of “Change to the directory /hardware/tools/avr/avr/include and enter: In -s <NDN-CPP root>/include/ndn-cpp”, follow this [guide](#) to include NDN-CPP-LITE, i.e. create a soft-link to “<NDN-CPP root>/include” inside “<depends on your OS>/Arduino/libraries/”.
4. Now everything should be in good shape.

Code Overview

The producer app code should be commented and readable. Written below is an overview of the app. Reader should be familiar with the concept of NDN interest/data and BLE peripheral/service/characteristic and C/C++. As a peripheral device, there are three important aspects.

First is “advertise”. NDN beacon advertises itself using RFduino’s default service uuid = ‘2220’. Its device name is used as part of its NDN [namespace](#). The setup for RFduinoBLE is done in the “void setup()” function in the Arduino sketch template.

Second is “read”. RFduinoBLE has very clean interface “void RFduinoBLE_onReceive(char *data, int len)” ready to use. In this function, I implemented the [BLE-level protocol](#) read. Once the raw NDN interest packet chunks are collected into a single byte array, I used NDN-CPP-LITE TLV’s decode api to parse the array into an InterestLite object. After prefix checking, I focused on the method type. For LED type, I analogWrite to the LED pins according to the given rgb values. For SETURL type, I store the URL. For GETURL type, I flip the send_flag and constructed an NDN data packet from the given namespace, locally stored URL, and default hmackey using NDN-CPP-LITE TLV’s encode api. For HELP type, I flip the send_flag and constructed an NDN data packet from the given namespace, help message, and default hmackey using NDN-CPP-LITE TLV’s encode api.

Third is “write”. RFduinoBLE has an easy-to-use send api “RFduinoBLE.send()”. This function needs to be called in the main function “void loop()” in the Arduino template sketch, which is why I only flipped a send_flag when a response NDN data packet is required. Loop() check this send_flag to determine whether to send the NDN data packet buffer content. The send part is created as a separate function implementing the [BLE-level protocol](#) write.

Among the three important aspects, “read” and “write” are talking to two characteristics with uuid “2221” and “2222”. These two characteristics are packaged into the single RFduino’s default service.

Consumer App

This refers to the sample Node.js command line app running on macOS.

Environment

Laptop: MacBook Pro (Retina, 13-inch, Late 2013)
OS: macOS 10.13.2
Language: Javascript
Bluetooth Library: noble (Node.js)
NDN Library: NDN-JS

Setup

1. Follow this noble [guide](#) to get Node.js and noble installed. Try out a few examples provided in the repo to make sure you have the correct setup.
2. Master branch noble is not working for macOS as of 1/8/2018. Follow this [issue](#), and run “npm install git://github.com/jacobrosenthal/noble.git#highsierra” instead during installation.

Code Overview

The producer app code should be commented an readable. Written below is an overview of the app. Reader should be familiar with the concept of NDN interest/data and BLE central/service/characteristic and Node.js. As a central device, there are three important aspects.

First is “discovering services”. The consumer app scan for devices advertising services with uuid = “2220”. On discovery, it tries to connect to the peripheral. On connect, it checks each characteristic enclosed. The NDN beacon has three characteristics. “2221” is a read characteristic. “2222” is a write characteristic. “2223” is unknown/unused.

Second is “read”. Reading from a characteristic is very different from the way it’s done on the RFduino side. In the noble library, subscription is required for continuous reading. Therefore, I first subscribe to the found read characteristic, and then register event handler which is

triggered when the data has changed in the read characteristic. The event handler in this case implements the [BLE-level protocol](#) read. Once the read completes (chunks assembled), a sample usage of the data packet (decode and log) is performed.

Third is “write”. When a write characteristic is found, the consumer app will perform the “tests”. This part is written in a scripting manner. It tries sending the NDN beacon each of NDN interests constructed at the beginning of the program (the interests are constructed with NDN-JS library directly). The writeToCharacteristic() function implements [BLE-level protocol](#) write.

Special Note

In the current implementation of NDN, there are two types of packets: 1) interest packet, and 2) data packet. Ideally, since NDN is a “pull” network, data transmission should be initiated with interest packet. This means that if any user input is needed during NDN beacon setup, the beacon itself should “pull” that information from the users by sending interest packets to namespace owned by the users. However, the low power consumption requirement for the beacons prevent this behavior as it requires the beacons to constantly ask for user input. As a result, I used interest packet to do the setup/controls the NDN beacon. The user inputs are integrated to the name of the interests (see [Method Types](#)). This behavior should be changed/updated once NDN interest packets can carry parameters.

Future Work

At this stage of the project, authorization and authentication are not implemented on either the producer app or the consumer app. This means that any user can modify the URL remembered by the NDN beacon. In the future, a way to only let authorized user to modified the URL of the NDN beacon needs to be implemented. Also, I only used default hmackey to sign the data packet and I do not verify the signature upon NDN data packet arrival. In the future, signature verification as well as public/private key exchange and encryption need to be implemented. Besides security related issues, packaging the BLE/NDN code into library/api is also needed. “Face” interface for NDN over BLE needs to be developed and unified. The current implementation only considers P2P connection. If possible, ad-hoc bluetooth network should be experimented.

Discussion & Conclusion

Over the past fall quarter and winter break, I learned quite a lot about NDN and BLE. Before starting on this project, my knowledge of networking remains at the “WWII technologies” about TCP/IP and UDP. This NDN project not only has given me a chance to have hands on experience with an innovative way of networking but also let me learned about issues I never thought of. For example, how the simple IP protocol has enabled the great advancement in today’s internet and how it fails/will fail future growth. I also learned about other technologies

such as CDN. As a result, I felt I was introduced to a domain of interesting problems that I have never learned about. These problems are big deals in the sense that there could be a “disrupt” in the next couple of years. That’s why I am glad I started learning about them now.

I have mixed feelings towards developing NDN applications. On one end, the idea is well introduced in words on the main NDN website. Everything is explained in plain language and required little knowledge to networking. On the other end, the documentations for the libraries are not the easiest thing to follow. I honestly spent a good amount of time searching among the repositories for a place to start. However, once I started understanding the code, I realized it’s written in a very clean and organized way such that things are easy to follow and conventions are obeyed across different libraries. This experience is like buying antiques at Flea market. In this case, what I would recommend is to have powerful propaganda and neat booth. In other words, some quality quick start tutorial into the code base will definitely help any new comers.

Besides NDN, I also want to talk a little bit about BLE. ble is actually much better than classic Bluetooth, in terms of both performance and design. Some libraries, like the RFduinoBLE comes in very handy. Examples are easy to follow and adopt. Other libraries, like the ones written in python for Mac can be a complete disaster and waste of time. Some libraries were working before a hardware/OS update but fail to keep up the good work after the update. Anyways, my point is that this discrepancy across platforms / between libraries has hindered a lot of people from developing for Bluetooth and IoT devices. Only with good and uniform library support can the field really flourish.

Overall, it has been a very rewarding experience developing this project.