
ndn-iot-package-over-posix

NDN-Lite Team

Oct 04, 2020

CONTENTS:

1	ndn-iot-package-over-posix	1
2	Overview	3
2.1	Core concepts	3
3	Download and Build	5
3.1	Docker image	5
3.2	Non Docker Build (macOS)	5
3.3	Instruction for developers	6
4	Quickstart Examples	7
4.1	Preparation	7
4.2	Share QR Code and bootstrap Device	8
4.3	Play with Example Command	9
4.4	Fetch a Published Content	10
5	Tutorial	11
5.1	Initialization	11
5.2	Connectivity	11
5.3	Load, Advertise and Register Services	12
5.4	Bootstrapping and Callbacks	12
6	Shared Secrets Generation	15
7	Demo Application	17
8	Tasks	19
8.1	Task 1: if-smoke-then-alarm (if smoke, then alarm)	19
8.2	Task 2: its-too-hot (if temp > 80, then air conditioning on)	19
8.3	Task 3: turn-off-when-i-leave (if no motion > 5 minute, turn off light)	20
9	Publish/Subscribe System	21
9.1	Topic	21
9.2	Subscriber and Publisher	21
9.3	API Reference	22
10	Trust Policy	25
11	Authors	27
12	Future Plans	29

NDN-IOT-PACKAGE-OVER-POSIX



NDN-IoT Package over POSIX is a integrated IoT package based on NDN-Lite for POSIX compatible OS, such as Raspberry Pi, Mac OS and Ubuntu. This package is built upon [ndn-lite](#)

Please see our [documentation](#) if you have any issues.

OVERVIEW

Our architecture provides a unique abstraction of devices from their distinct home service provided and locator in a way that allows developers to build applications that are insulated from the specifics of which device they are using. For example, there is home service “LED”. Locators of “LED” service are “/bedroom/device-123” and “/bedroom/device-456”.

Applications use the same abstraction. When an application interacts with the virtual representations of devices, it knows that the devices support certain actions/commands based on its home service associated. A device that has the “LED” service provided must support both the “on” and “off” command. In this way, all LED lights are the same, and it doesn’t matter what kind of LED light is actually involved.

The virtual representations of devices is in the format of {service, locator}. Locator aggregation is supported. For example, {LED, /bedroom} may represent devices {LED, /bedroom/device-123} and {LED, /bedroom/device-456}. Programs that implement the virtual representations on actual devices is called Device Program.

2.1 Core concepts

Service

Services are the interactions that a device allows. They provide an abstraction that allows applications to work with devices based on the home services they support, and not be tied to a specific manufacturer or model.

Consider the example of the “Switch” service. In simple terms, a switch is a device that can turn on and off. It may be that a switch in the traditional sense (for example an in-wall light switch), a connected bulb, or even a music player. All of these unique devices have a Device Program, and those Device Program’s support the “Switch” service. This allows applications to only require a device that supports the “Switch” service and thus work with a variety of devices including different manufacturer and model-specific “switches”. The application can then interact with the device knowing that it supports the “on” and “off” command (more on commands below), without caring about the specific device being used.

This code illustrates how a application might interact with a device that supports the “Switch” service through Pub/Sub APIs:

```
// try to turn on bedroom switches with a payload for "on" durability
uint8_t seconds = 80;
ps_event_t command_content = {
    .data_id = "on",
    .data_id_len = strlen("on"),
    .payload = &seconds,
    .payload_len = sizeof(seconds);
};
ps_publish_command(NDN_SD_SWITCH, "/bedroom", &command_content);
```

Pub/Sub

Publish/Subscribe System.

Command/Content

Command and content are represented by Pub/Sub events and be published or subscribed separately through Pub/Sub APIs.

Events are represented in {data-id, payload}. Data-id specifies the semantic identifier of a piece of data. For example, data-id for commands that turning switch on can be “on”, and data-id for content that switch report their on/off state can be “state”.

DOWNLOAD AND BUILD

We use docker container for development environment. If you're on macOS, you can also use the Non-Docker build approach.

3.1 Docker image

A [docker image](#) with compiled ndn-iot-package-over-linux package can be used as the base development environment.

The image is based on Ubuntu. It contains all dependencies for this package, including dependencies for the [Quickstart Examples](#). It is **highly recommended** to use this image to save time for preparing the development environment.

To use the image, you can download the latest version using git:

```
$ git clone https://github.com/shsssc/ndn-lite-docker-image
$ cd ndn-lite-docker-image
```

To build the image:

```
$ docker build --tag ndnlite:0.1 .
```

To use the image for development:

```
$ docker run -d -p6060:6060 --name ndnlite-container ndnlite:0.1 #start the container
→ with iot-controller on http://localhost:6060
$ docker exec -it ndnlite-container /bin/bash #run shell in the container
```

3.2 Non Docker Build (macOS)

Download the latest version with git:

```
$ git clone --recursive https://github.com/named-data-iot/ndn-iot-package-over-posix.
→ git
$ cd ndn-iot-package-over-posix
```

Create a new directory and configure with cmake:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

build all examples:

```
$ make -j2
```

3.3 Instruction for developers

Compile the documentation with Sphinx:

```
$ cd docs && pip3 install -r requirements.txt  
$ make html  
$ open _build/html/index.html
```

QUICKSTART EXAMPLES

The `ndn-lite-docker-image` has NFD and IoT System Controller pre-installed. It is **highly recommended** to use it to save time. If it's used as your development environment, please skip to *Share QR Code and bootstrap Device*.

4.1 Preparation

If you're using Docker image, no more preparation is needed and please skip to *Share QR Code and bootstrap Device*. If you're not using Docker image and start your build on macOS, additional software are required.

1. *Get Started with ndn-cxx*
2. *Get Started with NFD*

Start your NFD:

```
$ nfd-start
```

Notice: Please build `ndn-cxx` and NFD from source.

Download and execute `ndn-iot-controller`:

```
$ cd /path/to/controller
$ git clone http://github.com/named-data-iot/ndn-iot-controller
```

Install dependencies (if you're using macOS or brew):

```
$ brew install zbar leveldb
```

Set up development environment:

```
$ python3 -m venv ./venv
$ ./venv/bin/python -m pip install -r requirements.txt
```

Run the controller server:

```
$ ./venv/bin/python app.py
```

4.2 Share QR Code and bootstrap Device

Top posting: If you feel difficulties in finding those pre-generated QR code, you can simply download from our [repo](#)

<project-root> is the ndn-iot-package-over-posix directory. If you're already in container's root path, you need

```
$ cd ndn-iot-package-over-posix
```

to go into the project root.

Devices in smart homes, need to be bootstrapped by a IoT controller to obtain credentials. Thus the IoT controller should have some shared secrets between it and devices to validate bootstrap request. The shared secrets, in our project, are in two formats: a txt file that be loaded by device program, and a QR code that be uploaded to IoT controller by users. Three pairs of pre-generated shared secrets are in folder /<project-root>/devices. Device programs in /<project-root>/examples will use two of them.

Hard-coded bindings between example device program and QR code are:

Device Program	Pre-generated QR Code
tutorial-app	device-398.png
tutorial-app-sub	device-24777.png

Additionally, you can generate shared secrets by following instructions [Shared Secrets Generation](#).

Now it's time bring your devices online.

Open controller's UI in browser at 127.0.0.1:6060, then click Device Bootstrapping button, a blank for uploading QR code should show up, as shown in the following picture.

Device Identifier	Public Key	Symmetric Key	Operation
device-398	498CCE9F87B82B6D81621758E994F4A9A2F050129 4B49AD55D4B2AED3380845CD2B5AC4EEA5820993 8E436133828ED0CCD4EA70BFACBFC14ABF4DB6D 7AB1BAC1	6154F3998F5C4C90B31E2D2466D83156	remove

We upload device-398.png to the blank, and click bootstrap button, which enable controller waiting for bootstrapping request in the following 5 seconds.

Now run the corresponding device program inside this 5-second bootstrapping window:


```
$ cd /<project-root>/build
$ ./examples/tutorial-app
```

In this process, controller may ask for sudo, please give our access.

This device-398 has two functions:

1. Subscribe to LED command and adjust illuminance value based on command content
2. Publish a string `hello` to a pre-defined topic every 400000ms

Note: When you stop a running ndn-lite application or disconnected a device and want to reconnect, you have to re-bootstrap the device. Before that, please **delete the device from the controller**. To do so, please use the “device list” page in controller and hit `remove`. As shown in the image below. Failure to do so will crash the application with TLV Type (should be TLV_AC_KEYID) not correct error.



NDN IoT Controller

System Overview

Device Bootstrapping

Device List

Service List

Invoke Service

Access Control

Send Interest

Manage Policy

Device List

Device Identity Name	Device Identifier	Device Info	Operation
/ndn-iot/1/livingroom/device-398	device-398	1	<div>remove</div>

4.3 Play with Example Command

Click `Invoke Service` button, you shall see a form asking for interested service and command parameters:

Service Invocation

Service To

Invoke

Service Type

Command

Parameter

Given now only one device has been bootstrapped, only one service can be selected. Then, select Issue Command. Input any command id and input an integer between 0 and 100 as the parameter. This command is supposed to send LED brightness to the device. Send command by clicking express interest to invoke service, in the terminal which runs tutoriala-app, device side result should show.

4.4 Fetch a Published Content

Following similar steps with bootstrapping device-398 (please do not kill it), we can bootstrap device-24777 to the controller in another terminal by running tutorial-app-sub. This device subscribes to the pre-defined topic where device-398 publishes its string. After a while, the hello string should appear in the terminal.

TUTORIAL

After playing with quickstart examples, now we go through `tutorial-app.c` by lines to see the details. This is a typical LED Device Program that subscribes to command and publishes some hello messages in “LED” service.

5.1 Initialization

The device instance is initialized by loading pre-shared secrets. As you can see from the definition of `load_bootstrapping_info`, it will read some hard coded `.txt` file, which will give the initial crypto keys and device identifiers. `ndn_lite_startup` is an essential function to initialize various in-library states (e.g., key storage, service list) and platform-specific configurations.

```
// PARSE COMMAND LINE PARAMETERS
int ret = NDN_SUCCESS;
if ((ret = load_bootstrapping_info()) != 0) {
    return ret;
}
ndn_lite_startup();
```

5.2 Connectivity

Connectivity interfaces are abstracted as “face”s,

```
// CREAT A MULTICAST FACE
face = ndn_unix_face_construct(NDN_NFD_DEFAULT_ADDR, true);
// face = ndn_udp_unicast_face_construct(INADDR_ANY, htons((uint16_t) 2000), inet_
→addr("224.0.23.170"), htons((uint16_t) 56363));
// in_port_t multicast_port = htons((uint16_t) 56363);
// in_addr_t multicast_ip = inet_addr("224.0.23.170");
// face = ndn_udp_multicast_face_construct(INADDR_ANY, multicast_ip, multicast_port);
```

In quickstart examples, device instance is connected to controller via Unix socket, given they’re one the same host. Otherwise, you can specify a multicast face `udp4://224.0.23.170:56363` for data communication.

5.3 Load, Advertise and Register Services

Our package abstracts devices/applications (noted as instance below) into services - that is, what home service the device/application is associated with. This allows us to build applications that can work with any device that supports a given associated home service. Services, from instances' perspective, are categorized into *access-requested* that device instance subscribes to that service thus need access key, and *encryption-requested* that means device instance will need a encryption key to publish data into that service.

In this package, before running into the main loop, one should specify the services to be used. By calling `sd_add_or_update_self_service()`, instance will advertise given services to the IoT controller when entering the main loop so that IoT controller knows who is providing which service. If an instance is interested in publishing messages in certain services, it should call `ndn_ac_register_encryption_key_request()` to request encryption key for that service. If an instance is interested in receiving messages from certain services, access keys should be requested by calling `ndn_ac_register_access_key_request()`. Requesting encryption or access keys should be prior to any `ps_publish_to()` or `ps_subscribe_to()` calls so that the latter two have necessary keys to correctly execute.

Note: It is the instances who actually publish commands/content, not services. Services are only the abstractions of instances.

Some pre-defined services are in `/<project-root>/ndn-lite/ndn-services.h`. You can define your own services in similar approaches.

```
// LOAD SERVICES PROVIDED BY SELF DEVICE
uint8_t capability[1];
capability[0] = NDN_SD_LED;

// SET UP SERVICE DISCOVERY
sd_add_or_update_self_service(NDN_SD_LED, true, 1); // state code 1 means normal
ndn_ac_register_encryption_key_request(NDN_SD_LED);
//ndn_ac_register_access_request(NDN_SD_LED);
```

5.4 Bootstrapping and Callbacks

Before bootstrapping device onto controller, pre-shared crypto keys and identifiers should be loaded and wrap into the sign-on request.

```
// START BOOTSTRAPPING
ndn_bootstrapping_info_t bootstrapping_info = {
    .pre_installed_prv_key_bytes = secp256r1_prv_key_bytes,
    .pre_installed_pub_key_bytes = secp256r1_pub_key_bytes,
    .pre_shared_hmac_key_bytes = hmac_key_bytes,
};
ndn_device_info_t device_info = {
    .device_identifier = device_identifier,
    .service_list = capability,
    .service_list_size = sizeof(capability),
};
ndn_security_bootstrapping(&face->intf, &bootstrapping_info, &device_info, after_
↪bootstrapping);
```

`ndn_security_bootstrapping()` does this job. The first parameters requires a face input where in send the sign-on request to. The `after_bootstrapping()` callback defines the behavior of device instance right after a successful device bootstrapping. In the quickstart examples, the behavior is subscribes to the LED command and periodically publish content.


```

void
after_bootstrapping()
{
    ps_subscribe_to_command(NDN_SD_LED, "", on_light_command, NULL);
    periodic_publish(0, NULL);
    // enable this when you subscribe to content
    //ps_after_bootstrapping();
}

```

`on_light_command` defines the logic upon receiving the command. You can use this as a template when writing command callbacks.

```

void
on_light_command(const ps_event_context_t* context, const ps_event_t* event, void* _
↳userdata)
{
    printf("RECEIVED NEW COMMAND\n");
    printf("Command id: %.*s\n", event->data_id_len, event->data_id);
    printf("Command payload: %.*s\n", event->payload_len, event->payload);
    printf("Scope: %s\n", context->scope);

    int new_val;
    // Execute the function
    if (event->payload) {
        // new_val = *real_payload;
        char content_str[128] = {0};
        memcpy(content_str, event->payload, event->payload_len);
        content_str[event->payload_len] = '\0';
        new_val = atoi(content_str);
    }
    else {
        new_val = 0xFF;
    }
    if (new_val != 0xFF) {
        if ((new_val > 0) != (light_brightness > 0)) {
            if (new_val > 0) {
                printf("Switch on the light.\n");
            }
            else {
                printf("Turn off the light.\n");
            }
        }
        if (new_val < 10) {
            light_brightness = new_val;
            if (light_brightness > 0) {
                printf("Successfully set the brightness = %u\n", light_brightness);
                ps_event_t data_content = {
                    .data_id = "a",
                    .data_id_len = strlen("a"),
                    .payload = &light_brightness,
                    .payload_len = 1
                };
                ps_publish_content(NDN_SD_LED, &data_content);
            }
        }
        else {
            light_brightness = 10;
        }
    }
}

```

(continues on next page)

(continued from previous page)

```
        printf("Exceeding range. Set the brightness = %u\n", light_brightness);
    }
    else {
        printf("Query the brightness = %u\n", light_brightness);
    }
}
```

Symmetrically, there's a content subscription callback in `tutorial-app-sub.c`. You can use that as a template to write content subscription callbacks.

```
void
on_light_data(const ps_event_context_t* context, const ps_event_t* event, void*_
↳userdata)
{
    printf("RECEIVED NEW DATA\n");
    printf("Data id: %.*s\n", event->data_id_len, event->data_id);
    printf("Data payload: %.*s\n", event->payload_len, event->payload);
    printf("Scope: %s\n", context->scope);
}
```

Now you can play with `tutorial-app` and `tutorial-app-sub` in two terminals to see how the Pub/Sub pair works.

Note that `tutorial-app` should be online first so that `tutorial-app-sub` can request keys on an actual existing service. Because the “LED” service won't exist until former Device Program register it to the IoT controller.

SHARED SECRETS GENERATION

If you're not using image on build the package on macOS, install dependencies for QR code:

```
$ pip install pyqrcode  
$ pip install pypng
```

If you have already built the library, you can generate shared secrets by:

```
$ cd /project/root  
$ cd build  
$ ./examples/tutorial-gen-new-shared-info
```

This will generate a `tutorial_shared_info.txt` file in the `build` folder.

`tutorial_shared_info.txt` contains identity string, public/private key pair and a pre-shared key between identity and IoT controller.

Afterwards, you need to generate a QR code for this `.txt` file (may need `sudo` here).

```
$ cd /project/root  
$ python ./QR_encoder.py
```

Then a `shared_info.png` is generated in the project root path. The QR code encodes all information contained in `tutorial_shared_info.txt` and can later be uploaded through IoT controller's Web UI.

Finally, you can rename the `shared_info.png` and `tutorial_shared_info.txt` with whatever names you like and put into `/<project-root>/devices`. We recommend naming conventions used by existing examples.

The `.txt` file is used by device/application to obtain initial key materials and `.png` is used by IoT controller to gain shared secrets between itself and devices.

DEMO APPLICATION

Applications are written in the same way with Device Programs: request key materials, subscribe to something and publish something.

Application also use shared secrets like Device Programs. It's weird that application identity is still named as "device". We'll support different naming conventions in future.

Here, we present a template application: `if-hot-then-light`.

This template application implements the logic: if temperature in livingroom is above 80 degree, turn on the light in the livingroom and set the brightness to 30 percent. This application subscribe to "TEMP" service with locator "/livingroom" and publish command on "LED" service with locator "/livingroom".

Two Device Programs and one Application are involved:

Device Program/Application	Pre-generated QR Code	Role
tutorial-app.c	device-398.png	LED Light
tutorial-app-sub.c	device-24777.png	Temperature Sensor
app-template.c	device-63884.png	if-hot-then-light

When you play this demo application, bootstrap identities in three terminals in the order: device-398, device-24777, device-63884, so Device Program and Application can request keys on actually existing service.

LED light should print out the command, indicating command received.

This Application can be used as template to write your program. When adding new Applications or Device Programs, you should also add them to CMakeInputs. CMake configurations for examples are in `<project-root>/CMakeInputs/examples.cmake`.

TASKS

After getting familiar with the package by going through tutorial, you can try simple development tasks with this package.

Each of the task is a home automation senario where your application read result from one sensor and give commands to another device accrodingly. To facilitize the testing of your application, we provide [ndn-lite-mock-utils](#) to generate mock devices needed.

To generate a mocked smoke detector and a mocked alarm, you can use:

```
$ make TASK=if-smoke-then-alarm
```

And then the binary of the devices together with their credentials are ready to use in the same directory. The newly generated `command_receiver` can be used as the alarm, and `sensor` can be used as the smoke detector.

You can see the [code](#) and [documentation](#) of this library for more details.

Note: to prevent credential conflict, please **remove all devices from the `iot-controller` (especially `tutorial-app` and `tutorial-app-sub`)** as instructed in [Quickstart Examples](#) before you start these tasks.

Also, please use `device-63884` credentials in the `/devices` directory of the `ndn-iot-package-over-posix` for your automation applcation.

8.1 Task 1: if-smoke-then-alarm (if smoke, then alarm)

Try to develop a `if-smoke-then-alarm` automation logic. There should be a smoke detector that publishes content to smoke service, and a alarm that subscribes to alarm service. Data published by smoke detector can be binary: 0 -- no smoke, 1 -- smoke. This application should subscribe content from a smoke topic, and publish command to alarm topic. Payload carried in command can be binary, too: 0 -- alarm off, 1 -- alarm on. Three identities are invovled in total.

8.2 Task 2: its-too-hot (if temp > 80, then air conditioning on)

Try to develop an home automation that control your air conditioner. This application should subscribe content from some temperature sensors, and publish command to `air_conditioner` service. Whenever current temperature is above 80 degree, this application should try to turn the air conditioner.

8.3 Task 3: turn-off-when-i-leave (if no motion > 5 minute, turn off light)

Imagine users don't want waste electricity for living room lighting if no one is active working/moving. This application should subscribe content from motion sensors in living room, and publish command to lights in living room. Motion sensor periodically publish binary states: `active`, `inactive`, on `motion` topic. Lights subscribe to command that carry the desired illuminance value (as `tutorial-app` we presented). Whenever the inactive states last for over 5 minute, this application should issue command to set living room lights' illuminance value to 0. Our `tutorial-app-sub` is a good template to reuse for light logic, and you may need to implement motion sensor logic on your own.

PUBLISH/SUBSCRIBE SYSTEM

The Publish/Subscribe system constitute the core of package. Principally, every application data exchanges should be wrapped into the Pub/Sub.

9.1 Topic

Topic definition is foundation of building any Pub/Sub system. In this package, Topic is defined as series of name components connected by /. Based on longest prefix match, a topic can also be a *child topic*, or *parent topic* of another topic. For example:

Topic	Child Topic	Parent Topic
/A	/A/B /A/B/C	/
/A/B	/A/B/C	/
/A/B/C	None	/

In current design, topic hierachy has two level: *room* and *individual device*. Then an example mapping can be:

Topic	Child Topic	Parent Topic
/living	/living/device-398 /living/device-24777	/

9.2 Subscriber and Publisher

In this package, any instance that can send data to *topics* is publisher, and any instance that can receive data from *topics* is subscriber. A device instance can be publisher and subscriber at the same time.

9.2.1 Notice

If the data is reliably delivered is out of the scope of definition.

With a hierarchical topic layout, subscription to any topic is equivalent to subscription to the topic itself and also all child topics.

Subscribe to Topic	Equivalent to Subscribing to
/A	/A/B /A/B/C

Publishing to any topic is equivalent to publishing to the topic itself and also all parent topics.

Publish to Topic	Equivalent to Publishing on
/A/B	/A/B /A /

9.3 API Reference

API for Pub/Sub should at least three pieces: #. Topic targeted #. Data structure for Pub/Sub event data #. Callbacks

In this package we extend above three into four by adding `service`, which could reduce subscription/publishing range by specifying a interested service from application. Pub/Sub event data is defined as the combination of data payload, data-id (to further identify a event data inside a topic). Data freshness period is also important, but can work with default settings and leave it there.

```
typedef struct ps_event {  
    const uint8_t* data_id;  
    uint32_t data_id_len;  
    const uint8_t* payload;  
    uint32_t payload_len;  
    uint32_t freshness_period;  
} ps_event_t;
```

Besides Pub/Sub event's data structure, APIs are provided as follows:

```
/** subscribe  
 * If is not cmd, this function will register a event that periodically send an  
 *   ↳ Interest to the name  
 * prefix and fetch data.  
 * Subscription Interest Format: /home-prefix/service/DATA/identifier[0,2],MustBeFresh,  
 *   ↳ CanBePrefix.  
 *  
 * If is cmd, this function will register a Interest filter /home-prefix/service/CMD_  
 *   ↳ and listen to  
 * notification on new CMD content.  
 * Once there is a comming notification and the identifiers is under subscribed_  
 *   ↳ identifiers, an  
 * Interest will be sent to fetch the new CMD.  
 * Cmd Notification Interest Format: /home/service/CMD/NOTIFY/identifier[0,2]/action  
 * Cmd fetching Interest Format: /home/service/CMD/identifier[0,2]/action  
 */  
void
```

(continues on next page)

(continued from previous page)

```
ps_subscribe_to_content(uint8_t service, const char* scope,
                        uint32_t interval, ps_on_content_published callback, void*
↳userdata);
```

The scope refers the *topic* discussed above. Inputs like "", "/kitchen" are welcome. *userdata refers to pointer to user-specified data structure which wanted to be thrown into the callback when triggered. Similarly, other APIs are provided as follows. *Content* and *Command* are treated separately also based on consideration of reduce the subscription/publishing range.

```
void
ps_subscribe_to_command(uint8_t service, const char* scope, ps_on_command_published_
↳callback, void* userdata);

/** publish data
 * This function will publish data to a content repo.
 * Data format: /home-prefix/service/DATA/my-identifiers/timestamp
 */
void
ps_publish_content(uint8_t service, const ps_event_t* event);

/** publish command to the target scope
 * This function will publish command to a content repo and send out a notification_
↳Interest.
 * Cmd Notification Interest Format: /home-prefix/service/NOTIFY/CMD/identifier[0,2]/
↳action
 * Data format: /home-prefix/service/CMD/my-identifiers/timestamp
 */
void
ps_publish_command(uint8_t service, const char* scope, const ps_event_t* event);
```


TRUST POLICY

Trust policy refers to policies that restrict the Data only be signed to few permissioned keys. Recall from the Pub/Sub section that every event is a semantically named Data, and this Data is signed by semantically named key. For instance, a valid Data may look like:

```
Data ---+----> Data Name: /ndn/alice/example/1234
      |
      +----> Data Content: integer value
      |
      +----> Data Signature --+----> Signature Info ---> Key Locator: /ndn/alice/KEY
                        |
                        +----> Signature Value: 11c6991...
```

The `/ndn/alice/KEY` is the signing key for Data named `/ndn/alice/example/1234`. Trust policy play the role here to regulate the signing relations so that only alice's key can sign the Data under she's prefix `/ndn/alice`. This policy check is implemented in the Pub/Sub module with several default options:

1. **Same room:** Data Name and Key Locator should indicate they're in the same room. For example, `/my-home/kitchen/meta` should be signed by `/my-home/kitchen/device-123/KEY`.
2. **Controller only:** Data should only be signed by home controller. This applies to scenarios where commands are critical to home security (e.g., unlock doors) For example: `/my-home/front/unlock` should be signed by `/my-home/controller/KEY`
3. **Same producer:** Data Name should exactly match Key Locator. For example: `/my-home/kitchen/device-123/version` should be signed by `/my-home/kitchen/device-123/KEY`. Notice that this policy has subtle difference from the same room policy, which usually applies to Datas whose Names not explicitly reveal their producers.

Pub/Sub module will perform different policy checks based on received data types. By default command Datas are checked against controller only policy, and other Datas are checked against same producer policy.

CHAPTER
ELEVEN

AUTHORS

- Tianyuan Yu
- Sichen Song
- Zhiyi Zhang
- Xinyu Ma

FUTURE PLANS

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`