THESIS

SUPPORTING LOCALIZED INTERACTIONS USING NAMED DATA NETWORKING

Submitted by

Andres Calderon Jaramillo

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2017

Master's Committee:

Advisor: Christos Papadopoulos

Wim Bohm
Stephen Hayne

Copyright by Andres Calderon Jaramillo 2017

All Rights Reserved

ABSTRACT

SUPPORTING LOCALIZED INTERACTIONS USING NAMED DATA NETWORKING

A common application in the Internet of Things (IoT) is the access to devices in a specific location. For example, a user may walk into a room and use a mobile device to control the lights or to access the temperature reading. Similarly, things in a location need to advertise their services. For example, when a printer is moved into a room, it needs to make its presence known so that users in that room can access it with minimal configuration.

An application developer can achieve these tasks by referring to devices using intuitive names such as `/csu/mainCampus/csBuilding/room258/printer/activate`. To construct such a name, the developer must make the application aware of the current location. Furthermore, the device must enforce a location-based access control policy to ensure that only users in the same location as the device are allowed to access the device.

Our goal is to design a system that leverages the power of names in the Named Data Networking architecture to allow application developers to write code to access and advertise services in a location such as a room or a building. Our system provides a convenient level of indirection so that developers can use names such as `/thisRoom/printers/default/activate` to initiate a spontaneous interaction with local devices. In this thesis, we describe the system architecture and a prototype implementation. Furthermore, we explore trust and security issues and qualitatively compare our NDN-based solution against an IP-based solution.

DEDICATION

I would like to dedicate this thesis to my parents and grandparents.

TABLE OF CONTENTS

ABSTRACT ii
DEDICATION iii

Chapter 1 Introduction 1

Chapter 2 Background 3
2.1 Named Data Networking (NDN) 3
2.1.1 Packets in NDN 4
2.1.2 Basic Protocol 6
2.1.3 Signatures 9
2.1.4 Routing 12
2.1.5 Benefits and Challenges 12
2.2 Related Work 13
2.2.1 Location Awareness 14
2.2.2 Spontaneous Interaction 18
2.2.3 NDN and the IoT 19

Chapter 3 System Architecture 22
3.1 Requirements 24
3.1.1 Device Types 24
3.1.2 Location-Based Access Control 25
3.1.3 Support for Relative Prefixes 25
3.1.4 Safe Advertisement of Services 27
3.1.5 Support for Identity-Based Trust Models 28
3.1.6 Encryption Support 28
3.2 Overview 28
3.3 Namespaces 30
3.4 Localization Channel 31
3.5 Translation Service 32
3.6 Name Translation Process 34
3.6.1 Interest Emitted from an Application in a User Device 36
3.6.2 Interest Arriving at a Service Device 37
3.6.3 Data Emitted from a Device Controller in a Service Device 38
3.6.4 Data Arriving to a User Device 40
3.7 Device Onboarding Protocol 41
3.7.1 Description 41
3.7.2 Service Discovery 48
3.7.3 Examples 49
3.8 Caching Considerations 51

Chapter 4 Trust and Security Considerations . . . 54
4.1 Overview . . . 54
4.2 Identities . . . 54
4.3 Trust Management . . . 55
4.3.1 In User Applications . . . 59
4.3.2 In LB-NDN . . . 63
4.3.3 In Device Controllers . . . 65
4.3.4 In Advertisement Controllers . . . 65
4.3.5 In NFD . . . 66
4.4 Publication of Certificates . . . 66
4.5 Replay Attacks . . . 68
4.6 Cache Poisoning . . . 68

Chapter 5 Implementation . . . 70
5.1 Components . . . 70
5.1.1 Broadcaster . . . 70
5.1.2 Translation Service . . . 71
5.1.3 Advertisement Controller . . . 72
5.1.4 Pending Features . . . 73
5.2 From the Developer's Point of View . . . 73
5.2.1 Writing a Device Controller . . . 74
5.2.2 Writing a User Application . . . 78

Chapter 6 Evaluation . . . 84
6.1 Device Types . . . 84
6.2 Location-Based Access Control . . . 85
6.3 Support for Relative Prefixes . . . 87
6.4 Safe Advertisement of Services . . . 87
6.5 Support for Identity-Based Trust Models . . . 89
6.6 Encryption Support . . . 89
6.7 Network Topology Considerations . . . 91

Chapter 7 Qualitative Comparison to IP . . . 94
7.1 Envisioning an IP-Based Design . . . 94
7.2 IP vs. NDN . . . 100

Chapter 8 Conclusions and Future Work . . . 103

Bibliography . . . 107

Appendix A Validator Configuration Files . . . 116
A.1 For User Applications . . . 116
A.1.1 Outer Packet Validation . . . 116
A.1.2 Inner Packet Validation . . . 117
A.2 For LB-NDN . . . 119
A.3 For Device Controllers . . . 121

A.4 For Advertisement Controllers 122
A.5 For NFD 125

Chapter 1

Introduction

Consider the following hypothetical scenario: Colorado State University hires a team of mobile application developers to implement an system that allows a user such as a student to access devices using his mobile phone depending on his location. For example, a student should be able to get the temperature reading from the current room's sensor or send a document to the current building's printer. If the student leaves a location, he should lose access to its devices. Furthermore, it is possible for devices to move among locations, so the group of services provided in a specific place may change often.

The team identifies three types of well-defined physical scopes in the university: rooms, buildings, and campuses. One of the members in the team then remembers the famous remark attributed to David J. Wheeler, "all problems in computer science can be solved by another level of indirection" [1]. Hence, to simplify development and allow for spontaneous interaction with local devices, the developers decide to add a level of indirection that allows their system to reference the current location by name using relative prefixes that suggest spatial references: `/thisRoom`, `/thisBuilding`, and `/thisCampus`. Hence, if a user wants to access the default printer in the current building, the system can reference it by a name such as `/thisBuilding/printers/default`. Since the system needs to be able to handle devices that move, the developers have decided to equip them with an interface that allows them to use the layer of indirection to advertise their services. Thus, a printer could indicate that it wants to provide printing services for the current building.

To support this added level of indirection, the team needs a supporting system that takes care of the low-level details. For example, an application-level message that expresses a name such as `/thisBuilding/printers/default` could be translated into a message that can be routed by the network so that it gets to the correct printer. Furthermore, the supporting system must enforce the rule that users be in the same location as the devices they are trying to access.

The imaginary team of developers has partnered with us to design this supporting system. In preliminary discussions, we have decided to explore the Named Data Networking (NDN) architecture [2] which we believe is particularly well suited for this problem. In NDN, there are two types of packets: *Interest* and *Data*. A consumer uses an *Interest* packet to request data by name. A producer replies to an *Interest* with a *Data* packet. The network routes *Interest* packets directly on names and keeps state in the routers to relay *Data* packets back to the consumers. Hence, we can use meaningful names at the network layer level.

The main contribution of this thesis is the design of a system based on NDN that supports spontaneous interactions with local devices in scenarios such as the one in our imaginary problem. We will refine the system requirements and describe the low-level details of message translation, location-based access control, device onboarding, service discovery, and trust and security issues. Additionally, we will discuss how our system benefits from using NDN as the underlying network architecture as opposed to using IP.

The rest of the thesis is organized as follows: in Chapter 2, we review NDN and explore related work. In Chapter 3, we formalize the notion of location, enumerate the system requirements, and describe our design. In Chapter 4, we explore trust and security issues in our system in detail. In Chapter 5, we describe the current prototype implementation of our system and discuss how developers can benefit from it. In Chapter 6, we investigate the extent to which our system meets the requirements and discuss the limitations. Finally, in Chapter 7, we examine the advantages and disadvantages of using NDN as opposed to IP as the underlying network architecture.

Chapter 2

Background

2.1 Named Data Networking (NDN)

NDN is a network architecture proposed as part of the NSF Future Internet Architecture (FIA) project [3]. The FIA program investigated alternatives to the way we do networking today. In IP, the network layer is in charge of carrying packets from one end point to another. The network is in charge of finding a path between the two endpoints identified by IP addresses. In contrast, in NDN, the network layer is in charge of satisfying a consumer's request for data. The network is in charge of figuring out how to retrieve the requested data.

A major motivation for the development of NDN is that the current Internet is increasingly being used to distribute content [2]. Consider, for example, two users, A and B. User A needs to read a sensor located at a specific location. User B needs to download the front page of his favorite newspaper. For user A, IP is well suited because the sensor can be given a globally unique IP address that allows it to be reached from anywhere and distinguishes it from other devices on the Internet. For user B, the end-to-end capability offered by IP is not particularly important since the front page of the newspaper can come from multiple sources: the newspaper's primary server, a secondary server, a cache, etc. To use IP, a communication channel must first be established between B's computer and one of the sources. Hence the philosophy of identifying endpoints in the network using IP addresses does not naturally fit the task at hand. Nonetheless, in today's Internet, users such as B are becoming more and more common. NDN attempts to address this limitation.

The goal in this section is to examine the NDN concepts necessary to understand our work. In Section 2.1.1, we discuss the types of packets in NDN. In Section 2.1.2, we describe the basic protocol that routers follow to carry packets between producers and consumers. In Section 2.1.3, we examine signatures and authenticity validation. In Section 2.1.4, we briefly discuss routing in

NDN. We conclude with Section 2.1.5 where we examine the general benefits of using NDN and the challenges still present in the architecture.

2.1.1 Packets in NDN

In NDN, there are two types of packets: *Interest* and *Data* [2].

Interest

A consumer emits an *Interest* to request data. This packet contains the following elements [4]:

- **Name [5]:** a hierarchical name that specifies the content being requested. Internally, it is a group of *name components*. Each name component is a sequence of bytes. We can represent names textually using a URI-like format, e.g., `/csu/mainCampus` has two components: `csu` and `mainCampus`. Name components have no real meaning from a network point of view [2]: it is up to applications to interpret names.

- **Selectors [4]:** they are used to restrict the content that can match an *Interest*. We have the following types of selectors:

  - **MinSuffixComponents and MaxSuffixComponents:** allows an application to specify how many components beyond the prefix a matching *Data* may have.
  - **PublisherPublicKeyLocator:** allows an application to specify who should be the signer of the requested content.
  - **Exclude:** allows an application to indicate that it does not want certain name components from appearing after the requested prefix in the matching *Data* packet.
  - **ChildSelector:** allows an application to indicate a preference when an *Interest* matches multiple *Data* packets in a content store. We will discuss the content store later.
  - **MustBeFresh:** allows an application to tell a router that it does not want a *Data* packet from the content store if the packet's *FreshnessPeriod* has expired.

- **Nonce [4]:** in combination with the name, it allows for the detection of loops.

- **Guiders [4]:** they are used to affect how an *Interest* is forwarded. The two currently defined guiders are *InterestLifetime* and *ForwardingHint*. Our system does not support the latter.

Only the name and the nonce are required. The rest of the elements are optional [4].

Data

A producer satisfies an *Interest* by generating a *Data* packet. This packet contains the following elements [6]:

- **Name [5]:** a name for the content represented by the *Data* packet. Note that if an *Interest* does not specify selectors, the name of the *Data* packet can be longer than that in the *Interest*. For example, if the name in the *Interest* is `/csu/hello`, the name in the *Data* packet could be `/csu/hello/world`. Selectors can be used to control this behavior.
- **MetaInfo [6]:**
  - **ContentType [6]:** specifies the type of the content carried by the packet. For example, *BLOB* is a generic type and *KEY* represents a public key.
  - **FreshnessPeriod [6]:** allows a producer to specify the amount of time a packet is considered fresh in the content store. When a packet arrives at the content store, a timer counts down from the amount specified here. When it expires, the packet is marked stale and should not be used to satisfy *Interest* packets that demand a fresh packet.
  - **FinalBlockId [6]:** a name component that identifies the last fragment in a sequence so that consumers know when they have received everything.
- **Content [6]:** a sequence of bytes representing the payload.
- **Signature [7]:** a block that encodes signature information and the actual signature value for the packet. The signature information includes elements such as the signature type and the *KeyLocator* which allows a consumer to locate the certificate used to sign the packet.

The three elements in the *MetaInfo* are optional.

2.1.2 Basic Protocol

In NDN, we must be able to forward packets based on names. An NDN forwarder must keep track of the following structures [2]:

- **Pending Interest Table (PIT):** to keep track of forwarded *Interest* packets that have not been satisfied [2].
- **Forwarding Information Base (FIB):** to maintain enough information to forward *Interest* packets [8].
- **Content Store (CS):** to cache *Data* packets [8].

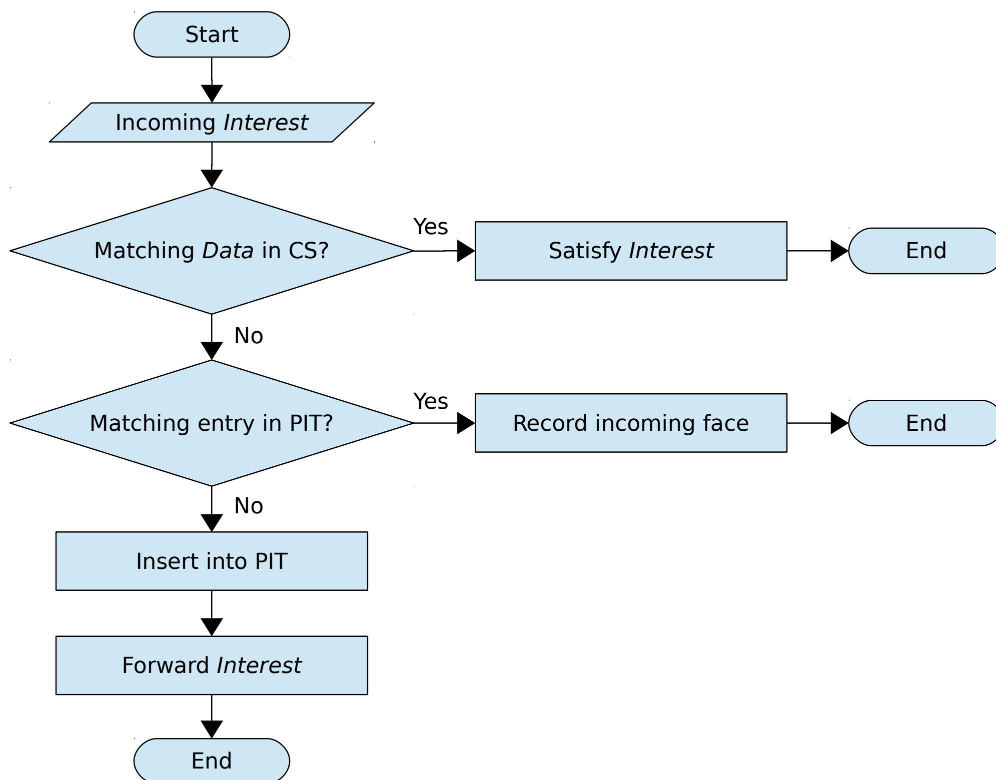

Figure 2.1: NDN protocol on an incoming *Interest*, based on Figure 3 in [2]

In NDN, we use the term *face* to refer to the port of entry or exit for packets. Faces generalize network interfaces [8]. They can represent interfaces to physical links, channels of communication between the forwarder and local applications, or overlay channels to remote nodes [8] (for example, to run NDN over TCP/IP).

The following occurs when an *Interest* is received by the forwarder [2, 8] (summarized in Figure 2.1):

1. If a matching *Data* packet is found in the content store, the *Interest* is satisfied. Otherwise, we continue to the next step.

2. We create a PIT entry for the *Interest*. If there is already a matching *Interest* in the PIT (e.g., if a request for the same content came from different faces), we simply add a reference to the incoming face to the existing entry and stop processing. Otherwise, we proceed to the next step.

Table 2.1: Example FIB

Prefix	Next Hop Faces
`/csu`	1, 2
`/csu/mainCampus`	3, 4, 5

3. The *Interest* is forwarded based on the information in the FIB and according to a *forwarding strategy*. Lookup in the FIB is done using a longest prefix match algorithm. For example, consider the FIB in Table 2.1 which shows for each prefix the IDs of the faces on which a matching *Interest* could be sent. Suppose the *Interest* in question has name `/csu/mainCampus/csBuilding/hello`. A longest-prefix match would yield the second entry. Hence, we need to decide on which of the three faces (3, 4, or 5). This decision is made by the forwarding strategy. For example, a best route strategy can choose the next hop with the

lowest routing cost [8]. Another strategy, such as the multicast strategy [8], might choose to forward the packet on multiple faces. The forwarding strategy can also decide when to forward the packet [2].

When a *Data* packet is received by the forwarder, the following will happen [2,8] (summarized in Figure 2.2):

1. If a matching entry is found in the PIT, the packet is sent through all the faces listed in the entry. Otherwise, the packet can be discarded.

2. The PIT entry is removed and the *Data* packet is added to the content store.

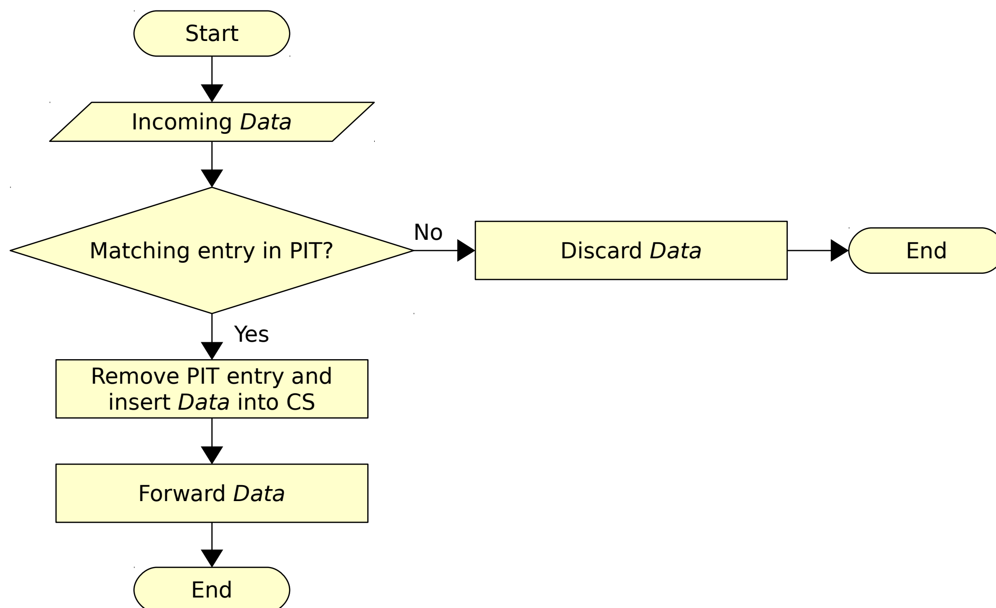

Figure 2.2: NDN protocol on an incoming *Data*, based on Figure 3 in [2]

One highlight of the protocol is that NDN packets do not carry endpoint identifiers as IP packets do. An *Interest* finds its way to a content producer by following the next hop in each router assuming that a routing protocol has previously populated the FIB. In order for a *Data* packet to find its way to the consumer, it uses the state in the PITs of the intermediate routers (we can think of *Interest* packets as leaving "breadcrumbs" [9] at each router so that the reply *Data* can get to the

consumer). This implies that a *Data* packet follows the reverse path of the matching *Interest* [2]. This also means that one *Interest* results in at most one *Data* packet per link [2].

The protocol is currently implemented by the Named Data Networking Forwarding Daemon (NFD) [10] which is supposed to run in every node in an NDN network. One notable difference with the basic protocol is that, for performance reasons, NFD creates a PIT entry before looking up a matching packet in the content store [8]. Another difference is that the PIT also contains recently satisfied *Interest* packets to help with measurements and to detect loops [8]. NFD allows one to set the `CHILD_INHERIT` flag on a route so that it can be used even when it is possible to match another route with a longer prefix [11].

In NDN, we do not need a separate transport layer such as TCP [2]. Demultiplexing is done by names instead of port numbers. Reliable delivery can be achieved by the application retransmitting *Interest* packets that timeout (as defined by the *InterestLifetime* guider). Congestion control can be achieved on a hop-by-hop basis because nodes can decide how fast to transmit *Interest* packets (since every *Interest* produces at most one *Data*) [12].

2.1.3 Signatures

In NDN, a digital signature binds content to a name. Every valid *Data* packet must contain a signature. This allow consumers to validate them for integrity and authenticity regardless of where they came from [2].

Recall that a *Data* packet includes information about the signature including the *KeyLocator* which allows consumers to obtain the public key needed to validate the signature [7]. The *KeyLocator* can be either 1) a digest to identify the public key or 2) a name that references a *Data* packet that contains the certificate or public key [7]. In our work, we use the latter *KeyLocator* type. Hence, a consumer can issue an *Interest* whose name is *KeyLocator* in order to retrieve the certificate.

We use the naming convention for certificates detailed in [13]. Essentially, we group keys under an *identity* represented by a namespace. For example, there could be an identity for a

Colorado State University certificate authority represented by the prefix `/csu/ca`. An identity can own multiple keys, and each one has its own name. For example, our certificate authority can have the following keys: `/csu/ca/ksk-12345` and `/csu/ca/ksk-12347` where `12345` and `12347` are timestamps. The `ksk` portion stands for *Key-Signing-Key*: keys with long lifetimes. Another type of key is *Data-Signing-Keys* (`dsk`): keys with short lifetimes [13].

Public keys can be distributed in the form of certificates to bind them to their corresponding name [13]. A certificate's name includes two special components: `KEY` and `ID-CERT`. Additionally, the last component of the name is a version. Thus, a certificate for the `/csu/ca/ksk-12345` private key can have the name `/csu/ca/KEY/ksk-12345/ID-CERT/%FD%02`. A certificate can be self-signed or it can be signed by a third party that vouches for the binding between the public key and its name [13].

The signature information also includes the type of signature. Most often, we will use RSA signatures with SHA-256 digests [7].

Even though signatures are required by NDN at the network layer, it is up to the application to enforce meaningful trust rules. To do this, it must be able to enforce a relationship between the name of a packet and the name of the signer. This can be achieved by the use of *trust schemas* [14]. A trust schema can be represented with a configuration file that defines the *trust rules* in the schema [15]. For example, consider the following configuration file:

```
validator
{
  rule
  {
    id "Example rule"
    for data
    filter
    {
      type name
      regex ^<myApp><><>+$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
```

```
        type name
        regex ^<localhost><myApp><KEY><ksk-.+><ID-CERT>$
      }
    }
  }

  trust-anchor
  {
    type file
    file-name "_localhost_myApp.ndncert"
  }
}
```

This file is expressing one trust rule: any *Data* packet whose name starts with `/myApp` followed by at least two components must be signed using a certificate whose name is of the form `/localhost/myApp/KEY/ksk-.+/ID-CERT` (note the use of regular expressions). Furthermore, the type of the signature must be RSA over a SHA-256 digest. This certificate has been set up as a *trust anchor* so that the consumer does not have to retrieve it from the network. Using configuration files, it is possible to express complex trust relationships. We make use of trust schemas in Chapter 4.

NDN also supports signed *Interest* packets. This is done by appending four components to the name of the *Interest* [16]. Thus, we could express a signed *Interest* with name

```
/csu/mainCampus/shutdown/<timestamp>/<nonce>/<signature-info>/<signature-value>
```

The `<timestamp>` and `<nonce>` components help in preventing replay attacks. The `<signature-info>` (for signature information) and `<signature-value>` components have the same roles as in a *Data* packet. We will be using signed *Interest* packets to support specific access control rules.

At the time of this writing, there is a new version of the specification for certificates [17] and trust schema configuration files [18]. The supporting library that we use, *jNDN* [19], does not seem to have implemented these specifications yet. Hence, we will continue with the specifications mentioned previously which have been marked deprecated [20]. Nonetheless, this should be no more than an implementation detail and the ideas presented in this work should remain valid.

2.1.4 Routing

The routing protocol is in charge of populating the FIB in the forwarder. The same routing algorithms used in IP can be adapted to be used in NDN. The difference is that in NDN, we advertise name prefixes instead of IP prefixes [2]. A prominent example is NLSR [21], a link state routing protocol for NDN. NLSR propagates prefix and adjacency information. For example, a router operator can specify what prefixes a router is able to handle. NLSR then uses *Interest* and *Data* packets with a dataset synchronization protocol (ChronoSync [22]) to propagate advertisements. Furthermore, the propagation of this information takes advantage of signatures to make sure advertisements originate from authorized routers.

2.1.5 Benefits and Challenges

Researchers have identified several aspects that make NDN an attractive architecture. Among them are:

- Names simplify application development because the need to have a mapping of "application configuration to network configuration" is reduced [2].

- NDN naturally supports multicasting because routers aggregate *Interest* packets by using the PIT [2].

- Since one *Interest* results in at most one *Data* packet per link [2], NDN achieves "hop-by-hop flow balance" [12].

- The name and the signature allows a *Data* packet to be self-contained. Since the packet does not contain endpoint identifiers, it can be used to satisfy the requests of multiple consumers. Therefore, unlike the case with IP, it makes sense to cache *Data* packets in the content store [2].

- It is harder to target particular network endpoints in an attack because of the absence of endpoint identifiers in *Interest* and *Data* packets [2].

- Unlike in IP, endpoints are not responsible for congestion control [12]. If congestion occurs and an *Interest* has to be retransmitted, it may not have to go all the way to the producer since it is possible that there is a cached version of the *Data* closer to the consumer [12].

Nonetheless, NDN has its own challenges currently under investigation by the community. Among them are:

- It is more difficult to scale routing based on names than routing based on IP addresses [12]. This problem is improved by the use of hierarchical namespaces [12].

- Even though it is harder to target specific network endpoints, there are still NDN-specific attacks. For example, *Interest flooding* attacks attempt to saturate routers and deny service to legitimate consumers [23]. This is analogous to DDoS attacks. The state stored in routers and the fact that a *Data* packet follows the reverse path of an *Interest* can be used to mitigate this [24]. Another attack is *content poisoning* in which a malicious producer injects bad *Data* packets in a router's cache [25]. Consumers can still detect this situation because of digital signatures, but they cannot do much about it on their own because the router's cache is poisoned.

- Devices that are constrained in terms of memory, computational capabilities, and power usage may find it difficult to implement NDN's stateful *Data* plane and signature verification schemes [26].

- It is not trivial for developers to specify trust relationships to verify signatures [14]. Trust schemas help with this aspect.

2.2 Related Work

Our work fits in the broad areas of ubiquitous and context-aware computing. Ubiquitous computing is the vision of letting computers "disappear into the background" [27]. Mark Weiser from Xerox PARC first coined this term in 1988 [28]. An early attempt to realize this concept

of integrating technology into everyday life was the *Active Badge* system: a tag a person wears to allow an infrastructure to track them so that other people can find their location and transfer phone calls to the right place [29]. Since then, we have started to see the idea of ubiquitous computing realized to an even greater extent. One of the main examples of this phenomenon is the smartphone [28]. An important concept in ubiquitous computing is the idea of *context awareness*: devices adapting their behaviors to the context in which they find themselves [30]. The context of a device may include aspects such as "location of use, the collection of nearby people, hosts, and accessible devices" [30].

In the following subsections, we will explore related work in the three main aspects of our research: location awareness, spontaneous interaction, and NDN applied to the Internet of Things (IoT).

2.2.1 Location Awareness

Among the different parameters of a device's context, location has been recognized as being particularly important [27, 31]. A core idea in our work is that of *location-based access control (LBAC)*.

In LBAC, access to devices or services is restricted based on the location of the requestor and/or of the object being requested. In [32], the authors extend a generic access control model to consider location information. Their work is able to deal with uncertainty as well as time-dependency in location measurements. In [33], the authors describe LRBAC, a formal model that extends role-based access control (RBAC) to take into account location information.

Closer to our line of work, but still related to LBAC, is the problem of *location verification* [34] in which it is necessary to check the location of a requestor against some criteria. In [35], the authors introduced *distance bounding protocols* which allow a party to obtain an upper bound on the physical distance to another party. The basic principle is to time the exchange of a challenge bit between the two parties. In [34], the authors present the *Echo* protocol to securely verify location claims. In this protocol, a prover p makes the claim to a verifier v about being in a certain region.

The verifier must decide whether it should accept the claim. To do so, v sends a nonce to p over an RF channel. Then, p replies with the nonce over an ultrasound channel. The verifier then uses the time elapsed since it sent the nonce to make its decision. The authors iteratively extend the protocol to account for processing delay, packet transmission time, and non-circular regions. Another example of a location verification system is described in [36]. This approach uses radio broadcasts. The idea is to have two types of sensors: verifiers that can receive a signal from a prover if the prover is inside a protected area, and rejectors that can receive the signal if the prover is outside the protected area.

Our approach to location verification is closer to the ones described in [37] and [31]. In [37], the authors consider a limited-range beacon that sends a token containing an encrypted timestamp. A device then uses this token as a cookie in future requests to prove its location. When the timestamp in the token has passed, the device must obtain another token to continue performing requests. In [31], the authors present the abstractions of *send-constrained* and *received-constrained* channels of communication and use them to describe three variants of a location authentication protocol based on a challenge-response mechanism.

Another idea related to our approach is presented in [38]. In this work, the authors describe an approach to bootstrapping trust between two devices that can exchange pre-authentication data over a *location-limited channel*. For example, a user can touch a printer in front of him with his device to exchange pre-authentication data. Then, the user device and the printer can continue communicating over a wireless channel and the user can be confident that his device is talking to the right printer [38]. The authors also extend the protocol to work for a group of devices so that they can obtain a group key. In their approach, devices send data over the location-limited channel. In contrast, we only require that devices be able to hear over the channel to listen for a secret that all the devices in a physical space share. The idea of establishing a shared secret over a location-limited channel is also seen in [39]. The idea of transmitting location information through a beacon is also discussed in [40].

It is also possible to carry out all communication over channels that are physically restricted to a desired range in order to enforce LBAC. In [41], the authors implement a network stack with a physical layer based on sound with a center frequency of 19 kHz. The rest of the network stack uses common protocols such as IP. Since sound is naturally limited to room boundaries, the authors were able to implement their idea of *room-area networks (RANs)*. Another system that uses an acoustic medium is *Sonicnet.js*, a JavaScript library for sending and receiving data over sound [42]. The *Quiet Modem Project* [43] is another group of libraries for achieving the same task. One problem with restricting all communication to such physically restricted channels is throughput. For example, the authors in [41] report that the physical layer implementation achieved 8100 bps in its fastest configuration. As the authors acknowledge, "[o]ff-loading traffic onto RF technologies after service discovery through the RAN will be desirable" [41]. Hence, in our approach, we only use physically restricted channels to distribute a shared secret that can be used to authenticate over a more robust channel.

Another area that is related to our work and falls within the umbrella of location awareness is that of *location-based services (LBS)*: services that "[take] into account the geographic location of an entity" [44]. Within this field, a work that is particularly relevant to this thesis is the *AROUND* architecture [45–47], a general model for supporting location-based services. Our work shares many similarities with this architecture. Among them are:

- Our concept of location is very similar to the concept of *location contexts* in the *AROUND* architecture. In fact, the authors define a location context as "a symbolic representation of an area in physical space that can be explicitly referenced by a global name and can be used across multiple networks and domains as a context for service discovery" [47].

- Location contexts can be associated with a type (e.g., a building) in the *AROUND* architecture [46]. We also use the idea of typed locations.

- The authors consider two models for the selection of services: distance-based and scope-based [47]. In the former, the concept of proximity is based on physical distance. In the latter,

proximity is based on a region of space that defines the scope of a service. The *AROUND* architecture aims to support both models, while our work is closer to a scope-based model.

- Servers in the *AROUND* architecture specify the location context to define the scope of their services [47]. We also require this in our work.

- The *AROUND* architecture identifies each location context using a globally unique URN. This is similar in spirit to the absolute NDN namespace used in our work (Section 3.3).

- The authors refer to a *contextualization process* that maps physical locations to location contexts [47]. This functionality is provided in our system by the localization channel (Section 3.4).

- The authors use *service types* [46]. This idea roughly corresponds to our concept of device categories in this thesis (Section 3.1.4).

Hence, to an extent, our work realizes many aspects of the *AROUND* model with NDN as the underlying network architecture. An interesting difference is the presence of a *name service* in the *AROUND* architecture which resolves location context names into references to servers that can inform applications about available services. This allows the architecture to have multiple servers handling the same location context to improve availability. In our work, the service that maintains information about available services can be located directly and there is only one such service per location.

Additionally, in the *AROUND* architecture, a query for available services may traverse multiple servers because the model explicitly accounts for containment and adjacency relationships among location contexts [47]. Our system neither assumes nor excludes containment relationships. Hence, if an application wants to know about nearby services in multiple locations that implement a containment relationship, it will need to query the services responsible for each location individually. We do not consider adjacency relationships.

2.2.2 Spontaneous Interaction

As described in [48], "[t]he principal idea of spontaneous interaction is to enable mobile users to associate their personal devices with devices encountered in their environment." Here are some examples of this idea from the literature:

- In [49], the authors propose augmenting physical objects with RFID tags to facilitate interaction with them. One of their proposed sample applications is a book equipped with a tag. When the tag is detected by a device, a document related to the book can be shown on the device. The authors also equipped rooms with infrared beacons that transmit room information to give context to devices. This is similar in spirit to our localization channel described in Section 3.4.

- In [50], the authors describe the *RAUM* network architecture that supports localized communication using packets called *events*. In one of the application examples, the authors use this system to allow a Palm Pilot to receive keyboard events from a nearby keyboard.

- In [51], the authors describe a web-based system to support *nomadic users* in the context of HP's *CoolTown* project. Nomadic users are mobile and expect to interact with electronic resources as they encounter them.

- In [48], the authors present the *RELATE* model which allows users to interact with nearby devices using relative spatial references. For example, in one of the applications of the model, a user device allows its owner to interact with other users' devices in a meeting room by displaying a graphical interface that shows the relative position of the other devices. In related work [52], a method is described to secure spontaneous interaction by using spatial references. In this thesis, we also use the idea of spatial references. However, our relative references (such as `thisRoom` and `thisBuilding`) are more abstract and may have a broader physical scope, such as an entire university campus.

In general, we can argue that the concept of spontaneous interaction is closely related to location awareness. Indeed, our system aims to support the development of spontaneous interaction

applications such that the concept of proximity is based on scope-based models [47] as explained previously.

2.2.3 NDN and the IoT

"The Internet of Things (IoT) is a vision for interconnecting all of the world's 'things' (...) through a common set of networking technologies" [26]. The IoT, being a manifestation of ubiquitous computing, is an active area of research. The NDN community is currently investigating how the IoT can benefit from the NDN architecture.

In [26], the authors make a case for why the current Internet architecture, IP, is not particularly suited to support the IoT. They identify the following challenges:

- The current solutions to the communication needs of IoT devices are complex and require the creation of an overlay on top of IP in order to support devices with diverse networking technologies.

- Channel- and session-based security may not be sufficient. It is also difficult to implement as each networking stack used by different communication technologies may have different security solutions. The cloud is often used to centralize security issues.

- Realizing an IoT framework using only local communication may require a difficult setup. Cloud services are often used to support IoT infrastructures even if they are local in nature.

The authors also explain how NDN can support the needs of the IoT: bootstrapping, service discovery, trust, control of access to data through encryption, data aggregation for analysis, publish-subscribe communication, dataset synchronization, and integration of local and global communication. A number of projects are given as examples of NDN applied to the IoT. Among them are:

- **NDN-BMS [53]:** a building management system focused on publishing sensor data.

- **NDN-IoTT [54]:** a framework for building simple IoT applications. It supports bootstrapping and discovery [26]. The controller node in this work is similar in spirit to our

advertisement controller (Section 3.7). Additionally, the idea of providing *keywords* in the definition of commands supported by nodes resembles our concept of device categories (Section 3.1.4).

- **NDN-ACE [55]:** a protocol that provides access control for actuators. It runs on constrained devices by delegating some tasks to a more powerful entity.

Furthermore, [26] mentions a number of challenges for realizing the IoT vision over NDN:

- Support for multiple naming hierarchies so that the same content can be organized in different ways
- Support for infrastructure-less environments that do not run routing protocols
- Support for constrained devices
- Support for push-style communication

In [56], existing cloud-centric IoT solutions are explored in more detail. The authors design a home entertainment application to show how it is possible to design NDN-based IoT systems that do not depend on the cloud. They recognized trust management and rendezvous as the two foundational blocks of IoT solutions such that other services can be implemented on top of them: cloud-based solutions simplify these tasks for users, but they fail to take full advantage of local communication. The authors propose a general NDN-based framework called *NDN-IoT* that realizes trust management and rendezvous using local communication without necessarily sacrificing usability while still leaving the option to use cloud services when needed. Our system also follows the same philosophy of keeping communication local. Similar to our advertisement controller, the *authentication server* in their work provides trust management for the device onboarding process.

Due to the use of meaningful and hierarchical names, location information can be naturally incorporated in applications built with NDN. For example, in the building management system

mentioned previously, the authors use the physical structure of a building to define the namespace [53]. Location information has also been used in ad-hoc environments. For example, in [57] the authors apply NDN to vehicular networks. The location of the sender is included in an *Interest* in order to decide which vehicle in the vicinity of another should be responsible for forwarding the *Interest* (this idea is also seen in [58]). In follow-up work [59], the authors introduce *GeoFaces* which further use geographical location to improve the forwarding of *Interest* packets. Ad hoc networking and node mobility has also been examined in other work (e.g., [60] and [61]).

Finally, note that context-dependent names have been conceived before (e.g., `/ThisRoom/Printer` in [62] or `/ThisRoom/projector` in [63]). One of our goals is to support this type of names for a given infrastructure.

Chapter 3

System Architecture

Our first task shall be to formalize a defining aspect of our work: location. To do so, let us start with the following definition taken verbatim from [33]:

Definition 3.1. A *location* Loc_i is a non-empty set of points $\{p_i, p_j, \ldots, p_n\}$ where a point p_k is represented by three co-ordinates.

Next, we will formalize the intuition of an infrastructure with well-defined physical locations for the purposes of our work:

Definition 3.2. An *infrastructure* is an entity consisting of the following components:

1. A non-empty set of locations $Locs = \{Loc_1, Loc_2, \ldots, Loc_m\}$.

2. A non-empty set $LocTypes = \{LT_1, LT_2, \ldots, LT_n\}$ where each element is a *location type*, a meaningful label that can be assigned to a location (similar to [47]).

3. A non-empty set $AbsPrefixes = \{AP_1, AP_2, \ldots, AP_m\}$ where each element is an *absolute prefix*, a network name that can be assigned to a location.

4. A non-empty set $RelPrefixes = \{RP_1, RP_2, \ldots, RP_n\}$ where each element is a *relative prefix*, a meaningful name that can be assigned to a location type.

5. A non-empty set of NDN routers $Routers = \{R_1, R_2, \ldots, R_p\}$.

6. A *location classification function* $TypeOf : Locs \rightarrow LocTypes$ that assigns a location type to a location. This function and the $Locs$ and $LocTypes$ sets must be defined in such a way that $\forall Loc_i, Loc_j \in L \left[TypeOf(Loc_i) = TypeOf(Loc_j) \implies Loc_i \cap Loc_j = \emptyset\right]$.

7. A one-to-one *relative naming function* $RelPrefixOf : LocTypes \rightarrow RelPrefixes$ that assigns a relative prefix to each location type.

8. A one-to-one *absolute naming function* $AbsPrefixOf : Locs \rightarrow AbsPrefixes$ that assigns an absolute prefix to each location.

9. A *router assignment function* $RouterOf : Locs \rightarrow Routers$ that assigns a router to each location.

Let us use our imaginary scenario described in Chapter 1 as an example. The infrastructure is Colorado State University. Below are the components of this infrastructure:

1. The university has many locations: for example, Loc_1 could be the set of 3D points enclosed by room 258 in the computer science building of the main campus; Loc_2 could correspond to a different room; Loc_3 could be the space enclosed by the computer science building.

2. $LocTypes$ was previously identified by the team:

   $LocTypes = \{\text{room}, \text{building}, \text{campus}\}$

3. $AbsPrefixes$ is flexible, but an example of one element could be `/csu/mainCampus` which is expected to identify the main campus location.

4. $RelPrefixes$ was previously identified by the team:

   $RelPrefixes = \{\texttt{/thisRoom}, \texttt{/thisBuilding}, \texttt{/thisCampus}\}$

5. $Routers$ is flexible. For example, there could be one router for each campus or one for each building.

6. The location classification function is straightforward to define because it is easy for a human to classify the different locations in a university according to the location types. The property described in the definition for this function ensures that two locations of the same type do not overlap. This makes intuitive sense. For example, rooms should have well-defined boundaries that prevent them from overlapping with other rooms.

7. The relative naming function was defined by the team:

   $RelPrefixOf = \{(\text{room}, \texttt{/thisRoom}), (\text{building}, \texttt{/thisBuilding}), (\text{campus}, \texttt{/thisCampus})\}$

8. The absolute naming function is straightforward to define. An example is the mapping that assigns the `/csu/mainCampus` prefix to the location enclosed by the main campus.

9. The router assignment function is flexible. For example, there could be one router per campus which is assigned to all the locations in that campus.

Definition 3.2 is minimal in the sense that it is enough for us to design our system. However, an organization can extend this definition to include more specific restrictions. For example, we may want to define the infrastructure in such a way that each room is contained by a building and each building is contained by a campus. This would capture the idea of a hierarchical space.

Equipped with a formalization of location, we will now proceed to define the scope of our system in terms of a set of requirements. From hereon, we will use the dot notation to refer to the different components of an infrastructure, e.g., $I.RelPrefixes$ represents the set of relative prefixes defined by infrastructure I.

3.1 Requirements

After lengthy discussions with imaginary university officials, we and the imaginary team of mobile developers were able to define the following formal requirements for the supporting system given an infrastructure I and devices in it. The motivation for each requirement is illustrated through an example use case.

3.1.1 Device Types

The system should distinguish between two types of devices: *user devices* and *service devices*. User devices run one or more applications that request data or actions from service devices. Each service device runs a device controller which is in charge of handling the requests issued by user applications. This device controller does not need to be provided by the manufacturer. It could be an adapter software that one of our mobile developers writes that allows the device to become part of the infrastructure. Devices should be able to switch roles.

Example

A professor enters a classroom as he gets emotionally ready to proctor a particularly hard exam. Upon entering the classroom, he opens an application in his phone that allows him to turn on the projector in the current room so that he can show information about the test (such as the time remaining). In this case, his phone is a user device and the projector is a service device. Students are to take the test using their laptops. The professor has the exam stored in his phone and wishes to distribute it to the students in the classroom. In this case, the professor's phone switches roles to become a service device that accepts requests for data from laptops (user devices) in the room.

3.1.2 Location-Based Access Control

The supporting system is in charge of enforcing the following rules:

- A service device accepts a request from a user device only if both devices are in the same location (see Definition 3.1).
- A service device can advertise its services in a location only if it is in that location.

Example

For the first rule, let us refer to the example in the previous requirement: the professor cannot turn on the projector using his mobile device until he is physically in the classroom. For the second rule, consider a student wanting to print a document in the building's default printer. A printer outside the building should not be able to offer its services to this student.

3.1.3 Support for Relative Prefixes

Recall that the team of developers decided to add a level of indirection so that their applications running on user devices can refer to the current location using relative prefixes such as `/thisRoom` or `/thisBuilding`. Additionally, the device controller running in a service device needs to be able to use relative prefixes to advertise its services and accept requests from user devices. Essentially, relative prefixes allow user applications and device controllers to communicate with each other

without knowing the addressing namespace of the underlying network. The supporting system is responsible for the low-level details of transforming messages containing relative prefixes to messages with absolute prefixes and vice versa.

Definition 3.2 provides us with the tools to formalize this problem. Let p_d represent the position of a user device (a 3D point). Suppose an application in the device issues a message with a relative prefix $RP \in I.RelPrefixes$. We must find an absolute prefix $AP \in I.AbsPrefixes$ such that $[AP = I.AbsPrefixOf(Loc)] \wedge [p_d \in Loc] \wedge [Loc \in I.Locs] \wedge [I.TypeOf(Loc) = I.RelPrefixOf^{-1}(RP)]$. If such absolute prefix cannot be found, it means that either the device's position cannot be associated with an infrastructure location or the location with which it is associated is not of the type implied by RP. At most, one absolute prefix corresponding to a relative prefix can be found. This is because of the property in Definition 3.2 that prevents locations of the same type from overlapping.

The system needs to be able to do the converse. Let us suppose the system gets a message with an absolute prefix $AP \in I.AbsPrefixes$. If p_d is the position of the service device getting the message, we must find a relative prefix $RP \in I.RelPrefixes$ such that $[RP = I.RelPrefixOf(I.TypeOf(Loc))] \wedge [p_d \in Loc] \wedge [Loc = I.AbsPrefixOf^{-1}(AP)]$. If the relative prefix is not found, it is because the device is not in the location implied by AP.

Example

Suppose a student wants to send a document to the printer in the current room which happens to be room 258 of the computer science building in the main campus. The printer's controller has advertised its presence using the `/thisRoom` prefix. The user application in the student's device refers to the current room using `/thisRoom`. This relative prefix is to be translated into an absolute prefix that the network can understand such as `/csu/mainCampus/csBuilding/rm258`. When the printer receives a message with this absolute prefix, it must be translated back into `/thisRoom` so that the printer's controller can accept the request since it used this prefix to advertise its services.

3.1.4 Safe Advertisement of Services

There should be two advertisement policies: open policies in which a service device can advertise its services freely and restricted policies in which a service device claiming to offer a well-known service (such as printing) must prove that it can really provide that service.

To support this, the team has decided to adopt the idea of *service device categories*, e.g., printers, temperature sensors, lights, etc. In an open policy, a device can advertise under the special category named `open`. In a restricted policy, it is assumed that each device has an identity in the form of a certificate. By following a chain of trust, we can verify that a device belongs to a specific category.

In general, we envision that the set of service categories is standardized by an industry consortium. Then, manufacturers can equip each device with a certificate that allows it to prove its membership in a category. However, in the absence of this, our mobile developers can manually equip each device controller with a certificate that makes the device part of an infrastructure-specific set of device categories.

Example

A scenario where we may want to use an open policy is if a professor wants to distribute lecture slides from her phone to the students in the classroom without possessing a manufacturer device certificate. Her phone is assigned a name by an onboarding protocol. She can then tell the class the assigned name and students can retrieve the slides using that name.

An example where we could use a restricted policy is a projector. Suppose a professor would like to show the answer key to a homework using the room's projector. If there is a malicious user in the room whose phone claims to be a projector and the network accepts the phone's advertisement, the answer key could arrive to this malicious device instead of a real projector.

3.1.5 Support for Identity-Based Trust Models

On top of the location-based access control requirement, it should be possible to implement policies that regulate the access of a user device to a service device based on the identity of the requestor.

Example

A classroom projector may demand that in order to be activated, the user must be a professor (in addition to being physically in the room).

3.1.6 Encryption Support

The system should allow user devices to encrypt their requests destined to service devices. It should also allow service devices to encrypt their responses.

Example

A professor would like to make sure that the document he is sending to the printer cannot be leaked to an attacker sniffing the network. In a different scenario, an authorized administrator would like to make sure that the video stream she obtains from a room's security camera cannot be sniffed on the way to her mobile device.

3.2 Overview

In the rest of this chapter, we will describe the main components of our system. We will often refer to the infrastructure shown in Figure 3.1. This figure corresponds to a simplified and hypothetical Colorado State University infrastructure. There are seven locations: two campuses, three buildings, and two rooms. Each location has a corresponding absolute prefix. There are eight routers, each one represented by a blue box and labeled with the prefix it advertises. Each location is assigned a different router. Having a physical router per location is convenient for explanation purposes, but as we will see in Section 6.7, our system is flexible in terms of the underlying network topology.

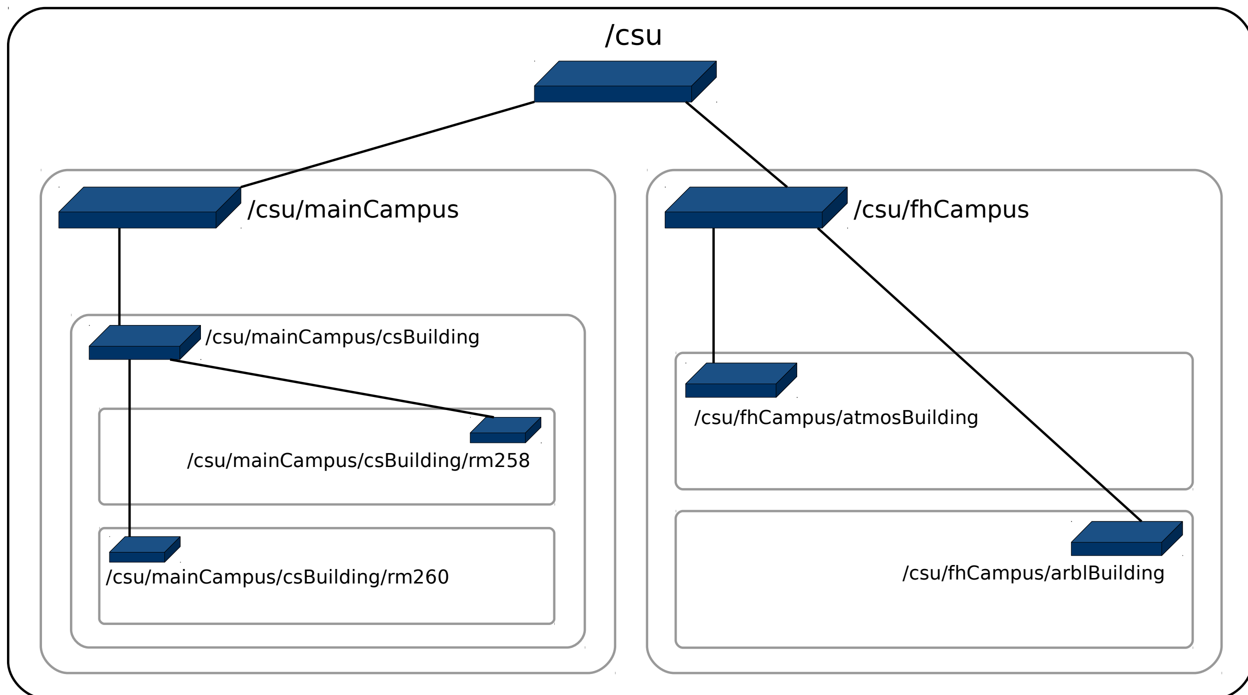

Figure 3.1: Infrastructure of the example scenario

As explained in Chapter 1, our design will use the NDN architecture. This will allow devices to refer to each other using intuitive names at the network level. When designing an NDN-based system, it is very important to define the naming scheme used by the different components. Hence, in Section 3.3, we introduce the main namespaces used by our system. Then, in Section 3.4, we describe the mechanism that will allow our system to enforce the location-based access control rule. In Section 3.5 and Section 3.6, we will describe the mechanism that translates messages with relative prefixes to messages with absolute prefixes and vice versa. Then, in Section 3.7, we discuss the mechanism that service devices can use to advertise their services in a location. Finally, in Section 3.8, we explore some considerations related to in-network caching.

Throughout this chapter, we will often refer to digital signatures. However, we will defer most of the trust and security details (such as identity names and trust hierarchies) until Chapter 4.

3.3 Namespaces

The use of relative and absolute prefixes defines two namespaces. Names using relative prefixes are said to be in the *relative namespace*. Such names also contain *service-dependent components* which are all the components after the relative prefix. For example, the name `/thisRoom/printers/p1/activate` consists of the `/thisRoom` relative prefix and the `printers/p1/activate` service-dependent components.

In general, the naming convention to access a service device using the relative namespace is as follows:

`/<rel-prefix>/<category>/<name>/<request-components>`

Where

- `<rel-prefix>` is a relative prefix.
- `<category>` is an element of a standardized set of service device categories as described in Section 3.1.4.
- `<name>` is the name assigned to the device which is unique within a category. More details about how this name assignment occurs are given in Section 3.7.
- `<request-components>` corresponds to one or more name components that specify what the device should do.

We will translate names in the relative namespace to names in the absolute namespace to allow packets with relative names to get to the correct place[1]. The absolute namespace consists of names that use absolute prefixes. The general convention to access a service device using the absolute namespace is as follows:

[1]We could make names in the relative namespace routable by the network if we impose certain requirements on the infrastructure. For example, we could require that each location has its own physical router (as in our example scenario) and that a user always connects to the appropriate router according to his location. However, we wish to avoid imposing this burden on the infrastructure and the users as it would complicate practical matters. Also, we do not use the *ForwardingHint* guider in this thesis. We consider exploring it in future work as it could allow us to route packets with relative names without transforming the names.

```
/<abs-prefix>/%C1.SDC/<category>/<name>/<request-components>/<HMAC>
```

Where <abs-prefix> is the absolute prefix and <HMAC> is a message authentication code that allows us to enforce location-based access control (more details are given in Section 3.6). Notice that the structure of a name to access service devices in the absolute namespace is much like the structure of a name in the relative namespace. There are two main differences: 1) the prefix unambiguously refers to a physical location, and 2) the %C1.SDC reserved component is included to separate the absolute prefix from the service-dependent components (this is similar in spirit to the router tag component in the NLSR namespace [21]).

Another type of name in the absolute namespace is the one used to access the advertisement controller for a given location (Section 3.7):

```
/<abs-prefix>/lb-ndn-ad/<request-components>/<HMAC>
```

3.4 Localization Channel

In order to enforce the location-based access control requirement, we need a mechanism to perform location verification. Our approach will be to distribute a "secret" within each location. Ideally, all devices in that location can hear it, and all devices outside the location cannot hear it. Devices in a location can use this shared secret to authenticate to other devices in the location. The secret will change periodically so that devices that leave a location lose the ability to authenticate. We will refer to this secret as the *local access code*. Our approach is similar to [37] and [31].

To describe the mechanism used to distribute the local access code, we will distinguish between two channels of communication: the *network* channel and the *localization* channel.

The network channel is powered by NDN and is in charge of carrying *Interest* and *Data* packets among devices.

The localization channel is in charge of broadcasting location information, including the local access code. There is no restriction on how it is implemented, but the broadcast must be limited to a location. The broadcast will carry the following pieces of information, which we collectively refer to as a *localization packet*:

1. The relative prefix corresponding to the location: $RP = I.RelPrefixOf(I.TypeOf(Loc))$

2. The absolute prefix corresponding to the location: $AP = I.AbsPrefixOf(Loc)$

3. The timestamp in the advertisement controller's public key name (this will be discussed in Section 4.3.1)

4. The local access code

5. A lifetime

6. A signature

This information must be broadcast periodically to account for devices that enter and leave a location. We will explain exactly how devices use the local access code in Section 3.6.

Different technologies can meet the requirements for the localization channel. For example, there could be a device in the room transmitting an ultrasound signal that encodes the localization packet. Another option is visible light communication in which information is transmitted using the visible light spectrum [64]. Our system does not impose any particular technology.

Note that our formalization of location allows a device to hear the localization packets from different locations of different types simultaneously. For example, a mobile device can hear the localization packets from the current campus, the current building, and the current room. Each location has its own broadcaster. However, since our formalization prevents two locations of the same type from overlapping, ideally, it should not be possible for a device to hear the localization packets from, say, two different rooms at the same time.

3.5 Translation Service

In order to support the level of indirection demanded by the mobile developers, we need a mechanism that takes care of the details of name translation and location-based access control. One option is to provide this support as a library that application developers can use in their programs. However, rather than introducing an API that is called from a specific programming

language, we and the team have chosen to provide a generic network service that runs alongside other applications on the device. A user application or device controller can then use the general NDN *Interest* and *Data* constructs.

Figure 3.2 shows how the namespace translation service fits in the network stack. It is simply an application that runs on top of the packet forwarder. We will call this service *LB-NDN* (Location-Based Named Data Networking). It runs on both user and service devices. The service has three main responsibilities:

- Implement the name translation process.
- Implement the details of the location-based access control requirement.
- Assist a service device controller in advertising its services.

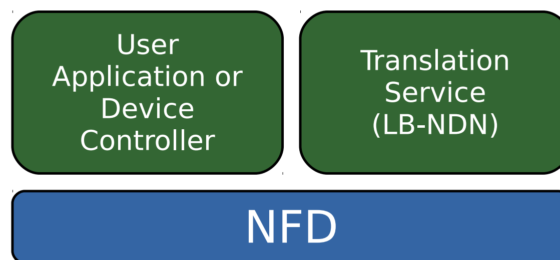

Figure 3.2: Architectural layout of the namespace translation mechanism

LB-NDN registers the following with the local NFD:

- Each relative prefix, i.e., `/thisCampus`, `/thisBuilding`, and `/thisRoom`.

  The reason for this registration is that *Interest* packets with these prefixes need translation. For example, suppose a user application wants to turn the room light on. First, it issues an *Interest* packet in the relative namespace, e.g., `/thisRoom/lights/default/on`. Next, NFD forwards this packet to LB-NDN. Finally, the *Interest* packet with the translated, absolute name is given to NFD so that it can be carried to its destination by the network.

  A special case is when the component following the relative prefix is `lb-ndn-ad`. This is used in the device onboarding protocol discussed in detail in Section 3.7.

- A prefix in the absolute namespace if LB-NDN is running on a service device so that it can accept *Interest* packets for the device. For example, if LB-NDN runs in a printer named `main` in room 258 of the CS building, the `/csu/mainCampus/csBuilding/rm258/%C1.SDC/printers/main` prefix should be registered. More details are given in Section 3.7.

3.6 Name Translation Process

A typical instance of user device/service device communication goes as follows:

1. A user application in the user device issues an *Interest* using the relative namespace, e.g., `/thisRoom/lights/1/on` (see Figure 3.3, step 1).

2. The packet goes through the local NFD service which knows that it should forward relative *Interest* packets to LB-NDN. LB-NDN transforms the *Interest* into the absolute namespace, e.g., `/csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/1/on/...` (see Figure 3.3, step 2).

3. LB-NDN hands the transformed *Interest* back to the local NFD (see Figure 3.3, step 3).

4. The network can route the packet to the destination service device where the NFD service receives it (see Figure 3.3, step 4).

5. NFD knows to forward the packet to LB-NDN to be translated back into the relative namespace (see Figure 3.3, step 5).

6. LB-NDN hands the translated packet to NFD (see Figure 3.3, step 6).

7. The NFD service knows to forward it to the device controller (see Figure 3.3, step 7).

8. The controller processes it and issues a *Data* packet in reply (see Figure 3.4, step 1).

9. The packet follows its way back to LB-NDN (see Figure 3.4, step 2).

10. LB-NDN transforms the *Data* packet into the absolute namespace and hands it to NFD so that it can be carried back to the user device (see Figure 3.4, step 3).

11. The network routes the packet to the user device (see Figure 3.4, step 4).

12. The NFD service in the user device receives the packet and hands it to LB-NDN so that it is transformed back into the relative namespace (see Figure 3.4, step 5).

13. LB-NDN hands the transformed *Data* packet to NFD (see Figure 3.4, step 6).

14. NFD gives the packet to the user application (see Figure 3.4, step 7).

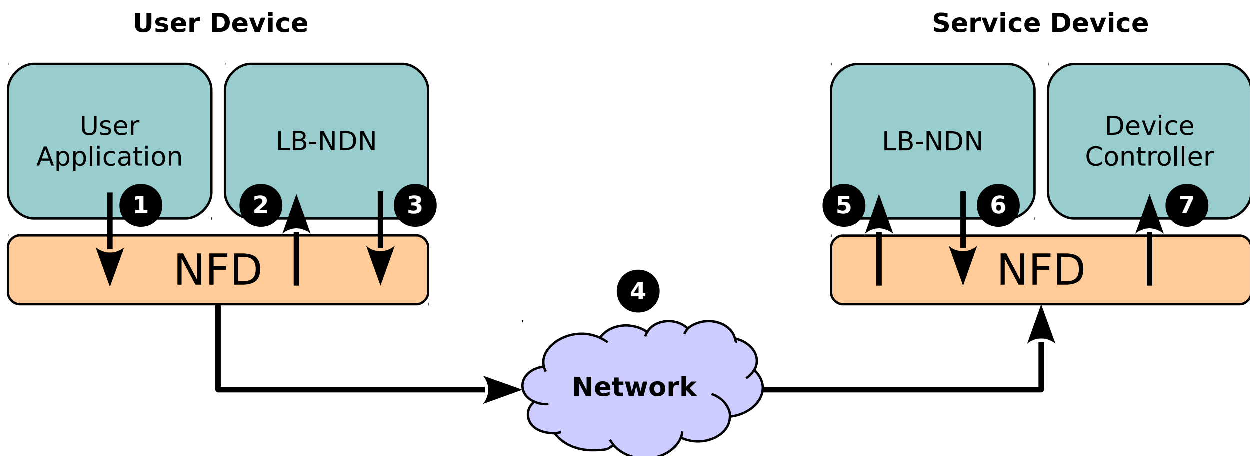

Figure 3.3: Flow of an *Interest* packet sent from the user application to a device

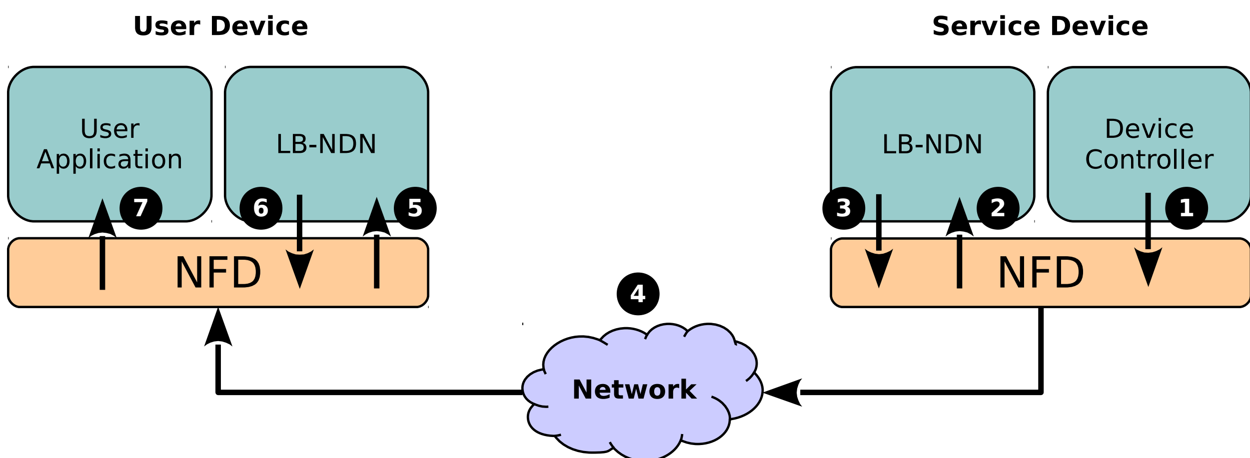

Figure 3.4: Flow of a *Data* packet sent from the device controller to the user device

The net effect was that the translation mechanism provided a layer of indirection that allowed the user application and the device controller to communicate using the relative namespace. In the next four subsections, we describe the translation process in detail for each of the packet transformation instances illustrated by the previous example.

3.6.1 Interest Emitted from an Application in a User Device

The algorithm to transform the *Interest* packet's name is as follows:

1. Create a new name initialized to the absolute prefix corresponding to the relative prefix in the original name. The information needed to perform this translation is obtained from the localization channel.

2. Append the `%C1.SDC` component.

3. Append the service-dependent components as they are.

4. Using the local access code as a key, compute the HMAC of the packet resulting from the previous steps and append it to the name. The local access code is obtained from the localization channel.

As an example, suppose that the user is in room 258 of the CS building and that the current local access code is 123. Now, consider the name `/thisRoom/lights/1/on`. The relative prefix, `/thisRoom` will get translated to `/csu/mainCampus/csBuilding/rm258`. The `%C1.SDC` component is added, the service-dependent components are left intact, and the HMAC is appended resulting in the transformed name `/csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/1/on/`$\texttt{HMAC}_{123}$, where $\texttt{HMAC}_{123}$ denotes the HMAC using 123 as the key.

It is important to note that this transformation scheme supports signed *Interest* packets. The user application can sign the *Interest* it emits. The signature-related components are considered part of the service-dependent components.

Note that if the components following the relative prefix in the incoming *Interest* are `lb-ndn-ad/advertise` or `lb-ndn-ad/withdraw`, this translation process is not invoked. Instead,

such *Interest* packets are processed as part of the device onboarding protocol (Section 3.7). If the components are `lb-ndn-ad/devices` or `lb-ndn-ad/device-info`, the packet is processed as part of service discovery (Section 3.7.2).

3.6.2 Interest Arriving at a Service Device

The routable *Interest* resulting from the previous scenario is emitted to the network and will eventually arrive at the service device[2]. Here, NFD knows that it should forward names in the absolute namespace to the translation service running on the device (Section 3.5).

The transformation algorithm is as follows:

1. LB-NDN checks the validity of the *Interest* packet using the local access code and the HMAC, i.e., the last component in the name. If this check does not pass, it is assumed that the originating user device is not in the same location as the service device and the *Interest* is discarded without further processing. If it passes, we proceed to the next step.

2. The HMAC is removed from the name.

3. The absolute prefix is replaced with the corresponding relative prefix. Recall that the absolute prefix is delimited by the `%C1.SDC` component. The information needed for this step is obtained from the localization channel.

4. The `%C1.SDC` reserved component is removed.

5. The service-dependent components are left intact.

6. Sign the resulting *Interest*. This is a certification that the packet has undergone validation with respect to the local access code.

The resulting name is in the relative namespace. Note that this name will contain any signature added by the originating user application. As we will discuss in Section 3.7, the device controller

[2]Note that we are assuming that when a device connects to a router (possibly through an access point), the device becomes capable of reaching any other router in the network, in much the same way as the *default gateway* functionality in IP.

will have previously registered a relative name in the local NFD (e.g., `/thisRoom/lights/1`) so that it can capture *Interest* packets in this namespace after the translation service has performed the transformation.

As an example, consider the absolute name generated in the previous subsection, `/csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/1/on/HMAC`$_{123}$. First, the HMAC is removed (after passing validation) resulting in `/csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/1/on`. Next the absolute prefix is replaced, the `%C1.SDC` component is removed, and the service-dependent components are left intact resulting in `/thisRoom/lights/1/on`.

There are two corner cases in this algorithm:

1. What if the local access code is not known or has changed since the *Interest* was issued from the user device?

2. What if the absolute prefix cannot be translated into a relative prefix? This could happen, for example, if the device is not physically in any room, but it somehow gets an *Interest* with name such as `/csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/1/on/HMAC`$_{123}$.

In either of these cases, the packet should be discarded and the user application must retransmit the original *Interest* after detecting a timeout in the original request.

3.6.3 Data Emitted from a Device Controller in a Service Device

If the device controller accepts the *Interest* (e.g., by verifying the signature added by the user application), it will reply with a *Data* packet with the appropriate prefix. For example, if the original *Interest* has name `/thisRoom/temperatureSensors/1/read`, a sensor may reply with a *Data* packet with name `/thisRoom/temperatureSensors/1/read/%FD%02` where `%FD%02` is a version number for the *Data* packet [65].

However, recall from Section 3.6.2 that LB-NDN added a signature to the *Interest*. Hence, it expects the signature components to be present in the name of the reply *Data* packet. To deal with this, the device controller encapsulates the *Data* packet that goes to the user application in another

Data packet whose name's prefix is the name of the *Interest* that came from LB-NDN. In our example above, the name of the outer *Data* packet would be `/thisRoom/temperatureSensors/1/read/<lb-ndn-sig>/%FD%02` where `<lb-ndn-sig>` are the signature components added by LB-NDN. The new packet should be signed using the manufacturer device identity so that LB-NDN knows that it originated from the device controller.

The packet will arrive at the local translator which must be careful in how it transforms the packet.

The algorithm is as follows:

1. Validate the signature in the outer packet and decapsulate it to obtain the inner packet, i.e., the one that will eventually make it to the user application.

2. Create a new *Data* packet. The content of this packet is the entire *Data* packet from the previous step encrypted using a symmetric encryption scheme with the local access code as the key. The name is generated in the next two steps.

3. Start with the absolute name of the *Interest* that will be satisfied by the *Data* packet.

4. Append the extra components added to the relative name by the device controller. These components should not be encrypted.

5. The new packet is signed by computing the HMAC of the packet using the local access code as the key.

As an example of the name generation, suppose that the light controller replies to the *Interest* from the previous subsection with a *Data* packet with name `/thisRoom/lights/1/on/%FC%00%01%E3%DD` where the last component is the timestamp when the requested action was taken [65]. First, we start with the name of the absolute *Interest*: /csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/1/on/$HMAC_{123}$. Next, we append the extra components added by the light controller resulting in /csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/1/on/$HMAC_{123}$/%FC%00%01%E3%DD.

Observe that the HMAC comes immediately before the extra components. This is because the NFD service at the user device that originally generated the *Interest* is expecting a *Data* packet with a prefix that includes the HMAC.

Why do we encapsulate the *Data* packet in step 2? The reason is that if LB-NDN were to directly modify the name in the original *Data* packet, it would produce an invalid packet because the signature no longer corresponds to the packet. The reason for encrypting the contents will be explained in Section 3.8. The encapsulation process is similar to the one used in ANDāNA [66].

3.6.4 Data Arriving to a User Device

When the NFD service at the user device gets the *Data* packet from the service device, it will forward it to LB-NDN. The transformation algorithm here is as follows:

1. Validate the HMAC in the packet.

2. Decapsulate the inner *Data* packet by decrypting the contents of the incoming *Data* packet using the local access code as the key.

3. Create a new *Data* packet. The content of this packet is the entire inner *Data* packet from the previous step (unencrypted). The name is the same as the inner packet's name.

4. Sign the new *Data* packet. This signature certifies that the packet comes from LB-NDN as opposed to a potentially malicious application on the user device.

The resulting *Data* packet has a name in the relative namespace. Therefore, it can be handed to the local NFD for delivery to the user application. The user application has to perform two validations: first, it makes sure that the packet came from LB-NDN; second, it "unwraps" the underlying *Data* packet and checks the signature in the resulting packet (signed by the device controller).

3.7 Device Onboarding Protocol

3.7.1 Description

Our system must provide a mechanism that allows a service device to make its services available to user devices. Per Section 3.1.3, such mechanism should allow the service device controller to use relative names to indicate the scope of its services. For example, a printer should be able to state that it wants to provide printing services for the current room by using the `/thisRoom` relative prefix. Suppose that the printer is in room 258 of the CS building. At a high level, two things must occur:

1. The printer must be given an appropriate name that can be reached by user devices, e.g., `/csu/mainCampus/csBuilding/rm258/%C1.SDC/printers/1`.

2. The network must become capable of routing packets to the printer. In our example, the `/csu/mainCampus/csBuilding/rm258/%C1.SDC/printers/1` prefix must be registered in the router that serves the `/csu/mainCampus/csBuilding/rm258` prefix.

In order to describe the specifics of the advertisement protocol, we will make the following assumptions:

1. Recall that each location has a designated router (Definition 3.2). This router can route *Interest* packets for the corresponding absolute prefix.

   Each absolute prefix is to be associated with an instance of a service which we will refer to as the *advertisement controller* or *LB-NDN-AD*. This service runs in the router that handles the corresponding absolute prefix and is in charge of handling advertisement requests from devices for the corresponding location. It registers the following prefixes with the router's NFD:

   (a) `/localhop/<abs-prefix>/lb-ndn-ad` to listen to service device advertisement requests.

   (b) `/<abs-prefix>/lb-ndn-ad` to support service discovery for user devices.

For example, the advertisement controller for `/csu/mainCampus/csBuilding` registers `/localhop/csu/mainCampus/csBuilding/lb-ndn-ad` and `/csu/mainCampus/csBuilding/lb-ndn-ad`.

Additionally, the advertisement controller must be aware of the local access code for the location that it handles. How this happens is implementation-dependent. One possibility is that the localization channel broadcasting device generates an *Interest* that goes to LB-NDN-AD and includes the encrypted local access code every time it changes.

2. The advertising service device must be directly connected to the designated router for the location under which it would be publishing. One result of establishing such a direct link should be that the NFD service in the device becomes capable of routing LB-NDN-AD *Interest* packets to this router using the `localhop` scope [67]. This is in addition to being able to reach other routers in the network. As an example, if a printer connects to the router that handles `/csu/mainCampus/csBuilding`, the printer's NFD service should forward all *Interest* packets with prefix `/localhop/csu/mainCampus/csBuilding/lb-ndn-ad` to that router.

3. Each device knows what category it belongs to.

Our mechanism must be careful about accepting advertisements since doing so involves a privileged operation on the network, i.e., changing the routing table on a router. Per Section 3.1.4, we will distinguish between two policies to establish a trust model for advertisement requests: *open* and *restricted*.

In an open policy, a device can freely advertise its services under the special device category `open`. In a restricted policy, a device claiming to be of a specific category needs to prove that it belongs to that category before an advertisement request from that device is accepted. In order to support restricted policies, we assume that the device is equipped with a *manufacturer device identity* which comprises a certificate that the manufacturer issues to allow the device to prove its membership in a category. For example, the name of the certificate for a printer could be `/hp/printers/sn-1234/dev-ctrl/KEY/ksk-12345678/ID-CERT`.

The name for the device is assigned by the advertisement controller based on some naming policy. A network administrator can specify the naming policy for each location. We will refer to such network administrator in charge of a specific location as a *location operator*.

An example of a naming policy is as follows:

- We want to allow open policy devices in room 258 of the CS building and they should be named in sequential order: `open/1`, `open/2`, etc.

- We want to allow open policy devices in room 258 and they should be named in sequential order within each category: `printers/1`, `printers/2`, `lights/1`, `lights/2` etc. Additionally, we would like to name one specific printer `printers/default`. Such printer is identified by a certificate.

The namespaces used for trust and the mechanism by which open and restricted policies can be defined and enforced will be discussed in Chapter 4. Let us now discuss the exact procedure our system follows to add a service device to the network:

1. The device controller initiates the process by issuing an *Interest* of the form

   ```
   /<rel-prefix>/lb-ndn-ad/advertise/<cat>
   ```

   Where `<rel-prefix>` is a relative prefix which specifies where the device wishes to advertise its services and `<cat>` is the category to which the device belongs, e.g., `printers`. This *Interest* should be signed using the certificate in the manufacturer device identity described above. If the device controller wants to use an open policy, then `<cat>` should be `open` and the packet should be signed with a self-signed certificate instead. This applies to all *Interest* packets generated by the device controller as part of the onboarding protocol. The reason for signing is so that LB-NDN-AD can validate the request.

2. LB-NDN captures the *Interest* generated in the previous step and generates a new *Interest* with name

   ```
   /localhop/<abs-prefix>/lb-ndn-ad/advertise/<cat>/<dev-ctrl-sign>/HMAC
   ```

Where

- `<abs-prefix>` is the translation of `<rel-prefix>` into the absolute namespace. For example, if `<rel-prefix>` is `thisRoom`, then `<abs-prefix>` could be `csu/mainCampus/csBuilding/rm258`.
- `<dev-ctrl-sign>` corresponds to the signature components added by the device controller.
- `HMAC` is the HMAC of the packet computed using the local access code as the key.

This packet does not need to be signed.

3. The *Interest* is sent to the network. Recall that we assume that there is a direct link between the service device and the router that handles the location under which the device wants to publish its services, i.e., the location corresponding to `<abs-prefix>`. Hence, the advertisement controller in that router will capture the *Interest*. Several elements are checked:

   - The HMAC is checked to make sure the originating device is in the location it wants to advertise its services in.
   - If `<cat>` is `open`, the `<dev-ctrl-sign>` signature is used only to check the packet's integrity since we will use an open policy (we assume that the device controller included the certificate as one of the components in `<dev-ctrl-sign>`). If `<cat>` is not `open`, we will use a restricted policy and the `<dev-ctrl-sign>` signature is checked to establish the membership of the device in a category. In either case, we need to reconstruct the original *Interest* (`/<rel-prefix>/lb-ndn-ad/advertise/<cat>`) before checking the signature. Note that `<rel-prefix>` does not need to be obtained from `<abs-prefix>`. For example, if the advertisement controller corresponds to a location of type *room*, then `<rel-prefix>` should be `thisRoom`.

If the validation is successful, the following will happen:

- The advertisement controller assigns a name to the device according to some naming policy. Then, it modifies NFD's RIB by adding a new route where the next hop is the face on which the *Interest* was received and the prefix is

  `/<abs-prefix>/%C1.SDC/<cat>/<name>`

  Where `<name>` is the name assigned to the device.

- The advertisement controller remembers the category, the name assigned to the device, and the device's certificate. This state has a lifetime. If the state expires, we consider the advertisement to have expired and the route created previously is deleted.

- A *Data* packet is sent in reply. It includes the result of the validation (see Table 3.1), the device's category, the name assigned to it, the prefix registered in NFD's RIB, and the advertisement's expiration timestamp.

Table 3.1: Validation result codes for an advertisement request

Code	Description
SUCCESS	The device was validated according to a policy and a route has been created.
ACCESS_DENIED	The advertisement is not allowed according to a policy. This is a permanent error, i.e., a retry will probably not change anything.

Similar to what occurs in the transformation procedure of Section 3.6.3, the generated *Data* packet is encapsulated. The name of the inner packet is `/<rel-prefix>/lb-ndn-ad/advertise/<dev-ctrl-sign>`. The content of the packet is as previously described. The name of the outer *Data* packet is the same as the incoming *Interest*, i.e., `/localhop/<abs-prefix>/lb-ndn-ad/advertise/<cat>/<dev-ctrl-sign>/HMAC`. The content is the entire inner *Data* packet. Both the inner and outer packets are signed using the

advertisement controller's identity, e.g., `/csu/mainCampus/csBuilding/rm258/lb-ndn-ad` (this applies to all *Data* packets generated by the advertisement controller).

4. The LB-NDN service at the service device receives the *Data* packet. It checks the signature to make sure the packet came from the advertisement controller. It then decapsulates the inner *Data* packet which contains the result of the validation, the device's category, the name assigned to it, the name of the route's prefix, and the advertisement's expiration timestamp. The signature of the inner packet does not need to be checked. If the reply is a success, LB-NDN remembers that there is an approved advertisement request and registers the route's prefix in the local NFD, i.e., `/<abs-prefix>/%C1.SDC/<cat>/<name>`.

   Then, it replies to the original *Interest* issued by the device controller with another *Data* packet. The new packet encapsulates the inner *Data* packet. The outer packet's name is the same as the inner packet's name. The packet is signed using the certificate in a local identity, i.e., `/localhost/lb-ndn`. This signature certifies that LB-NDN verified that the original packet was signed by the correct advertisement controller.

5. The device controller receives the *Data* packet. It can validate the outer signature to make sure it came from LB-NDN. Optionally, it can validate the inner signature to make sure the packet came from an advertisement controller. However, as long as we assume that LB-NDN is not compromised, the device controller can be confident of the packet's origin based on the outer signature. If the advertisement request was successful, it should register the following prefix with the local NFD service:

   `/<rel-prefix>/<cat>/<name>`

   The last two components are obtained from the *Data* packet.

6. The device controller must express another `advertise` *Interest* before the advertisement's expiration timestamp included in the *Data* packet. This way, LB-NDN and the device controller can keep the routes alive.

The device controller is now ready to accept packets from user devices. Any *Data* packets it generates should be signed using the same certificate as in step 1.

Note that there are multiple states in the protocol that may expire:

1. **Knowledge of the device in the advertisement controller:** this is the state that keeps track of an approved advertisement request. If this state expires, the route created by the advertisement controller for the device is deleted. Every successful `advertise` *Interest* renews this state.

2. **Knowledge of the device in the service device's LB-NDN:** this is the state that keeps track of the advertisement request approved by the advertisement controller. Once a request is approved, this state lives until the expiration timestamp included in the response by the advertisement controller. If this state expires, the route created by LB-NDN is deleted. Every successful `advertise` *Interest* renews this state.

3. **The route created by the device controller in the service device:** if this route expires, the device controller will no longer be able to capture *Interest* packets in the relative namespace. The device controller is responsible for keeping this route alive.

The reason for the expiration of these states is to minimize the number of stale routing entries and to account for service devices changing location. If a device deliberately wants to stop listening to *Interest* packets for a location, it should issue an *Interest* with name

`/<rel-prefix>/lb-ndn-ad/withdraw`

LB-NDN will send an *Interest* to the advertisement controller with name

`/localhop/<abs-specifier>/lb-ndn-ad/withdraw/<dev-ctrl-sign>/HMAC`

The effect should be that the states previously mentioned are deleted. Additionally, the device controller should unregister the relative name from the local NFD (e.g., `/thisRoom/printers/p1`).

What happens if the device is moved to a different location? LB-NDN can detect this situation because it would notice that the route's name in the response from the new advertisement controller

does not match the current location. When this happens, LB-NDN can unregister the old prefix register the new one.

3.7.2 Service Discovery

The fact that the advertisement controller assigns names and remembers service devices means that it is straightforward to implement a service discovery procedure. If a user device wants to know the list of devices in the current location, it can issue an *Interest* of the form

`/<rel-prefix>/lb-ndn-ad/devices`

LB-NDN will capture this packet and translate it to an *Interest* of the form

`/<abs-prefix>/lb-ndn-ad/devices/<HMAC>`

LB-NDN-AD can reply with a *Data* packet containing the list of devices. As we will see in Section 4.3.1, it will be necessary for a user application to obtain the certificate of the device that is authorized to use a specific name. To do this, the user application can issue an *Interest* with name of the form

`/<rel-prefix>/lb-ndn-ad/device-info/<cat>/<name>`

This packet will get translated to

`/<abs-prefix>/lb-ndn-ad/device-info/<cat>/<name>/<HMAC>`

LB-NDN-AD can reply with a *Data* packet containing the certificate of the device authorized to use this name[3] and the advertisement's expiration timestamp (see step 3 in the onboarding protocol). The advertisement's expiration timestamp is included as the last component in the *Interest* in order to support the *Exclude* selector. For example, suppose a user application needs to obtain the information associated with `/thisRoom/printers/1`. It issues an *Interest* with name `/thisRoom/lb-ndn-ad/device-info/printers/1`.

[3]In the case of a restricted policy device, it can simply reply with the name and digest of the certificate. We will see why in Section 4.3.1.

3.7.3 Examples

The following examples are intended to illustrate the onboarding protocol and some corner cases in practical settings. All scenarios occur under the infrastructure in Figure 3.1.

Scenario 1 (Connecting to the Wrong Router)

A printer is programmed to initiate the onboarding protocol for the `/thisRoom` prefix as soon as it is turned on. Suppose that initially, the printer is located in room 258 of the computer science building but a link is established to the building's router instead of the room's router. The LB-NDN service in the printer issues an *Interest* with name `/localhop/csu/mainCampus/csBuilding/rm258/lb-ndn-ad/advertise/printers/<dev-ctrl-sign>/HMAC`. However, since the printer is connected to the wrong router, the packet will go unsatisfied and the printer will continue attempting to start the protocol.

Scenario 2 (Typical Case)

The same printer establishes a link to the correct router. The `advertise` *Interest* will find its way to the advertisement controller for room 258 which approves it and assigns the name `1`. It then registers the route `/csu/mainCampus/csBuilding/rm258/%C1.SDC/printers/1`. When LB-NDN gets the response, it registers `/csu/mainCampus/csBuilding/rm258/%C1.SDC/printers/1` in the local NFD. The device controller then registers `/thisRoom/printers/1`. The printer is now reachable from user devices.

Scenario 3 (Moving to a Different Room)

The printer from the previous scenario is moved to room 260 and connected to the appropriate router. Eventually, the device controller expresses the periodic `advertise` *Interest*. The advertisement controller for the new room approves the request. It may or may not assign the same name that the device currently possesses. Let us assume it assigns a different name, e.g., `2`. It registers the route `/csu/mainCampus/csBuilding/rm260/%C1.SDC/printers/2`. When LB-NDN gets the response, it notices that the route's name no longer corresponds to room 258. Hence, it

unregisters the old prefix (`/csu/mainCampus/csBuilding/rm258/%C1.SDC/printers/1`) and registers `/csu/mainCampus/csBuilding/rm260/%C1.SDC/printers/2`. The device controller gets the response and notices that the assigned name has changed from `1` to `2`. Hence, it unregisters the old prefix (`/thisRoom/printers/1`) and registers `/thisRoom/printers/2`.

Scenario 4 (Moving to the Original Room)

The same printer is moved back to room 258. Assume that the printer is still known to the advertisement controller in this room because it did not spend too much time outside the room. When the device controller sends the periodic `advertise` *Interest*, the advertisement controller will reply with the old name, i.e., `1`. LB-NDN gets the response and notices that the route's name no longer corresponds to room 260. It unregisters the current prefix and registers `/csu/mainCampus/csBuilding/rm258/%C1.SDC/printers/1`. As in scenario 3, the device controller notices a change in the assigned name. It unregisters the current prefix and registers `/thisRoom/printers/1`.

Scenario 5 (Change in the Local Access Code)

Now, suppose that the printer issues the periodic `advertise` *Interest*, but the local access code changes by the time the request reaches the advertisement controller. This causes the advertisement controller to discard the packet and the original *Interest* to go unsatisfied. The device controller should retry as this situation is unlikely to happen often (unless the local access code changes very frequently).

Scenario 6 (Advertising in Multiple Locations)

The printer now wants to initiate the onboarding protocol for the `/thisBuilding` prefix while maintaining its outstanding advertisement in the room. Hence, it connects to the building's router (while maintaining its link to the room's router) and issues an *Interest* with name `/thisBuilding/lb-ndn-ad/advertise/printers` which causes LB-NDN to issue an *Interest* with name `/localhop/csu/mainCampus/csBuilding/lb-ndn-ad/advertise/printers/<dev-ctrl-sign>/HMAC`. The advertisement controller approves the request and assigns the name `5`. It creates the

route with name `/csu/mainCampus/csBuilding/%C1.SDC/printers/5`. LB-NDN registers that same prefix, and the device controller then registers `/thisBuilding/printers/5`. Now the printer is available in both the `/thisRoom` and the `/thisBuilding` scopes. Notice that none of the names for the `/thisBuilding` prefix conflict with the ones for the `/thisRoom` prefix.

Scenario 7 (Moving out of the Room)

The printer is now moved to a common area in the building. Three events may happen:

- The connection to the room's router is lost.
- The connection to the room's router is not lost, but LB-NDN detects that it cannot longer resolve the `/thisRoom` prefix because the localization packet expired.
- The connection to the room's router is not lost, but a new localization packet has been generated and LB-NDN cannot hear it (and the current localization packet has not expired).

In the first two cases, when the device controller issues an `advertise` *Interest*, it will go unsatisfied. In the last case, the advertisement controller will discard the packet because of an incorrect HMAC and the *Interest* will still go unsatisfied. In any case, the printer's controller will give up causing it to unregister the current prefix (i.e., `/thisRoom/printers/1`). The state in the advertisement controller and the LB-NDN service will eventually expire.

3.8 Caching Considerations

Since every *Data* packet in NDN is meaningful by virtue of its name, in-network caching makes sense. Imagine a user device requesting the latest reading from the room's default temperature sensor. It can issue an *Interest* with name `/thisRoom/temperatureSensors/default/read`. The name will get translated to `/csu/mainCampus/csBuilding/rm258/%C1.SDC/temperatureSensors/default/read/HMAC`. The *Data* packet coming out of a service device may have the name `/csu/mainCampus/csBuilding/rm258/%C1.SDC/temperatureSensors/default/read/HMAC/%FD%02` where the last component is the version of the *Data* packet. This packet may

get cached in intermediate routers. Fortunately, even if another user device outside the room issues an *Interest* with name `/csu/mainCampus/csBuilding/rm258/%C1.SDC/temperatureSensors/default/read` (notice the absence of the HMAC), neither the local access code nor the temperature reading will leak. The local access code is protected by the HMAC. Hence, the user device cannot generate arbitrary *Interest* packets destined to the temperature sensor. The temperature reading is protected by the fact that it was encrypted using the local access code as a symmetric key. Hence, cached *Data* packets are only useful to user devices in the correct location.

Nonetheless, caching introduces other problems. A device's absolute name in the network may later get associated with another device of the same type. Suppose a user device requests the temperature reading from `/csu/mainCampus/csBuilding/rm258/%C1.SDC/temperatureSensors/1`. The reading may get cached in the user device's NFD service. Later, someone takes the temperature sensor out of the room and brings in another sensor with more precision which happens to get the same name assigned. If the user device issues the same *Interest*, it might get the cached temperature reading from the old sensor.

We can start by disabling caching for relative prefixes so that relative *Interest* packets always go through LB-NDN. Then, as we discussed in Section 3.7.2, a user application can ask the advertisement controller for information about a specific relative name. It can send an *Interest* with name such as `/thisRoom/lb-ndn-ad/device-info/temperatureSensors/1`. The advertisement controller will reply with enough information that the user application can decide if a *Data* packet coming from the temperature sensor was signed by the sensor authorized to publish under `/thisRoom/temperatureSensors/1`. Hence, the user application can detect the undesirable situation in which an old *Data* packet was served from a cache. If this occurs, the user application can retransmit the *Interest* and use the *Exclude* selector to avoid getting the undesired *Data* packet. In Section 4.6, we explain how a user application can use this selector.

However, there is a subtle issue. Suppose that the application gets the device information from the advertisement controller for the current room which may indicate that the sensor can publish

under a name for the next 90 seconds. What happens if the user device is moved to a different room during this time? Then the user application, not knowing about the move, will use outdated information to validate *Data* packets. Solving this issue is not difficult. Recall that LB-NDN encapsulates incoming *Data* packets before handing them to the user application (Section 3.6.4). Hence, together with the incoming packet, LB-NDN can include the absolute prefix corresponding to the current location. That way, the user application can know if it is using the correct device information for validating service device *Data* packets.

Now, let us consider what happens after 90 seconds have passed in the previous scenario, but now assume that the user device does not switch rooms. In this case, the user application must check if the sensor can still publish under `/thisRoom/temperatureSensors/1`. To do so, it must retrieve the latest version of the *Data* corresponding to `/thisRoom/lb-ndn-ad/device-info/temperatureSensors/1`. This can be achieved by using the *Exclude* selector. We discuss this procedure in Section 5.2.2.

It is still possible for a malicious device to inject bad *Data* packets in the network. We will discuss cache poisoning in Section 4.6.

Chapter 4

Trust and Security Considerations

4.1 Overview

In Chapter 3, we described the functional characteristics of our system and discussed some access control problems. In this chapter, we will focus on issues related to trust and security. With respect to trust, we need to answer the following questions:

- When an entity receives a packet, should it accept it? This includes NDN packets and localization packets.

- Who is responsible for publishing certificates?

Sections 4.2-4.4 deal with these questions. With respect to security, we need to answer the following questions:

- How can we mitigate the risk of replay and cache poisoning attacks?

- How does our system support encrypted communication between user and service devices?

Section 4.5 and Section 4.6 deal with the first question. We will defer a discussion of encryption until Section 6.6.

4.2 Identities

There are several entities that handle packets in our system such as the user application, LB-NDN, routers, etc. NDN allows for signed *Interest* packets [16]. Furthermore, each *Data* packet is signed [6] so that the consumer can decide whether to accept the packet or not. In addition to using the local access code as a form of access control, our system must ensure that entities can verify that packets come from authorized sources. We will use signatures as

supported by NDN in order to allow entities to manage trust. However, before we explore this in Section 4.3, we will examine the different identities in our system.

In NDN, an identity groups keys and certificates under a namespace [13]. Table 4.1 summarizes all the entities involved in trust management in our system. For each entity, we provide one or more identities. Note that it is possible for an entity to have multiple identities for different purposes. Hence, for each identity, we provide an example of the namespace, a description, and the signer of the certificate for the identity. Observe that in order to simplify trust management, we will assume that each identity has only one private key and one certificate. The namespace examples are based on the infrastructure in Figure 3.1.

4.3 Trust Management

Table 4.1 defines two hierarchies: the *infrastructure trust hierarchy* anchored by the infrastructure certificate authority (depicted in Figure 4.1) and the *device trust hierarchy* anchored by the industry consortium certificate authority (depicted in Figure 4.2). In addition to these trust anchors, there is a self-signed certificate owned by LB-NDN to establish trust within a device and a self-signed certificate owned by a device controller to be used as part of an open policy.

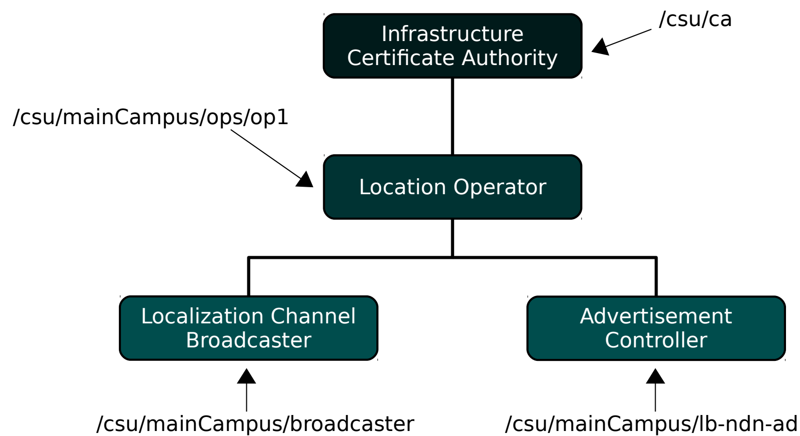

Figure 4.1: Trust hierarchy anchored by the infrastructure certificate authority

Table 4.1: Entities involved in trust management in our system

Entity	Identity Namespace Example	Description	Signer
User application in client device	`/andrescj/myApp`	Used optionally by a user application to authenticate against the service device's controller. For example, a projector may restrict its services to specific people.	Depends on the application
Device controller	`/hp/printers/sn-1234/dev-ctrl`	Used by the device controller to sign *Interest* packets as part of a restricted policy in the device onboarding protocol. It is also used by the device controller to sign *Data* packets destined to user applications.	Device manufacturer CA
	`/localhost/andrescj/phone`	Used by the device controller to sign *Interest* packets as part of an open policy in the device onboarding protocol. It is also used by the device controller to sign *Data* packets destined to user applications.	Self-signed
Localization channel broadcaster	`/csu/mainCampus/broadcaster`	Used by the localization channel broadcaster to sign localization packets.	Location operator
LB-NDN	`/localhost/lb-ndn`	In both types of devices, this is used by LB-NDN to sign *Data* packets destined to the user application or device controller so that they know the packets went through LB-NDN. In the case of advertisement controller *Data* packets, this signature certifies that they have been validated (Section 4.3.1). It is also used to sign *Interest* packets going to the device controller to indicate that the incoming packets underwent validation with respect to the local access code.	Self-signed

Advertisement controller (LB-NDN-AD)	`/csu/mainCampus/lb-ndn-ad`	Used to sign *Data* packets generated by the advertisement controller.	Location operator
Location operator	`/csu/mainCampus/ops/op1`	Used by a person to sign the certificates of the localization channel broadcaster and the advertisement controller for a location.	Infrastructure CA
Infrastructure CA	`/csu/ca`	Used by the global network administrator to sign the certificates of location operators.	Trust anchor
Device manufacturer CA	`/hp/printers/ca`	Used by manufacturers to sign the certificate of a particular product under a specific category. This identity essentially certifies that a manufacturer is authorized to make products of a specific category.	Industry consortium category CA
Industry consortium category CA	`/ic/printers/ca`	Used by the industry consortium that defines product categories in order to allow device manufacturers to claim that they make products of a specific category.	Industry consortium CA
Industry consortium CA	`/ic/ca`	Used by the industry consortium to define product categories.	Trust anchor

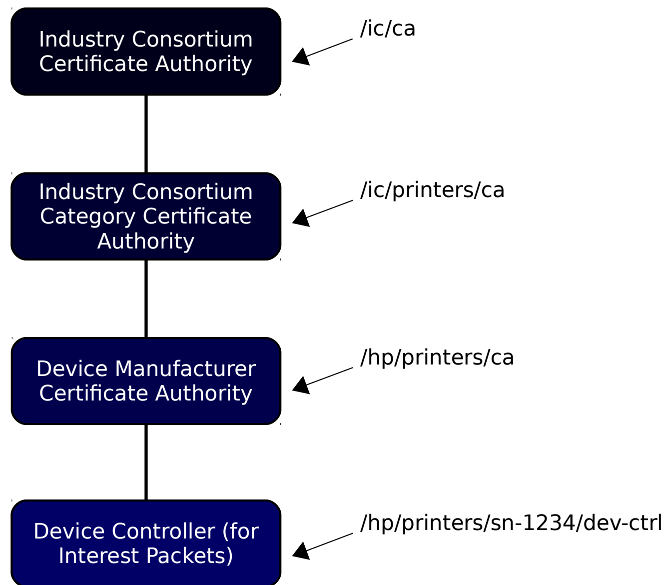

Figure 4.2: Trust hierarchy anchored by the industry consortium certificate authority

The infrastructure trust hierarchy is associated with a specific infrastructure and defines trust relationships within it. The infrastructure CA authorizes one or more location operators for each location. Any location operator for a given location can authorize the broadcaster and the advertisement controller for that location. Note that the `/csu` prefix is a placeholder for a local network prefix that can be reached from any of the routers in the infrastructure.

The device trust hierarchy is supposed to be independent of the infrastructure. The industry consortium CA defines device categories by signing the certificates of industry consortium category CAs. Each category of devices has one such CA, e.g., `/ic/printers/ca`, `/ic/lights/ca`, etc. The industry consortium category CA for a given category authorizes manufacturers to produce devices under that category by signing their device manufacturer CAs. For example, under the `printers` category, we could have the CAs of HP and Epson printers: `/hp/printers/ca` and `/epson/printers/ca`. Device manufacturers can then use these CAs to sign the identity of each device[4]. The `/ic` and the manufacturer prefixes (e.g., `/hp`) are placeholders for globally

[4]The namespace used for manufacturer device identities is similar in structure to the one used in [56].

reachable names. As mentioned previously, in the absence of an industry consortium, the device trust hierarchy can be replaced by an infrastructure-dependent hierarchy of device categories.

Our goal in the following subsections is to describe how each entity that accepts packets can decide whether or not to accept them. In order to express rules, we will be making extensive use of *validator configuration files* [15].

4.3.1 In User Applications

A user application expresses an *Interest* in the relative namespace and expects a *Data* packet in that same namespace. Recall that the packet the application is interested in is encapsulated by LB-NDN, and the outer packet is signed using a self-signed certificate that proves that the packet came from LB-NDN as opposed to a potentially malicious application on the user device. Section A.1.1 shows the configuration file needed to validate packets before decapsulation. Notice that the self-signed certificate is treated as a trust anchor.

After decapsulation, how does the user application decide whether or not to trust the inner *Data* packet? This question can be answered in two parts:

1. **Was the packet generated by a device of the *right* type?** To answer this question, recall that the device controller signs *Data* packets using either a self-signed certificate (in the case of an open policy) or a manufacturer device identity (in the case of a restricted policy). Since the application knows the category of the service device it is interested in, it can validate the signature using the device trust hierarchy. In the case of an open policy device, it can simply validate the signature as an integrity check (we will address how to obtain the public key shortly).

2. **Was the packet generated by the *right* device?** We are really asking if the packet was generated by the device who was assigned the name the user application is interested in. For example, if the application requested `/thisRoom/printers/default`, can it be confident that the reply packet was generated by a printer named `default` in the current room? Since the advertisement controller is in charge of assigning names, that is the right entity to ask.

The user application can issue an *Interest* with name `/thisRoom/lb-ndn-ad/device-info/printers/default`. The advertisement controller will reply with the digest of the certificate of the printer associated with the name `default`. It can then use this digest to make sure that the packet was signed with the expected certificate. In the case of an open policy device, the reply from the advertisement controller will include the full certificate that the application can use to check the signature in the packet. Recall that the reply also includes the advertisement's expiration timestamp. This is useful because it is essentially a guarantee that the name will not get associated with a different device at least until that timestamp[5]. Note that it is not necessary for the advertisement controller to publish the full certificate of a restricted policy device. This is because the certificate is assumed to have a globally reachable name. Another safeguard to make sure the packet was generated by a device in the current location is the HMAC added by LB-NDN in the service device. However, this only protects against devices outside the room. It does not guarantee that the packet was generated by a device authorized to publish under a specific name.

Note that a user application cannot use the *PublisherPublicKeyLocator* selector in outgoing *Interest* packets to take advantage of the fact that the advertisement controller can tell it who is allowed to publish under a specific name. The reason is that the *Data* packet coming from the network (which is in the absolute namespace) is not signed by the device controller.

A question emerges here: if the advertisement controller is already checking the device type, why does the user application need to do it? The reason is that we would like to mitigate the impact of a potentially compromised advertisement controller. If the application did not check the device type and the application controller got compromised, then anybody could potentially register say, a fake printer. The check at the advertisement controller serves more as a way to prevent bogus routing entries from being created than as a guarantee to the user application.

---

[5]Because of this, even when a device sends a `withdraw` request to the advertisement controller, the name should not get assigned to any other device until the expiration timestamp.

There is a subtle point here. When the user application gets a *Data* packet from the advertisement controller, how does it validate it? We could use the following procedure (suppose the original *Interest* was for `/thisBuilding/lb-ndn-ad/device-info/printers/default` and the current location is the computer science building):

1. The user application needs to make sure that the packet is signed by the advertisement controller. The application is not aware of the current absolute prefix, so it just checks that the signing certificate's name is of the form

   `/<1-or-more-components>/lb-ndn-ad/KEY/ksk-.../ID-CERT`

2. The user application issues an *Interest* to download the certificate, i.e., `/csu/mainCampus/csBuilding/lb-ndn-ad/KEY/ksk-.../ID-CERT`.

3. When the application gets the certificate, it makes sure that it is signed by a location operator, i.e., the signing certificate's name must be of the form

   `/<1-or-more-components>/ops/<1-component>/KEY/ksk-.../ID-CERT`

   The absolute prefix for the location operator's certificate has to match the absolute prefix for the advertisement controller's certificate, i.e., it must be `/csu/mainCampus/csBuilding/lb-ndn-ad`.

4. When the application gets the location operator's certificate, it makes sure that it is signed by the infrastructure CA, i.e., the signing certificate's name should be `/csu/ca/KEY/ksk-.../ID-CERT`.

Unfortunately, the fact that the user application is not aware of the current absolute prefix is problematic. To see this, imagine the following attack:

1. The attacker steals the key pair of the default printer in the ATMOS building (`atmosBuilding`). The relative namespace name of this printer is `/thisBuilding/printers/default`.

2. The attacker requests `/thisBuilding/lb-ndn-ad/device-info/printers/default` while in the ATMOS building.

3. Imagine, for the sake of argument, that the attacker can get from the ATMOS building to the computer science building in a few seconds. When someone requests a packet from the computer science's default printer, the attacker manages to reply with a malicious packet using the key pair stolen from the compromised printer.

4. The user application initially accepts the packet because it has been signed by a device of the right type. It then requests information about the name from the advertisement controller, i.e., `/thisBuilding/lb-ndn-ad/device-info/printers/default`.

5. The attacker manages to replay the packet that it got from the ATMOS building's advertisement controller which authorizes the compromised printer to use the `default` name in that building.

6. The user application accepts the packet because it has been signed by an advertisement controller (`/csu/mainCampus/atmosBuilding/lb-ndn-ad/KEY/ksk-.../ID-CERT`), and it is able to follow the chain of trust up to the infrastructure CA.

The root of the problem is that we need to express the following: if a device is in location *Loc*, then advertisement controller *Data* packets in the relative namespace must be signed by the advertisement controller that handles *Loc*.

To solve this problem, we will let LB-NDN be the entity responsible for validating advertisement controller *Data* packets. A user application will issue an *Interest* with name such as `/thisBuilding/lb-ndn-ad/device-info/printers/default`. When LB-NDN translates this packet, it can set the *PublisherKeyLocator* selector to the name of the correct advertisement controller's certificate. That way, it can make sure that the reply *Data* packet has been signed by the correct advertisement controller. When it gets the packet, it can validate the signature and encapsulate the packet as usual to give it to the user application. The user application just needs to

validate the outer packet's signature added by LB-NDN to be confident that the inner packet comes from the right advertisement controller.

How does LB-NDN know the name of the advertisement controller's certificate? Recall from Section 3.4 that one of the components of a localization packet is the timestamp in the advertisement controller's public key name. The name of the certificate is then

```
/<abs-prefix>/lb-ndn-ad/KEY/ksk-<timestamp>/ID-CERT
```

The configuration file to validate inner packets after decapsulation is shown in Section A.1.2. The configuration file also includes the rules to validate along the trust hierarchy as needed.

4.3.2 In LB-NDN

In a user device, LB-NDN does not need to validate the signature in *Interest* packets coming from an application because any application in a user device should be free to use its translation services. However, it should make sure that the *Interest* came from a local face so that a remote attacker cannot use the user device as a proxy. In a service device, it also does not need to check for a signature in regular *Interest* packets coming from the network. It only needs to check the HMAC to perform location-based access control.

In user devices, it needs to validate the HMAC in regular *Data* packets to make sure they were generated by a service device in the correct location. User applications are responsible for validating the authenticity of an incoming *Data* packet in terms of the device trust hierarchy. However, as discussed in Section 4.3.1, it is the responsibility of LB-NDN to validate *Data* packets coming from advertisement controllers. To do this, recall that the advertisement controller encapsulates *Data* packets and signs both the inner and outer packets using the same certificate. Thus, LB-NDN can simply validate the outer packet and be confident that both the inner and outer packets came from the advertisement controller.

In the context of the onboarding protocol for a service device, LB-NDN is a proxy for `advertise` and `withdraw` *Interest* packets. For `advertise` *Interest* packets, no signature validation is necessary. If the request gets approved by the advertisement controller, it can memorize the

certificate of the device controller. For `withdraw` packets, it can use the memorized certificate to make sure they were generated by the device controller. Just like in Section 4.3.1, LB-NDN can use the *PublisherKeyLocator* to make sure the reply *Data* packets come from the right advertisement controller. The signature in these packets also need to be validated just like any other *Data* packets coming from an advertisement controller.

Additionally, in a service device, LB-NDN receives *Data* packets from the device controller in order to satisfy requests from user devices. These packets are signed using the manufacture device identity so that LB-NDN knows that they originated in the device controller. Since LB-NDN memorized the certificate of the device controller, it can use it to check this signature.

A question emerges: if user applications and device controllers can trust LB-NDN to validate advertisement controller *Data* packets, why does LB-NDN need to re-encapsulate them? After validating the packet, LB-NDN could simply replace the signature in the inner packet with its own. One reason not to do this is that a user application or device controller may decide to be paranoid and re-validate the signature to make sure the inner packet came from *some* advertisement controller. Imagine, for example, that LB-NDN gets compromised in a user device. An attacker can then create arbitrary service discovery packets and make the user application believe that they came from an advertisement controller. If the user application re-validates the inner packet, it can at least be confident that it came from *some* advertisement controller. Since it is not aware of the current location, it cannot make sure that it came from the *right* advertisement controller, but at least now the attacker must replay packets from a legitimate controller. To simplify trust management, we have chosen not to be this paranoid. However, the option is there for user applications and device controllers.

The configuration file to validate LB-NDN-AD *Data* packets before decapsulation is shown in Section A.2. This applies to both user and service devices.

Finally, LB-NDN must also validate the authenticity of localization packets. Since we do not assume that localization packets are NDN packets, we do not show a validator configuration file for this.

4.3.3 In Device Controllers

Device controllers accept *Interest* packets from user applications. They are free to implement any identity-based restrictions. For example, a classroom projector's controller may restrict its services to professors. We do not discuss such trust models further except to say that our system supports them. Nonetheless, recall from Section 3.6.2 that LB-NDN adds its own signature (using a self-signed certificate) to certify that the *Interest* has undergone validation with respect to the local access code. The device controller needs to check this signature before processing the user application *Interest*.

In the context of the onboarding protocol, device controllers accept *Data* packets from the advertisement controller. Recall that LB-NDN validates these *Data* packets and encapsulates them. Hence, as long as we verify that the outer packet comes from LB-NDN, we can trust the inner packets.

The configuration file for both types of packets is shown in Section A.3.

4.3.4 In Advertisement Controllers

As part of the device onboarding protocol, an advertisement controller accepts *Interest* packets from service devices. For illustration purposes, suppose that we are configuring the advertisement controller for the computer science building. Per Section 3.7, a service device issues an *Interest* in the relative namespace which gets transformed into a routable packet. When the *Interest* makes it to the advertisement controller, the HMAC will get validated (to make sure the device is in the computer science building) and the original packet in the relative namespace is recovered. At this point, the advertisement controller must decide whether to approve the request or not. We have to handle both open and restrict policy devices:

1. **Open policy devices:** this is the case if the category of the service device is `open`. The signature on the *Interest* is used to verify the integrity of the packet. Recall that the entire self-signed certificate is included in the *Interest* so that the signature can be validated. When the advertisement controller approves the request, it will associate the device's certificate

with the name assigned to the device. We do not need a validator configuration file to enforce this rule.

2. **Restricted policy devices:** this is the case if the category of the device is not `open`. The signature on the *Interest* is used to verify that the device belongs to the category it wants to advertise as. The validation is done according to the device trust hierarchy. In this case, the certificate is not necessarily included in the packet, so the validator will have to use the *KeyLocator* field to download the certificate. See Section A.4 for the validator configuration file used to enforce a restricted policy.

   Note that the configuration file only takes care of `advertise` *Interest* packets. Recall that the advertisement controller memorizes a device's certificate upon approving an advertisement request. Hence, it can use this certificate to validate `withdraw` requests.

The advertisement controller also accepts service discovery requests such as `/thisBuilding/lb-ndn-ad/devices` and `/thisBuilding/lb-ndn-ad/device-info/printers/1`. These requests are satisfied freely: we do not need to check signatures (although we still check the HMAC).

4.3.5 In NFD

Local prefix registrations should be limited by NFD to trusted applications. This can be achieved by using NFD's configuration file. For example, the `localhost_security` portion of the `rib` section in NFD's configuration file to restrict local prefix registrations to LB-NDN and the device controller for an HP printer is shown in Section A.5.

4.4 Publication of Certificates

Now that we have discussed how to validate trust in our system, we need to decide who is responsible for publishing which certificates so that entities can retrieve them and realize the trust management procedures previously described.

Table 4.2 shows the same identities as Table 4.1. For each identity, we describe the method of publication of the corresponding certificate. Note that because of in-network caching, certificates

Table 4.2: Assignment of responsibility for the publication of certificates

Entity	Identity Namespace Example	Publication Method
User application in client device	`/andrescj/myApp`	The publication of this certificate is outside our scope because it is dependent on the application and the device.
Device controller	`/hp/printers/sn-1234/dev-ctrl`	Published by the manufacturer.
	`/localhost/andrescj/phone`	Published by the advertisement controller upon receiving a `device-info` request (Section 3.7.2).
Localization channel broadcaster	`/csu/mainCampus/broadcaster`	Published by the router responsible for handling the location corresponding to the broadcaster.
LB-NDN	`/localhost/lb-ndn`	This certificate does not need to be published: it is preloaded in the user application and the device controller as a trust anchor.
Advertisement controller (LB-NDN-AD)	`/csu/mainCampus/lb-ndn-ad`	Published by the advertisement controller responsible for handling the corresponding location.
Location operator	`/csu/mainCampus/ops/op1`	Published by the router responsible for handling the location corresponding to the operator.
Infrastructure CA	`/csu/ca`	This certificate does not need to be published: it is assumed to be preloaded in LB-NDN as a trust anchor.
Device manufacturer CA	`/hp/printers/ca`	Published by the manufacturer.
Industry consortium category CA	`/ic/printers/ca`	Published by an industry consortium repository.
Industry consortium CA	`/ic/ca`	This certificate does not need to be published: it is assumed to be preloaded in user applications and LB-NDN-AD as a trust anchor.

eventually maybe served by intermediate routers. The table shows who is initially responsible for distributing the certificate, i.e., the data producer.

4.5 Replay Attacks

Entities in our system are subject to attackers replaying *Interest* or *Data* packets. A full discussion of countermeasures is beyond our scope, but it is worth examining some possible solutions.

This problem is most critical for device controllers since they perform actions requested by user devices. For example, even if a projector restricts usage to professors, a malicious student could replay *Interest* packets previously generated and signed by a professor. This could allow the student to disrupt the normal operation of the device. Our general strategy to deal with replay attacks is to assume reasonably synchronized clocks among devices and use the validation procedure described in [16]: a packet can include a timestamp and a nonce. For each public key, we can keep track of the timestamp of the last packet received corresponding to that public key. If we receive a packet that has been seen before (based on the nonce) with a timestamp equal to or later than the latest timestamp, we reject it. If the packet has a timestamp less than the latest timestamp, it should also be rejected. The first packet for the public key is accepted if the timestamp is within an acceptable grace period.

4.6 Cache Poisoning

An attacker can take advantage of in-network caching by injecting bad *Data* packets into router caches [25]. When a poisoned router gets an *Interest*, it might serve bad cached packets. Even though the consumer application can detect this by validating the signature in a packet, it cannot do much about it. In theory, we can make routers validate each *Data* packet, which is unfeasible [25].

Mitigating content poisoning is an ongoing research issue in NDN [68], thus it is not within the scope of this thesis to have a full discussion on the topic. Nonetheless, one strategy to mitigate cache poisoning is to use the *Exclude* selector in an *Interest* [4]. When a consumer detects a bad

Data packet, it can reissue the *Interest* with the *Exclude* selector set to the digest of the bad *Data* packet [25]. In [25], the authors propose a ranking-based system to mitigate content poisoning. Cached content is assigned a rank based on the number of times it has been excluded.

The question then is, how does our system support the *Exclude* selector? Recall that LB-NDN encapsulates a *Data* packet before handing it to the user application. Along with the packet, LB-NDN could include the digest of the packet that came from the network[6]. If the user application determines that the inner *Data* packet is invalid, it can reissue the *Interest* and exclude the digest passed by LB-NDN. Then, LB-NDN can put the *Exclude* filter in the translated *Interest* as is. The user application can also exclude the digest of the outer packet. However, recall that we disable caching for packets with relative prefixes.

This procedure makes it harder for attacker to poison the content store of network routers and the NFD in the user device. If an attacker manages to poison the cache in a service device's NFD in order to satisfy *Interest* packets which LB-NDN expects to be satisfied by the device controller, LB-NDN can detect this situation because it expects *Data* packets to be signed by the device controller. Hence, it can also use the *Exclude* selector.

Remember that LB-NDN is also responsible for determining whether LB-NDN-AD *Data* packets are signed by the correct advertisement controller based on the device's location. The *PublisherKeyLocator* selector should ensure that LB-NDN only obtains packets who claim to be signed by the right advertisement controller. However, it is still possible for the signature validation to fail in the case of a bad *Data* packet. In this case, LB-NDN can reissue the *Interest* with the appropriate *Exclude* filter.

---

[6]This is another reason for LB-NDN to encapsulate packets before handing them to the user application or device controller: to pass information about the *Data* packet that came from the network.

Chapter 5

Implementation

Our current implementation of the system described in Chapter 3 and Chapter 4 is a proof-of-concept prototype. In Section 5.1, we describe the three main software components that comprise our implementation. In Section 5.2, we describe the implementation of a simple service device controller and a user application in order to explore how a developer could use our system.

All software runs on top of NFD which has a version for different platforms, including Android. NFD, in turn, runs on top of IP for the purposes of our implementation.

5.1 Components

5.1.1 Broadcaster

This component is in charge of broadcasting localization packets over an acoustic signal. To achieve this, we use the *Quiet* [43] library. The core module for emitting sound is written in C. However, we call this module from Java which is the main language used in our project. In order to receive information transmitted over sound, we also use the *Quiet* library. We have two versions of the receiver module: one which is suitable for running in devices such as a laptop or a Raspberry Pi and another which is suitable for Android devices. The latter is built upon the *org.quietmodem.Quiet* [43] library. The *Quiet* library has different profiles for transmitting information over sound. For example, the `ultrasonic-experimental` profile in *org.quietmodem.Quiet* uses a base frequency of 19000 kHz.

The broadcaster is configured to transmit a localization packet repeatedly. Furthermore, it is instructed to generate a new localization packet periodically. As an example, consider the following configuration for a broadcaster:

```
locations
{
  silenceInterval 500
```

```
  location
  {
    relPrefix /thisRoom
    absPrefix /csu/mainCampus/csBuilding/rm258
    adCtrlKeyID 1503547295642
    accessCodeSz 16
    packetLifetime 120
    regenerateInAdvance 30
    broadcaster
    {
      type quiet
      identity /csu/mainCampus/csBuilding/rm258/broadcaster
      adCtrl /localhop/csu/mainCampus/csBuilding/rm258/lb-ndn-ad
      adCtrlSecretBase64 **********
      quietProfile ultrasonic-experimental
    }
  }
}
```

This configuration repeatedly broadcasts a localization packet for room 258 with 500 ms of silence in between. Each packet has a lifetime of 120 seconds. This helps receivers know when they have left a location. A new localization packet will be generated 30 seconds before the expiration of the current localization packet. The local access code is set to be 16 bytes long. The `adCtrlKeyID` is necessary for LB-NDN to know the name of the certificate of the advertisement controller for the room (Section 4.3.1). The `adCtrl` and `adCtrlSecretBase64` directives allow the broadcaster to inform the advertisement controller about the current local access code. The `adCtrlSecretBase64` is a shared secret that allows the access code to be encrypted before being transmitted to the advertisement controller.

Finally, note that the configuration file allows us to specify more than one location per broadcaster. This is useful if we want to have the same broadcaster in a room transmit messages for the room, the building, and the campus.

5.1.2 Translation Service

This is a partial implementation of LB-NDN. We also have a laptop/Raspberry Pi version and an Android version. A translation service instance is associated with a module that receives localization packets from a broadcaster. The translation service is written in Java and uses the *jNDN* [19] to perform NDN operations.

The service must be configured with (among other options) the list of relative prefixes that it should serve. Figure 5.1 shows a screenshot of the translation service for Android.

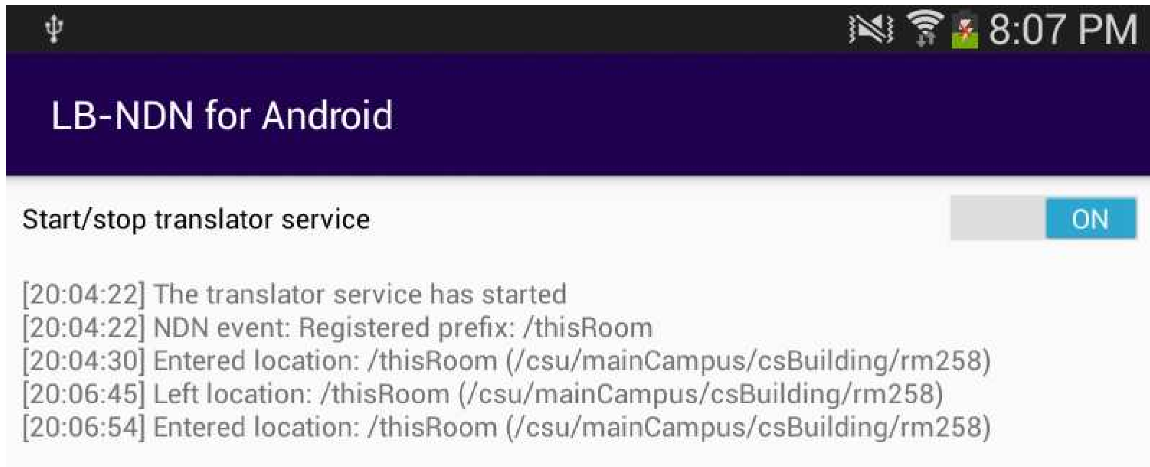

Figure 5.1: Screenshot of LB-NDN for Android

5.1.3 Advertisement Controller

This is a partial implementation of LB-NDN-AD. Only a laptop/Raspberry Pi version is available. It is written in Java and also uses the *jNDN* library. It currently supports both open and restricted policies. The device naming policy is hard coded: devices with specific public key names can be assigned a static name (e.g. `default`). Otherwise, names are assigned sequentially (i.e., `1`, `2`, `3`, ...) within a category and they are not reused even after an advertisement request expires.

The implementation is also able to satisfy `device-info` requests from user applications (Section 3.7.2). Currently, it returns the entire certificate of a service device in a `device-info` *Data* packet.

This is what the output of the advertisement controller may look like for the onboarding of a light in room 258:

```
[20:33:33] The advertisement controller has started
[20:33:33] NDN event: Enabled local fields
[20:33:33] NDN event: Registered prefix: /localhop/csu/mainCampus/csBuilding/rm258/lb-ndn-ad
```

```
[20:33:33] NDN event: Registered prefix: /csu/mainCampus/csBuilding/rm258/lb-ndn-ad
[20:34:00] Approved advertisement request for lights/default
[20:34:00] NDN event: Registered route: /csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/default -
    face ID: 390
[20:34:08] Renewing advertisement request for lights/default
[20:34:08] NDN event: Registered route: /csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/default -
    face ID: 390
```

The advertisement controller must be configured with (among other options) the lifetime of an approved advertisement request.

5.1.4 Pending Features

Our prototype implements the core features of our design including the majority of trust management. Nonetheless, it is not a full implementation, and among the most important features that are yet to be implemented are:

- Protection against replay attacks and cache poisoning.
- Authentication of localization packets.
- Handling of device listing requests (Section 3.7.2).
- Handling of onboarding protocol `withdraw` requests. Currently, devices must let an advertisement request expire.
- Validation of packets originating from NFD. This is important when registering and unregistering routes.
- Better input sanity checks.
- NFD registration restrictions as described in Section 4.3.5.

5.2 From the Developer's Point of View

Suppose we would like to write a device controller for a fancy mobile light which supports a `toggle` command. It is fancy because this light is equipped with a sensor that counts the number

of people in the room and transmits this information as the reply to every `toggle` command. Furthermore, we would like to program this light to advertise its presence in whatever room it may happen to be in within our infrastructure. Additionally, we would also like to write an Android user application to send commands to such a light as long as the user is in the same room.

In the next two subsections, we examine the most relevant fragments of code to achieve these tasks. All code is written Java and uses the *jNDN* library.

5.2.1 Writing a Device Controller

We provide a lengthy class `OnboardingProtocolManager` which handles the details of the onboarding protocol from the point of view of a device controller. The developer specifies a list of relative prefixes. For each relative prefix, an `OnboardingProtocolManager` object is created. In a typical scenario, the object repeatedly expresses an `advertise` *Interest* (e.g., `/thisRoom/lb-ndn-ad/advertise/lights`) until it gets an approved request. When it does, it registers the relative prefix assigned by the advertisement controller and memorizes the expiration timestamp. It then sends another `advertise` request before the current advertisement expires in order to renew it. If the advertisement cannot be renewed and it expires (possibly because the device was taken out of a location), the registered prefix is unregistered and the protocol starts again. This object takes care of signing requests and verifying the responses. Note that the `OnboardingProtocolManager` class is not part of the translator service: it is not aware of the current location and it uses the relative namespace to express *Interest* packets.

As an example, consider the following fragment which shows some constants and the constructor for the device controller:

```
/*
 * First, some constants for configuration
 */

// What identity to use for signing packets?
public static final Name IDENTITY = new Name("/company2/lights/sn-2397/dev-ctrl");

// Which file contains the trust schema for validating packets?
public static final String TRANSLATOR_VALIDATOR = "SampleDeviceController/translator_validator.conf";
```

```java
// What category to use in the onboarding protocol?
public static final String CATEGORY = "lights";

// What are the scopes to use in the onboarding protocol?
public static final String[] REL_PREFIXES = {"/thisRoom"};

// ...

public Main() throws SecurityException, IOException {
    // Initialize NDN elements
    face = new Face();

    // Create a key chain for signing and verifying packets
    ConfigPolicyManager cpm = new ConfigPolicyManager(TRANSLATOR_VALIDATOR);
    kc = new KeyChain(new IdentityManager(), cpm);
    kc.setFace(face);
    face.setCommandSigningInfo(kc, kc.getIdentityManager().getDefaultCertificateNameForIdentity(
        IDENTITY));

    // OnboardingProtocolManager is a class that uses the onboarding protocol
    opManagers = new ArrayList<>();

    for (String relPrefix : REL_PREFIXES) {
        opManagers.add(new OnboardingProtocolManager(relPrefix, CATEGORY, IDENTITY, this, kc, face));
    }

    // Member to control the status of the device controller
    keepRunning = new AtomicBoolean(true);
}
```

Since we want our light to advertise its presence in a room, we only create one `OnboardingProtocolManager` object. We specify that *Interest* packets coming from user applications are going to be handled in the current object (see the `this` keyword in the parameters passed to the `OnboardingProtocolManager` constructor). We also create a key chain which gives this object access to the keys necessary for signing and verifying packets. Notice the `ConfigPolicyManager` object: it allows us to specify a configuration file that defines the trust schema for validating packets (Section 2.1.3).

Because the `OnboardingProtocolManager` object takes care of the details of the onboarding protocol, we only need to worry about handling incoming *Interest* packets. Recall that LB-NDN adds a signature to certify that the incoming *Interest* was validated with respect to the local access code (Section 3.6.2). Hence, we must verify this signature. Furthermore, recall that the device controller must reply with an encapsulated *Data* packet (Section 3.6.3). The following

fragment shows the method in charge of handling verified *Interest* packets. The method that verifies incoming *Interest* packets is not shown.

```
private void handleVerifiedInterest(Interest interest, Face face) {
    Name interestName = interest.getName();

    // Get rid of the signature components
    Name nonSignName = interestName.getPrefix(-4);

    // Make sure we support this command
    if (nonSignName.size() != 4 || !nonSignName.get(-1).getValue().toString().equals("toggle")) {
        return;
    }

    // Append the current timestamp
    nonSignName.append(Name.Component.fromTimestamp(System.currentTimeMillis() / 1000));

    // Create some Data for the user application to consume
    Data reply = new Data(nonSignName);
    MetaInfo replyMI = new MetaInfo();
    replyMI.setFreshnessPeriod(600000);
    reply.setMetaInfo(replyMI);
    reply.setContent(new Blob("Number␣of␣people␣in␣the␣room␣=␣" + (new Random()).nextInt(10)));

    // Sign it
    try {
        kc.sign(reply, kc.getIdentityManager().getDefaultCertificateNameForIdentity(IDENTITY));
    } catch (SecurityException e) {
        e.printStackTrace();
        return;
    }

    // Encapsulate it
    interestName.append(nonSignName.get(-1));
    Data encReply = new Data(interestName);
    MetaInfo encReplyMI = new MetaInfo();
    encReplyMI.setFreshnessPeriod(1);
    encReply.setMetaInfo(encReplyMI);
    encReply.setContent(reply.wireEncode());

    // Sign it
    try {
        kc.sign(encReply, kc.getIdentityManager().getDefaultCertificateNameForIdentity(IDENTITY));
    } catch (SecurityException e) {
        e.printStackTrace();
        return;
    }

    // Send it
    try {
        face.putData(encReply);
    } catch (IOException e) {
```

```
        e.printStackTrace();
    }
}
```

Notice that we reply with a *Data* packet that has an extra component: the current timestamp. That way consumers know when the *Data* packet was produced. Furthermore, the freshness period of the inner packet is set to 10 minutes (600,000 ms). The freshness period of the outer packet is set to 1 ms. This period is not too important since LB-NDN adds a timestamp and a nonce as part of the signature components. Hence, it is unlikely to get a hit in the content store when LB-NDN expresses the translated *Interest* in the relative namespace.

Also, notice that the number of people in the room is computed using a random number. Instead of doing this, a developer would interface with the device in order to access actual sensor data.

The following is part of the output of a run of the device controller:

```
[22:27:05] Starting the onboarding protocol for /thisRoom
[22:27:14] Request for /thisRoom timed out!
[22:27:14] Starting the onboarding protocol for /thisRoom
[22:27:22] Request for /thisRoom timed out!
[22:27:22] Starting the onboarding protocol for /thisRoom
[22:27:33] Name assigned by the advertisement controller: default
[22:27:35] Registered prefix: /thisRoom/lights/default
[22:27:59] Renewing advertisement request for /thisRoom
[22:28:32] Renewing advertisement request for /thisRoom
[22:29:04] Renewing advertisement request for /thisRoom
[22:29:19] Received an Interest: /thisRoom/lights/default/toggle
[22:29:19] An incoming Interest was verified
[22:29:37] Renewing advertisement request for /thisRoom
[22:30:11] Renewing advertisement request for /thisRoom
[22:30:20] Request for /thisRoom timed out!
[22:30:20] Renewing advertisement request for /thisRoom
[22:30:29] Request for /thisRoom timed out!
[22:30:29] Renewing advertisement request for /thisRoom
[22:30:37] Request for /thisRoom timed out!
[22:30:37] Renewing advertisement request for /thisRoom
[22:30:45] Request for /thisRoom timed out!
[22:30:48] Unregistered prefix: /thisRoom/lights/default
[22:30:49] Starting the onboarding protocol for /thisRoom
[22:30:59] Request for /thisRoom timed out!
[22:30:59] Starting the onboarding protocol for /thisRoom
[22:31:07] Request for /thisRoom timed out!
```

Observe how the `OnboardingProtocolManager` object issues advertisement requests until it gets a reply. It then registers `/thisRoom/lights/default`. At this point, the object can be seen renewing

the advertisement. Simultaneously, the controller is able to accept incoming *Interest* packets. Eventually, the `OnboardingProtocolManager` object stops getting replies from the advertisement controller. The request expires and the prefix gets unregistered. At this point the object starts issuing advertisement requests again.

5.2.2 Writing a User Application

We will now highlight the core aspects of developing a user application that uses our system. The prototype is implemented in Android.

Our user application expresses an *Interest* in the relative namespace in order to toggle the light in the current room. It will then read the reply *Data* in order to display the number of people in the room.

Expressing an *Interest* is straightforward. The following fragment shows a method that does just this:

```
private void sendToggleInterest(String lightName) throws IOException {
    Name toggleCmd = new Name("/thisRoom/lights/" + lightName + "/toggle");

    Interest toggleInterest = new Interest(toggleCmd);
    toggleInterest.setInterestLifetimeMilliseconds(10000);

    lbndnEventProcessor.queueInterest(toggleInterest, this, this, this);

    clearMessages();
    onInfo("Queued␣Interest␣" + toggleCmd);
}
```

The `lightName` parameter is obtained from a UI text box.

Handling the reply *Data* is not as straightforward. There are multiple reasons for this: 1) the incoming packet must be decapsulated (Section 3.6.4); 2) validating the packet requires querying the correct advertisement controller about what device is authorized to publish under a given relative name; 3) validating the packet also involves checking the signature using the device trust hierarchy (Section 4.3).

Hence, to assist with this, we have another supporting class: `LBNDNEventProcessor`. This class handles incoming *Data* packets, validates them, and hands them to the developer for consuming.

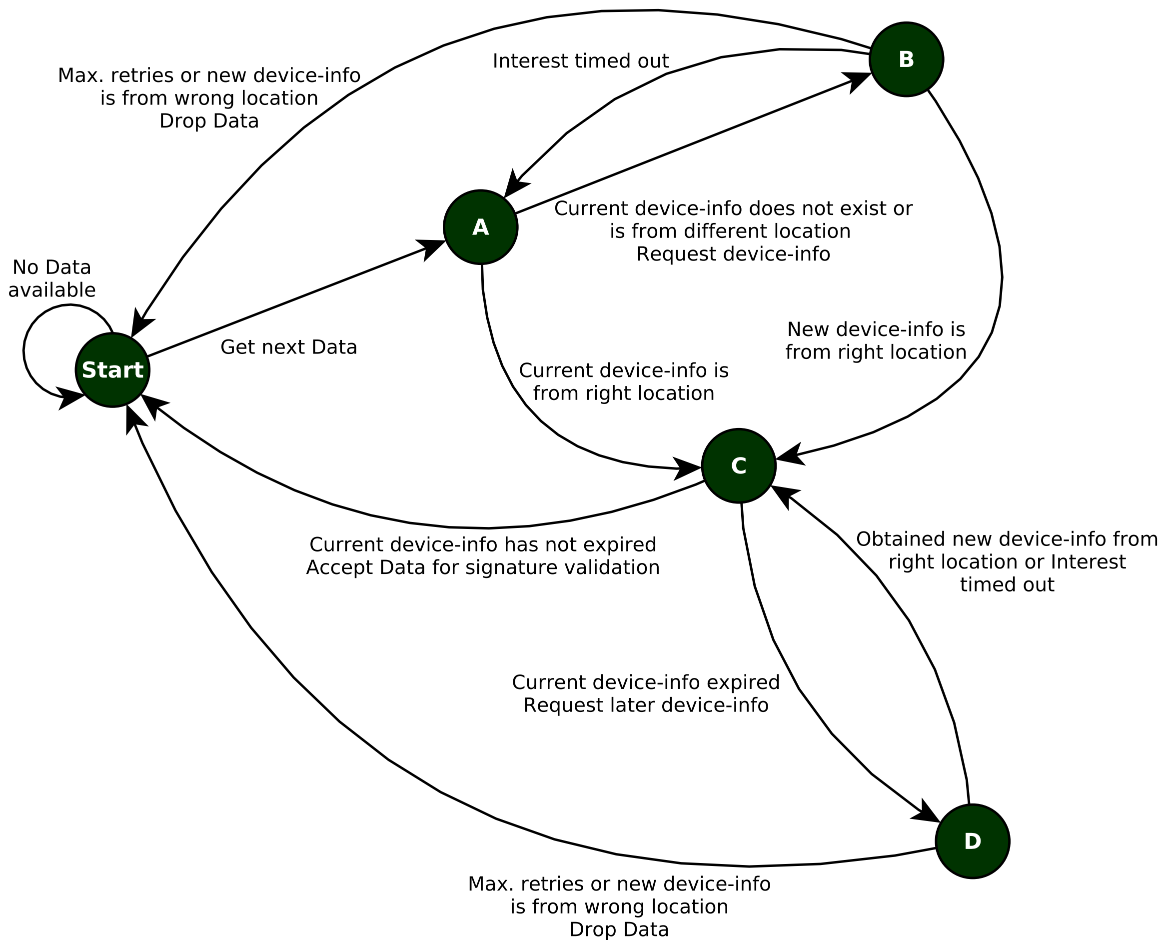

Figure 5.2: State machine for the validation of incoming *Data* packets in a user application

At its core is the state machine depicted in Figure 5.2. The idea of this state machine is simple: when we get a *Data* packet, we need to check if we already have a `device-info` *Data* packet for the device (Section 3.7.2). This also involves checking that the `device-info` packet was signed by the advertisement controller that handles the current location (recall that LB-NDN tags each incoming *Data* packet with an absolute prefix that can be used to do this check, see Section 3.8). After checking this, we must also check that the `device-info` packet has not expired: recall that the advertisement controller includes the timestamp when the advertisement for the service device expires (Section 3.7). If it has expired, we must query the advertisement controller for a more recent `device-info` packet. We can use the *Exclude* selector for this: the name of a `device-info` packet includes the expiration timestamp as the last component. Hence, we can use a filter to

indicate that we only want a `device-info` packet whose expiration timestamp is later than the current time.

Only when we have a non-expired `device-info` packet for the current location can we proceed to verify the signature. At that point, we can use the device trust hierarchy for restricted policy devices. We can use the certificate in the `device-info` packet to make sure that the certificate used to validate the `Data` packet corresponds to the one authorized by the advertisement controller. For open policy devices, we must use the certificate included in the `device-info` packet to verify the signature.

Up to this point, everything is taken care of by the `LBNDNEventProcessor` class. Once the signature is verified, the developer can use the *Data* packet. `LBNDNEventProcessor` takes care of decapsulation before handing it to the developer. The following fragment shows the method that handles a *Data* packet after it has gone through the `LBNDNEventProcessor` object:

```
public void onData(Interest interest, Data data) {
    // Get the timestamp
    long timestamp;

    try {
        timestamp = data.getName().get(4).toTimestamp();
        maxTimestamp = (timestamp > maxTimestamp) ? timestamp : maxTimestamp;
    } catch (EncodingException e) {
        onError(e.getMessage());
        return;
    }

    // ...
    displayMessage("DATA␣PACKET");
    displayMessage("\n");
    displayMessage("NAME:␣" + data.getName());
    displayMessage("TIMESTAMP:␣" + timestamp);
    displayMessage("CONTENT:␣" + data.getContent());
    displayMessage("SIGNED␣BY:␣" + KeyLocator.getFromSignature(data.getSignature()).getKeyName());
    // ...
}
```

Notice that we are extracting the timestamp added by the device controller which indicates when the packet was generated (this is entirely unrelated to the timestamp in the `device-info` packet). Furthermore, we are keeping track of the latest timestamp received. The reason we do this is because the freshness period of the packets generated by the device controller is 10 minutes.

What if we wanted to get the latest reading without waiting 10 minutes for content store entries to become stale? We can use the *Exclude* selector to indicate that we only want packets whose generation timestamp is later than the latest timestamp seen so far. The following fragment shows how the developer can easily achieve this:

```
private void sendToggleInterestWithExclude(String lightName) throws IOException {
    Name toggleCmd = new Name("/thisRoom/lights/" + lightName + "/toggle");

    Interest toggleInterest = new Interest(toggleCmd);
    toggleInterest.setInterestLifetimeMilliseconds(10000);

    Exclude excSelector = new Exclude();
    excSelector.appendAny();
    excSelector.appendComponent(Name.Component.fromTimestamp(maxTimestamp));
    toggleInterest.setExclude(excSelector);

    lbndnEventProcessor.queueInterest(toggleInterest, this, this, this);
    clearMessages();
    onInfo("Queued␣Interest␣" + toggleCmd + "␣-␣Exclude␣=␣" + excSelector.toUri());
}
```

Finally, to illustrate how a developer can take advantage of NDN in-network caching, our implementation also supports the *ChildSelector* (Section 2.1.1). This selector allows a user to retrieve the oldest or latest *Data* from a content store. For example, suppose that someone has used the light in the last 5 minutes by expressing *Interest* packets with the *Exclude* selector. Hence, the content store in the router possibly contains cached versions of the *Data* packet generated by the light, each with its own timestamp. If another user issues an *Interest* with the *ChildSelector* set to 1, he will get the *Data* packet with the latest timestamp in the content store. This is so as long as the local access code has not changed (once it changes, the name of the old *Data* packets will no longer match the name in the new *Interest* packets).

The following code fragment shows how a developer could use the *ChildSelector* filter:

```
private void sendToggleInterestWithChild(String lightName, int childSelector) throws IOException {
    Name toggleCmd = new Name("/thisRoom/lights/" + lightName + "/toggle");

    Interest toggleInterest = new Interest(toggleCmd);
    toggleInterest.setInterestLifetimeMilliseconds(10000);
    toggleInterest.setChildSelector(childSelector);

    lbndnEventProcessor.queueInterest(toggleInterest, this, this, this);
```

```
    clearMessages();
    onInfo("Queued␣Interest␣" + toggleCmd + "␣-␣ChildSelector␣=␣" + (childSelector == 0 ? "left" : "
        right"));
}
```

Figure 5.3 shows the output of our user application after expressing an *Interest* with no selectors. This run was executed after the application was first started, so we can observe how `LBNDNEventProcessor` must download the first `device-info` packet. Figure 5.4 shows what happens once the memorized `device-info` packet has expired and another *Interest* is expressed. In this case, `LBNDNEventProcessor` must request a more recent `device-info` packet using the *Exclude* filter.

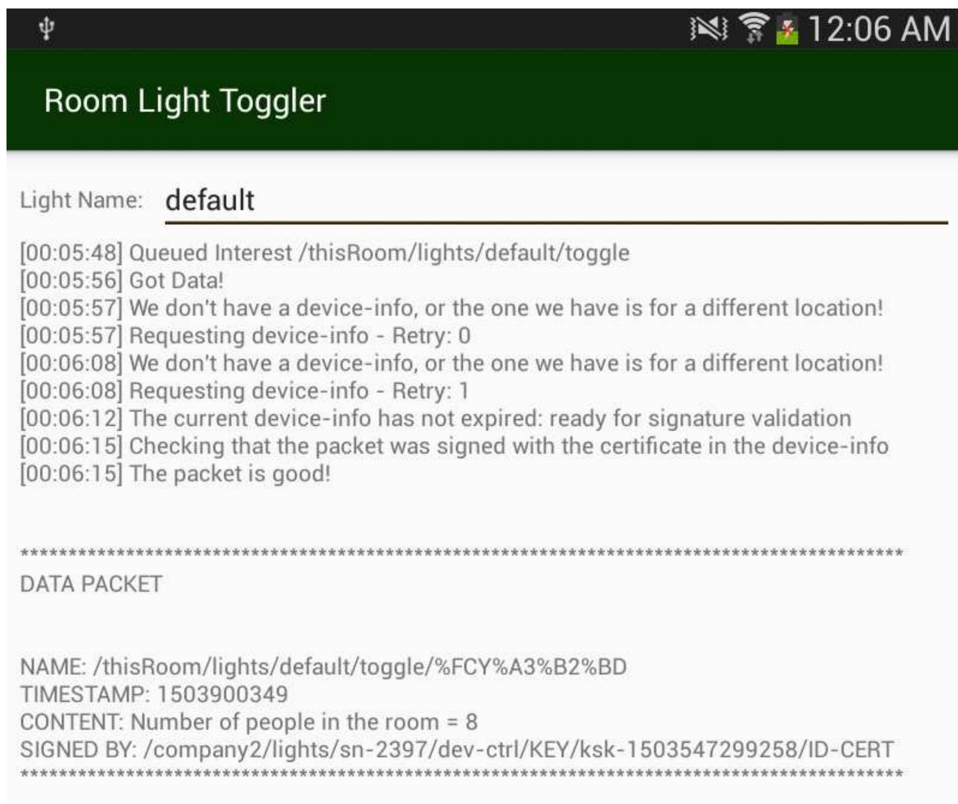

Figure 5.3: Initial run of our user application

Our example scenario was only intended to illustrate the capabilities of our system. In a real application, we would probably have separate verbs for toggling and obtaining the number of people in the room. Notice that in our application, not every `toggle` *Interest* results in an action

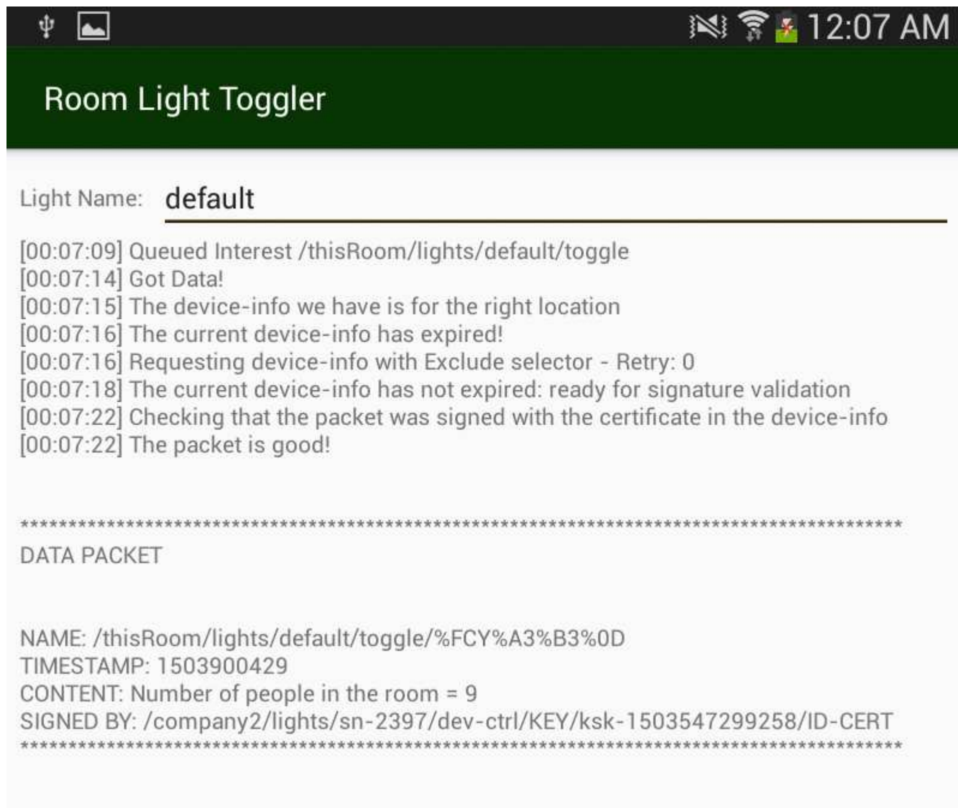

Figure 5.4: A run of our user application when the memorized `device-info` packet has expired

from the light: if the *Interest* is satisfied by a router, the light will never know about it. Hence, in a real application, one option is to include a nonce in every `toggle` *Interest* so that it always gets to the light.

Chapter 6

Evaluation

Our goal in this chapter is to qualitatively evaluate the system design proposed in Chapter 3 and Chapter 4. Note that we do not evaluate our current implementation in this chapter. Our strategy in sections 6.1-6.6 will be to analyze the extent to which our design meets the requirements explained in Section 3.1. We will also explore the limitations of the system where appropriate. In Section 6.7, we explore how our design behaves under diverse network topologies.

6.1 Device Types

Our system successfully distinguishes between user devices and service devices. A user application uses the relative namespace to request data or actions from a service device. Furthermore, it is possible for multiple user applications to share the same instance of LB-NDN in order to express *Interest* packets in the relative namespace. There are three elements that allow this to happen: 1) LB-NDN is a network service accessible by the use of NDN; 2) NFD keeps track of the packet's incoming face, so when LB-NDN replies with a *Data* packet, NFD will direct it to the originating application; 3) LB-NDN does not require user application *Interest* packets to be signed.

Note that it is possible for any device to become a service device by initiating an advertisement request using the appropriate category: the state kept by LB-NDN in the context of the advertisement protocol does not interfere with regular user application *Interest* packets. This also means that service devices can become user devices simply by using the relative namespace to generate *Interest* packets. Furthermore, the same application can act as both user application and device controller.

There is a corner case here: suppose a user application expresses an *Interest* for `/thisRoom/open/1` and that the destination device controller is in the same device as the user application. This means that they share the same NFD instance. Since the device controller registers `/thisRoom/`

`open/1`, NFD will perform a longest prefix match thus forwarding the *Interest* directly to the device controller without going through LB-NDN. The main problem is that the device controller expects a packet signed by LB-NDN (Section 4.3.3). Hence it will reject the packet. We do not expect this scenario to be common. However, if it must be supported without modifying the application or the device controller, one possibility is to modify NFD to force all relative names to be forwarded to LB-NDN. Another possibility is to make the communication between applications and LB-NDN occur over a non-NDN channel that guarantees that all relative names are handled by LB-NDN.

6.2 Location-Based Access Control

Our system design realizes location-based access control by assuming the existence of a localization channel that broadcasts messages within a specific location. This message contains a secret that should only be obtainable by devices that are in that location. This secret (i.e., the local access code) changes periodically in order to account for devices that leave a location. The local access code is specifically used in two ways: 1) to generate an HMAC for each *Interest* that leaves from LB-NDN to the network, and 2) to encrypt reply *Data* packets using a symmetric cipher and sign them with an HMAC.

The HMAC in an incoming *Interest* allows LB-NDN running in a service device to verify that the packet was generated by *something* in possession of the local access code. Ideally, this *something* is the LB-NDN service running in a user device that is in the same location as the service device. In practice, however, this strategy is subject to attacks. The HMAC protects the local access code from being obtained by an attacker sniffing the network. However, the local access code can be obtained in different ways. For example, an application can be created to listen to the localization channel and obtain localization packets. A user can then give the local access code to another user who is outside the location. The latter can then use the local access code to generate *Interest* packets in the absolute namespace with the correct HMAC and fool service devices into thinking the request originated from a user device in the same location. This is somewhat mitigated by the fact that local access code change periodically. This forces the outside

attacker to have a proxy in the location of the service device if he wants to use it for a long period of time.

We could investigate more secure and sophisticated location verification strategies such as those described in Section 2.2.1. However, the use of a secret that is shared by all devices in a location has an interesting benefit: if two applications in the same location express the same *Interest* (e.g., `/thisCampus/temperatureSensors/1/read`), the corresponding absolute names are going to be identical (e.g., `/csu/mainCampus/%C1.SDC/temperatureSensors/1/read/HMAC`). Hence, an NDN router can either aggregate the two packets if they arrive at roughly the same time, or it can satisfy one of them with cached *Data* if they do not arrive simultaneously.

Another limitation of our location-verification method is that an attacker could jam the localization channel. Nonetheless, the presence of a signature in a localization channel means that an attacker wishing to make a device think that it is in a different location by jamming the channel must replay a localization packet from that location. Again, the attacker must account for the fact that local access codes change periodically.

The periodic change in the local access code has at least two downsides:

- It may cause occasional packet loss: a user device may express an *Interest* and by the time the service device receives it, the local access code might have changed. This means that the service device will reject the packet. The converse applies: a service device may generate a *Data* packet and by the time the user device receives it, the local access code might have changed. We expect these losses to be transient. They can be fixed in the same way in both cases: the user application retransmits the *Interest*.

- Since different local access codes generate different HMACs for the same *Interest*, routers will not be able to satisfy *Interest* packets with cached *Data* generated using old local access codes. Hence, cached *Data* is only useful during the lifetime of a localization packet.

Finally, the usefulness of encrypting *Data* packets using the local access code as a symmetric key is limited: if an old local access code leaks, it is possible to decrypt *Data* packets that were

generated using that code. Hence, in sensitive applications, user applications should negotiate a key with the device controller to ensure that a compromised local access code cannot be used to decrypt old *Data*.

6.3 Support for Relative Prefixes

Our design supports the use of relative prefixes such as `/thisRoom`. This simplifies application development because developers do not have to worry about the details of enforcing location-based access control. They also do not need to worry about the details of the absolute namespace used in the underlying network architecture. Device controllers are also able to use relative prefixes to specify the scope of their services.

Nonetheless, support for these prefix is not completely transparent. The reason is that user applications and device controllers must decapsulate *Data* packets that went through LB-NDN (see Section 3.6.4 and Section 3.7). Additionally, the device controller must validate *Interest* packets coming from LB-NDN (Section 4.3.3) and encapsulate reply *Data* packets before giving them to LB-NDN (Section 3.6.3).

In spite of this, we consider that the value of the added layer of indirection outweighs the minor implementation inconvenience.

6.4 Safe Advertisement of Services

Our system allows for the use of open and restricted policies as described in Section 3.1.4. The point of supporting safe advertisement of services is to give the user confidence that he is communicating with the *right* device. We addressed this issue in Section 4.3.1. The conclusions were as follows:

- The user application validates the signature of a *Data* packet according to the device trust hierarchy to be confident that the packet came from a device of the right type.

- The user application requests information from the advertisement controller to make sure that the *Data* packet was signed by a device authorized to use a specific name. This is similar in spirit to how consumers retrieve trust schemas in [56].
- There are two safeguards to ensure that a *Data* packet came from a device in the user's location: 1) the device information obtained from the advertisement controller, and 2) the fact that LB-NDN checks the HMAC signature in incoming *Data* packets.

Hence, a user application can be reasonably confident that a *Data* packet originates from a device of the expected type which is in the user's location and has been assigned the name in the packet by the advertisement controller. However, it is possible to argue that these conditions are not sufficient in some cases to capture the notion of communicating with the *right* device.

One limitation is that the name of a device may not be enough to determine if that is the device a user wants to communicate with. For example, if an application expresses an *Interest* for `/thisRoom/light/1/on`, it may not be obvious how to locate light 1 in the room. This could be even worse if we used `/thisBuilding` instead. One could think about putting physical labels on the devices. However, labels are easy to forge, so the association between labels and network names is not trivial.

This problem can be mitigated by implementing a naming policy in an advertisement controller that associates specific devices with specific meaningful names. For example, a location operator for a building can say that a device with identity `/hp/printers/sn-1234/dev-ctrl` must be associated with name `printers/default`. Furthermore, this printer is to be positioned somewhere in the building in such a way that everybody can easily tell that it is the default printer for the building.

Another limitation is that packets may arrive at an entity outside the current location. Recall that a service device is required to periodically issue an `advertise` *Interest* to inform the advertisement controller that it is still alive and that it is still in the correct location. Hence, the network will route *Interest* packets to service devices as long as the devices are in the expected location (unless the device is taken out of that location in between advertisement requests).

However, this does not prevent packets from being sniffed. For this reason, encryption is necessary. For example, a user application could negotiate a key with the printer in the current building (see Section 6.6). As part of this negotiation, the user application generates a nonce expecting to get it back from the printer. If the application receives the nonce, it can be confident that the printer is in the current building. Then, it can send the encrypted document to the printer as an *Interest*. It would be difficult for an attacker to move the printer out of the building in between the receipt of the nonce by the application and the sending of the document.

6.5 Support for Identity-Based Trust Models

In our system, it is easy to restrict access to a service based on the identity of the requestor. Our base design only restricts access based on location. However, as mentioned in Section 3.6.1, user applications can sign *Interest* packets. The signature components will make it intact to the device controller. Network administrators can use this to their advantage in order build a trust hierarchy parallel to the infrastructure trust hierarchy (Section 4.3). This new hierarchy could assign identities to professors, students, staff, etc. Network administrators must ensure that the namespaces used in this hierarchy are reachable inside the network. They must also define who is responsible for the publication of certificates.

6.6 Encryption Support

Our system establishes an NDN channel of communication between user applications and device controllers over the relative namespace. It is then possible to use this channel to follow a key exchange protocol and establish a secure channel. For example, suppose we would like to send a document to the room's default printer `/thisRoom/printers/default` securely. First, we can download the printer's certificate. To do this, we can find the certificate's name by issuing a request for information to the advertisement controller `/thisRoom/lb-ndn-ad/device-info/printers/default`. After downloading the certificate, we can do the following:

1. Generate an AES key.

2. Encrypt the document using the key from the previous step.

3. Encrypt the AES key using the public key in the device's certificate.

4. Send an *Interest* with name

   `/thisRoom/printers/default/ENC`$_{AES}$`(print/<document>)/ENC`$_{pub}$`(<AES-key>)`

   Where `ENC`$_{AES}$ represents encryption using the AES key, and `ENC`$_{pub}$ represents encryption using the device's public key[7].

5. In the reply *Data* packet, the printer can encrypt the content using the AES key.

This is a simple example that assumes all printers have a standard interface, but more elaborate protocols could be implemented. Nonetheless, this illustrates an important fact: we cannot encrypt the entire *Interest* because the network needs everything up to the device's name (`default`) to route the packet.

Encryption support is important in our system. Recall that LB-NDN encapsulates an encrypted version of the *Data* packet generated by the device controller (Section 3.6.3). However, this is not enough for secure communication because everybody in the same location shares the same secret. Furthermore, as mentioned in Section 6.2, if an old local access code gets compromised, it can be used to decrypt old *Data* packets. Hence, it is important that user applications and device controllers do end-to-end encryption for important information.

Note that encryption support can be used to protect against replay attacks using a similar scheme as in Section 4.5 even if the service device does not require signed *Interest* packets. In the printer example, the user application can include a timestamp and a nonce in the request. The printer's device controller can then associate them with the symmetric key included in the request. That way, an attacker cannot replay the *Interest* to re-print our document.

[7]Here, for the sake of simplicity, we are assuming that the document is small enough that it can fit in a single *Interest* packet.

6.7 Network Topology Considerations

Our choice of absolute prefixes for our hypothetical infrastructure I in Chapter 3 implied a physical hierarchy. For example, `/csu/mainCampus/csBuilding/rm258` implies that room 258 is contained by the computer science building which in turn is contained by the main campus. This translated into a network topology that reflected the physical hierarchy. However, this does not need to be so.

Imagine for a moment that we redefine $I.AbsPrefixes$ and $I.AbsPrefixOf$ to assign less meaningful absolute prefixes to the locations in our infrastructure according to Table 6.1.

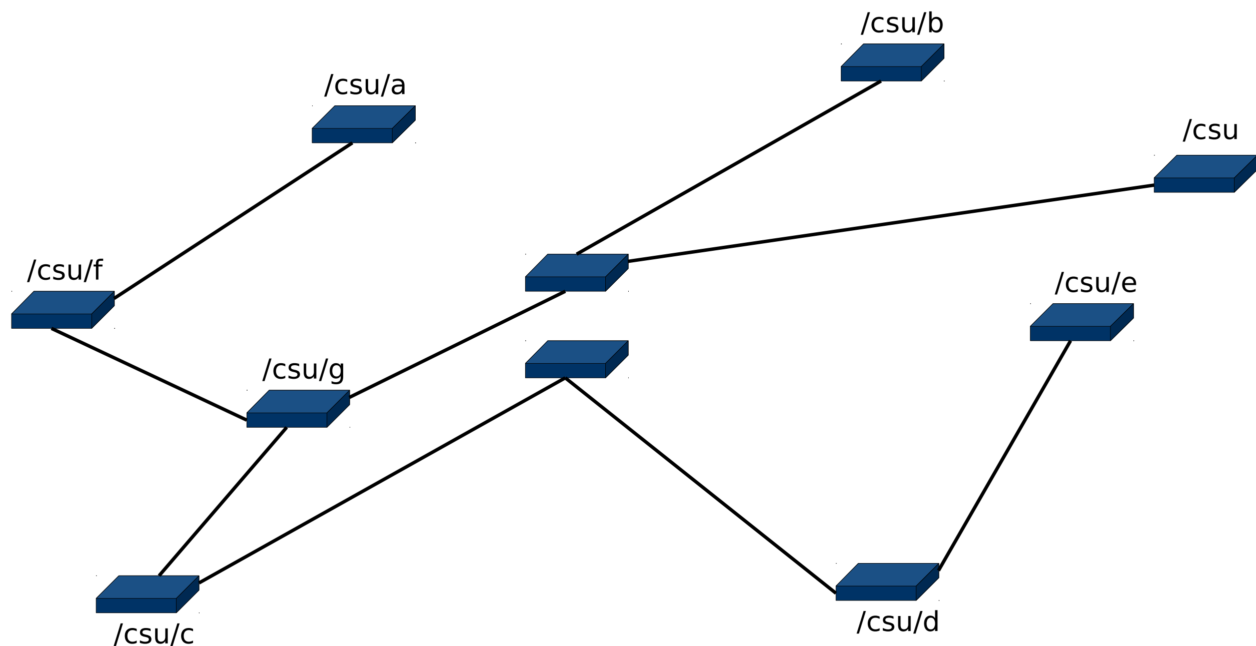

Figure 6.1: A complex topology for the example infrastructure

Furthermore, we can have a complex topology such as the one shown in Figure 6.1. Notice that each location still has its own router. Because of this, this topology does not break the onboarding protocol. As long as each service device is connected to the correct router, the advertisement controller can proceed as usual. Furthermore, suppose that a user device in room 258 is actually connected to the `/csu` router instead of `/csu/f`. As long as there is a routing protocol (such as

Table 6.1: Absolute prefix redefinition for the example infrastructure

Location	Absolute Prefix
Main campus	`/csu/a`
FH campus	`/csu/b`
CS building	`/csu/c`
ATMOS building	`/csu/d`
ARBL building	`/csu/e`
Room 258 in CS building	`/csu/f`
Room 260 in CS building	`/csu/g`

NLSR [21]), this device will still be able to reach anything under `/csu/f`. It is up to LB-NDN to enforce physical boundaries.

Next, we explore the possibility that one router can handle multiple contexts. Let us bring back our hierarchical absolute prefixes, and consider, for example, the simplified infrastructure in Figure 6.2.

In this scenario, the router for a given campus is also responsible for handling the prefixes for buildings and rooms. Would our system still work in such an infrastructure? The answer is yes. The `/csu/mainCampus` router is the designated router for every building and every room in the main campus. As such, there should be an instance of LB-NDN-AD for every building and every room running in this router (in addition to the instance for the main campus). These instances will not conflict with each other because they register different prefixes. Hence, a user device connected to any router can still reach these service devices. A similar argument can be made for the `/csu/fhCampus` router. Notice, however, that even if a router is handling multiple locations, there does not need to be a hierarchical topology. For example, it is possible, in theory, to have one router handle both the ATMOS building and room 258 of the computer science building. This works even if the naming convention for absolute prefixes is not hierarchical (as in Figure 6.1).

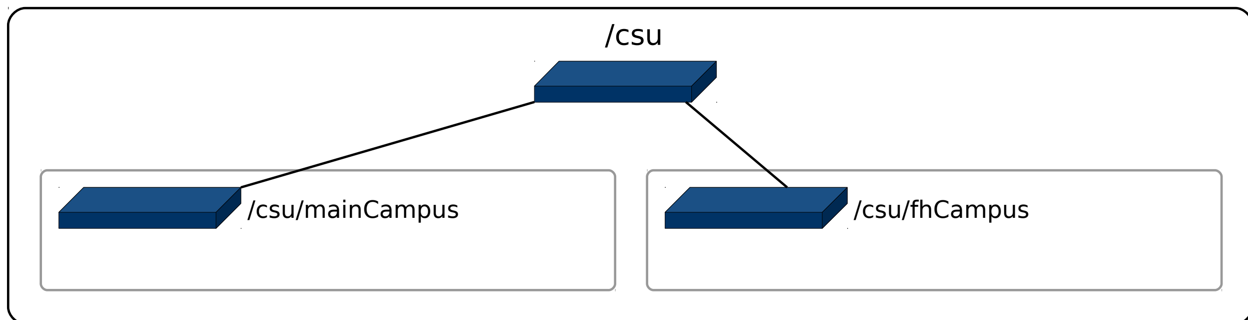

Figure 6.2: Simplified infrastructure in which routers can handle multiple locations

There are two main advantages to having a router handle multiple locations:

- Management and maintenance is easier.
- A service device may not have to establish a link to a different router when it switches locations. In Figure 6.2, a printer can move among rooms and buildings in the main campus without establishing new links.

Unfortunately, such an infrastructure can also suffer from the following:

- Single points of failure. In an infrastructure with more nodes, the advertisement controllers are distributed and the failure of a node does not affect the entire network.
- Routers can become overloaded by handling many prefixes.

Overall, our design is flexible enough to adapt to different topologies depending on the needs of the infrastructure.

Chapter 7

Qualitative Comparison to IP

In this section, we will attempt to conceive an IP-based design as equivalent as possible to our NDN-based design. We will then qualitatively compare the advantages of using IP vs. NDN as the underlying network architecture.

7.1 Envisioning an IP-Based Design

First, note that our system can be fully implemented over IP without any changes. That is because "NDN can run over anything that can forward datagrams (Ethernet, WiFi, Bluetooth, cellular, IP, TCP, etc.)" [2]. However, we would like to explore an alternative design that avoids implementing a network-wide overlay.

According to [26], there is a tendency to design IoT frameworks using a request-response communication model for named data. Hence, we will keep this model in our IP design. Note that in our NDN design, this model was realized by *Interest* and *Data* packets. In our IP design, we will need to implement request and response packets at the application layer. Therefore, trust management in the new design is not too difficult: just like in our NDN-based design, we can equip our requests and responses with digital signatures. We can then use the same trust hierarchies from Chapter 4.

We would like to allow user applications to use relative names to refer to devices in the current location just like in our NDN design. To support this, we will also have an advertisement controller per location. We will call it *LB-IP-AD*, analogous to our LB-NDN-AD. It will be responsible for assigning names to service devices. We will consider other responsibilities as we go.

The request packet sent from a user application can include the name of the service device (e.g., `/thisRoom/printers/p1`) and the request components (e.g. `print document X`). This packet can be sent to the translator service on the user device by using the IP address of the local interface and a

TCP port number (e.g., `127.0.0.1:30`). We will refer to the translator service as *LB-IP*, analogous to our LB-NDN.

LB-IP can act as a proxy for the user application and include an HMAC in the request to implement location-based access control. But then, how does LB-IP decide where to send the request packet? The request ultimately has to arrive at LB-IP in the service device which should be reachable using an IP address and a port number. We will assume this port number to be the same in all devices (e.g., TCP port 32).

To see how to get the request to the service device, we will assume the existence of the localization channel just like in our NDN design. Therefore, let us think about what the localization packet will look like in our design. We have a few options:

- We can leave the packet as is. In this case, LB-IP can translate the request packet using the same algorithm as in Section 3.6.1. This is not very helpful because LB-IP will still need to resolve the resulting name into an IP address. This resolution could be performed using mDNS so that service devices are responsible for translating their assigned names into IP addresses. However, matters may complicate if the user device and the service device are on different subnets [69].

- We can have the packet include the IP address corresponding to each device in the current location. That way, LB-IP can readily translate from a relative name to a network endpoint reference. However, this solution does not scale well as the number of devices in a location increases, especially if the localization channel is implemented using a non-reliable technology such as ultrasound.

- We can have the packet include the IP address of the advertisement controller. We can assume that the advertisement controller keeps a mapping of device names to IP addresses. LB-IP can make use of the advertisement controller in different ways:

  – It could send the request to it so that the advertisement controller acts as a proxy. This is not too much unlike LB-NDN-AD because in our NDN system, *Interest* packets

destined to service devices generally go through the router assigned to the device's location. However, relaying packets is a significantly greater responsibility assigned to LB-IP-AD when compared to LB-NDN.

- The advertisement controller can perform UDP hole punching to help establish a direct channel of communication between the user device and the service device.
- LB-IP can simply ask the advertisement controller to resolve a device's name into an IP address. It can then use this IP address to send the request directly to the service device.

It seems that the last option for the localization packet is the most scalable. Then, what is the best way of using the advertisement controller? If we assume that routers in our infrastructure do not perform NAT, then the UDP hole punching option might not be helpful. We can discuss the advantages and disadvantages of the other two options: using LB-IP-AD as a proxy vs. using it as a resolver.

The advantages of using the advertisement controller as a proxy over using it as a resolver are as follows:

- If the IP address of the service device changes, the user device does not have to worry about it as long as the advertisement controller keeps up-to-date name-to-IP mapping information.
- Since we have control over the design of the advertisement controller, we can make it cache responses from service devices in order to implement something similar to NDN's content store at the application layer.
- We can use use the advertisement controller to collect statistics about requests and responses in a given location.

The advantages of using the advertisement controller as a resolver over using it as a proxy are as follows:

- The communication is end-to-end. Depending on the topology, this may allow the user device to make a better use of the network. Additionally, the risk of overloading a not-so-powerful advertisement controller is less.

- There is better support for end-to-end encryption. Let us imagine what encryption would look like if we were to use the advertisement controller as a proxy: the user application can negotiate a key with the device controller even when using a proxy. The name of the service device can be encrypted by LB-IP after adding the HMAC to the request. This encryption can be done using a key negotiated with the advertisement controller. When the advertisement controller receives the request, it can decrypt the name and resolve it to an IP address. It can then negotiate a key with LB-IP in the service device in order to encrypt the name of the device once again before transferring the request. If we were to use the advertisement controller as a resolver and communicate with the service device directly, the LB-IP in the user device can negotiate a key directly with LB-IP in the service device.

- A transient failure in the advertisement controller may have less impact. This is because we would expect name resolution requests to occur less often than device communication.

Both approaches have their strengths, so we will not assume either one. The end result is the same: a request arrives at the service device. Much like in our NDN-based design, LB-IP in a service device must capture the request in order to enforce location-based access control based on the HMAC included on the request. After validating the request it must send it to the device controller. How does it know what the destination port number is? This could either be a fixed port or it can learn it during the device onboarding process.

The onboarding protocol has to be slightly different. No entity needs to create routing entries because IP decouples application names from network addresses. Instead, the advertisement controller needs to become aware of the IP address of the advertising service device. This may be problematic. We probably do not want to require static IP addresses because they increase configuration complexity. We could let the advertisement controller assign IP addresses during the

onboarding protocol. However, this would require the advertisement controller to keep track of other devices in the network not related to our system. A better choice would be to use the very pervasive DHCP protocol. Then, during the onboarding protocol, the service device can inform the advertisement controller about its assigned IP address. If the IP address changes (e.g., if the DHCP lease expires and the device gets a new address), the device can send an update to the advertisement controller.

Even though we can use the same trust management schemes as in our NDN design, public keys must be distributed differently. For example, each signature could have a reference to locate the corresponding signing key. This reference could be the IP address and port number of the server that distributes the public key.

Now, let us consider the effect of the network topology on the system. Recall from Section 3.7 that in the NDN design, each location has an assigned router and that a service device must establish a direct link to the router that handles the location under which it wants to advertise its services. The primary reason for this restriction is routing scalability. To see this, let us refer back to Figure 3.1. Imagine that we remove the direct-link restriction and allow a printer connected to the router for room 258 to advertise its services under the computer science building. Then, using a routing protocol, the router for room 258 will need to propagate a prefix that indicates that it can route *Interest* packets for the printer. That way, if a user device connects to any other router, it will still be able to reach the printer. Now, if every service device is free to connect to any router, then the routing tables may become large and disorganized. If we force devices to connect to specific routers, the routing system will scale better because prefixes are aggregated on a per-location basis.

As it turns out, we do not need to impose this restriction when using IP. The reason is that the network addresses are decoupled from application names. Furthermore, it is not necessary to run the advertisement controller in a router because it does not need to create routing entries. For example, consider the scenario in Figure 7.1. This figure shows a user device (a mobile phone), a service device (a printer), and the advertisement controller for room 258. All of these components are connected to different routers. The advertisement controller runs in a computer

outside of any router. The localization packet for the room will include the IP address of the advertisement controller, i.e., `192.168.3.1`. During the onboarding protocol, the printer informs the advertisement controller of its IP address, i.e., `192.168.1.43`. Regardless of whether we are using the advertisement controller as a proxy or as a resolver, a request from the user device can reach the printer. Furthermore, the routing tables in the routers do not become larger as we add more service devices.

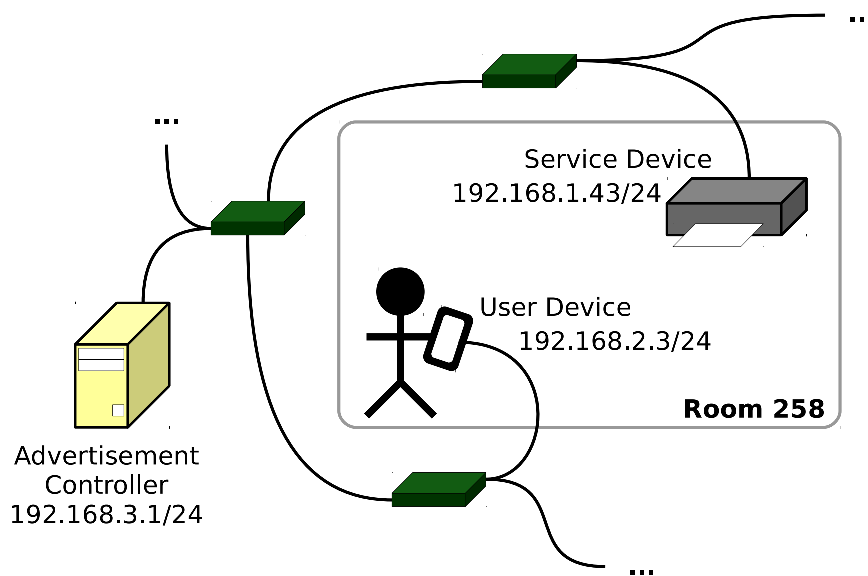

Figure 7.1: Portion of a hypothetical IP topology

There is a subtle corner case in our IP design. Suppose a service device sends a `withdraw` request to the advertisement controller. The latter can delete it from its list of known devices. However, if we use the advertisement controller as a resolver and the name-to-IP mapping has a lifetime to avoid resolving the name for every request, the network will still carry packets from the user device to the service device until the mapping expires and the user device realizes that it should not send requests to the service devices. This is an artifact of decoupling application names from IP addresses. In fact, as long as the IP address of the service device is known by other devices, it can be reached. In this case, LB-IP will need to serve as a firewall for the device controller: when there is no outstanding advertisement, LB-IP should reject incoming requests. In our NDN-based

design, we do not have this problem: as soon as a `withdraw` *Interest* is received, the network stops routing packets to the service device. In fact, as long as there is no name prefix registered in the service device's NFD, NFD will not forward packets to LB-NDN.

7.2 IP vs. NDN

Now that we have an idea of what an IP design will look like, we can compare it against NDN. Let us first discuss the advantages of the IP design:

- As previously discussed, the IP design is more flexible in terms of the underlying network topology. This is because IP decouples application names from network addresses. This also means that advertisement controllers do not need to run in routers.

- Since names are not needed for routing packets, we can encrypt entire requests. Hence, if an attacker intercepts packets, it is harder to know what is being requested. Note that it is not impossible: packets still carry unencrypted source and destination IP addresses. Hence, the attacker can still conceivably associate requests and responses with specific devices. In NDN, it is easier to know what is being requested because every *Interest* carries a meaningful name.

Now, let us discuss the advantages of using NDN:

- NDN supports in-network storage. This is because every *Data* packet is meaningful by virtue of its name. This feature is very useful in the following scenario: a school radio station wants to allow people on campus to tune in to it using their smartphones. Presumably, many different users will be connected to many different routers. These routers can cache the *Data* packets generated by the radio station. Thus, the radio station does not get overloaded with a unicast connection for each listener.

  Of course, in IP, this setup could be achieved with a multicast configuration. However, as explained in Section 2.1.5, NDN supports multicast naturally [2] "out-of-the-box." Also, if

we use the advertisement controller as a proxy in our IP design, we can cache responses from service devices. However, this is more limited than NDN's in-network storage in which every node can cache packets. If we wanted to achieve this same functionality in our IP design, we would need to make IP routers aware of application-layer packets. Unfortunately, since different IP packets make take different routes to the same destination, a router may not be able to assemble application packets correctly.

- Even though it is not easy to encrypt the full name of an *Interest* since the name is used for routing, recall that NDN packets do not carry network endpoint identifiers (Section 2.1.5). Hence, if an attacker intercepts an *Interest*, he can know what is being requested but not who the requestor is (unless the packet carries a signature) [66]. Privacy issues have been considered in the NDN literature, e.g., see ANDāNA [66].

- NDN does not have to deal with IP address changes because the name of a device allows the network to find the device.

- The distribution of public key certificates is more streamlined in NDN. This is because certificates are *Data* packets like any other.

- An NDN router could conceivably collect meaningful statistics about the packets that go through it. For example, a network administrator might be interested in knowing the percentage of *Interest* packets routed by a specific router that are destined to room 258 in the computer science building.

  We can achieve the same in IP but not as naturally. As mentioned previously, IP routers cannot correctly assemble all application-layer packets. Hence, if a router wants to collect meaningful statistics, it will need to learn the association between IP addresses and application names. This could be achieved by querying the advertisement controller for each location about this association.

Observe that both systems make use of local communication which is in line with the goal in [56]. This is done without a significant increase of network infrastructure. However, the network setup in IP may be slightly easier because routers do not run advertisement controllers.

Chapter 8

Conclusions and Future Work

In this thesis, we presented an NDN-based system that supports the development of applications that benefit from physically localized interactions. To achieve this, our system provides a level of indirection in the form of a relative namespace. Hence, user applications can use names such as `/thisRoom/lights/default/toggle` or `/thisBuilding/printers/default/activate`. Our system resolves such relative names into absolute names that can be routed by the network, e.g., `/csu/mainCampus/csBuilding/rm258/%C1.SDC/lights/default/toggle`. This translation mechanism is realized as a service which we call LB-NDN (Location-Based Named Data Networking). Devices that want to offer services in a location can also use the relative namespace to specify the scope of their services. In essence, our design abstracts away the details of localization.

The location awareness aspect of our system allows us to implement location-based access control (LBAC) in such a way that users can only access service devices that are located in the current physical scope of the user, e.g., a room or a building. Location awareness is achieved by using a *localization channel* that broadcasts a shared secret and other location information to all devices within a scope. This shared secret changes periodically to account for the movement of devices into and out of a location. Furthermore, we described a trust model that allows users to be confident that they are interacting with a device of the expected type. In order to support this, we adopted the idea of *service device categories* such as `printers` and `lights`. A device can prove its membership in a category by presenting a manufacturer-issued certificate.

We showed that our system can support additional trust models based on the identity of a requestor. For example, service devices can demand that user devices authenticate using an identity. This is in addition to the location-based access control rule enforced by our system. We also showed that our design can support encrypted communication between devices. Additionally,

we found that our system can adapt to a variety of network topologies, including non-hierarchical topologies.

The current prototype of our design is written mainly in Java and implements the core features of the system. To implement the localization channel, we use acoustic signals which are naturally limited to a given enclosed location like a room. We described how developers can write applications that make use of LB-NDN. By making use of two supporting classes, `OnboardingProtocolManager` and `LBNDNEventProcessor`, location-aware application development is greatly simplified.

Finally, we compared our NDN-based design against a hypothetical IP-based design. We found that while IP can be more flexible in terms of the underlying network topology and can allow for the encryption of application-layer names, NDN provides the following general benefits:

- In-network caching.
- Natural support for multicast communication.
- Support for security at the network layer.

Furthermore, since NDN does not require network identifiers for endpoints, our system does not need to deal with changing IP addresses. This also implies that it is harder for an attacker to figure out who the requestor is given a network packet. Finally, the use of names at the network layer means that routers could potentially collect meaningful application statistics.

We have identified several areas for future work:

- Our system is limited to a given infrastructure. In order to support the vision of ubiquitous computing, our system should be extended to support ideally any infrastructure. This, however, is a very ambitious goal. One obstacle would be to standardize a set of relative prefixes which is no easy task. For example, one university might want to use `/thisClassroom` instead of `/thisRoom` for classrooms. We would like to explore strategies that would facilitate the adoption of our system beyond a single infrastructure.

- From a practical point of view, the localization channel is difficult to implement. Acoustic signals tend to be unreliable and the situation may worsen in noisy environments such as a classroom. We would like to explore alternative media for broadcasting information. Furthermore, the idea of distributing a shared secret to perform location verification is inherently insecure: local users can potentially share the secret with remote users allowing them to access local resources. The fact that the secret code is assumed to change periodically somewhat mitigates this problem. Nonetheless, a shared secret works well with in-network caching. It would be interesting to explore alternative how other methods of location verification that are more secure could be integrated into our system.

- We would like to integrate event notifications in our system. For example, it would be useful for applications to be notified immediately upon detecting a change in location. This may allow them to trigger specific actions.

- We would like to optimize our system to run as smoothly as possible in constrained environments. As explained in Section 2.1.5, this is a difficulty in NDN.

- By using timestamps in network messages, our design essentially assumes devices with synchronized clocks. It would be interesting to attempt to remove this requirement as much as possible.

- Because of in-network storage, NDN introduces the possibility of cache poisoning in which an attacker can inject bad packets into the network to deny consumers access to legitimate content. In general, we found that dealing with caching at the network layer made certain tasks more difficult even in the absence of cache poisoning. For example, often times, a consumer must use the *Exclude* selector to retrieve the latest version of a *Data* packet. As future work in this area, we would like to explore how different content poisoning avoidance strategies can be effectively integrated into our system.

- In order to support the added level of indirection, our system had to perform transformation of network packets by translating the names of *Interest* packets and encapsulating *Data*

packets. Since the service in charge of packet transformation was designed as a network service, the encapsulation process was one of the elements which prevented user applications and device controllers from using the relative namespace transparently. In fact, in [70], the authors recognize that encapsulation introduces complex issues. We would like to explore how another type of object provided by NDN which was not considered in this thesis could potentially simplify our system: *Link* objects [70]. These objects can be used to get the *ForwardingHint* in an *Interest* [71]. This allows an *Interest* with a name that is not globally routable to be forwarded to a producer that can satisfy the *Interest* [70].

Bibliography

[1] Wikipedia. David Wheeler (British Computer Scientist) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/David_Wheeler_(British_computer_scientist)#Quotes, 2017.

[2] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. Named Data Networking. *SIGCOMM Comput. Commun. Rev.*, 44(3):66–73, July 2014.

[3] NSF Future Internet Architecture Project. http://www.nets-fia.net/.

[4] Interest Packet. https://named-data.net/doc/NDN-TLV/current/interest.html.

[5] Name. https://named-data.net/doc/NDN-TLV/current/name.html.

[6] Data Packet. https://named-data.net/doc/NDN-TLV/current/data.html.

[7] Signature. https://named-data.net/doc/NDN-TLV/current/signature.html.

[8] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yanbiao Li, Spyridon Mastorakis, Yi Huang, Jerald P. Abraham, Eric Newberry, Steve DiBenedetto, Chengyu Fan, Christos Papadopoulos, Davide Pesavento, Giulio Grassi, Giovanni Pau, Hang Zhang, Tian Song, Haowei Yuan, Hila B. Abraham, Patrick Crowley, Syed O. Amin, Vince Lehman, Muktadir Chowdhury, and Lan Wang. NFD Developer's Guide. Technical report, 10 2016.

[9] Lan Wang. Architectural Development and Routing Design in Named Data Networking. https://named-data.net/wp-content/uploads/2016/05/architectural_development_routing_design_ndn.pdf, 2015.

[10] NFD - Named Data Networking Forwarding Daemon. https://named-data.net/doc/NFD/current/.

[11] RIB Management. https://redmine.named-data.net/projects/nfd/wiki/RibMgmt.

[12] Named Data Networking: Motivation & Details. https://named-data.net/project/archoverview/.

[13] Security Library Tutorial. https://named-data.net/doc/ndn-cxx/current/tutorials/security-library.html.

[14] Yingdi Yu, Alexander Afanasyev, David Clark, kc claffy, Van Jacobson, and Lixia Zhang. Schematizing Trust in Named Data Networking. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*, ACM-ICN '15, pages 177–186, New York, NY, USA, 2015. ACM.

[15] Validator Configuration File Format. https://named-data.net/doc/ndn-cxx/current/tutorials/security-validator-config.html.

[16] Signed Interest. https://redmine.named-data.net/projects/ndn-cxx/wiki/SignedInterest.

[17] NDN Certificate Format Version 2.0. https://named-data.net/doc/ndn-cxx/0.5.1/specs/certificate-format.html.

[18] Yingdi Yu. Trust Schema Specification. https://github.com/named-data/ndn-cxx/blob/b56c54ebc18688a43cc248a95959c1ca21bb807c/docs/tutorials/security-trust-schema.rst, 2015.

[19] Jeff Thompson. jNDN: A Named Data Networking Client Library for Java. https://github.com/named-data/jndn.

[20] Named Data Networking Project Specifications. https://named-data.net/project/specifications/.

[21] Vince Lehman, A K M Mahmudul Hoque, Yingdi Yu, Lan Wang, Beichuan Zhang, and Lixia Zhang. A Secure Link State Routing Protocol for NDN. Technical report, 01 2016.

[22] Zhenkai Zhu and Alexander Afanasyev. Let's ChronoSync: Decentralized Dataset State Synchronization in Named Data Networking. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, Oct 2013.

[23] Alberto Compagno, Mauro Conti, Paolo Gasti, and Gene Tsudik. Poseidon: Mitigating Interest Flooding DDoS Attacks in Named Data Networking. In *38th Annual IEEE Conference on Local Computer Networks*, pages 630–638, Oct 2013.

[24] Alexander Afanasyev, Priya Mahadevan, Ilya Moiseenko, Ersin Uzun, and Lixia Zhang. Interest Flooding Attack and Countermeasures in Named Data Networking. In *2013 IFIP Networking Conference*, pages 1–9, May 2013.

[25] Cesar Ghali, Gene Tsudik, and Ersin Uzun. Needle in a Haystack: Mitigating Content Poisoning in Named-Data Networking. In *Proceedings of NDSS Workshop on Security of Emerging Networking Technologies (SENT)*, 2014.

[26] Wentao Shang, Adeola Bannis, Teng Liang, Zhehao Wang, Yingdi Yu, Alexander Afanasyev, Jeff Thompson, Jeff Burke, Beichuan Zhang, and Lixia Zhang. Named Data Networking of Things (Invited Paper). In *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 117–128, April 2016.

[27] Mark Weiser. The Computer for the 21st Century. *SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, July 1999.

[28] John Krumm. *Ubiquitous Computing Fundamentals*. Chapman & Hall/CRC, 1st edition, 2009.

[29] Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, January 1992.

[30] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, WMCSA '94, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society.

[31] Tim Kindberg, Kan Zhang, and Narendar Shankar. Context Authentication Using Constrained Channels. In *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 14–21, 2002.

[32] Claudio A. Ardagna, Marco Cremonini, Ernesto Damiani, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Supporting Location-based Conditions in Access Control Policies. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 212–222, New York, NY, USA, 2006. ACM.

[33] Indrakshi Ray, Mahendra Kumar, and Lijun Yu. LRBAC: A Location-Aware Role-based Access Control Model. In *Proceedings of the Second International Conference on Information Systems Security*, ICISS'06, pages 147–161, Berlin, Heidelberg, 2006. Springer-Verlag.

[34] Naveen Sastry, Umesh Shankar, and David Wagner. Secure Verification of Location Claims. In *Proceedings of the 2nd ACM Workshop on Wireless Security*, WiSe '03, pages 1–10, New York, NY, USA, 2003. ACM.

[35] Stefan Brands and David Chaum. Distance-Bounding Protocols. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, EUROCRYPT '93, pages 344–359, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[36] Adnan Vora and Mikhail Nesterenko. Secure Location Verification Using Radio Broadcast. *IEEE Transactions on Dependable and Secure Computing*, 3(4):377–385, Oct 2006.

[37] Deborah Caswell and Philippe Debaty. Creating Web Representations for Places. In *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*, HUC '00, pages 114–126, London, UK, UK, 2000. Springer-Verlag.

[38] Dirk Balfanz, Diana K. Smetters, Paul Stewart, and H. Chi Wong. Talking to Strangers: Authentication in Ad-Hoc Wireless Networks. In *NDSS*, 2002.

[39] Frank Stajano and Ross J. Anderson. The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. In *Proceedings of the 7th International Workshop on Security Protocols*, pages 172–194, London, UK, UK, 2000. Springer-Verlag.

[40] Simón Neira and Víctor M. Gulías. *Designing Transparent Location-Dependent Web-based Applications on Mobile Environments*, pages 37–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[41] Peter A. Iannucci, Ravi Netravali, Ameesh K. Goyal, and Hari Balakrishnan. Room-Area Networks. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 9. ACM, 2015.

[42] Boris Smus. Ultrasonic Networking on the Web. http://smus.com/ultrasonic-networking/, 2013.

[43] Brian Armstrong. Quiet Modem Project. https://github.com/quiet.

[44] Iris A. Junglas and Richard T. Watson. Location-based Services. *Communications of the ACM*, 51(3):65–69, March 2008.

[45] Rui José and Nigel Davies. Scalable and Flexible Location-based Services for Ubiquitous Information Access. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC ’99, pages 52–66, London, UK, UK, 1999. Springer-Verlag.

[46] Rui José, Adriano Moreira, Filipe Meneses, and Geoff Coulson. An Open Architecture for Developing Mobile Location-based Applications over the Internet. In *Proceedings of the 6th IEEE Symposium on Computers and Communications*, pages 500–505, 2001.

[47] Rui José, Adriano Moreira, Helena Rodrigues, and Nigel Davies. The AROUND Architecture for Dynamic Location-based Services. *Mobile Networks and Applications*, 8(4):377–387, August 2003.

[48] Hans Gellersen, Carl Fischer, Dominique Guinard, Roswitha Gostner, Gerd Kortuem, Christian Kray, Enrico Rukzio, and Sara Streng. Supporting Device Discovery and Spontaneous Interaction with Spatial References. *Personal and Ubiquitous Computing*, 13(4):255–264, May 2009.

[49] Roy Want, Kenneth P. Fishkin, Anuj Gujar, and Beverly L. Harrison. Bridging Physical and Virtual Worlds with Electronic Tags. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 370–377, New York, NY, USA, 1999. ACM.

[50] Felix Hupfeld and Michael Beigl. Spatially Aware Local Communication in the RAUM System. In *Proceedings of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, IDMS '00, pages 285–296, London, UK, UK, 2000. Springer-Verlag.

[51] Tim Kindberg and John Barton. A Web-based Nomadic Computing System. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 35(4):443–456, March 2001.

[52] Rene Mayrhofer, Hans Gellersen, and Mike Hazas. Security by Spatial Reference: Using Relative Positioning to Authenticate Devices for Spontaneous Interaction. In *Proceedings of the 9th International Conference on Ubiquitous Computing*, UbiComp '07, pages 199–216, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] Wentao Shang, Qiuhan Ding, Alessandro Marianantoni, Jeff Burke, and Lixia Zhang. Securing Building Management Systems Using Named Data Networking. *IEEE Network*, 28(3):50–56, May 2014.

[54] Adeola Bannis. Named Data Network Internet of Things Toolkit (NDN-IoTT). https://github.com/remap/ndn-pi.

[55] Shang Wentao, Yingdi Yu, Teng Liang, Beichuan Zhang, and Lixia Zhang. NDN-ACE: Access Control for Constrained Environments over Named Data Networking. Technical report, 12 2015.

[56] Wentao Shang, Zhehao Wang, Alexander Afanasyev, Jeff Burke, and Lixia Zhang. Breaking out of the Cloud: Local Trust Management and Rendezvous in Named Data Networking of Things. In *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 3–14, April 2017.

[57] Giulio Grassi, Davide Pesavento, Giovanni Pau, Rama Vuyyuru, Ryuji Wakikawa, and Lixia Zhang. VANET via Named Data Networking. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 410–415, April 2014.

[58] Lucas Wang, Alexander Afanasyev, Romain Kuntz, Rama Vuyyuru, Ryuji Wakikawa, and Lixia Zhang. Rapid Traffic Information Dissemination Using Named Data. In *Proceedings of the 1st ACM Workshop on Emerging Name-Oriented Mobile Networking Design - Architecture, Algorithms, and Applications*, NoM '12, pages 7–12, New York, NY, USA, 2012. ACM.

[59] Giulio Grassi, Davide Pesavento, Giovanni Pau, Lixia Zhang, and Serge Fdida. Navigo: Interest Forwarding by Geolocations in Vehicular Named Data Networking. In *2015 IEEE 16th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–10, June 2015.

[60] Michael Meisel, Vasileios Pappas, and Lixia Zhang. Ad Hoc Networking via Named Data. In *Proceedings of the 5th ACM International Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '10, pages 3–8, New York, NY, USA, 2010. ACM.

[61] Aytac Azgin, Ravishankar Ravindran, and Guoqiang Wang. A Scalable Mobility-Centric Architecture for Named Data Networking. *CoRR*, abs/1406.7049, 2014.

[62] David Goergen, Thibault Cholez, Jérôme François, and Thomas Engel. Security Monitoring for Content-Centric Networking. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 274–286. Springer, 2013.

[63] Thibault Cholez. Introduction to Content-Centric Networking and the CCNx Framework. In *6th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2012)*, 2012.

[64] Punith P. Salian, Sachidananda Prabhu, Preetham Amin, Sumanth K. Naik, and M. K. Parashuram. Visible Light Communication. In *2013 Texas Instruments India Educators' Conference*, pages 379–383, April 2013.

[65] NDN Project Team. NDN Technical Memo: Naming Conventions. Technical report, 07 2014.

[66] Steven DiBenedetto, Paolo Gasti, Gene Tsudik, and Ersin Uzun. ANDaNA: Anonymous Named Data Networking Application. In *19th Annual Network & Distributed System Security Symposium*, NDSS 2012, 2012.

[67] Namespace-based Scope Control. https://redmine.named-data.net/projects/nfd/wiki/ScopeControl.

[68] NDN Frequently Asked Questions (FAQ). https://named-data.net/project/faq/.

[69] Wikipedia. Bonjour (Software) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Bonjour_(software)#Overview, 2017.

[70] Alexander Afanasyev, Cheng Yi, Lan Wang, Beichuan Zhang, and Lixia Zhang. SNAMP: Secure Namespace Mapping to Scale NDN Forwarding. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 281–286, April 2015.

[71] Link Object. https://named-data.net/doc/NDN-TLV/current/link.html.

Appendix A

Validator Configuration Files

A.1 For User Applications

A.1.1 Outer Packet Validation

The following configuration file validates *Data* packets arriving to a user application before decapsulation:

```
validator
{
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;; Validation of LB-NDN Data packets ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  rule
  {
    id "Rule to validate Data packets coming from LB-NDN in user device"
    for data
    filter
    {
      type name
      regex ^<this.+><><>+$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
        type name
        regex ^<localhost><lb-ndn><KEY><ksk-.+><ID-CERT>$
      }
    }
  }

  trust-anchor
  {
    type file
    file-name "_localhost_lb-ndn.ndncert"
  }
}
```

A.1.2 Inner Packet Validation

The following configuration file validates *Data* packets arriving to a user application after decapsulation:

```
validator
{
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;; Validation of device controller Data packets ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  rule
  {
    id "Rule to validate Data packets issued by a device controller"
    for data
    filter
    {
      type name
      regex ^<this.+><><>+$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
        type name
        hyper-relation
        {
          k-regex ^<>+(<>)<sn-.+><dev-ctrl><KEY><ksk-.+><ID-CERT>$
          k-expand \\1
          h-relation equal
          p-regex ^<>(<>)<>+$
          p-expand \\1
        }
      }
    }
  }

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;; Device trust hierarchy validation ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  rule
  {
    id "Rule to validate a manufacturer device identity certificate"
    for data
    filter
    {
      type name
      regex ^<>+<><sn-.+><dev-ctrl><KEY><ksk-.+><ID-CERT><>$
    }
```

```
  checker
  {
    type customized
    sig-type rsa-sha256
    key-locator
    {
      type name
      hyper-relation
      {
        k-regex ^(<>+<>)<ca><KEY><ksk-.+><ID-CERT>$
        k-expand \\1
        h-relation equal
        p-regex ^(<>+<>)<sn-.+><dev-ctrl><KEY><ksk-.+><ID-CERT><>$
        p-expand \\1
      }
    }
  }
}

rule
{
  id "Rule to validate a device manufacturer CA certificate"
  for data
  filter
  {
    type name
    regex ^[^<ic>]<>+<ca><KEY><ksk-.+><ID-CERT><>$
  }
  checker
  {
    type customized
    sig-type rsa-sha256
    key-locator
    {
      type name
      hyper-relation
      {
        k-regex ^<ic>(<>)<ca><KEY><ksk-.+><ID-CERT>$
        k-expand \\1
        h-relation equal
        p-regex ^<>+(<>)<ca><KEY><ksk-.+><ID-CERT><>$
        p-expand \\1
      }
    }
  }
}

rule
{
  id "Rule to validate an industry consortium category CA certificate"
  for data
  filter
  {
    type name
```

```
      regex ^<ic><><ca><KEY><ksk-.+><ID-CERT><>$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
        type name
        regex ^<ic><ca><KEY><ksk-.+><ID-CERT>$
      }
    }
  }

  trust-anchor
  {
    type file
    file-name "_ic_ca.ndncert"
  }
}
```

A.2 For LB-NDN

The following configuration file validates advertisement controller *Data* packets before decapsulation:

```
validator
{
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;; Validation of LB-NDN-AD Data packets ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  rule
  {
    id "Rule to validate Data packets issued by an advertisement controller"
    for data
    filter
    {
      type name
      regex ^<>+<lb-ndn-ad>[<advertise><withdraw><devices><device-info>]<>*$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
        type name
        regex ^<>+<lb-ndn-ad><KEY><ksk-.+><ID-CERT>$
      }
```

```
  }
}

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Infrastructure trust hierarchy validation ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rule
{
  id "Rule to validate an advertisement controller certificate"
  for data
  filter
  {
    type name
    regex ^<>+<lb-ndn-ad><KEY><ksk-.+><ID-CERT><>$
  }
  checker
  {
    type customized
    sig-type rsa-sha256
    key-locator
    {
      type name
      hyper-relation
      {
        k-regex ^(<>+)<ops><><KEY><ksk-.+><ID-CERT>$
        k-expand \\1
        h-relation equal
        p-regex ^(<>+)<lb-ndn-ad><KEY><ksk-.+><ID-CERT><>$
        p-expand \\1
      }
    }
  }
}

rule
{
  id "Rule to validate a location operator certificate"
  for data
  filter
  {
    type name
    regex ^<>+<ops><><KEY><ksk-.+><ID-CERT><>$
  }
  checker
  {
    type customized
    sig-type rsa-sha256
    key-locator
    {
      type name
      hyper-relation
      {
        k-regex ^(<>+)<ca><KEY><ksk-.+><ID-CERT>$
```

```
          k-expand \\1
          h-relation is-prefix-of
          p-regex ^(<>+)<ops><><KEY><ksk-.+><ID-CERT><>$
          p-expand \\1
        }
      }
    }
  }

  trust-anchor
  {
    type file
    file-name "_csu_ca.ndncert"
  }
}
```

A.3 For Device Controllers

The following configuration file validates *Interest* and *Data* packets before decapsulation:

```
validator
{
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;; Validation of LB-NDN Data packets ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  rule
  {
    id "Rule to validate Data packets coming from LB-NDN in service device"
    for data
    filter
    {
      type name
      regex ^<this.+><><>+$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
        type name
        regex ^<localhost><lb-ndn><KEY><ksk-.+><ID-CERT>$
      }
    }
  }

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;; Validation of LB-NDN Interest packets ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
  rule
  {
    id "Rule to validate Interest packets coming from LB-NDN in service device"
    for interest
    filter
    {
      type name
      regex ^<this.+><><>+$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
        type name
        regex ^<localhost><lb-ndn><KEY><ksk-.+><ID-CERT>$
      }
    }
  }

  trust-anchor
  {
    type file
    file-name "_localhost_lb-ndn.ndncert"
  }
}
```

A.4 For Advertisement Controllers

The following configuration file enforces the restricted policy described in Section 4.3.4:

```
validator
{
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;; Restricted policy devices ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  rule
  {
    id "Restricted policy for non-open categories"
    for interest
    filter
    {
      type name
      regex ^<thisBuilding><lb-ndn-ad><advertise><>$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
```

```
    key-locator
    {
      type name
      hyper-relation
      {
        k-regex ^[^<ic>](<>)<sn-.+><dev-ctrl><KEY><ksk-.+><ID-CERT>$
        k-expand \\1
        h-relation equal
        p-regex ^<thisBuilding><lb-ndn-ad><advertise>(<>)$
        p-expand \\1
      }
    }
  }
}

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Device trust hierarchy validation ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rule
{
  id "Rule to validate a manufacturer device identity certificate"
  for data
  filter
  {
    type name
    regex ^<>+<><sn-.+><dev-ctrl><KEY><ksk-.+><ID-CERT><>$
  }
  checker
  {
    type customized
    sig-type rsa-sha256
    key-locator
    {
      type name
      hyper-relation
      {
        k-regex ^(<>+<>)<ca><KEY><ksk-.+><ID-CERT>$
        k-expand \\1
        h-relation equal
        p-regex ^(<>+<>)<sn-.+><dev-ctrl><KEY><ksk-.+><ID-CERT><>$
        p-expand \\1
      }
    }
  }
}

rule
{
  id "Rule to validate a device manufacturer CA certificate"
  for data
  filter
  {
    type name
```

```
      regex ^[^<ic>]<>+<ca><KEY><ksk-.+><ID-CERT><>$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
        type name
        hyper-relation
        {
          k-regex ^<ic>(<>)<ca><KEY><ksk-.+><ID-CERT>$
          k-expand \\1
          h-relation equal
          p-regex ^<>+(<>)<ca><KEY><ksk-.+><ID-CERT><>$
          p-expand \\1
        }
      }
    }
  }

  rule
  {
    id "Rule to validate an industry consortium category CA certificate"
    for data
    filter
    {
      type name
      regex ^<ic><><ca><KEY><ksk-.+><ID-CERT><>$
    }
    checker
    {
      type customized
      sig-type rsa-sha256
      key-locator
      {
        type name
        regex ^<ic><ca><KEY><ksk-.+><ID-CERT>$
      }
    }
  }

  trust-anchor
  {
    type file
    file-name "_ic_ca.ndncert"
  }
}
```

A.5 For NFD

The following file shows the `localhost_security` portion of the `rib` section in NFD's configuration to restrict local prefix registrations to LB-NDN and the device controller for an HP printer:

```
localhost_security
{
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;; Validation of local registration requests ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  rule
  {
    id "Restrict local prefix registrations"
    for interest
    filter
    {
      type name
      regex ^<localhost><nfd><rib>[<register><unregister>]<>$
    }
    checker
    {
      type fixed-signer
      sig-type rsa-sha256
      signer
      {
        type file
        file-name "_localhost_lb-ndn.ndncert"
      }
      signer
      {
        type file
        file-name "_hp_printers_sn-1234_dev-ctrl.ndncert"
      }
    }
  }
}
```