Title
NDNFit: An Open mHealth Application Built on Named Data Networking

Author
Zhang, Haitao

Publication Date
2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

NDNFit: An Open mHealth Application Built on Named Data Networking

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Haitao Zhang

2018

ABSTRACT OF THE DISSERTATION

NDNFit: An Open mHealth Application Built on Named Data Networking

by

Haitao Zhang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2018
Professor Lixia Zhang, Chair

Inspired by the Open mHealth application architecture, which emphasizes user-controlled data security, reusable common modules and inter-operability among different mHealth applications, this dissertation introduces NDNFit, a mobile health (mHealth) application built on the Named Data Networking (NDN) architecture, while offer users the familiar user experience as traditional mHealth applications. An equally important motivation for NDNFit is NDN's application-driven architecture development philosophy — NDNFit serves as a use case to experiment with integrating multiple NDN components into one coherent application ecosystem, as well as drive the design and development of NDN architecture and protocol.

During the design and implementation process of NDNFit, we identified and solved five problems. First, because NDN mandates that all data packets be authenticated, NDNFit builds a namespace and certificate management system to manage identity and certificate, and defines trust policies for consumers to verify data packets; to protect confidentiality, it employs NAC to encrypt Data content, and makes the first effort to obscure Data names. Second, NDNFit employs Named Function Networking (NFN) to implement data processing services, defines name conversion services to enable sharing of named functions across multiple mHealth applications, and designs key delegation mechanisms for named functions to properly secure processed data. Third, to provide reliable data storage service, NDNFit designs DSU command protocol for users to communication with the storage, and designs mechanisms to employ State Vector Synchronization (SVS) protocol to replicate data in a

distributed data storage system. Fourth, NDNFit introduces catalog and denial of existence packets for efficient data transferring among different components. Last, NDNFit recognizes the issue of application mobility and refine forwarding hint to enable data reachability and support producer application mobility.

The NDNFit design illustrates that NDN's data-centric approach to networking — naming and securing data directly, and sharing namespace between application layer and network layer — provides a superior solution over the existing TCP/IP based solution for Open mHealth application architecture, as NDN network primitives enable users' control over their own data and facilitate interoperability among multiple applications, without relying on underlying transport layers and other third party services. The NDNFit design also demonstrates the power of NDN naming conventions — how Data packets are named, and what is the relationship between different Data names — in simplifying application design, that good naming conventions can help in many aspects: enable data-centric security, speed up data dissemination, or even improve data reachability.

At the end of this dissertation, we present the initial implementation of NDNFit as well as its demonstration on NDN testbed. Experimental results show that NDNFit design works well, and the lessons we learned from NDNFit design and development can benefit those of future NDN-based applications.

The dissertation of Haitao Zhang is approved.

Peter L Reiher

Songwu Lu

Mario Gerla

Lixia Zhang, Committee Chair

University of California, Los Angeles

2018

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to sincerely thank my academic advisor Prof. Lixia Zhang for providing invaluable support throughout my Ph.D. program. I also would like to express my deepest gratitude to Prof. Jeff Burke (UCLA REMAP — The Center for Research in Engineering, Media and Performance) for guiding me through the NDNFit design and implementation, without whom this work would not have existed. I want to thank my colleagues from UCLA Internet Research Laboratory Alexander Afanasyev, Yingdi Yu, and others for their support and technical discussions, which provides valuable insights into this work. I particularly thank Yingdi Yu, Alex Afanasyev, Prof. Van Jacobson, and Prof. Alex Halderman (University of Michigan) for their security-related contributions. I am also enormously grateful to our collaborators Zhehao Wang and many other members of the NDN team for insightful discussions that built up my understanding of Named Data Networking architecture and Open mHealth application architecture.

VITA

2011	B.Eng. (Electronic Information Science and Technology), Tsinghua University, Beijing, China.
2013	M.E. (Information and Communication Engineering), Tsinghua University, Beijing, China.
2013	B.S. (Economics, double major), Peking University, Beijing, China.
2013–2014	Project Manager, China Mobile Communications Corporation (CMCC), Beijing, China.
2015	Graduate Technical Intern, Intel Corporation, Hillsboro, Oregon.
2016	M.S. (Computer Science), UCLA, Los Angeles, California.
2016	Research Intern, Futurewei Technologies, Inc. (Huawei R&D USA), Santa Clara, California.
2017	Software Engineer Intern, Uber Technologies, Inc., Palo Alto, California.
2014–2018	Graduate Student Researcher, Computer Science Department, UCLA, Los Angeles, California.
2015–2018	Teaching Assistant, Computer Science Department, UCLA, Los Angeles, California.

PUBLICATIONS AND PRESENTATIONS

Haitao Zhang, Zhehao Wang, Christopher Scherb, Claudio Marxer, Jeff Burke, Lixia Zhang, and Christian Tschudin. "Sharing mhealth data via named data networking." In *Proceedings*

of the 3rd ACM Conference on Information-Centric Networking, pp. 142-147. ACM, 2016.

Jeff Burke, and Haitao Zhang, "NDN Demos III: NDNHealth", *SIGCOMM 2017 Tutorial (Half-Day): Named Data Networking (NDN) Tutorial*, ACM, 2017

Yanbiao Li, Alexander Afanasyev, Junxiao Shi, Haitao Zhang, Zhiyi Zhang, Tianxiang Li, Edward Lu, Beichuan Zhang, Lan Wang, and Lixia Zhang, "NDN Automatic Prefix Propagation", *Technical Report NDN-0048*, NDN, 2018.

Zhiyi Zhang, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang, "Security Support in Named Data Networking", *Technical Report NDN-0057*, NDN, 2018.

Xin Xu, Haitao Zhang, Tianxiang Li, and Lixia Zhang, "Achieving Resilient Data Availability in Wireless Sensor Networks", In *Proceedings of the IEEE International Conference on Communications (ICN-SRA)*. IEEE, 2018.

(Under Submission) Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang, "An Overview of Security Support in Named Data Networking", IEEE Communication Magazine.

(Under Submission) Haitao Zhang, Yanbiao Li, Zhiyi Zhang, Alexander Afanasyev, and Lixia Zhang, "NDN Host Model", ACM SIGCOMM Computer Communication Review (CCR).

(Under Submission) Tianxiang Li, Xin Xu, Haitao Zhang, Spyridon Mastorakis, and Lixia Zhang, "Distributed Dataset Synchronization in Mobile Ad Hoc Networks over NDN", the 5th ACM Conference on Information-Centric Networking.

CHAPTER 1

Introduction

The advent of smart mobile devices and mobile telecommunication technologies, enables mobile health (mHealth) — "medical and public health practice supported by mobile devices, such as mobile phones, patient monitoring devices, personal digital assistants (PDAs), and other wireless devices" [KST11]. mHealth provides the potential to deliver anytime and anywhere healthcare services, overcoming "geographical, temporal, and even organizational barriers" [SRT15]. However, most existing mHealth applications are built independently from each other, with little sharing of collected data, management mechanisms, processing methods, and analysis tools [CHS12]. The lack of interoperability hinders users from getting deeper insight into their health using mHealth applications [ES10].

Open mHealth [ES10, CHS12] is an open application architecture designed to facilitate the development and reuse of common modules across multiple different mHealth applications, thus enabling sharing and interoperability among them. Inspired by the success of the Internet's open architecture, Open mHealth employs a hourglass architecture, making data as the narrow waist. However, Open mHealth's *data-centric* vision is fundamentally different from today's TCP/IP Internet architecture practice, which treats hosts and concepts built on top of hosts, such as sessions and connections, as its building blocks. Consequently, this mismatch makes it challenging to achieve the design goals of Open mHealth by using the current TCP/IP architecture. User-controlled data sharing is an example: in Ohmage [RAF12, THL15], an Open mHealth reference application, access to users' data is managed through OAuth [Ham10, Har12], which requires users to store all their raw data in a storage provider, consequently users must trust the storage provider to faithfully control the usage of their health data; however, when the storage provider is compromised, users

lose direct or complete control over the usage of their health data; what's worse, the storage provide may abuse users' data even they are not compromised.

Named Data Networking (NDN) [ZEB10, ZAB14, JST09], which is proposed as one of the future Internet architectures, changes the network layer — the "thin waist" — communication model from delivering data to destinations identified by IP addressed to fetching named data from the network. At NDN's application layer, *consumers* send *Interest* packets to retrieve *Data* packets, *producers* publish data *a priori* or after receiving *Interest* packets; at NDN's network layer, forwarders forward application-created Interest packets based on their names, and forward application-created Data packets reversely along the way through which the requesting Interest packets come. With Interest-Data exchange as the basic communication paradigm, consumer applications can communicate with producer applications directly. NDN also secures data directly, and has developed a basic set of supporting mechanisms, including Identity, Data Signature [SJ09], NDN Certificate [Yu15], Trust Schema [YAC15], NDN Certificate Management Protocol (NDNCERT) [ZYA17, ZAZ17] and Name-based Access Control (NAC) [YAZ15], for applications to secure Data packets directly, offering intrinsic secure communications independent from the underlying communication channels [ZYZ18].

The conceptual similarity between Open mHealth and NDN's data centric architecture motivates us to explore the implementation of Open mHealth over NDN. Both are centered on standardized data format and data exchange: Open mHealth is an open application architecture, and NDN is an open network architecture. Both aim to empower end users to directly authenticate and control access to their data, regardless of where the data is stored and how it is transferred.

This dissertation has three research goals (i) Design and implement an Open mHealth application — NDNFit — on top of NDN, to truly enable user control of their own data, for example, which processing and visualization services to use, and with whom to share data. (ii) Serve as another case study in NDN's application-driven architecture development. (iii) Experiment with integrating multiple NDN components, such as namespace design, trust management and data confidentiality and producer mobility support, into one coherent

application ecosystem.

This dissertation identifies five problems in realizing Open mHealth using NDN: how enable data security, how to build sharable functions for data processing, how to provide reliable data storage service, how to transfer data using Interest-Data exchange communication model, and how to make data reachable even if data names are not globally routable. To explore answers for those five questions, this dissertation designs and implements NDNFit, an NDN-based location tracking and analyzing mHealth application which is built following Open mHealth architecture. We briefly introduce those problems and their solutions.

1.1 Data Security in NDNFit

Data security problem is motivated by the fact that mHealth data is highly private, so that data consumers need to check origin, integrity and authenticity of fetched data; at the same time, data producers need to make sure their data is only shared with authorized consumers, and data producers should have ultimate control over access to their data.

In the traditional TCP/IP network, data security is typically achieved by first authenticating hosts and then securing channels between two hosts (e.g., TLS [DR08]). However, this solution may not work well in NDNFit, as all produced data is first always stored into some storage, then data consumers fetch data from the storage. To make this solution work, both data producers and consumers should build trust with the storage, in which case the storage becomes the single point of failure. Moreover, there are two problems: (i) data loses security protection when coming out of secured channels; (ii) data owners lose control of access to their data.

NDNFit directly makes use of NDN's solution of securing data directly — Data packets are signed and encrypted directly — to enable data security. Securing data directly requires solutions to manage producer/consumer identity and certificate, which is achieved by building a namespace and certificate management system; to provide data authenticity, NDNFit defines trust policies; to protect confidentiality, NDNFit motivates the development of NAC and utilizes NAC to encrypt Data content, and makes the first effort to obscure Data names.

1.2 Named Data Processing

In NDN, data sources register prefixes to receive Interests whose names are under those prefixes, consumers send Interests to request named Data packets. Network forwards Interests, based on names, to data sources. On receiving an Interest, a data resource responds with a Data packet that satisfies the request expressed in the Interest. This model brings two advantages: (i) naming data enables NDN to integrate networking, processing, and storage all into one coherent system, as they all just supply named data; (ii) using names at network layer enables one to choose and pick processing and storage service at network layer (instead of application layer, which requires some third parties between users and service providers).

Generally speaking, in NDN, processing servers register the processing services they can offer, accept Interests under those names, and name the data they generate under those names. However, in NDNFit, the requirement of requesting data processing service is that the processing servers accept Interests under users' prefixes, and the processed data is owned by users and need to be named accordingly (under users' prefixes). That is a mismatch between what processing services can provide and what callers want.

NDNFit solves this problem by introducing Interest name conversion which enables consumers directly send Interests for desired Data, and processed result encapsulation which enables processing services accept Interests and produce Data named under their own prefixes. At the same time, NDNFit performs proper name allocation, certificate issuance, data signing and encryption to ensure user control over the data.

1.3 Reliable Data Storage

In NDNFit, data produced by any components should be permanently stored in a storage, which then provides data to all consumers. This calls for a reliable data storage service that can be invoked by users.

To this end, (i) NDNFit designs protocols for users to communication with the storage service. (ii) To improve reliability, the storage service consists of many storage nodes, and

we design a new dataset synchronization protocol to replicate all Data packets in all nodes.

1.4 Data Transport

In theory, data consumers do not need to know location of Data packets, as long as they know names of desired Data packets, they can compose and send Interests to the network, then the network can fetch Data packet back. However, there is a problem: how do data consumers know names of desired Data packets?

To solve this problem, NDNFit designs and develops both naming conventions and supporting mechanisms, called catalog and denial of existence, to help consumers learn the names of desired data in real-time communication.

1.5 Data Reachability

All Data names in NDNFit start with prefix "`/org/openmhealth`", which represents Open mHealth and is location-independent, resulting Data packets not globally reachable. In other words, if consumers directly send Interests for NDNFit data, the NDN network may not know where to forward Interest packets and get Data packets back. So the problem is, how to help NDN network figure out how to forward Interest packets?

NDNFit define mechanisms, providing forwarding hints to consumers — by using naming conventions, NDNS system, or even real-time communication, so that consumers can provide that information to help network to decide where to forward Interests.

The rest part of this dissertation is organized as follows. In Chapter 2, we briefly introduce Open mHealth architecture and Named Data Networking (NDN), which serve as foundation of this dissertation. Chapter 3 introduces the experimental NDNFit application, identifies problems we need to solve in this dissertation, and presents the namespace designed for NDNFit. Chapter 4, 5, 6, 7, and 8 discuss and design solutions for the previously-mentioned five problems one by one. In Chapter 9, we show our first implementation and its demonstration on NDN testbed. Chapter 10 discusses lessons we learn through the design

and implementation of NDNFit, as well as remaining challenges. Chapter 11 shows our future plan. Chapter 12 concludes our work.

CHAPTER 2

Background

This chapter introduces an overview of the Open mHealth architecture and the Named Data Networking (NDN) architecture.

2.1 Open mHealth

As demonstrated in Figure 2.1, compared with stovepipe mHealth applications (i.e., mHealth applications that are designed and implemented independently), applications in Open mHealth architecture can share modules — analysis/visualization/feedback, processing, data APIs, protocols, data transport, data management and data capture — with each other. Those shared modules are built following open application programming interfaces (APIs) around a minimal set of common standards. With data APIs and protocols as the simple point of commonality to inter-connect other modules, those shared modules form an hour glass architecture.

In Open mHealth architecture, independently developed software components built following a common set of principles and APIs, which are specifically defined for reusable software modules, can be mixed and matched, and work together in a plug-and-play fashion. The reusable software components in Open mHealth can be classified into three types: Data Storage Units (DSUs), Data Processing Units (DPUs) and Data Visualization Units (DVUs).

Data flow in Open mHealth architecture is shown in Figure 2.2. DSUs store output data produced by mHealth applications and DPUs, and provide input data to DPUs and DVUs. Meanwhile, as data is highly private in mhealth application, data owners (i.e., users) want to have fully control of whom to share their data with (i.e., only authorized entities can

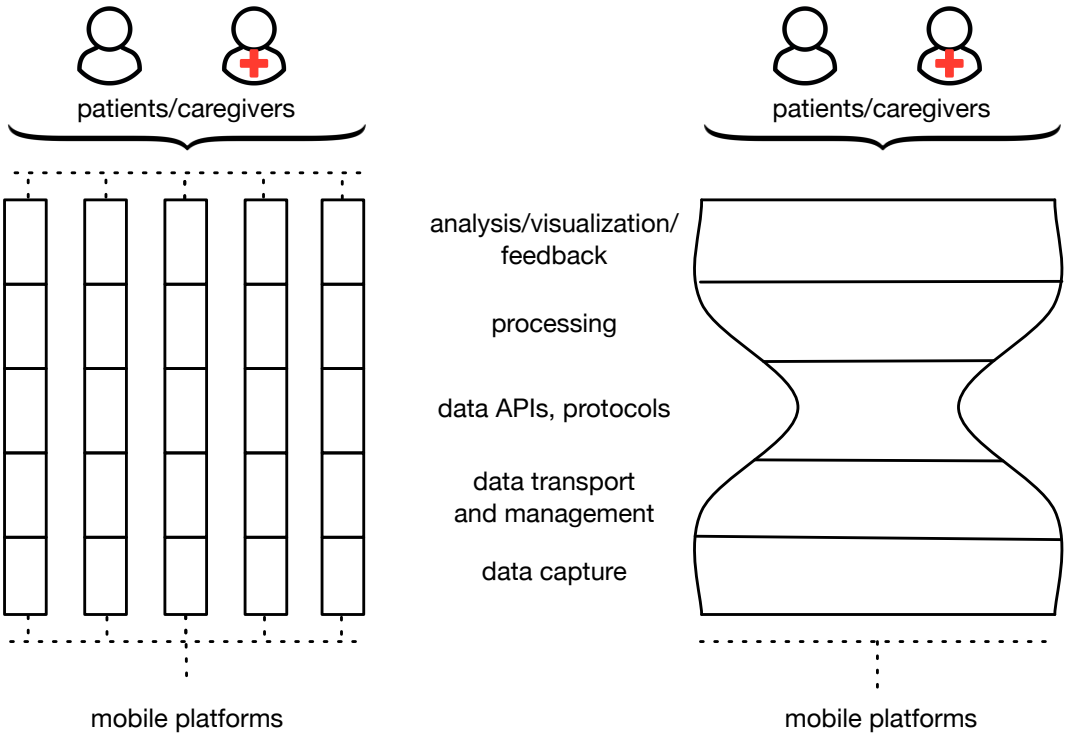


Figure 2.1: Stovepipe mHealth Applications versus Open mHealth Architecture

get access to their data), which is usually achieved by delegating DSUs the authority to manage access to data stored in them. DPUs take data as input, process them to produce some new data (e.g., extract a user's running path), and put those new data back to DSUs. DVUs take data as input, and present them in human-understandable ways. Each DPU and DVU typically performs one single function; when one function is not enough to produce the desired results, multiple DPUs and DVUs can be chained or composed.

With Open mHealth architecture, mHealth applications are not proprietary any more; instead, they can reuse those sharable DSUs, DPUs and DVUs, and interoperate with each other. A typical structure to "glue" mHealth applications and other components together to form a working system is shown in Figure 2.3. In this structure, mHealth applications focus on realizing data capturing related functions; they take DSUs, DPUs and DVUs as common services, and communicate with them using protocols defined by Open mHealth.

The Open mHealth architecture is expected to enable innovation both above and be-

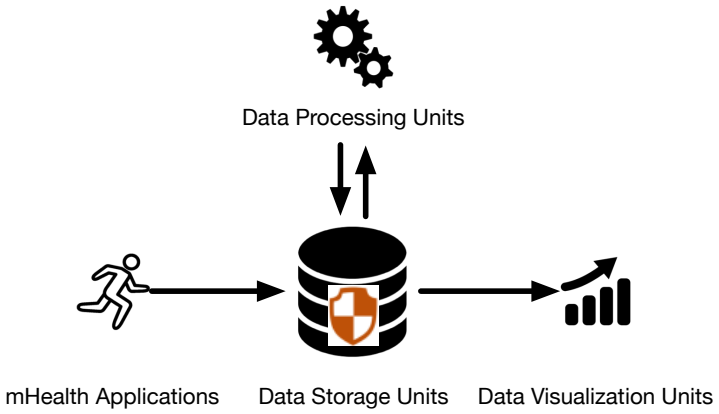


Figure 2.2: Data Flow in Open mHealth Architecture

low the thin waist, encourage cross-disciplinary collaborations, and maximize participation patients, caregivers, software developers and health researchers.

There are problems exist in realizing open mHealth using the traditional TCP/IP network architecture. The fundamental one arises from achieving user-controlled data sharing using session-based security. In session-based security, when data needs to be passed to another authorized host, an end-to-end secured channel is set up first, through which data is delivered. However, this practice suffers from two problems: (i) Data is no longer secured when it comes out of the secured channel between two hosts, so hosts need to be trusted; (ii) As other entities get data from data storage servers, users need to delegate access control to those servers, which is not truly user-controlled any more.

2.2 Named Data Networking

Similar to today's TCP/IP architecture, NDN architecture has an hourglass shape; however, NDN entirely changes the network thin waist of TCP/IP architecture and layers above it (see Figure 2.4). At the network layer, instead of identifying each host interface using an IP address and forwarding packets based on IP addresses, NDN identifies each piece of content using a *name* and forwarding packets based on names. In NDN, the application layer runs directly on top of the network layer; the network layer runs directly on top of the data link

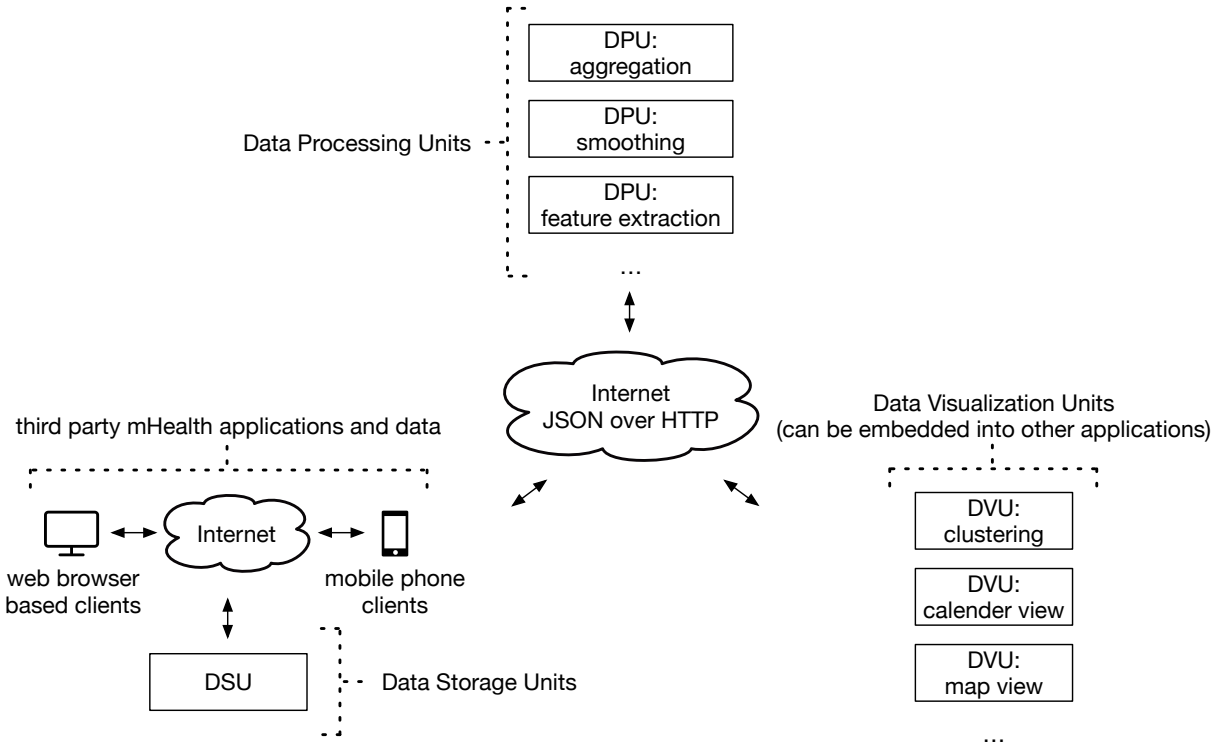


Figure 2.3: Modules of Open mHealth Architecture

layer or TCP/UDP/IP tunnels (running the network on top of TCP/UDP/IP tunnels is a temporary solution to enable NDN in the current TCP/IP network); there is not a separate transport layer, transport protocols are part of applications' logic.

NDN applications can play three different roles: *producer*, *consumer* and *data storage*. Producers divide application content into pieces, name each piece according to predefined naming conventions, and sign each using a signing key to create *Data* packets; consumers use the name or the name prefix of the desired Data packets to compose *Interest* packets; Data storage does not produce Interest or Data packets, they store Data packets produced by themselves or others. Consumers send Interest packets to NDN networks to ask for Data packets; producers and data storage reply Interest packets with Data packets when they receive Interest packets and have the desired Data.

NDN networks, or more specifically, NDN forwarders, forward Interest packets based on names, and forward Data packets along the reverse forwarding path of the requesting

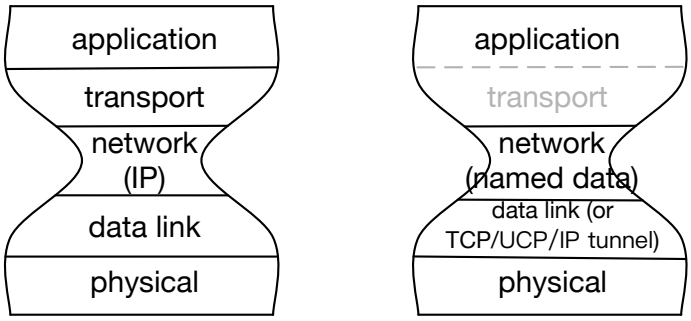


Figure 2.4: TCP/IP Architecture vs NDN Architecture

Interest packets recorded in NDN forwarders [ASZ16]. A data structure called Forwarding Information Base (FIB) is used to determine which next hops to send an Interests packet (i.e., where the corresponding Data packet may be found) when an NDN Forwarder needs to forward the Interests packet out. The FIB consists of a list of entries, each is keyed by a name prefix and contains one or multiple *faces* (In NDN, a face represents a communication channel between two applications [1], for example, a TCP channel between two NDN forwards, or an inter-process communication channel between an NDN forward and an application.) as potential next hops. When an Interest packet does not carry the *Forwarding Hint* [AYW15a, AYW15b], which represents a namespace forwarding delegation, an NDN forwarder finds a FIB entry whose key matches the Interest name using the "longest prefix match" algorithm, and forwards the Interest packet to all or some of [2] the next hop faces contained in the FIB entry; each forwarder does the same thing until the desired Data packet is found along the way or reaches a producer or Data cache. When an Interest packet carries the forwarding hint, NDN forwarders first forward the Interest packet based on the forwarding hint until it reaches the local network environment represented by the forwarding hint (i.e., when the forwarding hint is a prefix of the local network name), then NDN forwarders strip off the forwarding hint and forward the Interest packet based on its own name. Actually, NDN packet forwarding logic also involves the Pending Interest Table (PIT), the Content Store

[1] NDN fowarder is a special kind of application.

[2] The decision is made by the Forwarding Strategy that applied to the Interest namespace.

(CS) and the Forwarding Strategy, but they are out of scope of this dissertation; more details of NDN packet forwarding logic are described in [ZAB14] and [ASZ16].

NDN provides a built-in security framework that is based on public-key cryptography [Sal13]. Every NDN entity (i.e., network communication participant) should possess one or more identities (i.e., unique NDN names), and each identity should be associated with one or more cryptographic public-private key pairs. The private key should be kept securely by the identity owner; the corresponding public key should be certified by one or more NDN certificates, each is issued as a Data packet by either the identity itself or some other identities. A producer uses its private key to sign its Data packets; after retrieving a Data packet, consumers fetch the producer's public key certificate to verify signature of the Data packet, checking *integrity* of the Data packet. With trust anchor(s) and trust rule(s) configured, producers can produce Data packets by following the certificate chain; meanwhile, with the same trust anchor(s) and trust rule(s), consumers can verify *authenticity* of the Data packet by backtracking and verifying the integrity and authenticity of certificates along the certificate chain to the trust anchor. When *confidentiality* of the Data packet needs to be ensured, the producer can encrypt the content using a private key, and share the corresponding public key only with certain authorized consumers via some well-defined secured ways.

CHAPTER 3

Problem and Design Overview

This chapter first introduces NDNFit, an NDN-based location tracking and analyzing mHealth application designed to explore how we can implement Open mHealth using NDN, and a typical use case of NDNFit for readers to better understand its working process (Section 3.1). Then we summarize problems in realizing Open mHealth using NDN (Section 3.2), which are what we need to solve in this dissertation. Last, we design namespace for NDNFit based on its system requirements (Section 3.3).

3.1 NDNFit as a Pilot Application

As a pilot Open mHealth application that uses NDN to communicate, NDNFit tries to offer users the familiar user experience as traditional mHealth applications, while enables users to directly exercise security over their data, and compose their own data processing and presentation networks. NDNFit follows the Open mHealth paradigm, while adapting its REST-based communication model to a data dissemination approach using NDN. The application must be constructed from an system with each type of component envisioned by the Open mHealth, including data capture applications, DVU, DPU and DVU. Each component could be provided by different service providers at different stages, rather than a proprietary application or service.

To limit the problem scope, NDNFit focuses its function specifically on location tracking and analyzing. From a user's point of view, with NDNFit, she can (i) request an identity; (ii) provide authenticity of her data; (iii) grant consumers access to her data; (iv) collect real-time location data; (v) report captured data to DSUs; (vi) select DPUs to process her

data and extra features as new data; (vii) use DVUs to view statistics about her data.

A typical use case of NDNFit is shown as following. A user "Alice" runs the data capture application on her phone to collect time-location data when she is running. Alice uploads her data to a DSU when it is convenient, for example, when she connects her phone to a Wi-Fi network. Another user "Bob" fetches Alice's data and visualizes it using his DVU instance — for example, a JavaScript program running in browser. Alice invokes some DPUs to count her steps using her raw data, the resulting new data is still owned by Alice.

3.2 Problems in Realizing Open mHealth using NDN

When implemented, NDNFit can take advantages of NDN's built-in features: signed Data can be treated as the common data format, so no application-specific data format is needed; Interest-Data exchange can be used as the basic communication paradigm, so no application-specific API is needed.

However, there are still five problems need to be solved. First, security mechanisms are needed to manage identities, and provide data integrity, authenticity and confidentiality support. A basic requirement is, every entity evolved in the system should be uniquely identified. As health data is highly private and sensitive, data consumers should be able to verify that data they receive is the same as when providers send it, the data is indeed produced by the one who claims to be the producer, and a producer is trustworthy. More importantly, data owners (e.g., Alice in the previous example) should be able to control access to their data, that is, only consumers who are granted access by a data owner can get access to the owner's data, and the data owner can decide whether to grant access to her data or not. In NDN, as data names include abundance application semantics, proper mechanisms should also be adopted to prevent that information to be leaked to attackers.

Second, remote processing services are invoked by users (e.g., Alice) to process their data, producing new data that still belongs to users. To implement this function, we need to figure out how to invoke a remote processing service, how to provide parameters to the remote processing service, how to name the output of the remote processing service, how

invokers (i.e., users) authenticate the remote processing service, how the remote processing service authenticates invokers, how to protect privacy of both the invokers and the remote processing service, and how to properly exercise security on the processed output of the remote processing service.

Third, a reliable data storage service is required to store mHealth data. The DSU service, which should be able to reliably keep users' data for a long time period, is a built-in component of Open mHealth architecture. An implicit requirement for DSU is that, it can be invoked to store data produced by any data capture application, DPU or DVU, whenever needed. Notice that, the DSU service does not necessarily provide security features; instead, it just needs to keep and serve data as it is received.

Fourth, when work as consumers, components should be able to retrieve data correctly and efficiently, by using NDN's data fetching, or say, Interest-Data exchange, communication model. This requires consumer components to have reliable ways to learn Data names before sending Interests for Data packets.

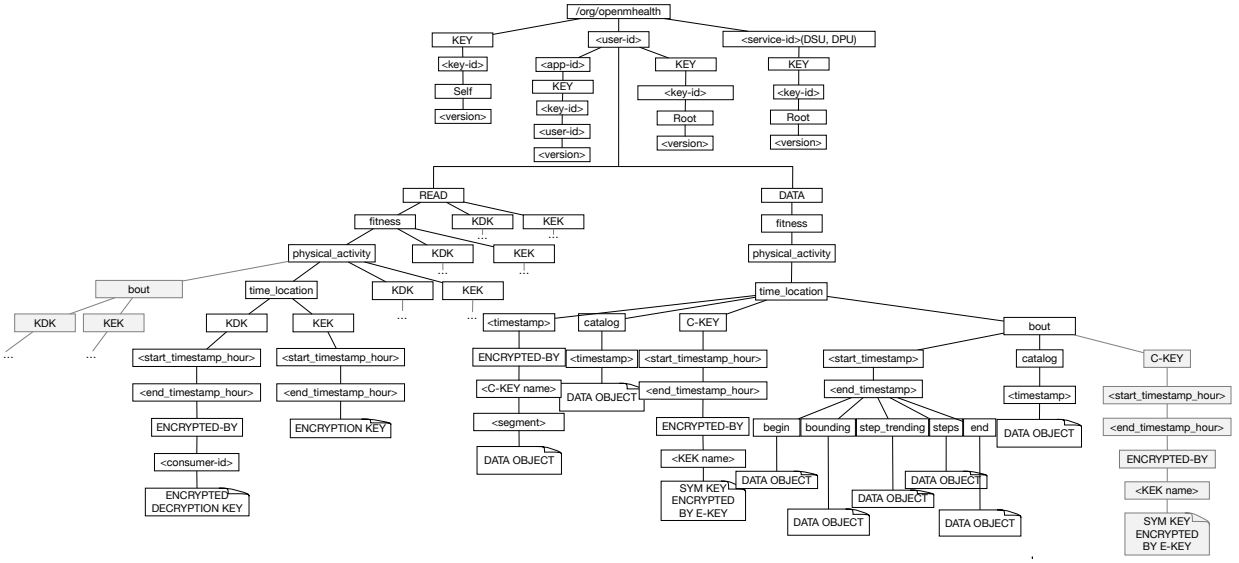
Last, NDNFit Data names are not globally routable, therefore consumer components need to learn data provider's reachability information and attach it to Interest packets, so that NDN network can retrieve Data packets correctly. A more complex case is when Data providers roam from one network to another, in which case consumer components need to keep their knowledge about data providers' attachment points up to date, so that they can provide proper reachability information to help network to forward Interest correctly. A typical example is, when Alice takes her mobile phone, which runs the NDNFit data capture application, from one ISP (e.g., her home network ISP) to another (e.g., her campus network ISP), the DSU can still fetch data from the data capture application.

3.3 Namespace Design

NDNFit namespace is designed iteratively to meet several requirements brought up by NDNFit designs: (i) to name data with meaningful classifications; (ii) to reflect data ownership; (iii) to reflect trust relationships among different entities; (iv) to indicate data encryption

and decryption information; (v) to disseminate data naming information. Although readers can fully understand those requirements only after reading and understanding designs in the following several chapters, we list them here to give an overview; readers are highly recommended to come back to go through them again to gain a deeper understanding.

The current NDNFit namespace is illustrated in Figure 3.1. Here are the detailed explanations:


/org/openmhealth
KEY
<key-id>
Self
<version>
<user-id>
<app-id>
KEY
<key-id>
<user-id>
<version>
KEY
<key-id>
Root
<version>
<service-id>(DSU, DPU)
KEY
<key-id>
Root
<version>
READ
DATA
fitness
KDK
KEK
physical_activity
KDK
KEK
bout
time_location
KDK
KEK
KDK
KEK
KDK
KEK
<start_timestamp_hour>
<end_timestamp_hour>
ENCRYPTED-BY
<consumer-id>
ENCRYPTED DECRYPTION KEY
<start_timestamp_hour>
<end_timestamp_hour>
ENCRYPTION KEY
fitness
physical_activity
time_location
<timestamp>
catalog
C-KEY
ENCRYPTED-BY
<C-KEY name>

DATA OBJECT
<timestamp>
<start_timestamp_hour>
DATA OBJECT
<end_timestamp_hour>
ENCRYPTED-BY
<KEK name>
SYM KEY ENCRYPTED BY E-KEY
<start_timestamp>
<end_timestamp>
begin
bounding
step_trending
steps
end
DATA OBJECT
DATA OBJECT
DATA OBJECT
DATA OBJECT
bout
catalog
<timestamp>
DATA OBJECT
C-KEY
<start_timestamp_hour>
<end_timestamp_hour>
ENCRYPTED-BY
<KEK name>
SYM KEY ENCRYPTED BY E-KEY

Figure 3.1: NDNFit Namespace and Data Objects

1. The first two components, "`/org/openmhealth`", compose the root prefix of Open mHealth, which is the prefix for all potential Open mHealth applications;

2. The self-signed certificate, "`/org/openmhealth/KEY/<key-id>/Self/<version>`", serves as the trust anchor of all Open mHealth applications — including NDNFit, as will be seen in Section 4.1;

3. Each user (the human user) has at least one identity, represented by the component "`<user-id>`";

4. Each DSU, DPU and DVU also has at least one identity, represented by the component "`<service-id>`";

5. Each "<user-id>" and "<service-id>" is associated with a certificate, "/org/openmhealth/<user or service id>/KEY/<key-id>/Root/<version>", which is issued by the Open mHealth trust anchor;

6. Each user can run multiple mHealth data capture applications, each of which has at least one identity, represented by the component "<app-id>";

7. Each "<app-id>" is associated with a certificate, "/org/openmhealth/<user-id>/<app-id>/KEY/<key-id>/<user-id>/<version>", which is issued by the user who owns this mHealth data capture application;

8. Each user namespace also has two sub-namespaces, "DATA" and "READ", where "DATA" contains location data and features extracted from location data, "READ" contains encryption and decryption data used in data access control;

9. The components "fitness", "physical_activity" and "time_location" classify location data according to 3 hierarchical levels;

10. Each location Data packet is named with a component "<timestamp>". "timestamp" name components are represented in ISO 8601 format, i.e., YYYYMMDDThhmmss. When the data size for a specific time range (starts from "<timestamp>", and ends at "<timestamp>" + a predefined time interval) exceeds the NDN maximum packet size, the data is split into segments, and named with component "";

11. The components "/catalog/<timestamp>" provide a shortcut for consumers to quickly learn names of Data packets covered by its time range (starts from "<timestamp>", and ends at "<timestamp>" + a predefined time interval), as will be seen in Section 7.1.3;

12. Those location data can be further annotated by various DPU services, generating data under sub-namespace "bout". Example basic annotations include starting point information ("bout/<start_timestamp>/<end_timestamp>/begin"), ending point information ("bout/<start_timestamp>/<end_timestamp>/end"), the location bounding edge ("bout/<start_timestamp>/<end_timestamp>/bounding") and the step count

("bout/<start_timestamp>/<end_timestamp>/steps") of all location data covered by the time range (starts from "<start_timestamp>", and ends at "<end_timestamp>"); using those basic annotations, advanced annotations can be produced, for example, step count trending ("bout/<start_timestamp>/<end_timestamp>/step_trending") is derived from the basic step count annotation data covered by the time range. The components "/catalog/<timestamp>" serve the same purpose as mentioned before.

13. The "/org/openmhealth/<user-id>/READ" sub-namespace and "/<prefix>/C-KEY" sub-namespace will be explained in Section 4.4.2.

CHAPTER 4

Data Security

This chapter illustrates how NDNFit provides standardized identity management mechanisms (Section 4.1), guarantees data integrity (Section 4.2) and data authenticity (Section 4.3), as well as enables data owners to directly protect data content and name confidentiality (Section 4.4). As NDN is a data-centric architecture, we try to achieve those goals in a data-centric way.

4.1 Identity Management

In NDN, every entity should have at least one name, and every name should be unique within its scope [1]. Thus in NDNFit, users can take names as identities. Using names as identities brings several advantages:

1. An identity is guaranteed to be unique within the Open mHealth system.

2. An identity has built-in application semantics. For example, we can learn from the first two components that "`/org/openmhealth/Alice`" is an identity used in Open mHealth.

3. A user can name her data using names picked from namespaces represented by her identities; those names can also indicate data's ownership. For example, a name "`/org/openmhealth/Alice/DATA/fitness/physical_activity/time_location/20180501100000/0`" indicates that this Data belongs to an Open mHealth user "Alice".

[1] Scope means the realm a name is used. For example, a NDNFit user name "`/org/openmhealth/Alice`" is used in the global network, so it should be globally unique; the NFD name "`/localhost/nfd`" is used in the current NFD, so it only needs to be locally unique.

4. A user can assign identities to other entities, by allocating sub-namespaces of her identities to them. For example, the Open mHealth user "`/org/openmhealth/Alice`" can assign identity "`/org/openmhealth/Alice/NDNFit`" to the NDNFit data capture application she uses to collection location data.

Following the same idea, other Open mHealth components — data capture applications, DSUs, DPUs and DVUs — in NDNFit can take their names as identities as well.

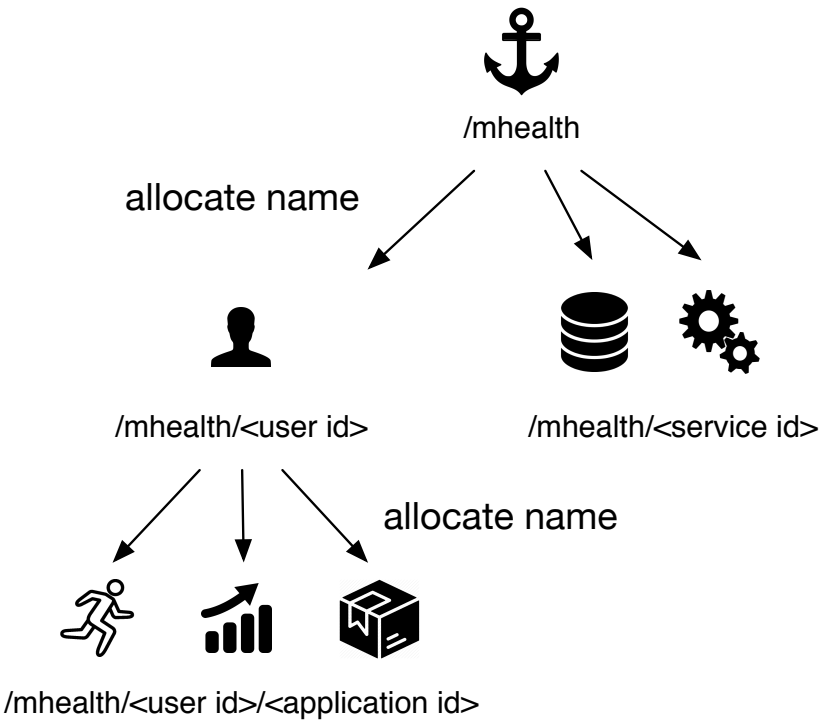


Figure 4.1: Identity Management in NDNFit

We have defined rules to standardize the identity (i.e., the namespace) assignment and management process. (i) There is an Open mHealth system root, who is manually assigned the globally unique identity "`/org/openmhealth`". (ii) All NDNFit users and sharable Open mHealth components (DSUs and DPUs) request unique identities, "`/org/openmhealth/<user or service id>`", from the Open mHealth system root. (iii) Those sharable Open mHealth components are not allowed to assign identities to other entities. (iv) NDNFit users can assign identities, "`/org/openmhealth/<user id>/<application id>`", to applications they use,

including but not limiting to NDNFit data capture application, Access Control Manager (an application produces Data for controlling access to users' Data), and DVU application. (v) Users' applications are not allowed to assign identities to other entities; they produce Data according to predefined naming conventions. See Figure 4.1 for the identity management logic.

4.2 Data Integrity

Every NDN Data packet should be cryptographically signed by its producer to generate the "Signature" (Figure 4.2). The "Signature" is a built-in part of every Data packet. The correctness of "Signature" ensures data integrity, i.e., data is not changed during its transmission in the network, regardless of how and where the Data packet is retrieved from.

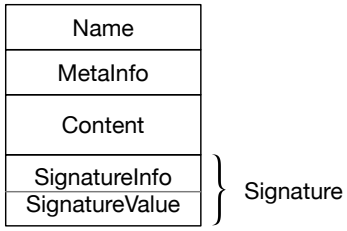


Figure 4.2: NDN Data Packet

We briefly explain how the signature works. Every producer has a public-private signing key pair K_{pub} and K_{pri}. To generate the "SignatureValue", the producer first configures the "SignatureInfo", including the algorithm of generating the "SignatureValue" (such as Digital Signature Algorithm, DSA, [Kra93]) and other information; then it calculates SHA256 (Secure Hash Algorithm 2, the hash function with digest — hash value — that is 256 bits) [NIS13] digest of the whole Data packet, excluding the "SignatureValue" itself

$$SHA256\_digest = SHA256(Name, MetaInfo, Content, SignatureInfo) \tag{4.1}$$

last, it creates the "SignatureValue" with the chosen signing algorithm and its own private

key

$$SignatureValue = Sign_{(K_{pri})}(SHA256\_digest) \tag{4.2}$$

If a consumer can get the producer's public key, it first calculates SHA256 digest of the whole Data packet using equation 4.1, then it verifies the "SignatureValue" with the chosen signing algorithm (indicated by "SignatureInfo" of the Data packet) and the producer's public key

$$Verify_{(K_{pub})}(SHA256\_digest, SignatureValue) \tag{4.3}$$

4.3 Data Authenticity

NDN Data signature can also help consumers to verify data authenticity (i.e., a Data packet is produced by the entity it claims to be produced by), as long as consumers have the producer's public key K_{pub} and the producer keeps its private key K_{pri} safe. That is because, other entities cannot guess or calculate a producer's private key, even if they have the producer's public key [2]; meanwhile, without the correct private key, other entities cannot produce signature that can be verified by consumers using Equation 4.3. Therefore, the key problem in verifying data authenticity is, how consumers get the producer's public key K_{pub} and make sure it is trustworthy.

NDN provides built-in mechanisms to solve the previous problem:

1. If consumers get the producer's public key K_{pub} using a safe out-of-band method, then no other actions are needed. For example, the system root identity ("`/org/openmhealth`")'s public key — contained in a self-signed certificate — is pre-installed in all Open mHealth components and applications; therefore, they can verify authenticity of Data packets produced by the system root.

2. Otherwise, a producer names its public key K_{pub} according to predefined conventions, "`/<identity>/KEY/<key id>`", to bind the key with its identity. For example, the NDNFit data capture application with an identity "`/org/openmhealth/alice/ndnfit`"

[2] In theory, other entities can calculate a producer's private key; but in practice, that takes too long.

can name its public key K_{pub} "/org/openmhealth/alice/ndnfit/KEY/abcdEFG"; when consumers get a public key with this name, they know that it should belong to "/org/openmhealth/alice/ndnfit".

3. To certify that this public key belongs to the NDNFit data capture application identity "/org/openmhealth/alice/ndnfit", the application requests a certificate from another identity (the certificate issuer). The certificate is an NDN Data packet, whose name is "/org/openmhealth/alice/ndnfit/KEY/<key id>/<issuer id>/<version>", and whose content is the public key bits, and is signed by the issuer. When creates Data packets, the application also puts information about the certificate into "SignatureInfo" field; when get Data packets, consumers extract information about the public signing key certificate from the "SignatureInfo" field, and fetch the certificate as well. For example, the data capture application can request a certificate from identity "/org/openmhealth/alice", an example certificate name can be "/org/openmhealth/alice/ndnfit/KEY/abcdEFG/alice/2";

4. To verify authenticity of this certificate, consumers should also get the certificate issuer's public key, which is exactly the same problem we are trying to solve now. Following the same process, consumers fetch a chain of certificate. To determine whether Data packets are trustworthy or not, consumers are configured trust rules, which define (i) acceptable relationships between Data packet (including certificate packet) names and their public signing key certificate names; and (ii) acceptable trust anchors (the certificates or public keys that do not need to be further verified). When names of the Data packet and the corresponding chain of certificate stick to the trust rules, and a consumer has one of the certificates pre-installed, the consumer accepts the Data packet; otherwise, the consumer rejects the Data packet.

In NDNFit, we defined rules to standardize the (public signing key) certificate issuance management process, which are similar to the identity assignment and management rules defined in Section 4.1. (i) The Open mHealth system root has a public key "/org/openmhealth/KEY/<key id>" (and the corresponding private key), and a self-signed certificate "/org

/openmhealth/KEY/<key id>/Self/<version>", where the component "self" indicates that this certificate is signed by itself. (ii) All NDNFit users and sharable Open mHealth components (DSUs and DPUs) generate their own public-private key pairs, and name the public key "/org/openmhealth/<user or service id>/KEY/<key id>"; they request certificates, "/org/openmhealth/<user or service id>/Root/<key id>", from the Open mHealth system root, where the component "Root" indicates that this certificate is signed by the Open mHealth system root. (iii) NDNFit users can issue certificates, "/org/openmhealth/<user id>/<application id>/KEY/<key id>/<user id>/<version>", where the second component "<user id>" indicates that this certificate is signed by that user identity, to applications they use, which generate the public key "/org/openmhealth/<user id>/<application id>/KEY/<key id>". See Figure 4.3 for the certificate management logic.

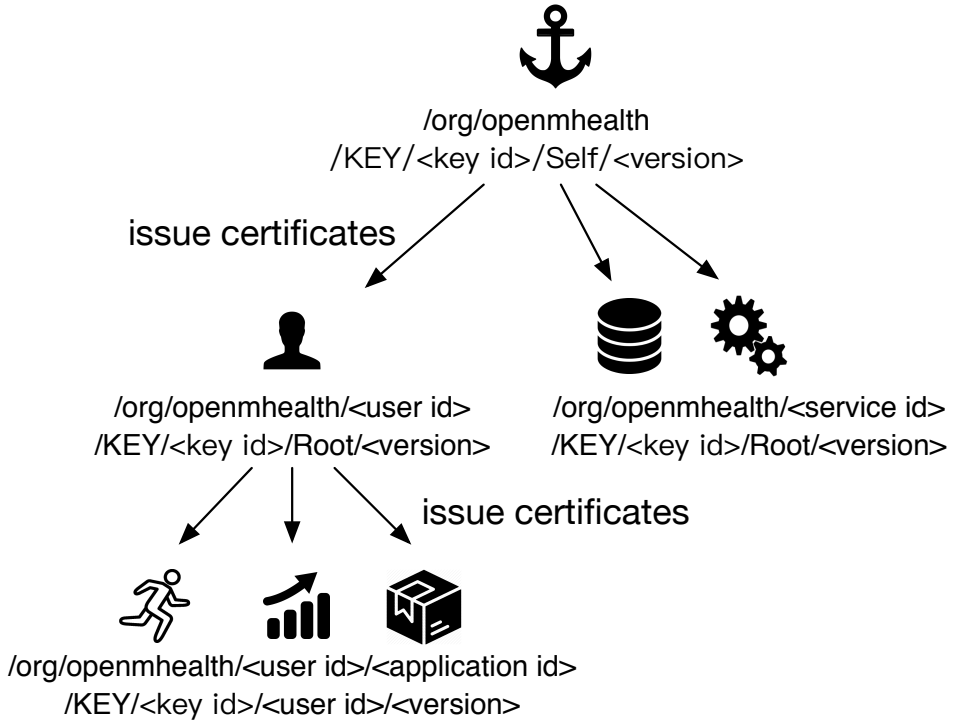


Figure 4.3: Certificate Management in NDNFit

Data consumers' trust rules should follow the certificate management logic defined in the previous paragraph. Figure 4.4 shows the trust rules used in verifying Data produced by NDNFit data capture applications. Notice that components with the same name are

literally the same. Trust rules used in verifying Data produced by Open mHealth sharable components are introduced in Section 5.3.1, which need more background to understand. In practice, those trust rules are expressed using trust schema [YAC15], where the relationship between Data names and their expected public signing key certificate names can be expressed using regular expressions. The trust schema for data produced by the NDNFit data capture application is preconfigured and is published as a Data packet, named "`/org/openmhealth/trust_schema/NDNFit/<version>`" and signed by the Open mHealth root, enabling data consumers to fetch it from the network, consult it and verify any received NDNFit data consistently, regardless of where it comes from or where it is stored, eliminating the dependency on session-based security. This model can also be easily extended to the non-hierarchical, web-of-trust style models, which are not discussed in this dissertation.

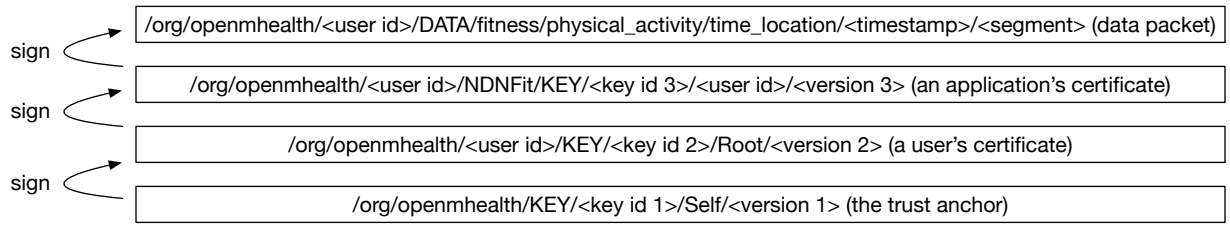

/org/openmhealth/<user id>/DATA/fitness/physical_activity/time_location/<timestamp>/ (data packet)
sign
/org/openmhealth/<user id>/NDNFit/KEY/<key id 3>/<user id>/<version 3> (an application's certificate)
sign
/org/openmhealth/<user id>/KEY/<key id 2>/Root/<version 2> (a user's certificate)
sign
/org/openmhealth/KEY/<key id 1>/Self/<version 1> (the trust anchor)

Figure 4.4: Trust Rules Used in Verifying Data Produced by NDNFit

4.4 Data Confidentiality

In NDNFit, entities play three different roles: data owner, data producer and data consumer. For example, the owner of Data "`/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501100000/0`" is "`/org/openmhealth/alice`", its producer is "`/org/openmhealth/alice/NDNFit`"; entities who want to fetch this Data packet is its consumers.

Data confidentiality refers to that only consumers who are granted access by data owners, but not producers, can get information from certain Data packets. Confidentiality can be achieved through encryption.

Among four fields — Name, MetaInfo, Content and Signature — of an NDN Data packet

that contain information, (i) It is less possible to encrypt information contained in the Signature field, as those information is needed to verify data authenticity; (ii) The name or its prefix is contained in Interest packets for retrieving this Data packet, and will be used for network forwarding (when there is no forwarding hint), so we need to consider the affects on forwarding when try to encrypt name; (iii) Similarly, the MetaInfo field may contain information that can be used by routers, for example, the FreshnessPeriod can be used by routers to decide whether a Data packet is stale or not, so it is less possible to encrypt it; (iv) The Content field is only inspected by applications, so it is safe to encrypt it without affecting other functions. Therefore, NDNFit focuses on content confidentiality (Section 4.4.2) and name confidentiality (Section 4.4.3).

4.4.1 General Confidentiality Solutions

Even in NDN, securing point-to-point sessions is still possible. Key exchange protocols like Diffie-Hellman [DH76] can still help both sides — one side serves as a data owner and data producer at the same time, the other side is a data owner — of the session to negotiate encryption and decryption keys for the sessions, with both of them well-aware of the key information [MUW17].

However, in NDNFit, session-based security cannot achieve the goal of user-controlled data sharing (see Section 2.1 for the analysis); data owners want to take advantages of named and secured data to directly authorize multiple consumers and share Data with them. Motivated by Section 4.3, where per-packet signature and trust schema enable session-independent data authenticity, NDNFit seeks to make use of per-packet encryption and usable key distribution mechanisms to enable session-independent data confidentiality. Again, in NDN, since Data names are structured and convey rich semantics, we explore approaches that leverage systematic naming conventions to achieve the goal.

4.4.2 Content Confidentiality

The mechanism to achieve content confidentiality, which treats NDNFit as a typical use case, is Name-based Access Control (NAC) [YAZ15]. This section introduces the basic NAC mechanism first (Section 4.4.2.1); then we design specific naming conventions for employing NAC in NDNFit (Section 4.4.2.2), and use an example to show how NAC works in NDNFit (Section 4.4.2.3); finally, we summarize the work (Section 4.4.2.4).

4.4.2.1 Name-Based Access Control

Figure 4.5 shows the relations among entities, keys and Data packets in NAC. There are a data owner (such as an NDNFit user "alice"), a data producer (such as the NDNFit data capture application which produces data for "alice"), and a consumer (such as Bob's DVU). The producer produces data for the data owner and encrypts the data content using proper encryption keys — i.e., the keys appointed by the data owner, such that the consumer can decrypt the data owner's data correctly if Alice grants it access to her data by properly distributing the corresponding decryption keys to it. Here is a more detailed explanation for how NAC works:

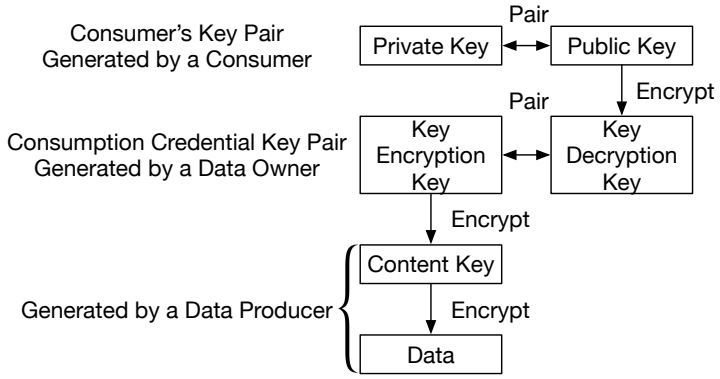


Figure 4.5: Relations among Entities, Keys and Data Packets in Name-based Access Control

Key Generation (by Data Owner): The data owner generates a list of Consumption Credential Key Pairs, each includes a Key Encryption Key (KEK) and a Key Decryption

Key (KDK). For each Consumption Credential Key Pair, the data owner produces a Data packet D_{KEK} carrying the KEK in plaintext, and produces a Data packet D_{KDK} carrying the KDK encrypted by the consumer's public key, which is contained in a Data packet D_{public} fetched from the consumer (assume that the consumer has already produced D_{public}). Notice that the D_{KEK} should be produced before the data producer produces Data for the data owner, and the D_{KDK} should be produced before the data consumer fetches Data produced by the data producer.

Data Production (by Data Producer): When producing data (for the data owner), the producer first produces a list of symmetric key called Content Keys (C-KEYs). Then it fetches the proper D_{KEK} from the network (assume that the data owner has already produced a list of D_{KEK}), uses them to encrypt the C-KEYs, and produces a Data packet D_{C-KEY} for each C-KEY to carry the encrypted C-KEY. Finally, it produces content, encrypts it using the proper C-KEY, and packets it into Data D_{data}.

Data Consumption (by Data Consumer): When the consumer consumes the owner's data, it first fetches D_{data}, and learns from D_{data} that this Data is encrypted by C-KEY; in order to decrypt D_{data}, it fetches the corresponding D_{C-KEY}, and learns that this Data is encrypted by KEK; in order to decrypt D_{C-KEY}, it fetches the corresponding D_{KDK}, and learns that this Data is encrypted by its own public key. Finally, the consumer decrypts D_{KDK} using its own private key $D_{private}$, getting KDK; decrypts D_{C-KEY} using KDK, getting C-KEY; decrypts D_{data} using C-KEY, getting the data content.

We analyze how NAC can achieve the goal of user-controlled data sharing. The above data consumption process shows that, to get the original data, consumers need to get KDKs (and some other keys). As the data owner controls the generation and secured distribution of Consumption Credential Key Pairs (i.e., KEKs and KDKs), it can control who can get access to her data by deciding whom to distribute KDKs.

4.4.2.2 Naming Conventions for Employing NAC in NDNFit

We designed specific naming conventions for employing NAC in NDNFit, which can help data owners, producers and consumers figure out names of different keys, thus automating the data (including keys) production and consumption, as well as data encryption and decryption process.

In NDNFit, a data owner has multiple sub-namespaces under her name prefix "`/org /openmhealth/<user id>`". Among them, a sub-namespace "`/org/openmhealth/<user id> /READ`" is used for publishing Consumption Credential Key Pairs (i.e., KEKs and KDKs), and a sub-namespace "`/org/openmhealth/<user id>/DATA`" is used for publishing data and C-KEYs.

When producing Consumption Credential Key Pairs, a data owner names plaintext KEK "`/org/openmhealth/<user id>/READ/<data types>/KEK/<time range>`", and names the corresponding encrypted KDK "`/org/openmhealth/<user id>/READ/<data types>/KDK/<time r ange>/ENCRYPTED-BY/<Consumer Public Key>`". The "`<data types>`" component can be "", "`fitness`", "`fitness/physical_activity`" or "`fitness/physical_activity/time_location`", and "`<time range>`" is "`<start hourpoint>/<end hourpoint>`". Those two together determine capability of a Consumption Credential Key Pair, i.e., which C-KEYs should be encrypted using the KEK of the Consumption Credential Key Pair. "`<Consumer Public Key>`" is the the public key name of the consumer the data owner wants to grant access to. "`ENCRYPTED-BY`" indicates the key "`<Consumer Public Key>`" is used to encrypt the KDK. Notice that if the data owner wants to grant access to multiple consumers, the same KDK should encryped seperately for each consumer, therefore a same KDK has multiple encrypted versions.

When producing data, a producer names encrypted Data packet "`/org/openmhealth /<user id>/READ/fitness/physical_activity/time_location/<timestamp>/ENCRYPTED-BY/< C-KEY name>`", and names encrypted C-KEY "`/org/openmhealth/<user id>/READ/fitness /physical_activity/time_location/C-KEY/<hourpoint>/ENCRYPTED-BY/<KEK name>`". Similar as before, "`ENCRYPTED-BY`" indicates the key "`<C-KEY name>`" or "`<KEK name>`" is used

to encrypt the data or the C-KEY. A data packet should be encrypted by a C-KEY only when the data packet's "`<timestamp>`" falls into "`[<hourpoint>, <hourpoint> + 1)`", where "`<hourpoint>`" is a component of the C-KEY name. A C-KEY packet should be encrypted by a KEK only when the C-KEY's "`<hourpoint>`" falls into "`[<start hourpoint>, <end hourpoint>)`", where "`<start hourpoint>/<end hourpoint>`" are components of the KEK name, and when "`<data types>`" component of the KEK name is a prefix of "`fitness /physical_activity/time_location`". Notice that if multiple KEKs satisfy the conditions, the C-KEY should be encrypted for each KEK.

Notice that in the previous naming conventions, C-KEYs are produced for per hour point. According to the defined relationship between data and C-KEYs, Data packets produced in the same hour are encrypted using the same C-KEY. Therefore, a consumer can get access to all Data packets produced in the same hour if it is granted access to a single Data packet. That means, the minimum access control granularity is decided by the granularity of C-KEYs; if C-KEYs are produced for per hour point, the minimum access control granularity is one hour. We can modify the minimum access control granularity as needed.

4.4.2.3 A NAC Example

We use an example to illustrate how NAC works in NDNFit. Assume that a user Alice defines a naive data sharing policy for her data: Bob's DVU is allowed to read all her time location data generated between May 1st, 2018, 10am (inclusive) and May 1st, 2018, 10pm (exclusive). Alice wants the minimum access control granularity to be 1 hour.

Following the data sharing policy, Alice generates a Consumption Credential Key Pair shown in Figure 4.6.

/org/openmhealth/alice/READ/fitness/physical_activity/time_location/KEK/20180501T100000/20180501T220000

/org/openmhealth/alice/READ/fitness/physical_activity/time_location/KDK/20180501T100000/20180501T220000
/ENCRYPTED-BY/<the name of Bob's DVU public key>

Figure 4.6: Alice's Consumption Credential Key Pair Names

Assume that the NDNFit data capture application wants to produce a Data packet "`/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T100523/0`" for Alice. It needs to generate a plaintext C-KEY, whose name should be "`/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T100000`". Following NAC naming conventions, the Data packet should be named by the C-KEY, the C-KEY should be encrypted by the KEK, they are named properly as shown in Figure 4.7.

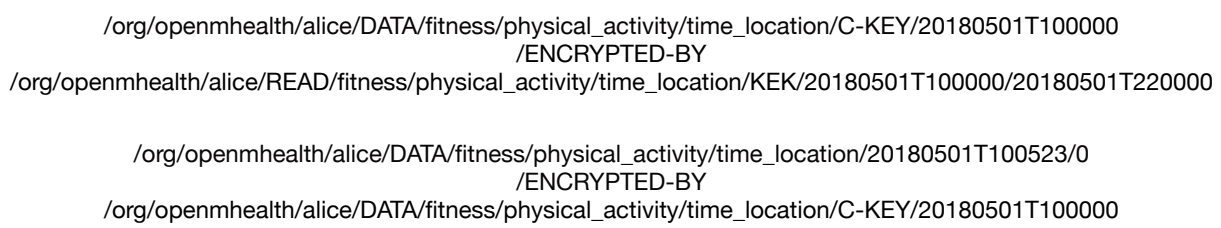


Figure 4.7: Alice's Data and C-KEY Names

When Bob's DVU consumes Alice's Data, the Interest-Data exchanges should be the ones shown in Figure 4.8.

Figure 4.8: The Interest-Data Exchange When Bob's DVU Consumes Alice's Data

4.4.2.4 Summary

To summarize, NAC provides a building block to enable data owners to control access when their data is produced, independently of how they are exchanged. Granting access to data neither requires online service negotiation. Access is granted by encrypting data's C-KEYs using KEKs of the consumption credentials whose access priviledge covers the data, and encrypting KDKs of those consumption credentials for authorized consumers. This process is handled by using naming convention to deal with the problems of consumption credential (i.e., KEK and KDK) naming and encrypting. Specifically, data owners direct how data is encrypted by describing (i) relationships between data and C-KEYs; (ii) relationships between C-KEYs and consumption credentials; (iii) for whom the consumption credentials' KDKs are encrypted.

Meanwhile, as access management is simply handled by consumption credential (i.e., KEK and KDK) naming and encrypting, data owners can delegate access management authority of part or all of their data to other entities (for example, the Access Manager application), by authorizing them to publish consumption credentials. To be specific, data owners need to issue proper certificates to those entities and define proper trust rules for keys produced by them (as seen in Section 4.3).

4.4.3 Name Confidentiality

We make the first step to achieve name confidentiality, that is, we try to hide human-readable user ID in names. The basic idea is using a secret mapping to map human-readable user IDs to a fixed-sized random IDs, so attackers cannot know whom a Data packet belongs to by merely checking its name.

Figure 4.9 shows the relations among entities, keys and Data packets in name mapping. There are the Open mHealth root, a data owner, such as an NDNFit user "alice", and a data consumer who wants to learn the user ID mapping. We use "alice" as an example to show how the ID mapping works.

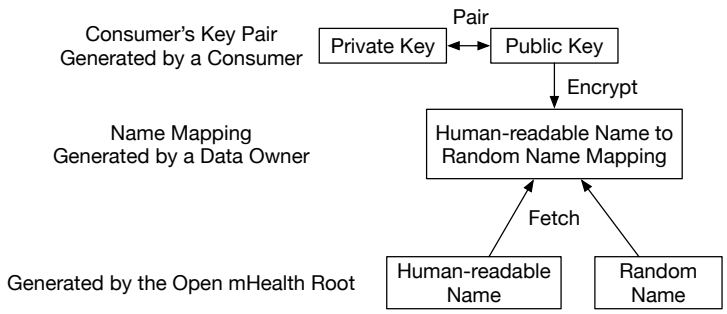


Figure 4.9: Relations among Entities, Keys and Data Packets in Naming Mapping

ID Mapping Generation: When "`alice`" requests identities from the Open mHealth root, the root allocated two identities, one is human-readable, i.e., "`/org/openmhealth/alice`", and another one contains a fixed-sized random ID, such as "`/org/openmhealth/ESZafeDQaG`", to "alice". After generating a public-private key pair for each identity, naming them, and requesting a certificate for each public key, the user "alice" creates a Data packet named "`/org/openmhealth/alice/mapped_id/ENCRYPTED-BY/<Consumer Public Key>`" and signed by the public key corresponding to "`/org/openmhealth/alice`", which contains his mapped user ID "`ESZafeDQaG`" encrypted by the consumer's public key.

Data Production: The user "alice" only uses the mapped user ID "`ESZafeDQaG`" to produce data. That is, he allocates sub-namespaces of "`/org/openmhealth/ESZafeDQaG`" to other entities, and issues certificates to them using the public key corresponding to "`/org/openmhealth/ESZafeDQaG`". As a result, all his data, either produced by himself or other applications, is under "`/org/openmhealth/ESZafeDQaG`".

ID Mapping Consumption: When a consumer wants to fetch "`alice`"'s data, it first sends an Interest "`/org/openmhealth/alice/mapped_id/ENCRYPTED-BY/<Consumer Public Key>`" to fetch the encrypted mapped ID, and decrypts it using its own private key, getting the plaintext mapped user ID "`ESZafeDQaG`". Then the consumer uses ID "`ESZafeDQaG`" to fetch "`alice`"'s data.

CHAPTER 5

Data Processing

According to users' requirements and depending on available processing functionality, their (raw) data, produced for by NDNFit data capture applications, are transformed and complemented by processing and analysis in DPUs. To achieve the goal, this chapter designs a framework where new data is produced, as processing results of existing data, on demand. Specifically, data processing and analysis are provided by named functions, a concept introduced by Named Function Networking (NFN) (Section 5.1); to interact with a named function, proper parameters need to be specified (Section 5.2); security in function call can be achieved using the mechanisms introduced in Chapter 4 (Section 5.3). named functions can be easily chained to provide complex data processing and analysis functions (Section 5.4).

5.1 Named Function Networking (NFN)

Named Function Networking (NFN) is a specific use case of NDN, that is, it treats functions as data and names them. Instead of using a single data reference, NFN operates on complex expressions (i.e., NFN names) which can reference named functions as well as (NDN or NFN) parameters. For example, given a function name `func` and a list of input data names $\texttt{data}_1$, $\texttt{data}_2$, ... $\texttt{data}_n$, an NFN name is an expression `call n func` $\texttt{data}_1$, $\texttt{data}_2$,...$\texttt{data}_n$, representing the application of `func` on $\texttt{data}_1$, $\texttt{data}_2$, ... $\texttt{data}_n$, where `n` is the total number of parameters.

An NFN-enabled node is able to analyze NFN names, pull input data and directly compute the final result if the NFN-enabled node has both the function and input data in hand; otherwise, the NFN-enabled node can forward partial sub-expressions and combine retrieved

intermediate results if a single NFN-enabled node cannot calculate the final result directly, pull executable code if the named function is published as named data and the NFN-enabled node does not have it yet, depending on its capabilities. In the spirit of named data, results are expressed in a location-agnostic way that lets the named function network orchestrate the computation.

To be compatible with the traditional NDN architecture, NFN names can be converted into NFN-enhanced NDN names and vice versa (i.e., the mapping is bidirectional). An NFN-enhanced NDN name consists of a routable prefix and a workflow definition. The routable prefix is the name of the input data or the name of the function call. Since NDN networks apply longest prefix matching to the Interest name, the Interest is routed to a node which has a copy of the Data or, in case of a function, is capable of executing it. The workflow is represented as a λ-expression. For example, the NFN name `call n func` $\texttt{data}_1$`,` $\texttt{data}_2$`, ...` $\texttt{data}_n$, can be converted into an NFN-enhanced NDN name consisting of a routable prefix and a workflow definition (represented by a λ-abstraction), `func @x (call n x` $\texttt{data}_1$`,` $\texttt{data}_2$`, ...` $\texttt{data}_n$`)`, with which the Interest will be routed to a node that is capable of executing the function `func`, or $\texttt{data}_i$ `@x (call n func` $\texttt{data}_1$`,` $\texttt{data}_2$`, ... x, ...` $\texttt{data}_n$`)` (where the `@` replaces the λ), with which the Interest will be routed to a node that has a copy of the Data $\texttt{data}_i$.

In Open mHealth, where DPUs need to provide sharable functions across multiple mHealth applications, we can implement them using NFN. Specifically, each DPU runs a named function `func`, and should be invoked using NFN-enhanced NDN names `func @x (call n x` $\texttt{data}_1$`,` $\texttt{data}_2$`, ...` $\texttt{data}_n$`)`, where the routable prefix is the name of the function.

5.2 Function Call

There are naming issues when implementing DPUs using NFN. First, to invoke a named function, a consumer needs to compose an Interest packet, whose name needs to specifically list all input Data packet names; however, this may not be feasible when the number of input Data packets is too large, as listing all input Data packet names may result in a

large Interest packet whose size exceeds the maximum NDN packet size — 8800 bytes. For example, we need a DPU which can count a user's steps based on the time-location data provided by the user. When implementing in NFN, the named function could be named "`/org/openmhealth/step_counter`" (which follows NDNFit's naming convention for naming DPU services); it takes a list of time-location Data names, such as "`/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501100000/0`", as input. According to NFN's naming convention, one Interest to invoke the function (and the corresponding processing result) can be named "`/org/openmhealth/step_counter/@x (call 120 x /org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501100000/0,... /org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501115900/0)`". There are 120 input Data packet names in the Interest name, which is definitely larger than 8800 bytes.

Second, in NFN, Data packets produced by named functions has the same names as the calling Interest packets, whereas those Data names do not follow NDNFit's naming conventions. For example, the Data packet produced by the named function "`/org/openmhealth/step_counter`" in the previous example is the same as the Interest name, whereas NDNFit requires the processing result to be named "`/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/bout/20180501100000/20180501115959/steps`".

Third, when a consumer in NDNFit wants to fetch a Data packet that should be produced by a DPU, but the Data packet has not been produced yet, the consumer does not know whether to invoke a DPU or not; if yes, which DPU should be invoked and how to invoke it. For example, in order to display a user's step count, a DVU need to fetch a Data packet "`/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/bout/20180501100000/20180501115959/steps`"; but if this Data has not been produced yet, how should it know that it should compose a different Interest to invoke the named function "`/org/openmhealth/step_counter`"?

To solve the naming issues, we have designed the following solutions. First, to avoid Interest packets (or more specific, input Data packet name list in their names) used for invoking named functions grow too large, we can compress the input Data packet name list

based on specific use cases, and have named functions accept and interpret those Interest having compressed input Data packet name list. For example, currently, all named functions used in DPUs require a list of input Data packets with the same prefix but different "`timestamp`"s, and all segments for the same "`timestamp`" are always needed at the same time, that is to say, Interests for invoking DPUs are named "`/<function name>/@x (call n /<prefix>/timestamp_1/<segment>, ... /<prefix>/timestamp_n/<segment>)`". To deal with this case, we compress input Data packet names in those Interest, generating new names "`/<function name>/@x (call n x <prefix>, start_timestamp, end_timestamp)`", with which named functions can deduce the original input Data name list. More compression algorithms can be introduced when they are needed.

Second, DPUs (named functions) are still invoked by using Interests with names "`/<function name>/@x (call n x <parameter list>)`"; however, the "`<parameter list>`" in Interests now contains a new parameter — the name for the output Data packet, which is a name that follows NDNFit naming conventions for processed results (i.e., "`/org/openmhealth/<user id>/DATA/fitness/physical_activity/time_location/bout/<suffix>`"). To be more specific, Interests for invoking DPUs are now named "`/<function name>/@x (call n x <input parameter list>, <output data name>)`". With this design, when DPUs produce Data packets, they produce encapsulated Data packets, with the outer names same as the Interest names, and the inner names as "`<output data name>`" taken from the Interest names (see Figure 5.1 for the details). In this way, consumers sending an Interest packet can get a Data packet matching the original Interest packet; meanwhile, consumers can decapsulate the Data packet to get a Data packet following NDNFit naming conventions for processed results.

Third, an Interest for retrieving a Data packet that should be produced by a DPU is always sent to a DSU, which first checks whether a matching Data packets exists or not; if no existing matching Data packet is found, the DSU sends the Interest to a name conversion service, which converts this Interests name, creating an NFN-enhanced NDN name "`/<function name>/@x (call n x <input parameter list>, <output data name>)`", then the name conversion service invokes the DPU that can produce the required Data. To this end, the

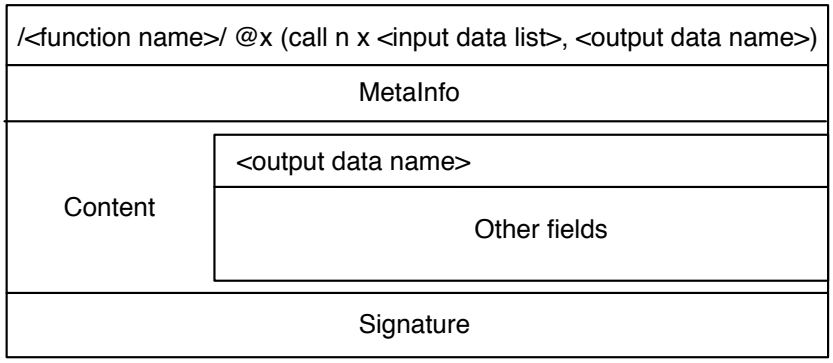


Figure 5.1: The Structure of Encapsulated Data Produced by DPU: The Inner Data Name Is A Processed Data Name, The Outer Data Is A Function Call Name

name conversion services need to be configured name conversion conventions, for converting NDNFit processed data names to NFN-enhanced NDN names.

Figure 5.2 shows an example of name conversion rules configured in the name conversion services: how to convert an NDNFit step count data (one kind of processed data) name for an Interest name used to invoke a DPU to generate this step count Data. When a name conversion service receives an Interest with name shown in Figure 5.2(a), it learns that it is for an step count Data packet. According to its configurations, it knows that the Data can be produced by the DPU (named function) "`/org/openmhealth/step_counter`"; it also knows that this named function requires 4 parameters — 3 inputs and 1 output Data name, and it knows how to generate those 4 parameters. Based on all those knowledge, the naming service composes a NFN-enhanced NDN name shown in Figure 5.2(b).

5.3 Security in Function Call

Interaction with DPUs, i.e., function call, introduces two new security problems: (i) Although NDNFit processed Data packets, i.e., the inner Data shown in Figure 5.1, are produced by DPUs, their owners are certain NDNFit users but not DPUs. In this case, consumers still need to verify authenticity of those Data packets, and check whether they belong to NDNFit users or not; meanwhile, NDNFit users, as data owners, need to control access to processed

/org/openmhealth/<user-id>/data/fitness/physical_activity/time_location/bout/<start_timestamp>/<end_timestamp>/steps

(a) NDNFit Step Count Data (One Kind of Processed Data) Name for

/org/openmhealth/step_counter/
@x (call 4 x
/org/openmhealth/<user-id>/data/fitness/physical_activity/time_location,
<start_timestamp>,
<end_timestamp>,
/org/openmhealth/<user-id>/data/fitness/physical_activity/time_location/bout/<start_timestamp>/<end_timestamp>/steps)

(b) The Interest Name Used to Invoke a DPU to Generate Step Count Data

Figure 5.2: An Example of Name Conversion between an NDNFit Processed Data Name and an NFN-enhanced NDN Name

Data. (ii) It usually takes DPUs some, if not many, resources to perform functions, as they need to analyze invoking Interests, fetch, verify and process input Data packets, and produce final results. Based on this, attackers can exhaust DPUs' resources by sending many invoking Interests. To avoid this kind of attack, DPUs need to verify authenticity of Interests sent by legit consumers. In this section, we explain how security of processed Data and invoking Interest are enforced.

5.3.1 Security of Processed Data

According to identity management rules defined in Section 4.1 and certificate management rules defined in Section 4.3, a DPU can request an identity "`/org/openmhealth/<service id>`" (As a DPU is a named function, a DPU identity is also a named function name), generate its corresponding public-private key pair, and obtain public key certificate "`/org/openmhealth/<service id>/KEY/<key id>/Root/<version>`".

A DPU can use the private key of identity "`/org/openmhealth/<service id>`" to sign the outer Data packets of produced encapsulated Data packets, as those packets are owned by the DPU. Consumers can use trust rules defined in Figure 5.3 to verify authenticity of the outer Data packets. Similar to trust rules used in verifying Data produced by NDNFit, trust rules for verifying outer Data packets produced by DPUs are expressed using trust

schema; this trust schema is also preconfigured and published as a Data packet, named "/org/openmhealth/trust_schema/DPUs/<version>" and signed by the Open mHealth root, enabling data consumers to fetch it from the network, consult it and verify any received DPU data consistently.

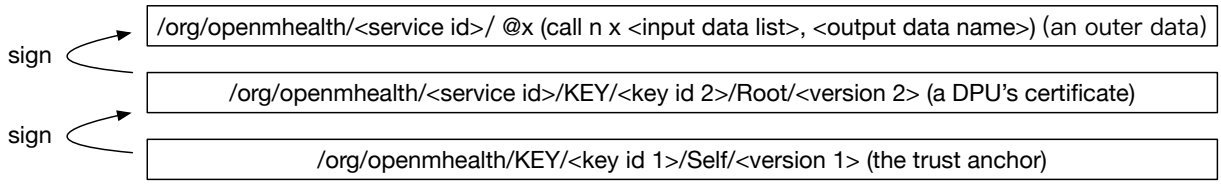


Figure 5.3: Trust Rules Used in Verifying Outer Data of Encapsulated Produced by DPUs

However, the inner Data packets of produced encapsulated Data packets, i.e., processed data, are owned by NDNFit users, the previous signing and verification rules do not work, as NDNFit users do not show up in the identity management rules, certificate management rules and trust rules.

We define new rules specifically for this case, that is, (i) an NDNFit user allocates sub-namespace "/org/openmhealth/<user id>/<service id>" to a DPU she selects to process her data; (ii) the DPU generates a public-private key pair for this identity, and requests a public key certificate "/org/openmhealth/<user id>/<service id>/KEY/<key id>/<user id>/<version>" from the NDNFit user; (iii) the DPU uses the private key of identity "/org/openmhealth/<user id>/<service id>" to sign inner Data named "/org/openmhealth/<user id>/DATA/fitness/physical_activity/time_location/bout/<suffix>"; (iv) consumers can use trust rules defined in Figure 5.3 to verify authenticity of the inner Data. Trust schema corresponding to those trust rules is preconfigured and published as a Data packet, named "/org/openmhealth/trust_schema/<user id>/processed_data/<version>" and signed by the user.

Mechanisms designed in Section 4.4 can be directly applied by DPUs to protect confidentiality of the inner Data. As long as confidentiality of the inner Data is properly enforced, we do not need to have separate mechanisms to protect confidentiality of the outer Data, as it is only used for mapping the invoking Interest and delivering inner Data back to consumers,

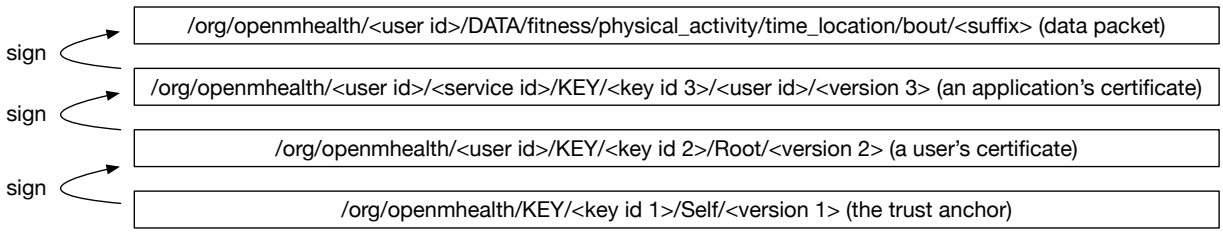


Figure 5.4: Trust Rules Used in Verifying Inner Data of Encapsulated Produced by DPUs

which does not leak sensitive information.

5.3.2 Security of Invoking Interest

NDN has a mechanism to issue an authenticated interest — Signed Interest [NDN15]. Similar to NDN Data packets, Signed Interests has a signature, which is embedded into the last component of the Interest name. To be specific, compared with normal Interests, Signed Interests have two additional components — "`<SignatureInfo>`" and "`<SignatureValue>`" — in their names, as shown in Figure 5.5. How the signature ("`<SignatureInfo>`" and "`<SignatureValue>`") works is similar to mechanisms in Section 4.2, with a difference that SHA256_digest is calculated using Equation 5.1 instead of Equation 4.1.

/<normal interest name>/<SignatureInfo>/<SignatureValue>

Figure 5.5: The Format of Signed Interest Name

$$SHA256\_digest = SHA256(Name, MetaInfo, Content, SignatureInfo) \tag{5.1}$$

By using Signed Interests, consumers can sign DPU invoking Interests. Figure 5.6 shows signed invoking Interest name format used in NDNFit. Upon receiving an invoking Interest, a DPU first checks whether it is signed or not. If not, the DPU simply discards it; otherwise, the DPU checks integrity and authenticity of the Interest. The DPU will analyze the Interest and produce data accordingly only when the Interest is signed by an authenticated identity.

As for now, all Open mHealth DPUs are designed to accept invoking Interests signed by all identities — such as NDNFit users, data capture applications, DPUs, DVUs, DSUs — of the Open mHealth architecture. To achieve this goal, we have designed trust rules, that are, as long as an invoking Interest's signature can be tracked along a certificate chain back to the Open mHealth root certificate "`/org/openmhealth/KEY/<key id>/Self/<version>`", the invoking Interest is treated as authenticated.

/org/openmhealth/<service id> @x (call n x <input data list>, <output data name>)

/org/openmhealth/<service id> @x (call n x <input data list>, <output data name>)/<SignatureInfo>/<SignatureValue>

Figure 5.6: Normal Invoking Interest Name vs Signed Invoking Interest Name

5.4 Function Chaining

According to Open mHealth, some DPU processed data is derived from other DPU processed data. For example, step count trending "`/org/openmhealth/<user id>/DATA/fitness/physical_activity/time_location/bout/<start_timestamp>/<end_timestamp>/step_trending`" is derived from "`/org/openmhealth/<user id>/DATA/fitness/physical_activity/time_location/bout/<start_timestamp>/<end_timestamp>/steps`". As the final data is produced by a chain of DPUs/functions, we call that this kind of processed data is produced by a *function chain*. Figure 5.7 demonstrates this concept.

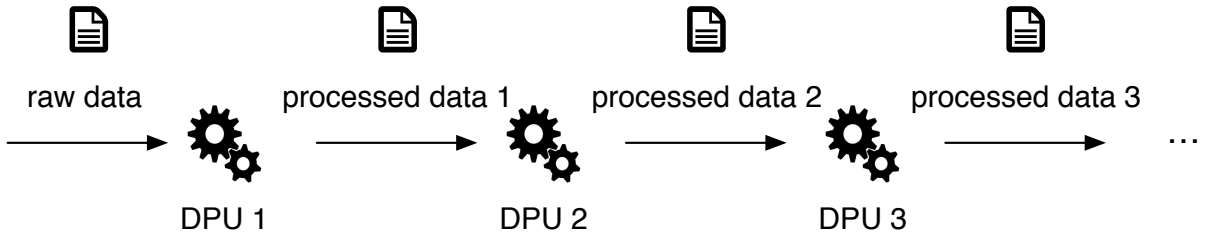


Figure 5.7: The Concept of Function Chain

Function chaining in NDNFit is easily supported as NDN is a data-centric architecture.

(i) A DPU/function does not need to explicitly invoke other DPUs/functions; as long as it sends Interests to fetch input Data packets, the Open mHealth system will help to forward those Interests to data storage or DPUs/functions that can produce those Data packets. (ii) As Interest security (i.e., Interest authenticity) and Data security (i.e., Data integrity, authenticity and confidentiality) are achieved at packet level, to enforce secured communications, DPUs/functions only need use to use proper keys to sign and encrypt packets.

CHAPTER 6

Data Storage

DSU is used for users to store their data permanently and reliably. To this end, in NDNFit, each DSU is designed to be a distributed storage system, that is, each consists of a list of storage nodes, and each Data packet is replicated in all storage nodes (Figure 6.1). To build such a distributed storage system, three problems need to be solved: how users communicate with the storage system (Section 6.1), how to synchronize data among different storage nodes of the system (Section 6.2), and how the storage system stores data (Section 6.3).

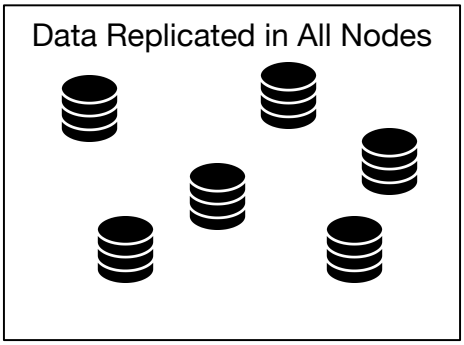


Figure 6.1: The Structure of a DSU

6.1 DSU Protocol

The DSU protocol specifies how to insert data in and delete data from the distributed DSU[1]. Three requirements determine the DSU protocol design: (1) End users publish and consume data under the user-specific namespace (in NDNFit, it is "`/org/openmhealth/<user id>`"),

[1]How data is fetched from DSU is not part of the DSU protocol, which is discussed in Section 8.3

not the DSU namespace (e.g., "`/org/openmhealth/<DSU id>`"); (2) End users need to use the DSU name to command the DSU; (3) End users do not need to understand the internal system structure of the DSU (e.g., what data is stored on which node).

We design *DSU Command*, a specially-formatted Signed Interest, to fulfil those requirements. DSU Command name is formatted as Figure 6.2, where "`<DSU prefix>`" refers to specific prefix DSU is listening, "`<command verb>`" refers to the name of command, and "`<DSUCommandParameter>`" refers to parameters of DSU command; the remaining components are components of Singed Interest for access control. The security consideration of DSU Command is similar to that of invoking Interest as described in Section 5.3.2, with a further requirement — the relationship between Data name (prefix) (See the following two paragraphs for details) contained in the "`<DSUCommandParameter>`" and the identity who signs the Interest follows trust rules defined in Figure 4.4.

/<DSU-prefix>/<command-verb>/<DSUCommandParameter>
/<timestamp>/<random-value>/<SignatureInfo>/<SignatureValue>

Figure 6.2: The Format of DSU Command Interest Name

To insert a Data packet, the "`<command verb>`" is set to "`insert`". The "`<DSUCommandParameter>`" contains the name of the Data packet to be inserted and a forwarding hint when the Data name is not globally routable, and is formatted as "`(<data name>, <forwarding hint>)`". When "`<data name>`" is globally routable, the "`<forwarding hint>`" can be set to "`null`". The process of data insertion is shown in Figure 6.3. On receiving an insertion command Interest, the DSU verified it, extracts the Data packet name and forwarding hint, composes and sends an Interest to fetch the Data; on receiving the Data packet, the DSU composes a Data packet to respond the insertion command Interest.

To delete a set of Data packets under a prefix, the "`<command verb>`" is set to "`delete`". The "`<DSUCommandParameter>`" contains the prefix of the Data packet names to be deleted, and is formatted as "`(<data name prefix>)`". The process of data deletion is shown in Figure 6.4. On receiving an insertion command Interest, the DSU verified it, extracts the

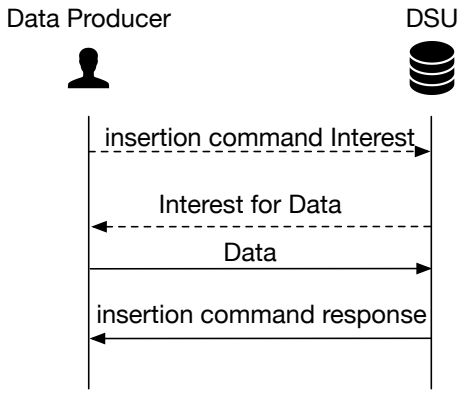


Figure 6.3: The Process of Inserting Data into DSU

Data packet name prefix, sends a Data packet to respond the deletion command Interest; then it starts to delete all Data packets under the given name prefix. Notice that the response just confirms that the deletion has been started. The DSU tries its best to delete all Data packets under the given prefix, but there is no guarantee that all such Data packets will be successfully deleted. In fact, not deleting all such Data packets, or even not deleting such Data packets at all, is not a problem, as in NDN, each Data packet is globally unique and can be cached by any entity.

How those DSU Commands are routed to the DSU is discussed in Section 8.3.

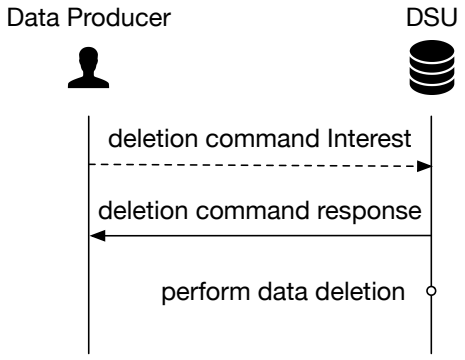


Figure 6.4: The Process of Deleting Data from DSU

6.2 Dataset Synchronization among DSU Nodes

To store data reliably, each Data packet is replicated in all DSU nodes. However, according to Section 6.1, each Data packet is only inserted into one DSU node by DSU users. Therefore, we make use of a *distributed dataset synchronization* protocol, which achieves data-centric multi-party communication, to replicate user-inserted Data packets to multiple DSU nodes. As existing sync protocols cannot fulfil our requirement of fast synchronization when many Data packets are inserted into different nodes simultaneously, we employs a new sync protocol — State Vector Synchronization Protocol [XZL18] — to DSU. Notice that the design has not been fully implemented yet, and it is left for our future work.

6.2.1 Distributed Dataset Synchronization in NDN

Several NDN sync protocols in NDN have been developed over the last few years. [SYW17] provides a detailed analysis of these efforts up to mid 2017, including CCNx 0.8 Sync [Pro12], iSync [FAC15], CCNx 1.0 Sync [Mos14], ChronoSync [ZA13], RoundSync [HCS17], and PSync (or PartialSync) [ZLW17]. It extracts out the following main lessons.[2]

- To simplify the sync protocol design, it is best for each producer to name its data using a sequence number.
- The dataset synchronization problem should be separated from the synchronization of the corresponding namespace.
- Once a sync protocol pulls in all the data , the total data ordering (across all producers) should be left to the application to decide based on its specific needs.
- Given multiple nodes may produce data at the same time, Sync Interests should not be used to retrieve new data.
- A digest of the dataset state can serve as a compact presentation of the state, but cannot directly tell which data items are missing when digests have different values.

[2]These lessons do not apply to pSync–as the name indicates, pSync supports partial sync.

Although informed guesses may resolve this issue in some cases, when none of them works, one has to resort to exchange the sequence number list of all producers [ZA13].

It also summarizes three key design aspects of existing sync protocols, which is the basis of data synchronization among DSU nodes design.

Group Communication Namespace: A shared group communication namespace is required for members to exchange messages for data synchronization. Messages sent to this namespace should be received by all other members in the same group, in the absence of packet loss. For example, when a node needs to publish its dataset update to the other nodes in the group, it could send the update message to the group communication namespace.

Dataset State Representation: Sync is conducted in group. Each member in a sync group keeps a local set of dataset shared with other members. Each member also keeps a local dataset state, which represents its local knowledge of the shared dataset namespace, i.e., what *Data* is in the shared dataset. Thus the node can determine the previously unaware data by comparing dataset states with others. The member could then fetch some or all of the previously unaware data based on its own desire.

Dataset Synchronization Mechanism: Each member in the sync group can generate new *Data* in its shared dataset at any given time, causing a difference in the dataset state among group members. A sync protocol needs to notify other members of the dataset state update and provide a mechanism for them to synchronize their dataset states to achieve an overall agreed dataset state. After dataset state synchronization, a member could then fetch some or all of the dataset update based on its own desire.

Based on the above lessons and key design aspects, VectorSync [SAZ17, SAZ18] adopts ChronoSync's naming convention of numbering each sensor's data by a monotonically increasing sequence number, but departs from ChronoSync design in two basic ways:

- Instead of a digest, its Sync Interest carries a version vector which includes a list of all producer's latest sequence numbers, ordered by the producer names.

- Sync Interests are used to notify dataset state changes only; missing data derived from the Sync Interest is fetched using separate Interest-data exchanges.

Because its version vector in each Sync Interest contains the sequence number of each producer but not the producer's name, VectorSync uses a group manager to control the group membership and the ordering of the sequence numbers in Sync Interests.

6.2.2 State Vector Synchronization (SVS) Protocol

The State Vector Synchronization (SVS) Protocol design is based on VectorSync. Instead of relying on the complex group management mechanism to maintain membership, SVS performs dataset state synchronization by explicitly enumerating member names together with their data sequence numbers, which we call *State Vector*. This section introduces SVS protocol in the following order: naming convention, state vector and dataset synchronization mechanism.

6.2.2.1 Naming Convention

Each member of a SVS synchronization group may produce data at any time. Figure 6.5(a) shows the name format of Data produced by a member running SVS. "`<member id>`" is the member ID used to identify the data producer. Each member in the sync group has an unique member ID. "`<data sequence number>`" is the data sequence ID used to identify a unique piece of data generated by this member. Each time a member generates a new piece of data, the data sequence number increments by 1. The pair of "`<member ID>`" and "`<data sequence number>`" uniquely identifies a Data packet.

(a) Data Name Format (b) Group Synchronization Interest Format

Figure 6.5: Naming Convetion in SVS

Figure 6.5(b) shows the name format for the Sync Interest. The Sync Interest carries the

State Vector (see Section 6.2.2.2) of the member, and is transmitted via multicast. "`<group id>`" is the group multicast prefix, which also identifies the group. "`[encoded state vector]`" is the encoded State Vector.

6.2.2.2 State Vector

Since the sequence number of each member's Data increments monotonically, we can represent a member's knowledge about the state of its group's shared dataset using a *State Vector* [XZL18]. A *State Vector* is a vector of [member ID : sequence number] pairs, where member ID represents a member in the group, and sequence number is the latest sequence number the *State Vector* owner knows of *Data* generated by that member. The [member ID : sequence number] pair allows the State Vector to explicitly list group members and distinguish each member in the list, thus avoiding the need to have a separate group membership management mechanism as required in VectorSync. Note that, (1) a member has the knowledge about a dataset, i.e., a *State Vector*, does not mean it has fetched all Data packets in the dataset, whether and when to fetch a specific Data packet is a separate problem. (2) if knowledge about the group's dataset is not synced up among all members in the group, different members may have different *State Vectors*. Figure 6.6 shows an example representation of the state of the group's shared dataset, which is owned by member A (assume that A is a unique id). Specifically, member A has the knowledge about Data of B, D, E and itself. The latest sequence number sensor A knows of Data published by itself is 128, and 127 for B, 129 for D, 127 for E.

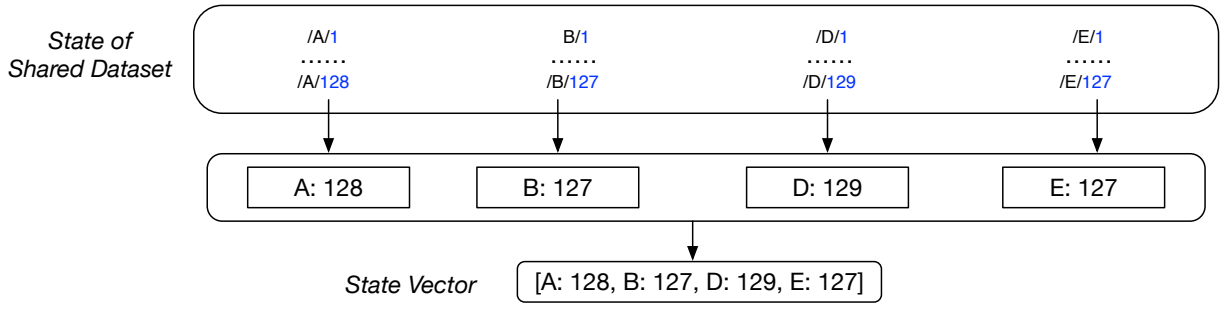


Figure 6.6: The Process of Deleting Data from DSU

6.2.2.3 Dataset Synchronization Mechanism

A member encodes its State Vector in a Sync Interest and sends it out to all other members of the group in three different cases. First, when a member generates new data, it updates its own State Vector to reflect its updated knowledge about state of the group's shared dataset. In this case, it sends out a Sync Interest to inform other members of the new Data. Second, when a member receives a Sync Interest containing stale information (we show how to decide whether a State Vector contains stale information in the next paragraph). In this case, it helps another member to catch up with its own knowledge about the state of the group's shared dataset. Third, when a member does not have new Data packets generated and does not receive Sync Interests during a pre-defined time period. In this case, the Sync Interest keeps the group alive.

When a member receives a Sync Interest, it compares the encoded State Vector ($S_{received}$) with its own State Vector (S_{local}). If $S_{received}$ contains less member IDs than S_{local}, or any sequence number for any member ID contained in $S_{received}$ is smaller than that contained in S_{local}, we say $S_{received}$ has stale information; similarly, if S_{local} contains less member IDs than $S_{received}$, or any sequence number for any member ID contained in S_{local} is smaller than that contained in $S_{received}$, we say S_{local} has stale information. If S_{local} has stale information, the member merges those two State Vectors to be S_{merged},

$$S_{merged} = [M_i : N_i] \cup [M_j : N_j] \cup [M_k : N_k]$$

$$\begin{aligned} &where\ M_i \in S_{received}\ \&\&\ M_i \notin S_{local},\ N_i\ is\ the\ sequece\ number\ corresponds\ to\ M_i\ in\ S_{received} \\ &M_j \in S_{local}\ \&\&\ M_j \notin S_{received},\ N_j\ is\ the\ sequece\ number\ corresponds\ to\ M_j\ in\ S_{local} \\ &M_k \in S_{local}\ \&\&\ M_k \in S_{received},\ N_k = max(the\ sequece\ number\ corresponds\ to\ M_k\ in\ S_{received}, \\ &\qquad the\ sequece\ number\ corresponds\ to\ M_k\ in\ S_{local}) \end{aligned} \quad (6.1)$$

and updates its S_{local} to S_{merged}; if $S_{received}$ has stale information, the member sends a Sync Interest with its own State Vector (the most up to date one) to the group.

If S_{local} has stale information, the member can figure out the Data packet set covered by

S_{merged} but not S_{local}, and decides whether to compose Interests to fetch them based on its own requirement.

6.2.2.4 Why Can SVS Support Fast Synchronization

We claim that SVS can support fast data synchronization well for two reasons. First, as every member of the group has its own sequence number in the State Vector, a member's information is not mixed with other members by hashing functions used in either computing digest or IBF. By simply comparing the local State Vector and a received State Vector, a member can directly figure out which Data packets are missing without any help from other members or without any guessing.

Second, membership information is explicitly listed in the State Vector. By simply checking the member IDs, a member knows who are in the group, and can directly start to fetch its data without any help from a group membership manager.

6.2.3 Use SVS to Achieve Data Synchronization among DSU Nodes

This section employs SVS to achieve Data synchronization among DSU nodes. There are two problems: (1) what "`<member id>`" and "`<group id>`" should be used by DSU nodes? (2) how to covert data insertion and deletion into DSU node data production? (3) when to produce a Data packet?

6.2.3.1 Naming Convention

As dataset is Synchronized inside a DSU among DSU nodes, the "`<member id>`" (of DSU nodes) is not used and visible outside the DSU. Therefore, they can be simply named "`/A`", "`/B`", "`/C`", ... Compared with longer member IDs, such simple member IDs can not only speed up packet forwarding, but also reduce the size of State Vector, thus the size of Sync Interest packet. As the maximum NDN packet limit is 8800 bytes, and NDN Interest packets cannot be fragmented, this can further increase the number of members can be fit into the State Vector (or say, the Sync Interest), increasing the upper limit of group size.

As Sync Interest should be multicast to all nodes within a DSU, the "`<group id>`" used in Sync Interest can be set as "`/ndn/broadcast`", which by default makes Interests be multicast to all next hops.

6.2.3.2 Action-based Synchronization

We use the idea similar to action-based synchronization [SA15] to covert data insertion and deletion into DSU node data production. To be specific, a DSU node publish *action Data packet* (Figure 6.7) to tell other nodes what actions they need to perform. An action packet produced by a DSU node is named "`/<member id>/<sequence number>`", which follows the SVS protocol naming conventions.

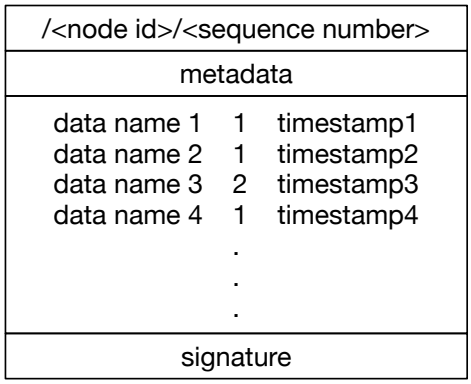


Figure 6.7: The Format of Action Packet

The content of an action packet is a list of [data name, action, timestamp] tuples, each line corresponds to an action commanded by an DSU Command received by the DSU node which produces the action packet. In a tuple, "`data name`" is the name of Data to be inserted or the name prefix under which all the Data packets to be deleted; the action is either "1", representing "insert", or "2", representing "delete"; the timestamp is extracted from the DSU Command name. For example, a DSU Command "`/<DSU-prefix>/insert/(name1, null) /20180501000000/<random-value>/<SignatureInfor>/<SignatureValue>`" can be turned into an action [name1, 1, 20180501000000].

Again, as dataset is synchronized inside a DSU, and action Data packets are not visible to

outside network, the dataset synchronization process does not face severe security problems. Therefore, action Data packets can be signed and verified in a simpler way, that is, all DSU nodes share the same symmetric key, they all use this key to sign and verify data. This simple approach brings two advantages: (1) There is no need to define complex signing and trust rules any more; (2) As signature is computed using symmetric key, compared with public key cryptography, the signing and verification can be greatly speeded up.

6.2.3.3 Synchronization Delay and Packet Number Balancing

DSU Commands, each is converted into an action, do not arrive at the same time. While we package multiple actions together into a single action Data packet, it introduces some synchronization delay: those earlier actions and the latest action packaged in the same Data packet are synchronized at the same time, which delays the synchronization of those earlier actions. Meanwhile, we do not want put a single action into an action Data packet, as this creates too many action Data packets, which (1) increases the number of Interests for action Data packets sent by other DSU nodes; (2) lowers the bandwidth usage as the portion of action Data packet Name, Metadata, and Signature fields increases (every action packet is separately named and signed).

To balance synchronization delay and action packet number, an action Data packet is produced whichever the following two conditions is met: (1) the accumulated action size exceeds a certain size limit (e.g., 5000 bytes), so that we package as many actions as possible into an action Data packet while the size of the action Data packet does not exceed NDN packet size limit (i.e., 8800 bytes). (2) the time interval between the earliest suspended action and the latest suspended action exceeds a certain time limit (e.g., 500 milliseconds), so that the synchronization delay is limited.

6.3 Low-level Data Storage

Data packets are finally stored somewhere, which we call low-level Data storage in DSU.

In NDN, every Data packet has its unique name, and should be stored without any changes. Meanwhile, in NDNFit, Data packets are only queried using names. In consideration of performance and scalability, Google BigTable [CDG08], or Key-Value store database like Reids [Car13], Amazon DynamoDB [DHJ07], Microsoft Azure Cosmos DB [Paz18] and so on fit the scenarios pretty well, and can be used as the low-level Data storage. With Key-Value store database, NDN packet names are used as keys, and the entire Data packets are values.

However, due to a lack of standardization for Key-Value store database, currently, we use a traditional SQL (Structured Query Language) database — SQLite [OA10] as the low-level Data storage. With SQLite, each Data packet is stored as a row; each row has two columns — "name" and "packet", where "name" stores a Data packet name, and "packet" stores the entire Data packet.

CHAPTER 7

Data Transport

In previous chapters, we always assume that data consumers know names of Data packets they need, so that they can simply compose and send Interests for Data packets, then the NDN network can fetch Data back. This chapter discusses how consumers figure out names of desired Data packets, thus they can fetch Data packets using NDN's basic Interest-Data exchange communication model.

There are three different types of Data packets in NDNFit: raw Data packets, processed data packets and NAC keys. Raw Data packets, which contain time-location data samples, are created and named by the NDNFit data capture applications. Processed Data packets, which contain statistics of users' data, are created and named by DPUs (see Chapter 5). C-KEYs are created by the NDNFit data capture applications, while KEKs and KDKs are created by the access control manager applications. Section 7.1, Section 7.2 and Section 7.3 discuss how data consumers learn names of raw Data packet names, processed data packet names and NAC key names, respectively.

7.1 Raw Data Transport

To determine how consumers fetch raw data, we should first understand how NDNFit data capture applications produce raw data (Section 7.1.1). Then we look into mechanisms designed for consumers to learn names of raw Data packets: name prediction (Section 7.1.2), catalog (Section 7.1.3), hierarchical catalogs (Section 7.1.4), denial of existence (Section 7.1.5).

7.1.1 Data Production in NDNFit Data Capture Application

Data production includes four steps: raw data capturing, data formatting, data packaging, data naming and signing.

7.1.1.1 Raw Data Capturing

A mobile device's operating system can capture timestamps, GPS longitude and latitude tuples at an application-defined granularity. Making use of this function, NDNFit data capture application can define data capture granularity and have the mobile OS to capture raw data.

NDNFit defines the granularity of raw data (timestamps, GPS longitude and latitude tuples) capture to be one tuple per second, for the reason that Human's running speed (average man's speed for short periods) and walking speed (preferred walking speed) are 6.7m/s[spe17] and 1.4m/s[BBH06], respectively, whose order of magnitude are both 1m/s. Choosing that granularity, the order of magnitude of distances between adjacent time-location data samples is 1 meter, which is fine enough to characterize human's running and walking features.

7.1.1.2 Data Formatting

NDNFit designs an NDNFit data format JSON schema [jso17], shown following this paragraph, which is a convention for producers to format raw time-location data to be JSON objects, and for consumers to check whether received Data contents are correct formatted as JSON objects. According to this JSON schema, a time-location data sample is a `(latitude, longitude, timestamp)` tuple, and an NDNFit raw Data packet contains a list of such tuples as its content.

```
{
  ``$schema'': ``http://json-schema.org/draft-04/schema#'',

  ``type'': ``array'',
```

```
  ''items'': {
    ''type'': ''object'',
    ''properties'': {
      ''lat'':{
        ''type'': ''number'',
        ''minimum'': -90,
        ''maximum'': 90
      },

      ''lng'':{
        ''type'': ''number'',
        ''minimum'': -180,
        ''maximum'': 180
      },

      ''timeStamp'':{
        ''type'': ''number''
      }
    },

    ''required'': [
          ''lat'',
          ''lng'',
          ''timeStamp''
    ]
  }
}
```

7.1.1.3 Data Packaging

Tradeoffs exist in how many time-location data samples to put into a packet, as it directly affects the size of NDNFit Data packet content, which further affects: 1) The number of Interest sent by consumers to retrieve data, (Given the number of samples, putting more samples into a packet reduces the number of Data packets, thus the number of Interest) 2) Network bandwidth usage rate, (Putting more samples into a packet increases the proportion of real data in a packet, thus network bandwidth usage rate) 3) The average and longest delay from when a sample is generated to the Data packet containing this sample is available for fetching, (Given the data capture granularity, putting more samples into a packet increases the average and longest delay).

To balance the requirements those three factors, NDNFit chooses to create a Data packet every minute. First, at most 60 Data packets need to be generated to include all the data samples captured within an hour. That means, a consumer only needs to send at most 60, a relative small number of, Interests to fetch all those data. Second, given that the raw data capture granularity is one tuple per second, a typical Data packet contains 60 formatted data samples, thus real data accounts for more than 80% of the packet's total size.[1] From the perspective of network bandwidth usage rate, the overhead of fetching data is "relative" low. Third, the average and longest delay from when a sample is generated to the Data packet containing this sample is made available are half and one minute. What's more, the current (soft) limit for NDN packet size is 8800 bytes, with the designed packaging mechanism, NDNFit does not need to segment Data, simplifying Data packet processing on both the producer and consumer sides.

7.1.1.4 Data Packet Naming

Data packets are named "`/org/openmhealth/<user-id>/DATA/fitness/physical_activity/time_location/<timestamp>/<segment>`" by the NDNFit data capture applications, where

[1]This is calculated based on NDN Packet Format Specification `https://named-data.net/doc/NDN-packet-spec/current/`, NDNFit data format json schema. The progress of doing this calculation is skipped in this article.

the prefix `/org/openmhealth/<user-id>` is the same as the NDNFit data capture application's identity "`/org/openmhealth/<user-id>/ndnfit`". They are signed by the NDNFit data capture application using the private key associated with the identity "`/org/openmhealth /<user-id>/ndnfit`".

However, the previous naming rules need to be adjusted when NAC (see Section 4.4.2 for details) is used. Following NAC naming conventions, encrypted raw Data packets should be named "`/<data name>/ENCRYPTED-BY/<C-KEY name>/<segment>`", where "`<data name>`" is the name shown in the previous paragraph, but without "`<segment>`".

The `timestamp` component is left unspecified in the name. As a Data packet contains a list of data samples, a natural thought is to make the `timestamp` component be the timestamp of the first sample in the data sample list. But that makes Data packet names unpredictable for consumers. In order to make Data packet names predictable, and simplify catalog creation (see section 7.1.3 for details), NDNFit packages data samples falling into the same minute interval into the same Data packet, and set `timestamp` name component to be the minute point (`timestamp` name components are represented in ISO 8601 format, so they are always of the format YYYYMMDDThhmm**00**).

7.1.2 Name Prediction

With raw data name conventions defined in Section 7.1.1, consumers can predict names of all Data packets that are possibly produced by an NDNFit data capture application. Therefore, consumers can employ NDN's Interest-Data exchange communication model to retrieve Data. After sending Interests, for those that get replied, there are Data packets produced; for those that do not get replied, there is no Data packets produced. As a result, the consumers successfully fetch all Data packets back.

We use an example to demonstrate how this works. An NDNFit user Bob wants to get Alice's raw Data packets produced between May 1st, 2018, 10am (inclusive) and May 1st, 2018, 11am (exclusive). He can send a list of Interests, with names shown in Figure 7.1. Notice that those Interest names are prefixes of Data packet names. NDN performs longest

name prefix match to decide which Data packets can satisfy an Interest, and replies it with one of them. With the replied Data packet, the consumers can learn more information about data naming, and send more Interests to retrieve more Data packets when needed. In this specific case, if Alice only has three Data packets generated for 10:10am (two segments) and 10:11am (one segment), whose names are shown in Figure 7.2, then Bob will get two Data packets first, whose segment numbers are both 0 (the first and the third names in Figure 7.2). Bob learns from segment 0 for 10:10am that there is segment 1 for 10:10am, and learn from segment 0 of 10:11am that there is no more segments for 10:11am, thus he sends another Interest, with the specific Data packet name (the second name in Figure 7.2), to fetch segment 1 for 10:11am. Finally, Bob get all Data packets produced by Alice.

```
/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T100000
/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T100100
                                    ...
/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T105900
```

Figure 7.1: Names of Interests for Retrieving Alice's Data Produced between May 1st, 2018, 10am (inclusive) and May 1st, 2018, 11am (exclusive)

```
/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T101000
                                    /ENCRYPTED-BY/<C-KEY name>/0
/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T101000
                                    /ENCRYPTED-BY/<C-KEY name>/1
/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T101100
                                    /ENCRYPTED-BY/<C-KEY name>/0
```

Figure 7.2: Alice's Data Names Produced between May 1st, 2018, 10am (inclusive) and May 1st, 2018, 11am (exclusive)

7.1.3 Data Catalog

The name prediction solution presented in 7.1.2 suffers from a problem, that is, when Data packets are sparse in the time space, the overhead of fetching Data is too high. For example, in the example used in Section 7.1.2, Bob sends 60 Interests to try to fetch Alice's raw Data packets produced between May 1st, 2018, 10am (inclusive) and May 1st, 2018, 11am (exclusive), only two get replied. We argue that sparse Data is not a rare case. For example, Alice may not use the data capture application to produce Data all the time; instead, she may only want to capture her time-locations when she is doing exercise. As a result, Alice does not have a Data packet produced for every minute (or she only produce a few Data packets every day); meanwhile, among all time, for which minutes Alice has Data packets produced is not predictable.

In order to solve the problem in an efficient way, NDNFit introduces *Catalog*, which are Data packets inspired by the ideas of manifests [Moi14] and scientific catalog [FSD15]. The name of a catalog is formatted as "`/org/openmhealth/<user-id>/DATA/fitness/physical_activity/time_location/catalog/<timestamp>`". Catalogs are also published by the NDNFit data capture application, and they are published at a predefined interval i. In mathematics word, if a catalog's name has a timestamp t, timestamps of all other catalogs' names must be $t + n * i$, where n is an integer. Catalog packet structure shown in Figure 7.3. The content of a catalog is a list of timestamps at which raw Data packets are produced, those Data packets share a common prefix "`/org/openmhealth/<user-id>/DATA/fitness/physical_activity/time_location`" with the catalog, and those timestamps fall into the time interval $[t, t + i)$, where t is the timestamp of the catalog. For example, if the predefined catalog publishing interval i is 1 hour, and catalogs are named at exact hour points, the catalog packet with "`timestamp`" name component 20180501T100000 contains "`timestamp`" between 20180501T100000 (inclusive) and 20180501T100000 (exclusive).

Similar to data packaging, tradeoffs exist in choosing the catalog interval. Based on similar analysis, NDNFit chooses to create a catalog packet every hour.

With catalogs, consumers can first send Interests to fetch them, extract a list of times-

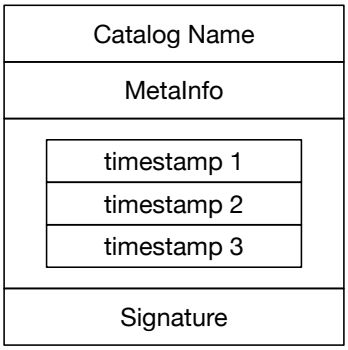


Figure 7.3: The Catalog Packet Structure

tamps, then compose and issue Interests to fetch Data packets. As for how to get all catalog names produced by an NDNFit data capture application, we can use the name prediction solution presented in Section 7.1.2. While using name prediction to fetch catalogs suffers the same high-overhead problem when catalog is sparse in time space, the number of catalogs is much smaller than that of the raw Data packets themselves. Therefore, the problem is not that severe now. To further relieve the problem, NDNFit designs hierarchical catalogs (Section 7.1.4) and deny of existence (Section 7.1.5).

To guarantee integrity and authenticity, all catalogs should be signed by the NDNFit data capture application. Consumers can verify them by using trust rules similar to those defined by Figure 4.4.

7.1.4 Hierarchical Catalogs

As explained in the previous section, catalogs work better because the number of them, which contain information of raw Data packets, is much smaller than that of the raw Data packet names contained in them.

One way to further reduce the number of packets that are needed for consumer to get all raw Data packets produced by the NDNFit data capture application is using *hierarchical catalogs*. That is, we create catalogs for catalogs, recursively. The names for different levels of catalogs are of format "`/org/openmhealth/<user-id>/DATA/fitness/physical_activity/time_location/catalog/<level name>/<timestamp>`".

Figure 7.4 shows an two-level — day-level and hour-level — hierarchical catalog example. In this example, a user generates two hour-level catalogs and a day-level catalog for 20180501. Without the day-level catalog, consumers need to enumerate all the possible hour-level catalogs. With this two-level hierarchical catalogs, as long as consumers learn the name of the day-level catalog, they can first send Interests to fetch the day-level catalogs, extract hour-level catalog names; then they issue Interests for the hour-level catalogs, extract raw Data names; last, they compose Interests to fetch raw Data packets.

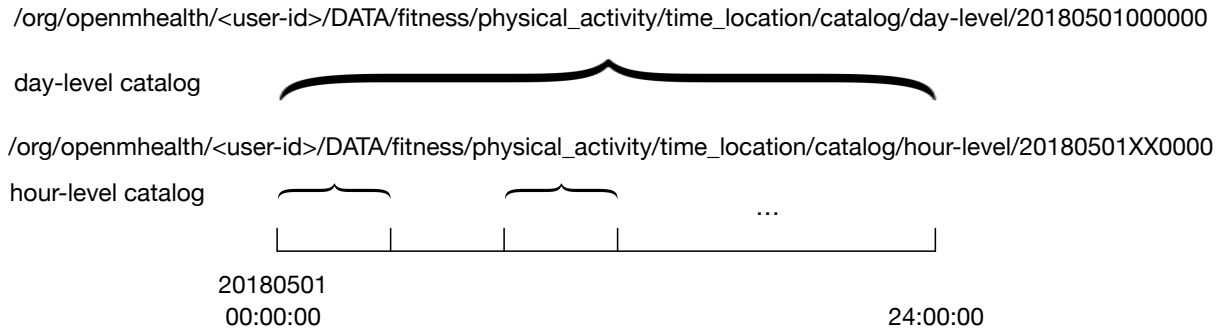


Figure 7.4: A Hierarchical Catalog Packet Example

More higher levels of catalogs can be used when it is needed. Again, all catalogs should be signed by the NDNFit data capture application, and consumers verify them by using trust rules similar to those defined by Figure 4.4.

7.1.5 Denial of Existence

Take another look of catalogs, they actually summarize raw Data *existence* information and provide them to the consumers. Similarly, we can provide a new mechanism which summarizes raw Data *non-existence* information and provide them to the consumers.

One such mechanism used in NDNFit is *Deny of Existence Packet*, which indicates that some Data packets do not exist. To be specific, a Data packet named "`/<prefix>/non-existence/<start_timestamp>/<end_timestamp>`" indicates that all Data packets between "`/<prefix>/<start_timestamp>`" (inclusive) and "`/<prefix>/<end_timestamp>`" (exclusive) do not exist. With this kind of packet, when a consumer sends an Interest for a non-

existing Data packet "/<prefix>/<timestamp for non existing packet>", where "<timestamp for non-existing packet>" falls in "[<start_timestamp>, <end_timestamp>)", a producer or a data storage can reply it with an encapsulated Data packet, whose inner packet is the deny of existence packet "/<prefix>/non-existence/<start_timestamp>/<end_timestamp>" (see Figure 7.5). After verifying the inner packet, the consumer learns that all Data packets between "/<prefix>/<start_timestamp>" (inclusive) and "/<prefix>/<end_timestamp>" (exclusive) do not exist, and it does not need to fetch them. Notice that the outer packet can be signed by any identity, or even be not signed, as it is only an "envelope" for delivering the inner Data packet.

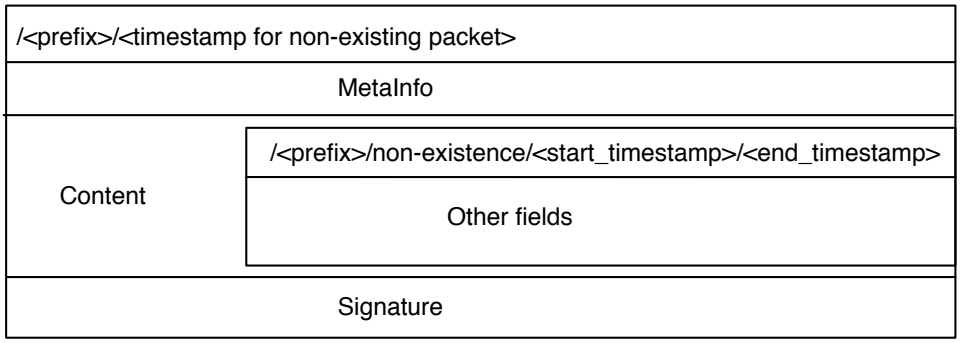


Figure 7.5: The Format of a Data Packet Replied to an Interest for a Non-Existing Data Packet

This deny of existence mechanism can be used to indicate the non-existence of both raw Data packets and catalogs. But if catalog mechanism is employed, it only needs to be employed on the highest-level catalog, lower-level catalog names and raw Data packets names can be extracted from the higher-level catalogs.

7.2 Processed Data Transport

Similar to raw data packets, processed Data packets are named following conventions "/org /openmhealth/<user-id>/DATA/fitness/physical_activity /time_location/bout/<start_timestamp>/<end_timestamp>/<data type>". The naming and signing issue have been dis-

cussed in Chapter 5. When NAC is employed, the encrypted processed Data should be named "/<process data name>/ENCRYPTED-BY/<C-KEY name>", where "<process data name>" is the same as the one used in the previous paragraph.

Processed data can be generated on demand, while raw data cannot. As a result, when consumers need some processed data, they compose and send Interests according to processed data naming conventions (without the suffix "ENCRYPTED-BY/<C-KEY name>"), expecting Data packets that are produced and cached, either in DSUs or in-network cache, before, or Data packets that are produced in real time by processing services.

7.3 NAC Key Transport

Retrieving NAC keys without retrieving the corresponding data is useless, as what consumers really need is the data. Thus, after Data packets (either raw data packets or processed data packets) are retrieved by consumers, they can further retrieve corresponding C-KEYs, KEKs and KDKs, according to NAC naming conventions described in Section 4.4.2.

CHAPTER 8

Data Reachability

The Open mHealth architecture prefix "`/org/openmhealth`" is not designed to be globally routable. That means, even a consumer knows the name of a desired NDNFit Data packet (by using the mechanisms designed in Chapter 7), if it sends Interests with the Data name or its prefix (e.g., a raw Data packet name "`/org/openmhealth/alice/DATA/DATA/fitness/physical_activity/time_location/20180501T101000`") out directly, the NDN network may not be able to get the Data packet back, as the network does not know how to forward it to a data source[1]. Therefore, NDNFit should also design mechanisms to make data reachable to consumers.

In NDNFit, there are four kinds of data sources (Content Store of NDN routers is not included): NDNFit data capture application, access management application, DPUs and DSUs. Among them, NDNFit data capture application, access management application are producers, DPUs are data processors, and DSUs are permanent data storage. Defined by Open mHealth architecture, all data — in NDNFit, it's data produced by NDNFit data capture application, access management application and DPUs — should be stored into DSUs, where all other components should get data from. Based on the previous analysis, in this chapter, we divide data reachability problem into three categories: how NDNFit data capture applications make data reachable to DSUs (Section 8.1. How access management applications make data reachable to DSUs is similar, so we do not discuss it separately), How DPUs make data reachable to DSUs (Section 8.2), and how DSUs make data reachable to other Open mHealth components (Section 8.3).

[1]the network can get the Data packet back when the corresponding Data packet is cached in routers or saved in data storage along the Interest forwarding path.

The basic idea for solving data reachability problem is using Forwarding Hint [AYW15a, AYW15b]. Forwarding Hint is an optional field of Interest packets, which contains a list of name delegations. Each delegation implies that the requested Data packet can be retrieved by forwarding the Interest using the delegation. If an Interest has a Forwarding Hint, it will be forwarded according to the Forwarding Hint first, until the Interest reaches a router whose name is a prefix of the Forwarding Hint; then the Interest will be further forwarded according to its own name. If an Interest has more than one Forwarding Hints, it will be forwarded according to the one with the highest "weight"[2].

In NDNFit, when a consumer sends an Interest for an NDNFit Data, it should also attach at least one Forwarding Hint to the Interest, which can help the NDN network figure out how to forward the Interest to a potential data source that has the corresponding Data packet. In the following sections, we focus on providing solutions for NDNFit consumers to learn the correct Forwarding Hints.

8.1 NDNFit Data Capture Applications Make Data Reachable to DSUs

NDNFit data capture applications run on mobile devices. As mobile device may move between networks, the NDNFit data capture applications may also move between networks. Therefore, in this case, there are two problems: DSUs learn forwarding hint to for reaching a NDNFit data capture application, (Section 8.1.1), and support mobile producer (i.e., the NDNFit data capture application) mobility (Section 8.1.2).

8.1.1 Data Reachability

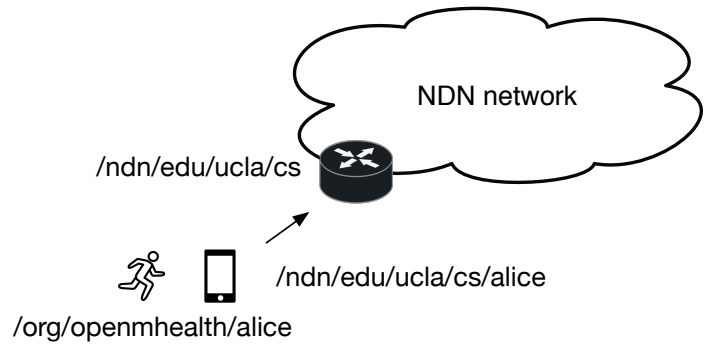
The Data reachability problem can be divided into two sub-problems, depending on whether the identity of the mobile device that runs the NDNFit data capture application is globally reachable via its attachment router or not.

[2]Readers are referred to [AYW15a, AYW15b] for more details.

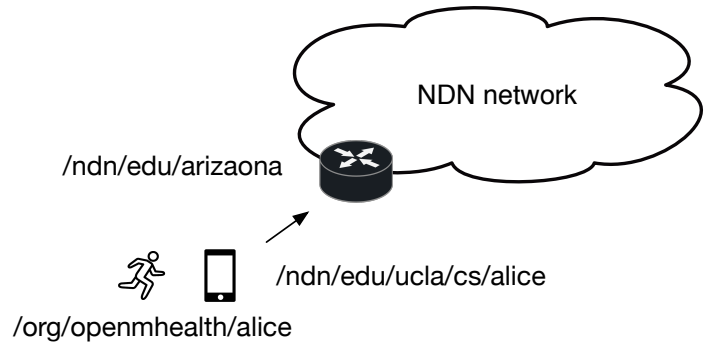
In the case shown in Figure 8.1(a), the mobile device has an identity "`/ndn/edu/ucla/cs/alice`". As (1) it is attached to its home router "`/ndn/edu/ucla/cs`" (a router is an entity's home router when the router's identity is a prefix of the entity's prefix), (2) its home router announces a global reachable prefix "`/ndn/edu/ucla/cs`" in the routing system, and (3) it registers its prefix "`/ndn/edu/ucla/cs/alice`" on the home router (This is done by using "Auto Prefix Propagation" [LAS], in which the mobile sends prefix propagation command Interest to the router. The router uses trust rules defined in Figure 8.2 to verify the Interest.), Interests with the prefix "`/ndn/edu/ucla/cs/alice`" or Interest with this prefix as a forwarding hint can be routed to the mobile device. To make use of global reachability of the mobile device's identity, the NDNFit data capture application can get the mobile device's identity, and report it to the DSU (we will discuss how to do these two soon), such that the DSU can attach this prefix as a forwarding hint to all Interests with prefix "`/org/openmhealth/alice`", making them be routed to the mobile device.

In the case shown in Figure 8.1(b), again, the mobile device has an identity "`/ndn/edu/ucla/cs/alice`". But the mobile device is not attached to its home router, instead, it is attached to a router "`/ndn/edu/arizona`", which announces a global reachable prefix "`/ndn/edu/arizona`" in the routing system. Even if the mobile device registers its prefix "`/ndn/edu/ucla/cs/alice`" on the router, as long as the router does not propagate the prefix to the routing system (which is usually the case, as announcing all prefixes registered causes routing table explosion problem), Interests with the prefix "`/ndn/edu/ucla/cs/alice`" or Interest with this prefix as a forwarding hint cannot be routed to the mobile device. However, if the mobile device can register a prefix under "`/ndn/edu/arizona`" (we will discuss how to do this soon), then Interests with the that prefix or Interest with that prefix as a forwarding hint can be routed to the mobile device. In that case, the NDNFit data capture application can get this routable prefix, and uses it similarly to the way described in the previous paragraph, making Interests be routed to the mobile device.

We then discuss how the NDNFit data capture application gets the mobile device's routable prefixes (either the mobile device's identity in case one or the routable prefix in case two) and how to report them to the DSU. Typically, in NDN, node identities and its



(a) Producer Reachability Problem When the Mobile Itself is Reachable



(b) Producer Reachability Problem When the Mobile Itself is Not Reachable

Figure 8.1: The Producer Reachability Problem

routable prefixes are held by NFD running on the node. To get those prefixes, we just need to add a new function into NFD, that is, when it receives an Interest "`/localhost/nfd/routable-prefixes`", it collects its routable identities, makes a Data packet, and sends it back to the consumer (in our case, it is the NDNFit data capture application).

To report those prefixes to the DSU, the NDNFit data capture application can send a special interest to the DSU, with the non-routable prefix as the second last component, and those prefixes encoded as the last component of the Interest name (as Interest packet does not carry payload like "content" in Data packet); to make the Interest trustworthy and not replayable, the Interest name should also contain its creation timestamp, and be signed by the NDNFit data capture application (Figure 8.3). On receiving such an Interest, the DSU

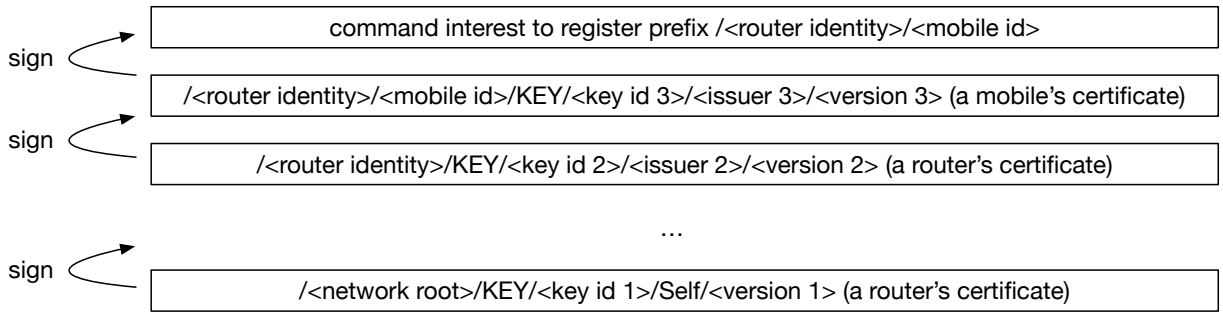


Figure 8.2: The Trust Rules for Routers to Verify Prefix Propagation Command Interest

can verify its signature by using the trust rules shown in Figure 8.4, check if this Interest is a replyed Interest by comparing the timestamp with the timestamp in the last received routable prefix reporting Interest for the same non-routable prefix, decode the third last component of the Interest to get routable prefixes, map them to the fourth last component and save the [non-routable prefix, routable prefixes, timestamp] mapping locally. When the DSU needs to fetch data produced by an NDNFit data capture application for this non-routable prefix, it looks up the mapping, pick one routable prefix and attach it as the forwarding hint to Interests for those data.

/org/openmhealth/<DSU id>/routable-prefixes-report
/<timestamp>/<non-routable prefix>/[a list of routable prefixes]/<Signature Info>/<Signature Value>

Figure 8.3: The Format of Routable Prefixes Report Interest Name

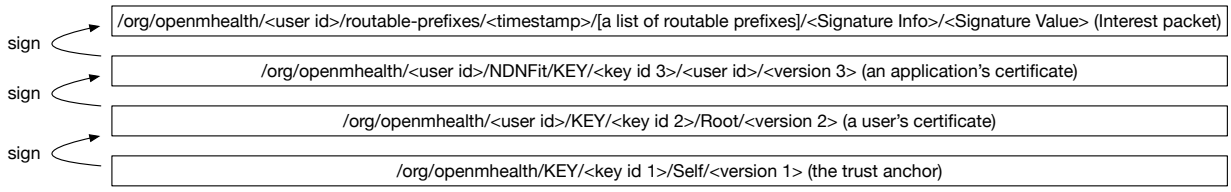


Figure 8.4: The Trust Rules Used to Verify Prefixes Report Interest

Next, we discuss how the mobile device gets the router's routable prefixes and what prefix, under the router's identity namespace, the mobile device should register on the router, and how to register it in case two. "Discover local hub prefix" protocol (`https://named-data.`

net/doc/NFD/current/misc/local-prefix-discovery.html) provides a way for the mobile device to discover global reachable prefixes of the router it attaches to. Specifically, when the mobile device sends an Interest "/localhop/nfd/rib/routable-prefixes" to the router, the router collects its routable identities, makes a Data packet, and sends it back to the mobile device.

We design the prefix to be registered as "/<router identity>/guest/<mobile device identity>", where prefix "/<mobile device identity>" makes it aggregatable, component "guest" is a special marker, and suffix "/<mobile device identity>" guarantees that the prefix is globally unique (as "/<mobile device identity>" is globally unique). To allow the mobile device to register this prefix on the router using "Auto Prefix Propagation", we also modify trust rules used by routers to verify prefix propagation command Interests shown in Figure 8.2, allowing the mobile device to sign command interest to register prefix "/<router identity>/guest/<mobile device identity>".

Figure 8.5 summarizes the mechanism to make Data reachable to the DSU, and show how the designed mechanisms are used step by step. Those five steps can be divided into three phases: step 1 and step 2 make the mobile device globally reachable; step 3 and step 4 make the NDNFit data capture application globally reachable to the DSU; in step 5, the DSU fetches data from the NDNFit data capture application with the routable prefixes provided by the application.

8.1.2 Producer Mobility Support

The producer mobility can be demonstrated in Figure 8.6, when a DSU starts sending Interests "/org/openmhealth/alice/<suffix>" at first, the mobile device running an NDNFit data capture application that produces desired data is attached to the router "/ndn/edu/ucla/cs", and the data capture application reports the routable prefix "/ndn/edu/ucla/cs/alice" to the DSU, such that the DSU can use it as the forwarding hint in Interests, then the network routes all those Interests according to forwarding hint to the NDNFit data capture application. At some time point, the mobile device moves to attach to a new router

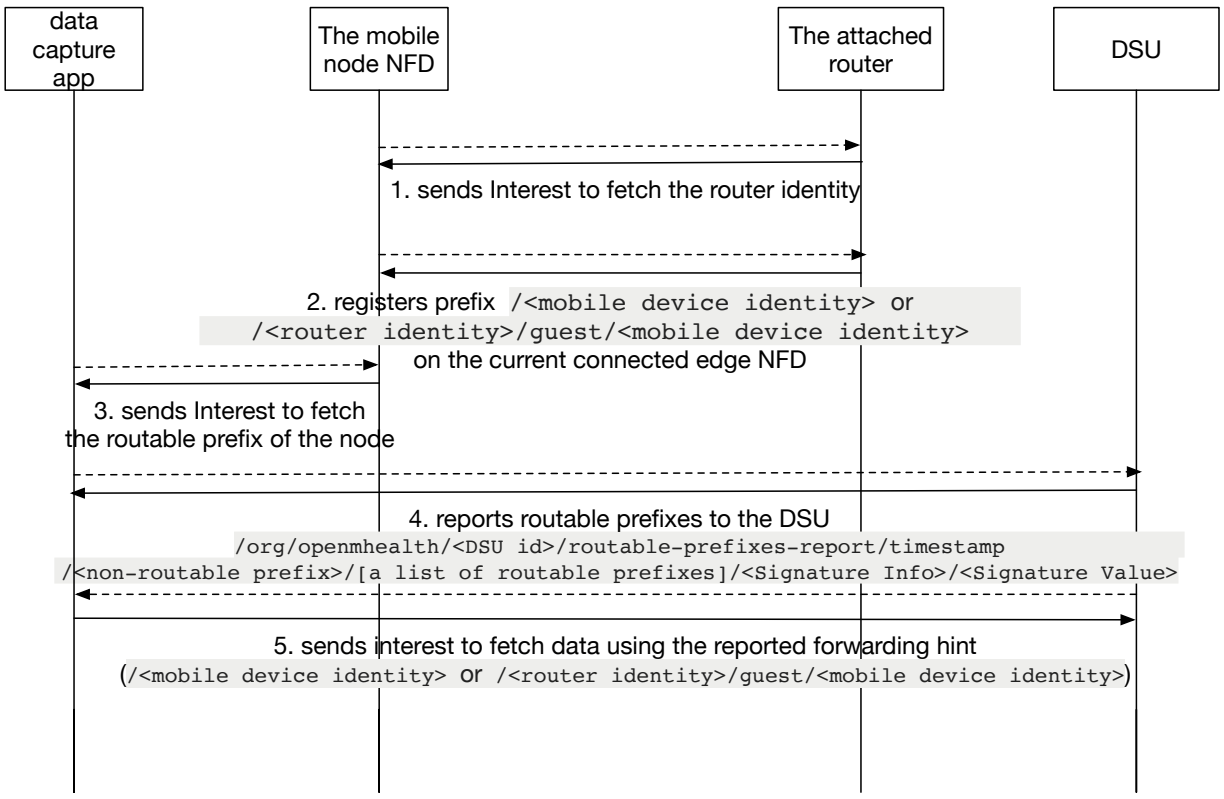


Figure 8.5: The Mechanism to Make Data Reachable to The DSU

"`/ndn/edu/arizona`", in which case the NDNFit data capture application cannot be reached using the previous Interests any more.

To solve the producer mobility problem, we noticed that "data depot+mapping" (where "data depot" means a permanent or temporary place to store data, "mapping" requires the mobile device point to report to a data depot the mapping from its "stable" data name/namespace to its "temporary" current point of attachment name/namespace)[ZAZ13, ZAB16] solution is the best fit for NDNFit, as the DSU plays exactly the role of "data depot" and forwarding hint can be used as the "mapping" mechanism, which in additional doesn't require to modify NDN network infrastructure. In the "data depot+mapping" solution, wherever the mobile device point connects to a new point of attachment, it should report a new "mapping" to the "data depot".

As discussed in Section 8.1.1, to solve the data reachability problem, the mobile device needs to get the router's routable prefixes, and the NDNFit data capture application needs

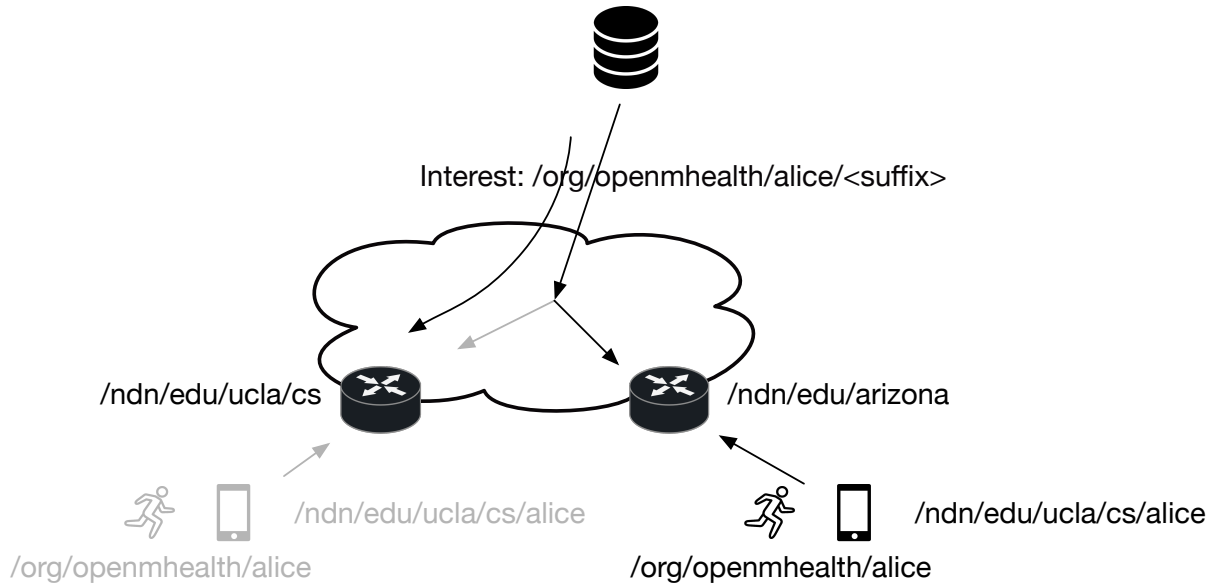


Figure 8.6: The Producer Mobility Problem

to get the mobile device's routable prefixes, and reports to the DSU. To support producer mobility, this process just needs to be repeated every time the mobile device attaches to a new router.

8.2 DPUs Make Data Reachable to DSUs

In Section 5.2, we designed that DSUs send Interests for processed data to name conversion services first, then those name conversion services translate Interest names into NFN-enhanced NDN names, compose Interests to invoke DPUs. This process is shown in Figure 8.7.

Forwarding hints are needed when DSUs send Interests, whose names have prefix "`/org/openmhealth`" which is not globally routable, to the name conversion services. As different users can choose different DPUs to process their data, each user may have her own name conversion service, based on their own choices. A user can report her name conversion service choice, with the forwarding hint used to reach it, to the DSU, using Signed Interest similar to that shown in Figure 8.3, such that DSUs know which forwarding hints to use when

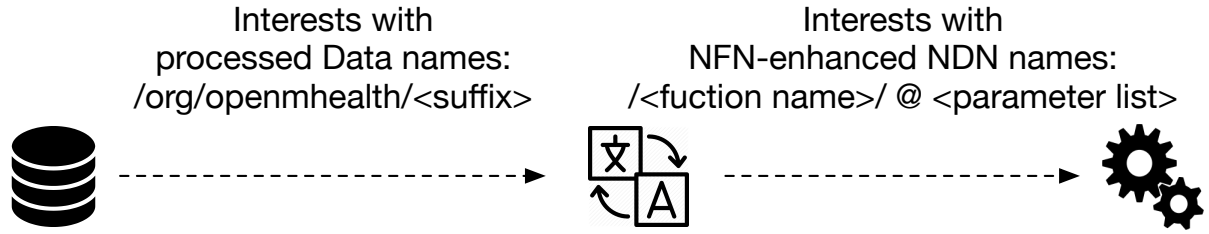


Figure 8.7: The Process of A DSU Sends Interests to A DPU

sending Interests for processed data.

Forwarding hints are also needed when a name conversion service sends Interests to a DPU. To solve this problem, we make use of the NDNS [Afa13, AJY17], a DNS-like name service for NDN, for DPUs to insert ["`/org/openmhealth/<service id>`" : forwarding hints to reach DPUs] mappings, and for name conversion services to query the forwarding hint used to invoke DPU (i.e., named function) "`/org/openmhealth/<service id>`" (Figure 8.8). A mapping is created by DPUs as a Data packet, whose name is "`/org/openmhealth/<service id>/forwarding-hint`", and whose content contains a list of names that can be used as forwarding hints for routing Interests whose name has prefix "`/org/openmhealth/<service id>`". To prevent attackers insert mappings into the NDNS system, DPUs should sign the mapping, and name conversion services should verify the mapping using trust rules similar to those shown in Figure 5.3.

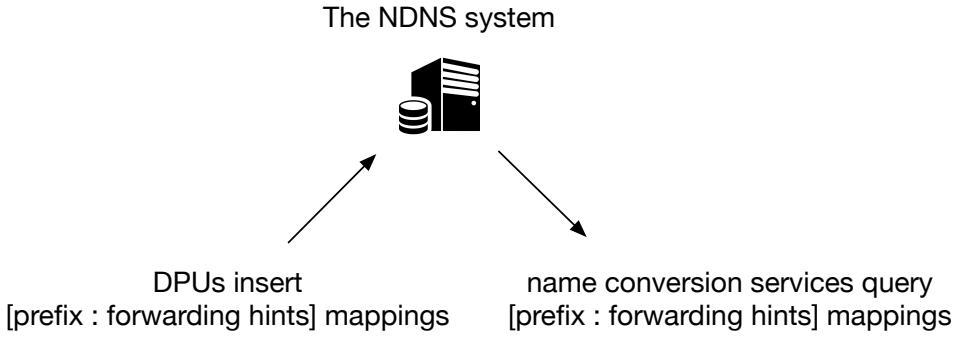


Figure 8.8: DPUs Insert Forwarding Hints into NDNS and Name Conversion Services Query Forwarding Hints from NDNS

DPU computation takes time, long or short, to finish. When the computation takes time shorter than an invoking Interest's lifetime to finish, the DPU can directly reply the Interest with processing Data to the name conversion service. When the computation takes time longer than an invoking Interest's lifetime to finish, the DPU can reply with an application level negative acknowledgement (NACK) [MWZ15], which is a special Data packet, to the name conversion service; the NACK packet contains a time interval, indicating how long it takes the computation to finish, after which the name conversion service should send the same Interest to fetch the Data again. To avoid the case that (1) different Data packets replied to the same invoking Interest have the same name, and (2) the previous NACK packet is returned again, the NACK packet should be named "`/<the invoking Interet name>/<timestamp>`", and have a short fressness period, making it possible to be filtered out when the Interest has a "MustBeFresh" field set. This process is shown in Figure 8.9.

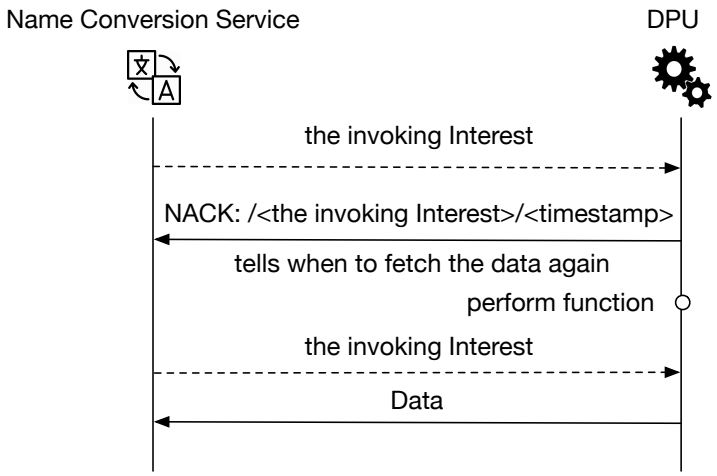


Figure 8.9: The Communication between a Name Conversion Service and a DPU when the Computation Takes Long

After sending an Interest to a DPU, depending on what response it gets from a DPU, the name conversion service replies the DSU's Interest (the Interest that is translated into the Interest sent to the DPU) differently. If the name conversion service gets a Encapsulated Data packet containing the processed result directly, it decapsulates the Data packet, getting the required processed Data packet, and sends it to the DSU. If the name conversion service

gets a NACK packet, it takes out the time interval, and replies the DSU with a NACK packet containing the same time interval. For the same consideration, the NACK packet is named "/<the processed data name>/<timestamp>", and has a short fressness period.

8.3 DSUs Make Data Reachable to Other Open mHealth Components

According to the design, when components, except the entity who produces the desired data and DSUs, need some NDNFit Data (who is named "/org/openmhealth/<suffix>"), or when data produers need to command DSUs to perform data insertion or deletion, their Interests should always go to DSUs. Again, as users can choose her own DSUs, there should be a way for consumers (i.e., those components) to learn which forwarding hint should be used, such that Interests for NDNFit Data can be routed to the user-chosen DSUs.

Similar to Section 8.2, to solve this problem, again, we make use of the NDNS, for users to insert ["/org/openmhealth/<user id>" : forwarding hints to reach DSUs] mappings, and for consumers to query the forwarding hint used to fetch Data under prefix "/org/openmhealth /<user id>" (Figure 8.8). The Data packet containing a mapping is created by users, and is named "/org/openmhealth/<user id>/forwarding-hint-DSU". To prevent attackers insert mappings into the NDNS system, users should sign the mapping, and consumers verify the mapping using trust rules similar to those shown in Figure 4.4.

CHAPTER 9

Implementation, Demonstration and Evaluation

This chapter first describes the implementation (Section 9.1) and demonstration (Section 9.2) of NDNFit on NDN research testbed, then we briefly evaluate the design and implementation (Section 9.3) of NDNFit.

9.1 Implementation

The NDNFit design has been implemented in a proof-of-concept version of NDNFit. In this section, we overview the system modules first, followed with detailed explanations.

By summarizing designs from the previous chapters, the NDNFit application should contain 11 different modules, grouped into 5, shown in Figure 9.1.

There is a *name allocator*, which allocates sub-namespaces of "`/org/openmhealth`" to users and sharable modules (i.e., DPUs and DVUs), and a *certificate issuer*, which issues certificates to those entities. As name allocation and certificate issuance are always closely bundled as the first two steps of bootstrapping an NDNFit entity, so they are always deployed together, called *Open mHealth Trust Anchor*. The trust anchor is implemented using Python, and runs on the Linux platform.

An NDNFit user needs to run four different applications on her mobile device. An *NDN-Android* application (Figure 9.2(a)) is used to turn an Android mobile device to an NDN node, and provide NDN network layer. a *data capture* application (Figure 9.2(b)) is used to produce raw data; an *identity manager* application (Figure 9.2(c)) is used to request user names and certificates from the Open mHealth trust anchor, and allocate names and certificates to applications (a human user does not exist in cyber space, so she uses

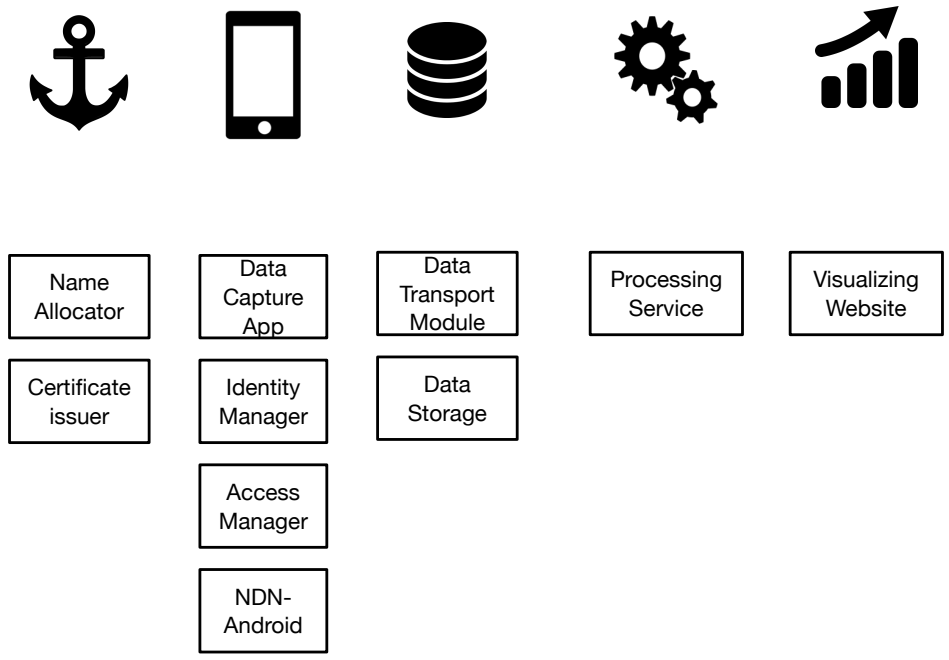


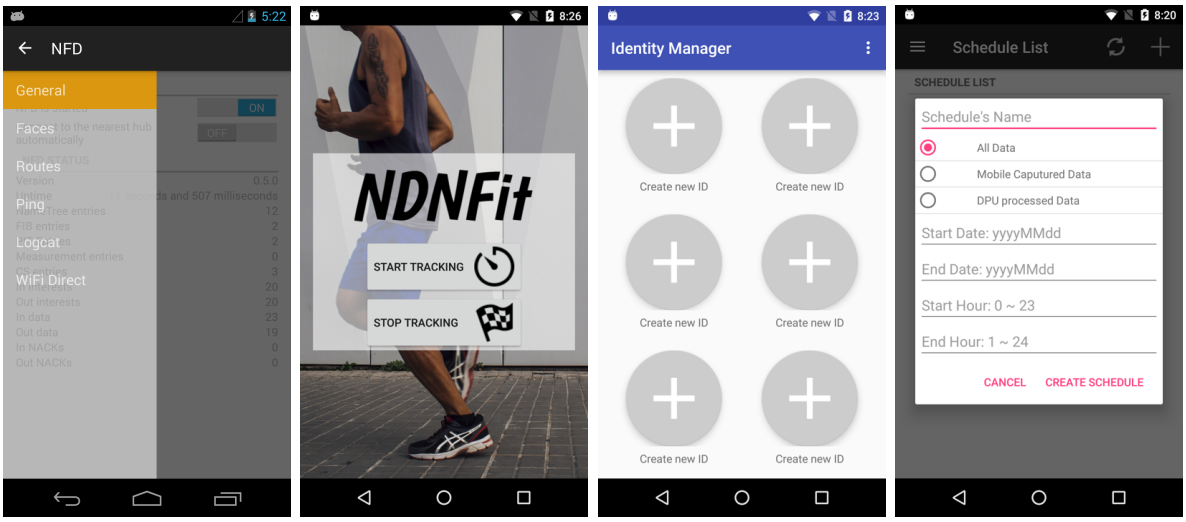
Figure 9.1: The NDNFit Modules

the application to represent her); an *access manager* application (Figure 9.2(d)) is used to interpret users' data sharing policies, and produce NAC keys (again, a human user does not exist in cyber space, so she uses the application to perform access control); Those applications are implemented using Java, and runs on the Android platform.

DSUs consist of a list of storage nodes, each has a *Data Storage* that stores and serves Data packets, and a *data transport module* to retrieve Data packets from other components and insert them into the *Data Storage*. DSUs are implemented using C++, and runs on the Linux platform.

DPUs are used to process users' data. The current implemented DPU can count steps, and calculate users' location bounding edges during a given time period. They are implemented using Python, and runs on the Linux platform.

DVUs (Figure 9.3) are used by NDNFit users to visualize their data. The current implemented DVU can display users' location bounding edges, and walking/jogging/running paths. They are implemented using JavaScript, and runs in web browsers. To make DVU



(a) NDN-Android Application (b) Data Capture Application (c) Identity Manager Application (d) Access Manager Application

Figure 9.2: Four Application Running on Mobile Devices

code available to users, we also build a website to serve those code. When users use web browsers to visit the website, the DVU code is downloaded automatically.

Those components collaborate to form a system. A user launches NDN-Android to configure her mobile device to an NDN node. After installing the identity manager application, a user employs it to request a namespace from the trust anchor, generate a self-sign certificate for the identity, and ask the trust anchor to issue a certificate. Then the user uses the access manager application to configure policies for consumers to get access to her data. The data capture application, upon its first launch, requests an identity and a corresponding certificate from a chosen user; the identity manager receives its request, and notifies the user to make a decision. Once the initial setup is finished, the user can use the data capture application to produce properly encrypted and signed data. When the mobile device has network connection, the user-chosen DSUs will fetch data from the data capture application and store the data. Lastly, system sharable DPUs and other users' DVUs (with access granted) can fetch data to process or display as needed.

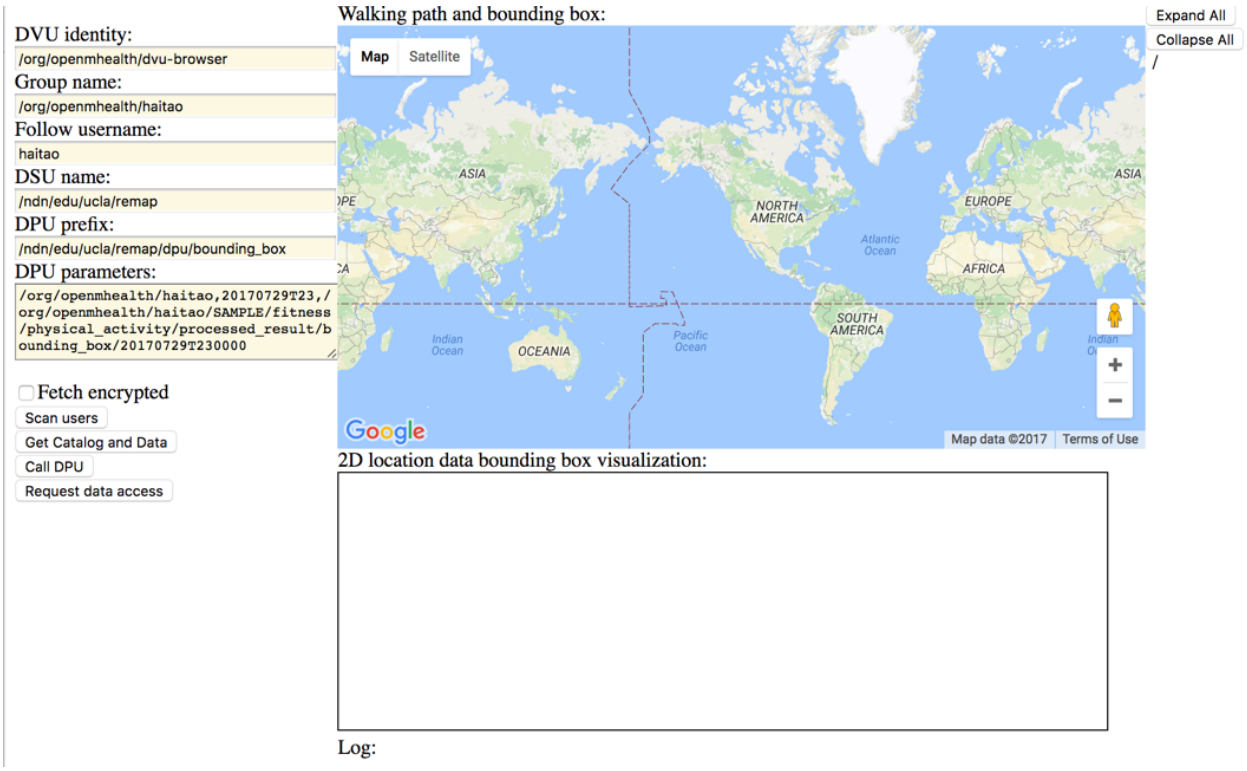


Figure 9.3: The Data Visualization Unit

9.2 Demonstration on NDN Testbed

To experiment with NDNFit, we not only demonstrated the implementation on local network, but also demonstrated it on the NDN research testbed (`https://named-data.net/ndn-testbed`), making the service available to the public. This section introduces the demonstration of NDNFit on the NDN research testbed.

9.2.1 The NDN Research Testbed

The NDN research testbed, which consists of NDN software routers installed at participating institutions and links among those routers, is a public available NDN network deployed on the current global IP network. As of time of writing this dissertation, there are 43 routers and 119 links, and links are created as TCP or UDP tunnels. Figure 9.4 shows the status of the current NDN research testbed (it also shows demonstration of NDNFit, which will be introduced in Section 9.2.2).

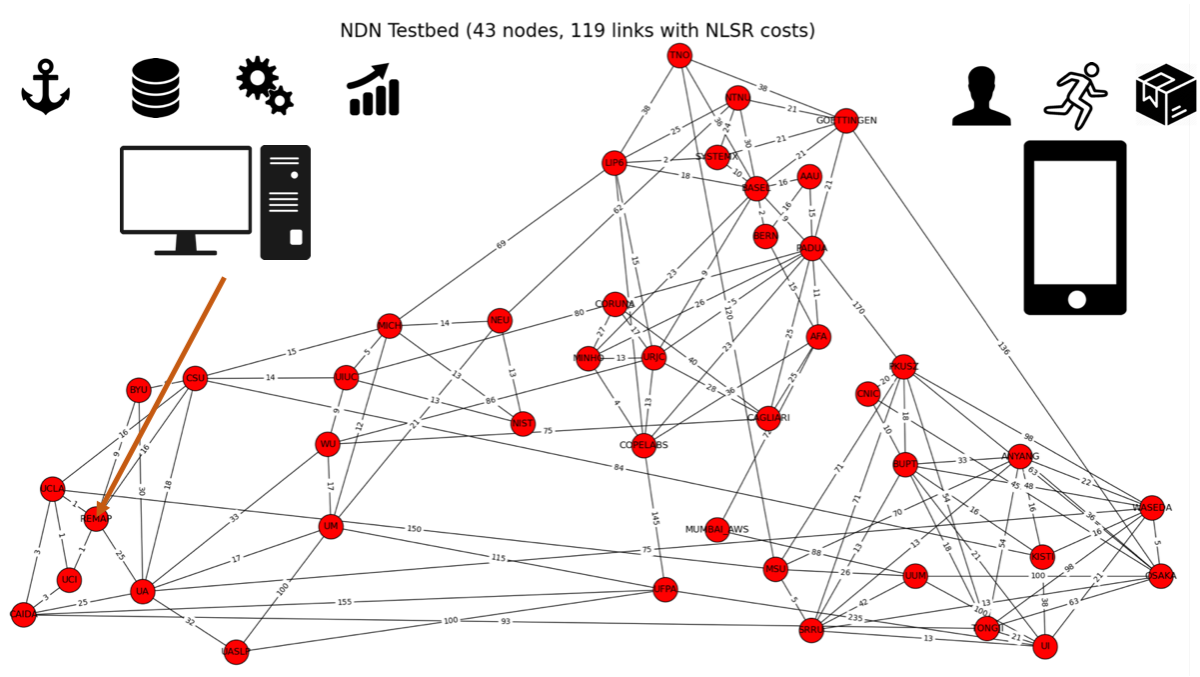


Figure 9.4: The Status of the Current NDN Research Testbed & the Demonstration of NDNFit

9.2.2 NDNFit Demonstration

Figure 9.4 also shows the demonstration of NDNFit on the NDN research testbed. A machine running the Open mHealth trust anchor, the DSU, the DPU and the DVU website connects to UCLA REMAP node. Interests with prefix "`/org/openmhealth`" are by default forwarded to UCLA REMAP node, and further to the machine.

As long as a user has an Android device, she can download the four NDNFit related applications, connect to any NDN testbed node, and start to use NDNFit just as a traditional IP application.

9.3 Evaluation: Compared with IP Based Open mHealth Applications

We perform a comparative evaluation on NDNFit. We will not compare NDNFit with IP-based Open mHealth applications in terms of performance. That's because not only NDNFit,

but also NDN network, is still at its early development stage, they have not been optimized for system performance. Thus NDNFit should not be expected to outperform IP-based Open mHealth applications. Instead, we will compare NDNFit with IP-based Open mHealth applications in terms of extra dependencies (Section 9.3.1), security overhead (Section 9.3.2), and whether the goal of user-controlled data sharing is really achieved (Section 9.3.3).

The existing IP-based Open mHealth reference applications, e.g., Ohmage[THL15], use the traditional cloud-based approach for network communication and data storage, and OAuth [Ham10, Har12] authentication for user-controlled data sharing. Data exchange and processing within those Open mHealth applications are standardized through a combination of APIs, data names, and data formats.

9.3.1 Extra Dependencies

With the traditional cloud-based approach, in the IP-based Open mHealth applications, (i) data capture applications running on mobile devices always communicate with the cloud hosts, they cannot communicate with each other directly; (ii) to communicate with the cloud, data capture applications should figure out IP addresses of cloud hosts, either by hard-coding or DNS query; (iii) as data is not secured directly, all data containers (e.g., the cloud hosts) and communication channels (e.g., between data capture applications and the cloud hosts) should be secured. Those functions introduce dependencies on cloud servers, DNS systems and protocols for securing hosts and communication channels, making the IP-based Open mHealth applications rather complex.

In contrast, except for basic NDN network functions, NDNFit functions do not rely on extra services. That is, Data packets are named and secured by applications, so that data containers (e.g., the cloud hosts) and communication channels do not need to be secured; Interests for Data packets are forwarded based on names or forwarding hints, and can be satisfied by any data resources having the desired Data packets.

9.3.2 Security Overhead

In the IP-based Open mHealth applications, whenever data needs to be transferred from one container to another, those two communication sides need to first set up a secured channel and authenticate each other, then encrypt data and deliver it over the secured channel. While in NDNFit, data only needs to be secured (signed and encrypted) once at its generation time, and consumers can be authenticated by the data owner at any time, no more security actions are needed when it is transferred in the network.

This analysis suggests that security overhead in the IP-based Open mHealth applications is proportional to the number of data transfer. In contrast, security overhead in NDNFit is constant, which can perform better than the IP-based Open mHealth applications if data needs to be transferred many times.

9.3.3 User-Controlled Data Sharing

In the IP-based Open mHealth applications, as data is not secured directly, when a user uploads her data to a storage server, the server gets the original data, and can share the data with anyone at its will. The user has to rely on the server to provide access control of her data — in other words, the user has to delegate access control of her data to the server. In this case, even if the server follows user-defined rules to control access of her data, this is not really user-controlled data sharing.

In contrast, NDNFit secures (signs and encrypts) data directly at its generation time. Without the right decryption keys, no entity can decrypt data, even the storage server. The decryption keys are distributed only by the data owner herself. In a word, user-controlled data sharing is achieved via securing data directly and users control decryption key distribution.

CHAPTER 10

Discussion

As a pilot NDN-based application, the design and implementation of NDNFit help us better understand NDN architecture, and motivate us to develop new mechanisms to solve NDN-based application problems. This chapter summarizes lessons learnt from NDNFit, including the advantages of NDN (Section 10.1), mechanisms that can be generalized (Section 10.2), and challenges in developing NDN-based applications (Section 10.3).

10.1 Advantages of NDN

With Ohmage's approach and the NDNFit's experience in mind, the advantages of NDN are briefly described below.

Names have significant power in NDN. NDNFit uses data names to organize identity and trust management, access control, data production, distributed data processing, data transport, and producer mobility support. NDN allows application-defined data names to be used for forwarding at the network layer, unifying application and network behavior in a way that can't be achieved in IP. Our design practice in NDNFit suggests that namespace design is of primary importance, and can be used to drive development of secure communications.

Named data simplifies protocols and security mechanisms in a data-centric system. To produce secured data in NDNFit, an application must obtain the appropriate private signing keys and encryption keys; to consume data in NDNFit, an application must obtain the appropriate trust schema, public signing keys and decryption keys, and compose Interests to fetch data — both can be done through standard data naming, signing, verification, encryption, decryption and Interest-Data exchange. While, for now, each step in

this process is a new territory, there are already significant advantages over standard IP web services approaches, where the mechanisms of data-centric authenticity and user-controlled data sharing do not really exist. Further, in NDN, various communication functions are unified through expressive naming schemes visible to the network layer. As such, data, trust schema, and access control schemes can all be easily fetched using the same primitives supported by any NDN network.

Data-centric security empowers data owners to have fully control over their data. Data owners in NDNFit sign and encrypt their data directly, making security features bound together with data content. Therefore, Data packets are always secured, no matter how they are transferred or where they are stored. NDNFit further employs NAC to empower data owners to securely control distribution of data encryption keys, thus data owners can have fully control over their data without dependency on any third parties.

Robustness to diverse communication situations is more inherent. NDN provides intrinsic data caching, multicast, multi-path forwarding, and disruption recovery mechanisms. Open mHealth applications built on NDN should not need to address those at the application layer, which gives them the potential of being more efficient and robust than those built on IP.

10.2 General Mechanisms Developed with NDNFit

The experience of design and implementation of NDNFit demonstrates that some mechanisms developed with NDNFit are mature enough to be generalized and to be used in other NDN-based applications. Those include Name-based Access Control, catalog for helping consumers learn data availability information and exact data names, and in-application producer mobility support mechanism. They can be generalized because no changes to the basic NDN protocols are introduced; instead, those mechanisms mainly require well-defined naming convention. Specifically,

1. Name-based Access Control requires clearly-defined consumer authentication policies,

and relationships among the names data, C-KEYs and KEKs/KDKs.

2. Catalog introduces a level of indirection into NDN's Interest-Data exchange communication model, producers and consumers should agree on catalog naming conventions.

3. Producer application mobility support mechanism just needs mobile producers knows the naming conventions of reporting Forwarding Hints to consumers.

10.3 Challenges in Developing NDN-Based Applications

While NDNFit benefits a lot from NDN, challenges remain in developing NDN-based applications. Many map to larger NDN research challenges.

How to choose the right naming conventions is still an art. Name plays a core role in NDN-based applications. There is still a lack of principles about how to best balance the conflicts between application and network's preferences on data naming, given it is now shared between the two.

It remains difficult to make new approaches easily grasped and debugged by developers, and communicated to users. NAC is an example of this. It provides significant power — enable user-controlled data sharing — to NDNFit, but requires work in making application development straightforward. Moreover, NDNFit is the first application to use NAC, we also need to design complementary mechanisms to make NAC user-friendly and help debug the library.

CHAPTER 11

Future Work

Although the initial design, implementation and demonstration of NDNFit have shown the feasibility and advantages of building Open mHealth applications on top of NDN, NDNFit is still at its early stage. To fully explore NDN's advantages of supporting user-controlled, data-centric and inter-operable applications, there are more tasks to be done, including but not limited to the following four.

11.1 Name Confidentiality

NDN names have rich application semantics, which leaks users' information as well as opens a door for attackers to analyze users' behavior. For example, from the name "`/org/openmhealth/alice/DATA/fitness/physical_activity/time_location/20180501T100000/0`", we know this Data packet belongs to a user Alice, this Data packet contains time location data, and it is generated at May 1st, 2018, 10:00am. By collecting many such Data packets produced by Alice, some attackers may be able to conclude that Alice goes out every day around 10:00am, and it's safe to sneak into her home during that time period.

To protect user information leakage, we need to design mechanisms to properly obscure data names (Section 4.4.3 shows our first try to obscure user identity). However, names play the central role in NDN, some important functions depend on proper naming to work correctly: NDN network layer forwarding Interests according to their names, NAC defines naming conventions for deriving key names from data names, consumers make use of naming conventions to predict data names or catalog names. Therefore, when obscuring names, we need to make sure those functions are not affected.

11.2 NDN Auto-Configuration

NDNFit is a distributed application, which has many components, such as data capture applications, DSUs, DPUs, DVUs and trust anchor, involving many NDN tools and mechanisms, such as NDN-Android, trust schema, NAC, forwarding hint, auto prefix propagation, remote prefix registration and so on. Immaturity of those tools and mechanisms blocked the design, implementation and demonstration of NDNFit many times and brought pains to the developers.

In the future, we plan to properly categorize those tools and mechanisms, integrate related ones in a systematic way, realizing the goal of auto-configuring NDN network as well as making NDN node/application installation and launch in a "plug-and-play" fashion. Ultimately, we hope NDNFit can provide users with similar experience with that of the current TCP/IP based Open mHealth applications, that is, users only need to care about functionalities of the applications, but not details of network related configurations.

11.3 Scalable Data Storage

The distributed data storage is still a weak point of NDNFit. The current design and implementation has only solved data replication problem. Many important features are left as future work. The following two are among them:

1. Replicate data in k-out-of-n storage nodes. Not like the current design and implementation, to guarantee reliability, data is not necessarily replicated in all storage nodes. K-out-of-n replication is usually enough, where n is a large number (e.g., 100), and k is a small number (e.g., 3). One mechanism to achieve k-out-of-n replication for NDN data packets can be Consistent Hashing [KLL97].

2. Support storage node addition and deletion. Recall that each DSU consists of a list of storage nodes, which is a distributed data storage system. To extend data storage volume, new nodes are added into the DSU; broken nodes are removed from DSU. When node addition or deletion happens, the running system should not be affected.

We are now exploring how to do this in a reliable way.

11.4 Large Scale Experiments

So far, NDNFit has only been demonstrated and tested within a small scale — on the NDN testbed and with a few users, and the systems works well. Only with large scale — larger network deployment, more users, heavier traffic — experiments, can the potential scalability and reliability problems be revealed. Meanwhile, NDNFit is the first NDN-based Open mHealth application; ideally, we should develop and test NDN-based Open mHealth with multiple similar applications — such as sleeping monitoring and heartbeat tracking applications, so that we can further prove NDN's advantages of supporting user-controlled, data-centric and inter-operable applications.

CHAPTER 12

Conclusion

Motivated by Open mHealth's vision of an interoperable application architecture of fitness data processing, and Named Data Networking's vision of a data-centric network architecture, NDNFit is designed as a demonstration application that follows Open mHealth architecture, while illustrates potential benefits of the NDN architecture in supporting mobile health applications.

This dissertation summarizes five problems and their solutions in designing and implementing NDNFit: (i) how to manage identities, and guarantee data integrity, authenticity and confidentiality. (ii) how to build sharable data processing and visualizing services. (iii) how to build a reliable distributed data storage service. (iv) how consumer components retrieve data efficiently from data provider components. (v) how to solve data reachability and producer mobility problems.

The design and implementation NDNFit validate NDN's application-driven approach to architecture design. Specifically, it motivated and verified the design of Name-based Access Control mechanism, named function invocation mechanism, catalog for consumers to learn data existence information and exact data names, and Forwarding Hint refinement to support producer application reachability issue in mobile scenario.

NDNFit achieved the goal of user-controlled data sharing that the original Open mHealth failed, by letting users name and secure their data directly. This also removes dependencies and constraints emerging from relying on underlying transport layers other third parties for security. For example, when a user needs to retrieve data from another user securely, in the IP-based implementation, the data owner needs to set up a secure communication channel (for example, using TLS) with a cloud storage server, then uploads its data via the channel;

the consumer needs to employ the same process to download data from the cloud storage server. This solution depends on a third part storage server, and the storage server, but not the data owner, controls the data sharing process with the consumer. While in the NDN-based implementation, once the Data packets are encrypted by the data owner, those Data packets stay secured, no matter where they are stored; after the data consumer is granted access, she can fetch and decrypt the owner's data (the names of data and keys are constructed automatically according to naming conventions), without dependency on any third parties.

The NDNFit design also suggests the power of networking via application-named data, the usefulness of naming conventions to solve a number of difficult problems and simplify application design. One example is, when a client needs to communicate with a processing or visualizing service, in the IP-based implementation, it needs to use Domain Name System (DNS) to resolve the domain name of the service to an IP address first, then it uses this IP address to communicate with the machine hosting the service; while in the NDN-based implementation, as long as it learns the name of the service, it can use the name to communicate with service directly. Compared with the IP-based Open mHealth implementation, the key difference of the NDN-based Open mHealth implementation is, instead of solving each design problem separately, it tries to solve all problems by designing naming conventions or disseminating names properly.

Albeit benefiting much from NDN's architecture design, NDNFit also shows that, dealing with routing configuration and packet forwarding is a pain point for application developers. This is because NDN network directly uses application-defined names in network layer packet forwarding, which implicitly involves application developers in routing configuration and packet forwarding problems. While application developers design the application namespace, they need to be careful about whether names are globally routable or not, if yes, where the names will be routed to, if no, how to make the Data packets reachable. We believe that this problem is caused by the fact that NDN is still in its early stage. As more and more network auto-configuration tools are designed and implemented, application developers can get rid of those forwarding-related dirty work.

In addition, as a pilot NDN-based application, NDNFit also touches a more general problem — how to build a user-facing application using NDN. We hope that future NDN-based application developers can learn and benefit from the design and implementation of NDNFit; meanwhile, we hope the mechanisms, protocols and tools developed for NDNFit can be reused by future NDN-based applications.

REFERENCES

[Afa13] Alexander Afanasyev. *Addressing operational challenges in Named Data Networking through NDNS distributed database*. University of California, Los Angeles, 2013.

[AJY17] Alexander Afanasyev, Xiaoke Jiang, Yingdi Yu, Jiewen Tan, Yumin Xia, Allison Mankin, and Lixia Zhang. "NDNS: A DNS-Like Name Service for NDN." In *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*, pp. 1–9. IEEE, 2017.

[ASZ16] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, et al. "NFD Developer's Guide." Technical Report NDN-0021, Revision 7, NDN, October 2016.

[AYW15a] Alexander Afanasyev, Cheng Yi, Lan Wang, Beichuan Zhang, and Lixia Zhang. "Map-and-encap for scaling ndn routing." Technical Report NDN-0005, NDN, 2015.

[AYW15b] Alexander Afanasyev, Cheng Yi, Lan Wang, Beichuan Zhang, and Lixia Zhang. "SNAMP: Secure namespace mapping to scale NDN forwarding." In *Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on*, pp. 281–286. IEEE, 2015.

[BBH06] Raymond C Browning, Emily A Baker, Jessica A Herron, and Rodger Kram. "Effects of obesity and sex on the energetic cost and preferred speed of walking." *Journal of Applied Physiology*, **100**(2):390–398, 2006.

[Car13] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[CDG08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. "Bigtable: A distributed storage system for structured data." *ACM Transactions on Computer Systems (TOCS)*, **26**(2):4, 2008.

[CHS12] Connie Chen, David Haddad, Joshua Selsky, Julia E Hoffman, Richard L Kravitz, Deborah E Estrin, and Ida Sim. "Making sense of mobile health data: an open architecture to improve individual-and population-level health." *Journal of medical Internet research*, **14**(4), 2012.

[DH76] Whitfield Diffie and Martin Hellman. "New directions in cryptography." *IEEE transactions on Information Theory*, **22**(6):644–654, 1976.

[DHJ07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store." In *ACM SIGOPS operating systems review*, volume 41, pp. 205–220. ACM, 2007.

[DR08] T. Dierks and E. Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.2." RFC 5246, RFC Editor, August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`.

[ES10] Deborah Estrin and Ida Sim. "Open mHealth architecture: an engine for health care innovation." *Science*, **330**(6005):759–760, 2010.

[FAC15] Wenliang Fu, Hila Ben Abraham, and Patrick Crowley. "Synchronizing namespaces with invertible bloom filters." In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pp. 123–134. IEEE, 2015.

[FSD15] Chengyu Fan, Susmit Shannigrahi, Steve DiBenedetto, Catherine Olschanowsky, Christos Papadopoulos, and Harvey Newman. "Managing scientific data with named data networking." In *Proceedings of the Fifth International Workshop on Network-Aware Data Management*, p. 1. ACM, 2015.

[Ham10] Eran Hammer-Lahav. "The oauth 1.0 protocol." 2010.

[Har12] Dick Hardt. "The OAuth 2.0 authorization framework." 2012.

[HCS17] Pedro de-las Heras-Quirós, Eva M Castro, Wentao Shang, Yingdi Yu, Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. "The design of RoundSync protocol." Technical Report NDN-0048, NDN, 2017.

[jso17] "JSON schema." `http://json-schema.org/`, 2017.

[JST09] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. "Networking Named Content." In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pp. 1–12. ACM, 2009.

[KLL97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web." In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663. ACM, 1997.

[Kra93] David W Kravitz. "Digital signature algorithm.", July 27 1993. US Patent 5,231,668.

[KST11] Misha Kay, Jonathan Santos, and Marina Takane. "mHealth: New horizons for health through mobile technologies." *World Health Organization*, **64**(7):66–71, 2011.

[LAS] Yanbiao Li, Alexander Afanasyev, Junxiao Shi, et al. "NDN Automatic Prefix Propagation." Technical Report NDN-0045, NDN.

[Moi14] Ilya Moiseenko. "Fetching content in Named Data Networking with embedded manifests." Technical Report NDN-0025, NDN, 2014.

[Mos14] Marc Mosko. "CCNx 1.0 Collection Synchronization." In *Technical Report*. Palo Alto Research Center, Inc., 2014.

[MUW17] Marc Mosko, Ersin Uzun, and Christopher A Wood. "Mobile Sessions in Content-Centric Networks." In *IFIP Networking*, 2017.

[MWZ15] Ilya Moiseenko, Lijing Wang, and Lixia Zhang. "Consumer/producer communication with application level framing in named data networking." In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*, pp. 99–108. ACM, 2015.

[NDN15] NDN Project Team. "Signed Interest." Online: `https://redmine.named-data.net/projects/ndn-cxx/wiki/SignedInterest`, 2015.

[NIS13] NIST. "Descriptions of SHA-256, SHA-384, and SHA-512." http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf, May 2013.

[OA10] Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

[Paz18] José Rolando Guay Paz. "Introduction to Azure Cosmos DB." In *Microsoft Azure Cosmos DB Revealed*, pp. 1–23. Springer, 2018.

[Pro12] ProjectCCNx. "CCNx Synchronization Protocol." Online: `https://github.com/ProjectCCNx/ccnx/blob/master/doc/technical/SynchronizationProtocol.txt`, 2012.

[RAF12] Nithya Ramanathan, Faisal Alquaddoomi, Hossein Falaki, Dony George, Cheng-Kang Hsieh, John Jenkins, Cameron Ketcham, Brent Longstaff, Jeroen Ooms, Joshua Selsky, et al. "Ohmage: an open mobile system for activity and experience sampling." In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2012 6th International Conference on*, pp. 203–204. IEEE, 2012.

[SA15] Weiqi Shi and Alexander Afanasyev. "RepoSync: Combined action-based and data-based synchronization model in Named Data Network." In *Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on*, pp. 275–280. IEEE, 2015.

[Sal13] Arto Salomaa. *Public-key cryptography*. Springer Science & Business Media, 2013.

[SAZ17] Wentao Shang, Alexander Afanasyev, and Lixia Zhang. "VectorSync: distributed dataset synchronization over named data networking." In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pp. 192–193. ACM, 2017.

[SAZ18] Wentao Shang, Alexander Afanasyev, and Lixia Zhang. "VectorSync: Distributed Dataset Synchronization over Named Data Networking." Technical Report NDN-0056, NDN, 2018.

[SJ09] Diana Smetters and Van Jacobson. "Securing Network Content." Technical report, Citeseer, 2009.

[spe17] "The Limits of Human Speed." `https://www.ncsf.org/enew/articles/articles-limitsofhumanspeed.aspx`, 2017.

[SRT15] Bruno MC Silva, Joel JPC Rodrigues, Isabel de la Torre Díez, Miguel López-Coronado, and Kashif Saleem. "Mobile-health: A review of current state in 2015." *Journal of biomedical informatics*, **56**:265–272, 2015.

[SYW17] Wentao Shang, Yingdi Yu, Lijing Wang, Alexander Afanasyev, and Lixia Zhang. "A Survey of Distributed Dataset Synchronization in Named Data Networking." Technical Report NDN-0053, NDN, 2017.

[THL15] Hongsuda Tangmunarunkit, Cheng-Kang Hsieh, Brent Longstaff, S Nolen, John Jenkins, Cameron Ketcham, Joshua Selsky, Faisal Alquaddoomi, Dony George, Jinha Kang, et al. "Ohmage: A general and extensible end-to-end participatory sensing platform." *ACM Transactions on Intelligent Systems and Technology (TIST)*, **6**(3):38, 2015.

[XZL18] Xin Xu, Haitao Zhang, Tianxiang Li, and Lixia Zhang. "Achieving Resilient Data Availability in Wireless Sensor Networks." In *Proceedings of the IEEE International Conference on Communications (ICN-SRA)*. IEEE, May 2018.

[YAC15] Yingdi Yu, Alexander Afanasyev, David Clark, et al. "Schematizing trust in Named Data Networking." In *Proceedings of the 2nd ACM Conference on ICN*, pp. 177–186. ACM, 2015.

[YAZ15] Yingdi Yu, Alexander Afanasyev, and Lixia Zhang. "Name-Based Access Control." Technical Report NDN-0034, NDN, 2015.

[Yu15] Yingdi Yu. "Public Key Management in Named Data Networking." Technical Report NDN-0029, NDN, 2015.

[ZA13] Zhenkai Zhu and Alexander Afanasyev. "Let's chronosync: Decentralized dataset state synchronization in named data networking." In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pp. 1–10. IEEE, 2013.

[ZAB14] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, et al. "Named Data Networking." *ACM SIGCOMM Computer Communication Review*, 2014.

[ZAB16] Yu Zhang, Alexander Afanasyev, Jeff Burke, and Lixia Zhang. "A survey of mobility support in named data networking." In *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*, pp. 83–88. IEEE, 2016.

[ZAZ13] Zhenkai Zhu, Alexander Afanasyev, and Lixia Zhang. "A new perspective on mobility support." Technical Report NDN-0013, NDN, 2013.

[ZAZ17] Zhiyi Zhang, Alexander Afanasyev, and Lixia Zhang. "NDNCERT: Universal Usable Trust Management for NDN." In *Proceedings of the 4th ACM Conference on ICN*, 2017.

[ZEB10] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D Thornton, Diana K Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, Christos Papadopoulos, et al. "Named Data Networking (NDN) Project." Technical Report NDN-0001, NDN, 2010.

[ZLW17] Minsheng Zhang, Vince Lehman, and Lan Wang. "Scalable name-based data synchronization for named data networking." In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*, pp. 1–9. IEEE, 2017.

[ZYA17] Zhiyi Zhang, Yingdi Yu, Alexander Afanasyev, and Lixia Zhang. "NDN Certificate Management Protocol (NDNCERT)." Technical Report NDN-0050, NDN, 2017.

[ZYZ18] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. "An Overview of Security Support in Named Data Networking." Technical Report NDN-0057, NDN, 2018.