

This revision is **OUTDATED** and preserved for historical purposes only.

Check for the most recent revision at  
<https://named-data.net/publications/techreports/>

# NFD Developer's Guide

Alexander Afanasyev<sup>1</sup>, Junxiao Shi<sup>2</sup>, Beichuan Zhang<sup>2</sup>, Lixia Zhang<sup>1</sup>, Ilya Moiseenko<sup>1</sup>, Yingdi Yu<sup>1</sup>, Wentao Shang<sup>1</sup>, Yanbiao Li<sup>1</sup>, Spyridon Mastorakis<sup>1</sup>, Yi Huang<sup>2</sup>, Jerald Paul Abraham<sup>2</sup>, Steve DiBenedetto<sup>3</sup>, Chengyu Fan<sup>3</sup>, Christos Papadopoulos<sup>3</sup>, Davide Pesavento<sup>4</sup>, Giulio Grassi<sup>4</sup>, Giovanni Pau<sup>4</sup>, Hang Zhang<sup>5</sup>, Tian Song<sup>5</sup>, Haowei Yuan<sup>6</sup>, Hila Ben Abraham<sup>6</sup>, Patrick Crowley<sup>6</sup>, Syed Obaid Amin<sup>7</sup>, Vince Lehman<sup>7</sup>, and Lan Wang<sup>7</sup>

<sup>1</sup>University of California, Los Angeles

<sup>2</sup>The University of Arizona

<sup>3</sup>Colorado State University

<sup>4</sup>University Pierre & Marie Curie, Sorbonne University

<sup>5</sup>Beijing Institute of Technology

<sup>6</sup>Washington University in St. Louis

<sup>7</sup>The University of Memphis

NFD Team

## Abstract

NDN Forwarding Daemon (NFD) is a network forwarder that implements the Named Data Networking (NDN) protocol. NFD is designed with *modularity* and *extensibility* in mind to enable easy experiments with new protocol features, algorithms, and applications for NDN. To help developers extend and improve NFD, this document explains NFD's internals including the overall design, major modules, their implementations, and their interactions.

## Revision history

- **Revision 5 (Oct 27, 2015):**
  - Add description of CS CachePolicy API, including information about new LRU policy
  - BroadcastStrategy renamed to MulticastStrategy
  - Added overview of how forwarder processes Link objects
  - Added overview of the new face system (incomplete)
  - Added description of the new automatic prefix propagation feature
  - Added description of the refactored management
  - Added description of NetworkRegionTable configuration
  - Added description about client.conf and NFD
- **Revision 4 (May 12, 2015):** New section about testing and updates for NFD version 0.3.2:
  - Added description of new ContentStore implementation, including a new async lookup model of CS
  - Added description of the remote prefix registration
  - Updated Common Services section
- **Revision 3 (February 3, 2015):** Updates for NFD version 0.3.0:
  - In Strategy interface, beforeSatisfyPendingInterest renamed to beforeSatisfyInterest
  - Added description of dead nonce list and related changes to forwarding pipelines
  - Added description of a new `strategy_choice` config file subsection
  - Amended unix config text to reflect removal of "listen" option
  - Added discussion about encapsulation of NDN packets inside WebSocket messages
  - Revised FaceManager description, requiring canonical FaceUri in create operations
  - Added description of the new access router strategy

---

This revision is **OUTDATED** and preserved for historical purposes only. Check for the most recent revision at <https://named-data.net/publications/techreports/>

- **Revision 2 (August 25, 2014):** Updated steps in forwarding pipelines, `nfd::BestRouteStrategy` is replaced with `nfd::BestRouteStrategy2` that allows client-based recovery from Interest losses
- **Revision 1 (July 1, 2014):** Initial release

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	NFD Modules . . . . .	6
1.2	How Packets are Processed in NFD . . . . .	7
1.3	How Management Interests are Processed in NFD . . . . .	8
<b>2</b>	<b>Face System</b>	<b>9</b>
2.1	Face . . . . .	9
2.2	Transport . . . . .	10
2.2.1	Unix Stream Transport . . . . .	11
2.2.2	TCP Transport . . . . .	11
2.2.3	UDP unicast Transport . . . . .	11
2.2.4	UDP multicast Transport . . . . .	11
2.2.5	Ethernet Transport . . . . .	11
2.2.6	WebSocket Transport . . . . .	11
2.2.7	Internal Transport . . . . .	12
2.2.8	Dummy Transport . . . . .	12
2.3	Link Service . . . . .	12
2.3.1	Generic Link Service . . . . .	12
2.3.2	Vehicular Link Service . . . . .	12
2.3.3	Developing a New Link Service . . . . .	12
2.4	FaceManager, Protocol Factory, and Channel . . . . .	12
<b>3</b>	<b>Face System (2014)</b>	<b>13</b>
3.1	Face URI . . . . .	13
3.2	Protocol Factory Abstraction . . . . .	15
3.3	Channel Abstraction . . . . .	15
3.4	Face Abstraction . . . . .	16
3.5	WebSocket Face and Encapsulation of NDN Packet . . . . .	17
3.6	Extending NFD Face System . . . . .	17
<b>4</b>	<b>Tables</b>	<b>18</b>
4.1	Forwarding Information Base (FIB) . . . . .	18
4.1.1	Structure and Semantics . . . . .	18
4.1.2	Usage . . . . .	19
4.2	Network Region Table . . . . .	19
4.3	Content Store (CS) . . . . .	19
4.3.1	Semantics and Usage . . . . .	19
4.3.2	Implementation . . . . .	20
4.4	Interest Table (PIT) . . . . .	22
4.4.1	PIT Entry . . . . .	22
4.4.2	PIT . . . . .	23
4.5	Dead Nonce List . . . . .	23
4.5.1	Structure, Semantics, and Usage . . . . .	23
4.5.2	Capacity Maintenance . . . . .	24
4.6	Strategy Choice Table . . . . .	24
4.6.1	Structure and Semantics . . . . .	24
4.6.2	Usage . . . . .	25
4.7	Measurements Table . . . . .	25
4.7.1	Structure . . . . .	25
4.7.2	Usage . . . . .	26
4.8	NameTree . . . . .	26
4.8.1	Structure . . . . .	26
4.8.2	Operations and Algorithms . . . . .	28
4.8.3	Shortcuts . . . . .	28

<b>5</b>	<b>Forwarding</b>	<b>30</b>
5.1	Forwarding Pipelines . . . . .	30
5.2	Interest Processing Path . . . . .	30
5.2.1	Incoming Interest Pipeline . . . . .	31
5.2.2	Interest Loop Pipeline . . . . .	32
5.2.3	ContentStore Miss Pipeline . . . . .	32
5.2.4	ContentStore Hit Pipeline . . . . .	33
5.2.5	Outgoing Interest Pipeline . . . . .	33
5.2.6	Interest Reject Pipeline . . . . .	34
5.2.7	Interest Unsatisfied Pipeline . . . . .	34
5.2.8	Interest Finalize Pipeline . . . . .	35
5.3	Data Processing Path . . . . .	35
5.3.1	Incoming Data Pipeline . . . . .	35
5.3.2	Data Unsolicited Pipeline . . . . .	36
5.3.3	Outgoing Data Pipeline . . . . .	36
<b>6</b>	<b>Forwarding Strategy</b>	<b>38</b>
6.1	Strategy API . . . . .	38
6.1.1	Triggers . . . . .	38
6.1.2	Actions . . . . .	39
6.1.3	Storage . . . . .	39
6.2	Built-in Strategies . . . . .	39
6.2.1	Best Route Strategy . . . . .	40
6.2.2	Multicast Strategy . . . . .	40
6.2.3	Client Control Strategy . . . . .	40
6.2.4	NCC Strategy . . . . .	40
6.2.5	Access Router Strategy . . . . .	40
6.3	How to Develop a New Strategy . . . . .	41
6.3.1	Should I Develop a New Strategy? . . . . .	41
6.3.2	Develop a New Built-in Strategy . . . . .	41
<b>7</b>	<b>Management</b>	<b>43</b>
7.1	Managers . . . . .	44
7.1.1	Face Manager . . . . .	44
7.1.2	FIB Manager . . . . .	46
7.1.3	Strategy Choice Manager . . . . .	46
7.1.4	Forwarder Status Manager . . . . .	46
7.2	Management Support Classes . . . . .	46
7.2.1	Manager Base . . . . .	46
7.2.2	Internal Face and Internal Client Face . . . . .	47
7.2.3	Command Validator . . . . .	47
7.2.4	General Configuration File Section Parser . . . . .	47
7.2.5	Tables Configuration File Section Parser . . . . .	47
<b>8</b>	<b>RIB Management</b>	<b>48</b>
8.1	Initializing RIB Manager . . . . .	48
8.2	Communicating with RIB Manager . . . . .	50
8.2.1	Registering a Route . . . . .	50
8.2.2	Unregistering a Route . . . . .	50
8.3	RIB Entry . . . . .	50
8.4	Prefix Registration Flags . . . . .	51
8.4.1	Examples . . . . .	51
8.4.2	Cost Inheritance . . . . .	51
8.5	FIB Updater . . . . .	52
8.6	On Request . . . . .	52
8.7	Auto Prefix Propagator . . . . .	52
8.7.1	What Prefix to Propagate . . . . .	53

8.7.2	When to Propagate . . . . .	54
8.7.3	Secure Propagations . . . . .	54
8.7.4	Propagated-entry State Machine . . . . .	54
8.7.5	Configure Auto Prefix Propagator . . . . .	55
8.8	Extending RIB Manager . . . . .	55
<b>9</b>	<b>Security</b>	<b>57</b>
9.1	Interface Control . . . . .	57
9.2	Trust Model . . . . .	57
9.2.1	Command Interest . . . . .	57
9.2.2	NFD Trust Model . . . . .	58
9.2.3	NFD RIB manager Trust Model . . . . .	58
9.3	Local Key Management . . . . .	58
<b>10</b>	<b>Common Services</b>	<b>59</b>
10.1	Configuration File . . . . .	59
10.1.1	User Info . . . . .	59
10.1.2	Developer Info . . . . .	61
10.2	Basic Logger . . . . .	61
10.2.1	User Info . . . . .	61
10.2.2	Developer Info . . . . .	62
10.3	Hash Computation Routines . . . . .	62
10.4	Global Scheduler . . . . .	62
10.5	Global IO Service . . . . .	63
10.6	Privilege Helper . . . . .	63
<b>11</b>	<b>Testing</b>	<b>64</b>
11.1	Unit Tests . . . . .	64
11.1.1	Test Structure . . . . .	64
11.1.2	Running Tests . . . . .	64
11.1.3	Test Helpers . . . . .	64
11.1.4	Test Code Guidelines and Naming Conventions . . . . .	65
11.2	Integration Tests . . . . .	65
	<b>References</b>	<b>66</b>

# 1 Introduction

NDN Forwarding Daemon (NFD) is a network forwarder that implements and evolves together with the Named Data Networking (NDN) protocol [1]. This document explains the internals of NFD and is intended for developers who are interested in extending and improving NFD. Other information about NFD, including instructions of how to compile and run NFD, are available on NFD's home page [2].

The main design goal of NFD is to support diverse experimentation with NDN architecture. The design emphasizes *modularity* and *extensibility* to allow easy experiments with new protocol features, algorithms, and applications. We have not fully optimized the code for performance. The intention is that performance optimizations are one type of experiments that developers can conduct by trying out different data structures and different algorithms; over time, better implementations may emerge within the same design framework.

NFD will keep evolving in three aspects: improvement of the modularity framework, keeping up with the NDN protocol spec, and addition of new features. We hope to keep the modular framework stable and lean, allowing researchers to implement and experiment with various features, some of which may eventually work into the protocol specification.

## 1.1 NFD Modules

The main functionality of NFD is to forward Interest and Data packets. To do this, it abstracts lower-level network transport mechanisms into NDN Faces, maintains basic data structures like CS, PIT, and FIB, and implements the packet processing logic. In addition to basic packet forwarding, it also supports multiple forwarding strategies, and a management interface to configure, control, and monitor NFD. As illustrated in Figure 1, NFD contains the following inter-dependent modules:

- **ndn-cxx Library, Core, and Tools** (Section 10)

Provides various common services shared between different NFD modules. These include hash computation routines, DNS resolver, config file, Face monitoring, and several other modules.

- **Faces** (Section 2)

Implements the NDN Face abstraction on top of various lower level transport mechanisms.

- **Tables** (Section 4)

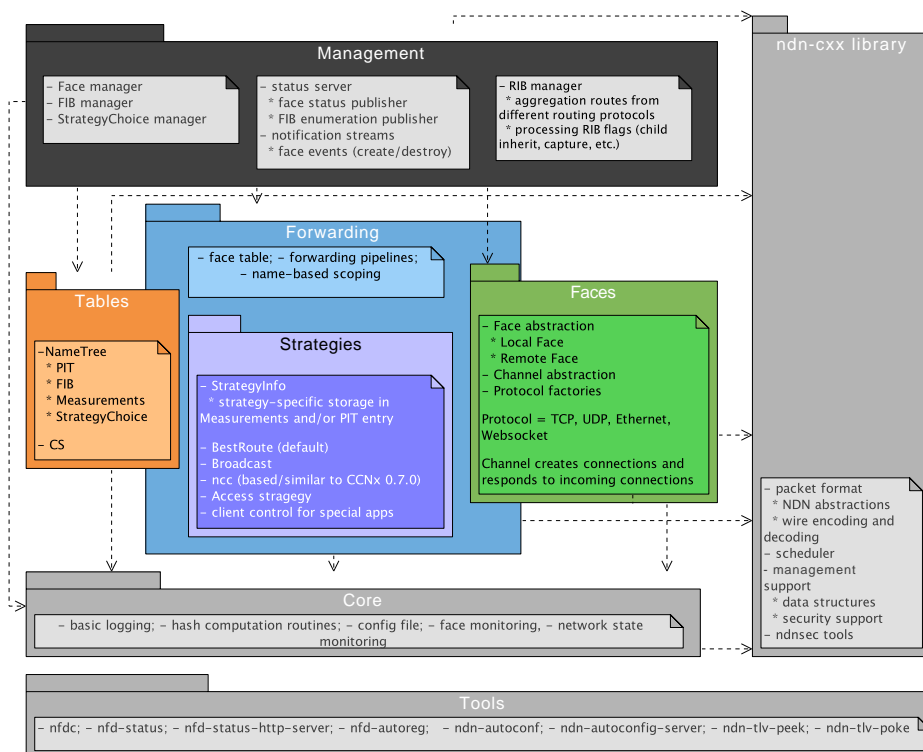


Figure 1: Overview of NFD modules

Implements the Content Store (CS), the Pending Interest Table (PIT), the Forwarding Information Base (FIB), StrategyChoice, Measurements, and other data structures to support forwarding of NDN Data and Interest packets.

- **Forwarding** (Section 5)

Implements basic packet processing pathways, which interact with Faces, Tables, and Strategies (Section 6).

Strategies is a major part of the forwarding module. It implements a framework to support different forwarding strategies in the form of forwarding pipelines, described in detail in Section 5.

- **Management** (Section 7)

Implements the NFD Management Protocol [3], which allows applications to configure NFD and set/query NFD's internal states. Protocol interaction is done via NDN's Interest/Data exchange between applications and NFD.

- **RIB Management** (Section 8)

Manages the routing information base (RIB). The RIB may be updated by different parties in different ways, including various routing protocols, application prefix registrations, and command-line manipulation by sysadmins. The RIB management module processes all these requests to generate a consistent forwarding table, and syncs it up with NFD's FIB, which contains only the minimal information needed for forwarding decisions.

The rest of this document will explain all these modules in more detail.

## 1.2 How Packets are Processed in NFD

To give readers a better idea on how NFD works, this section explains how a packet is processed in NFD.

Packets arrive at NFD via *Faces*. “Face” is a generalization of “interface”. It can be either a physical interface (where NDN operates directly over Ethernet), or a overlay tunnel (where NDN operates as an overlay above TCP or UDP). In addition, NFD communicates with local application via a UNIX-domain socket, which is also a Face. The Face reads incoming stream or datagrams via operating system API, strips away link-layer headers if any, and delivers network layer packets (NDN-TLV Interest or Data) to the forwarding.

A network layer packet (Interest or Data) is processed by *forwarding pipelines*, which define series of steps that operate on the packet. NFD's data plane is stateful, and what NFD does to a packet depends on not only the packet itself but also the forwarding state, which is stored in *tables*.

An incoming Interest is first inserted to the *Interest table* (PIT), where each entry represents a pending Interest or a recently satisfied Interest. If the Interest is not a duplicate, a lookup is performed on the *Content Store* (CS), which is an in-network cache of Data packets. If there is a matching Data packet in CS, that Data packet is returned to the requester; otherwise, the Interest needs to be forwarded.

A *forwarding strategy* decides how to forward an Interest. NFD allows per-namespace strategy choice; to decide which strategy is responsible for forwarding an Interest, a longest prefix match lookup is performed on the *Strategy Choice table*, which contains strategy configuration. The strategy responsible for an Interest (or, more precisely, the PIT entry) makes a decision whether, when, and where to forward the Interest. While making this decision, the strategy can take input from the *Forwarding Information Base* (FIB), which contains routing information that comes from local application's prefix registrations and routing protocols, use strategy-specific information stored in the PIT entry, and record and use data plane performance measurements in Measurements entry.

After the strategy decides to forward an Interest to a specified Face, the Interest goes through a few more steps in forwarding pipelines, and then it is passed to the Face. The Face, depending on the underlying protocol, adds link-layer header, fragments if necessary, and sends the link-layer packets as outgoing stream or datagrams via operating system API.

An incoming Data is processed differently. Its first step is checking the Interest table to see if there are PIT entries that can be satisfied by this Data packet. All matched entries are then selected for further processing. If this Data can satisfy none of the PIT entries, it is unsolicited and it is dropped; as a special case, an unsolicited Data from a local application is cached in the Content Store. Otherwise, the Data is added to the Content Store. Forwarding strategy that is responsible for each of the mat PIT entries is notified. Through this notification, and another “no Data comes back” timeout, the strategy is able to observe the reachability and performance of paths; the strategy can remember its observations in the *Measurements table*, in order to improve its future decisions. Finally, the Data is sent to all requesters, recorded in downstream records (in-records) of the PIT entries; the process of sending a Data via a Face is similar to sending an Interest.



### 1.3 How Management Interests are Processed in NFD

NFD Management protocol [3] defines three inter-process management mechanisms that are based on Interest-Data exchanges: control commands, status datasets, and notification streams. This section gives a brief overview how these mechanisms work and what are their requirements.

A **control command** is a signed (authenticated) Interest to perform a state change within NFD. Since the objective of each control command Interest is to reach the destination management module and not be satisfied from CS, each control command Interest is made unique through the use of timestamp and nonce components. For more detail refer to control command specification [4].

When NFD receives the command Interest, it directs the Interest to a special Face, called the *Internal Face*.<sup>1</sup> When an Interest is forwarded to this Face, it is dispatched internally to a designated *manager* (e.g., Interests under `/localhost/nfd/faces` are dispatched to the Face manager, see Section 7). The manager then looks at the Interest name to decide which action is requested. If the name refers to a valid control command, the manager validates the command (checks the signature and validates whether the requester is authorized to send this command), and performs the requested action if validation succeeds. The response is sent back to the requester as a Data packet, which is processed by forwarding and Face in the same way as a regular Data.

The exception from the above procedure is RIB Management (Section 8), which is implemented as a separate daemon. All RIB Management control commands, instead of Internal Face, are forwarded toward the RIB daemon, the same way as forwarding to any local application (RIB daemon “registers” itself with NFD for the RIB management prefix when it starts).

A **status dataset** is either periodically (e.g., NFD status) or on-demand (e.g., NFD Face status) generated dataset with some internal NFD status. These datasets can be requested by anybody using a simple unsigned Interest directed towards the specific management module, as defined in the specification [3]. An Interest requesting a new version of a **status dataset** is forwarded to the internal Face and then the designated manager the same way as control commands. The manager, however, will not validate this Interest, but instead generate all segments of the requested dataset and put them into the forwarding pipeline. This way, the first segment of the dataset will directly satisfy the initial Interest, while others will satisfy the subsequent Interests through CS. In the unlikely event when subsequent segments are evicted from the CS before being fetched, the requester is responsible for restarting the fetch process from the beginning.

**Notification streams** are similar to status datasets (can be accessed by anybody with an unsigned Interests), but they operate slightly differently. The subscribers that want receive notification streams are still sending Interests and these Interests are still directed toward the designated manager. However, it is expected that these Interests will not be satisfied immediately, and the subscribers are expected to re-express the notification stream Interests when they expire. Whenever a notification is generated, the manager puts a Data packet into the forwarding, satisfying all outstanding notification stream Interests, and the notification is getting delivered to all subscribers.

<sup>1</sup>There is always a FIB entry for the management protocol prefix that points to the Internal Face.

## 2 Face System

NOTE: A major refactoring is underway since Aug 2015 in NFD face system. This section describes the new design. Section 3 describes the old design from 2014, which will be removed after refactoring completes.

*Face* is the generalization of network interface. Similar to a physical network interface, packets can be sent and received on a face. A face is more general than a network interface. It could be:

- a physical network interface to communicate on a physical link
- an overlay communication channel between NFD and a remote node
- an inter-process communication channel between NFD and a local application

The face abstraction provides a best-effort delivery service for NDN network layer packets. Forwarding can send and receive Interests, Data, and Nacks through faces. The face then handles the underlying communication mechanisms (e.g. sockets), and hides the differences of underlying protocols from forwarding.

Section 2.1 introduces the semantics of face, how it's used by forwarding, and its internal structure consisting of transport (Section 2.2) and link service (Section 2.3). Section 2.4 describes how faces are created and organized.

### 2.1 Face

NFD, as a network forwarder, moves packets between network interfaces. NFD can communicate on not only physical network interfaces, but also on a variety of other communication channels, such as overlay tunnels over TCP and UDP. Therefore, we generalize “network interface” as “face”, which abstracts a communication channel that NFD can use for packet forwarding. The face abstraction (`nfd::Face` class) provides a best-effort delivery service for NDN network layer packets. Forwarding can send and receive Interests, Data, and Nacks through faces. The face then handles the underlying communication mechanisms (e.g. sockets), and hides the differences of underlying protocols from forwarding.

In NFD, the inter-process communication channel between NFD and a local application is also treated as a face. This differs from traditional TCP/IP network stack, where local applications use syscalls to interact with the network stack, while network packets exist on the wire only. NFD is able to communicate with a local application via a face, because the network layer packet format is same as the packets on the wire. Having an unified face abstraction for both local application and remote hosts simplifies NFD architecture.

**How forwarding uses faces?** The `FaceTable` class, which is part of forwarding, keeps track of all active faces. A newly created face is passed to `FaceTable::add`, which assigns the face a numeric `FaceId` for identification purpose. After a face is closed, it's removed from the `FaceTable`. Forwarding receives packets from a face by connecting to `afterReceiveInterest` `afterReceiveData` `afterReceiveNack` signals, which is done in `FaceTable`. Forwarding can send network layer packets via the face by calling `sendInterest` `sendData` `sendNack` methods.

**Face attributes** A face exposes some attributes to display its status and control its behavior.

- **FaceId** is a numeric ID to identify the face. It's assigned a non-zero value by `FaceTable`, and cleared when a face is removed from the `FaceTable`.
- **LocalUri** is a `FaceUri` (Section 2.2) that represents the local endpoint. This attribute is read-only.
- **RemoteUri** is a `FaceUri` (Section 2.2) that represents the remote endpoint. This attribute is read-only.
- **Scope** indicates whether the face is local for scope control purposes. It can be either *non-local* or *local*. This attribute is read-only.
- **Persistency** controls the behavior when an error occurs in the underlying communication channel, or when the face has been idle for some time.
  - A *on-demand* face closes if it remains idle for some time, or when an error occurs in the underlying communication channel.
  - A *persistent* face remains open until it's explicitly destroyed, or when an error occurs in the underlying communication channel.
  - A *permanent* face remains open until it's explicit destroyed; errors in the underlying communication channel are recovered internally.
- **LinkType** indicates the type of communication link. It can be either *point-to-point* or *multi-access*. This attribute is read-only.

- **State** indicates the current availability of the face.
  - *UP*: the face is working properly
  - *DOWN*: the face is down temporarily, and is being recovered; sending packets is possible but delivery is unlikely to succeed
  - *CLOSING*: the face is being closed normally
  - *FAILED*: the face is being closed due to a failure
  - *CLOSED*: the face has been closed
- **Counters** gives statistics about the count and size of Interests, Data, Nacks, and lower layer packets sent and received on the face.

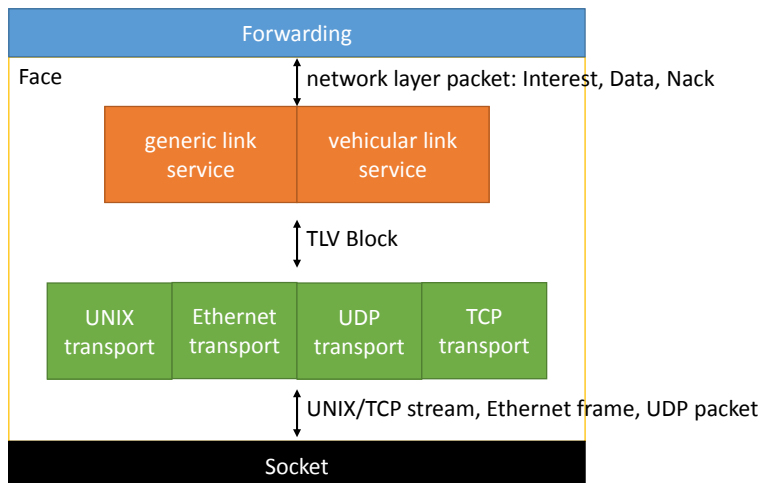


Figure 2: Face = LinkService + Transport

**Internal structure** Internally, a face is composed of a **link service** and a **transport** (Figure 2). The transport (Section 2.2) is the lower part of a face, which wraps the underlying communication mechanism (such as sockets or libpcap handles), and exposes a best-effort TLV packet delivery service. The link service (Section 2.3) is the upper part of a face, which translates between network layer packets and lower layer packets, and provides additional services such as fragmentation, link failure detection, retransmission.

The face is implemented as `nfd::face::Face` class. (NOTE: During refactoring, this class is temporarily named `nfd::face::LpFace` to be distinguished from the old `nfd::Face`). The Face class is designed to be non-inheritable (except for unit testing), and the link service and transport passed to its constructor fully defines its behavior. Within the constructor, both the link service and the transport are given a pointer to each other and to the face, so that they may call into each other with lowest runtime overhead.

## 2.2 Transport

A **transport** (`nfd::face::Transport` base class) provides best-effort packet delivery service to the link service of the face. The link service may invoke `Transport::send` to send a packet. When a packet arrives, `LinkService::receivePacket` will be invoked. Each packet must be a complete TLV block; the transport makes no assumption on the TLV-TYPE of this block. In addition, each received packet is accompanied with an `EndpointId` which indicates the sender of this packet, which is useful for fragment reassembly, failure detection, and other purposes on a multi access link.

**Transport attributes** The transport provides `LocalUri`, `RemoteUri`, `Scope`, `Persistency`, `LinkType`, `State` attributes. The transport also maintains lower-layer packet counters and byte counters on incoming and outgoing directions. These attributes and counters are accessible through the face.

If a transport's persistency is set to *permanent*, the transport is responsible for taking necessary actions to recover from underlying failures. For example: a UDP transport should ignore socket errors; a TCP transport should attempt to re-establish TCP connection if the previous connection is closed. The progress of such recovery is reflected in the `State` attribute.

In addition, the transport provides the following attributes for use by link service:

- **Mtu** indicates the maximum packet size that can be sent or received through this transport. It can be either a positive number, or a special `MTU_UNLIMITED` that indicates there's no limit on packet size. This attribute is read-only. The transport may change the value of this attribute at any time, and the link service should be prepared for such changes.

**FaceUri** **FaceUri** is a URI that represents the endpoint or communication channel used by a transport. It starts with a scheme that indicates the underlying protocol (e.g. `udp4`), followed by a scheme-specific representation of the underlying address. It's used in `LocalUri` and `RemoteUri` attributes.

The rest of this section describes each individual transport for different underlying communication mechanism, including their `FaceUri` format, implementation details, and feature limitations.

### 2.2.1 Unix Stream Transport

Unix stream transport (`nfd::face::UnixStreamTransport`) is a transport that communicates on stream-oriented Unix domain sockets.

NFD listens for incoming connections via `UnixStreamChannel` at a named socket whose path is specified by `face_system.unix.pat` configuration option. A `UnixStreamTransport` is created for each incoming connection. NFD does not support making outgoing Unix stream connections.

The static attributes of a Unix stream transport are:

- **LocalUri** `unix://path` where *path* is the socket path; e.g. `unix:///var/run/nfd.sock`
- **RemoteUri** `fd://file-descriptor` where *file-descriptor* is the file descriptor of the accepted socket within NFD process; e.g. `fd://30`
- **Scope** `local`
- **Persistency** `on-demand`; other persistency settings are disallowed
- **LinkType** `point-to-point`
- **Mtu** `unlimited`

### 2.2.2 TCP Transport

TCP transport (`nfd::face::TcpTransport`) is a transport that communicates on TCP tunnels over IPv4 or IPv6.

NFD listens for incoming connections via `TcpChannel` at a port number specified by `face_system.tcp.port` configuration option. A `TcpTransport` is created for each incoming connection. NFD can also make outgoing TCP connections.

The static attributes of a TCP transport are:

- **LocalUri** and **RemoteUri**
  - IPv4 `tcp4://ip:port` ; e.g. `tcp4://192.0.2.1:6363`
  - IPv6 `tcp6://ip:port` where *ip* is in lower case and enclosed by square brackets; e.g. `tcp6://[2001:db8::1]:6363`
- **Scope** `local` if remote endpoint has loopback IP, non-local otherwise
- **Persistency** `on-demand` for transport created from accepted socket, persistent for transport created for outgoing connection; changing between on-demand and persistent is allowed; permanent is unimplemented, but will be supported in the future
- **LinkType** `point-to-point`
- **Mtu** `unlimited`

### 2.2.3 UDP unicast Transport

TODO: describe `nfd::face::UnicastUdpTransport`

### 2.2.4 UDP multicast Transport

TODO: describe `nfd::face::MulticastUdpTransport`

### 2.2.5 Ethernet Transport

TODO: describe `nfd::face::EthernetTransport`

### 2.2.6 WebSocket Transport

TODO: describe `nfd::face::WebSocketTransport`

### 2.2.7 Internal Transport

TODO: describe `nfd::face::InternalTransport`

### 2.2.8 Dummy Transport

TODO: describe `nfd::face::tests::DummyTransport`

## 2.3 Link Service

A **link service** (`nfd::face::LinkService` base class) works on top of a transport and provides a best-effort network layer packet delivery service. A link service must translate between network layer packets (Interests, Data, and Nacks) and link layer packets (TLV blocks). In addition, additional link services may be provided, to bridge the gap between the desire of forwarding and the capabilities of the underlying transport. For example, if the underlying transport has a Maximum Transmission Unit (MTU) limit, fragmentation and reassembly will be needed in order to send and receive network layer packets larger than MTU; if the underlying transport has a high loss rate, per-link retransmission may be enabled to reduce loss and improve performance.

### 2.3.1 Generic Link Service

**Generic link service** (`nfd::face::GenericLinkService`) is the default service in NFD. Its link layer packet format is NDNLPv2 [5].

As of NFD 0.4.0, the following features are implemented:

1. encoding and decoding of Interest, Data, and Nack
  2. fragmentation and reassembly (indexed fragmentation)
  3. consumer controlled forwarding (NextHopFaceId field)
  4. local cache policy (CachePolicy field)
  5. incoming face indicates (IncomingFaceId field)
- Other planned features include:
1. failure detection (similar to BFD [6])
  2. link reliability improvement (repeat request, similar to ARQ [7])
- TODO: internal structure

### 2.3.2 Vehicular Link Service

**Vehicular link service** is a planned feature to implement a link service suitable for vehicular networks.

### 2.3.3 Developing a New Link Service

OUTLINE:

1. considerations of developing a new LinkService

## 2.4 FaceManager, Protocol Factory, and Channel

OUTLINE:

1. purpose of Protocol Factory
2. purpose of Channel
3. face creation workflow: from configuration
4. face creation workflow: from create face command

### 3 Face System (2014)

NOTE: A major refactoring is underway since Aug 2015 in NFD face system. Section 2 describes the new design. This section describes the old design from 2014, which will be removed after refactoring completes.

The face system in NFD is separated into three logical abstractions: protocol factories, channels, and faces. A **protocol factory** create channels or faces of specific protocols (e.g., the TCP protocol factory creates TCP faces). A **channel** represents NFD-side endpoint for unicast communications (i.e., “listening” socket or socket from which connection will be established). We call this endpoint “*local*”, while the “*remote*” endpoint represents the other side of the communication (a network node, an application, or a set of remote peers). Both endpoints in NFD are described using concept of FaceURI, which defines protocol and protocol-specific parameters of the endpoint (see Section 3.1). A **Face** is an abstraction which implements communication primitives to actually send and receive Interest and Data packets. Depending on nature of communication, a Face can represent slightly different elements. For “unicast” communications (TCP, unicast UDP, UNIX sockets, WebSocket [8]), a Face represents a connection between local and remote endpoints for the specific protocol, i.e, a connection between NFDs on different network nodes or local NFD and local application. For multi-access communications (Ethernet, IP multicast), a Face represents a virtual connection between a local endpoint and zero or more remote peers.

The overall interaction between these abstractions is illustrated in Figure 3 and each abstraction is described in detail in the following sections. In short, these interactions can be summarized as: protocol factories create channels, channels create faces, and faces are actually responsible for sending and receiving Interest and Data packets through the protocol-specific tunnel.

The current implementation is heavily based on the Boost.Asio library [9] and uses asynchronous operations as much as possible to avoid blocking the rest of the daemon while performing potentially lengthy network operations.

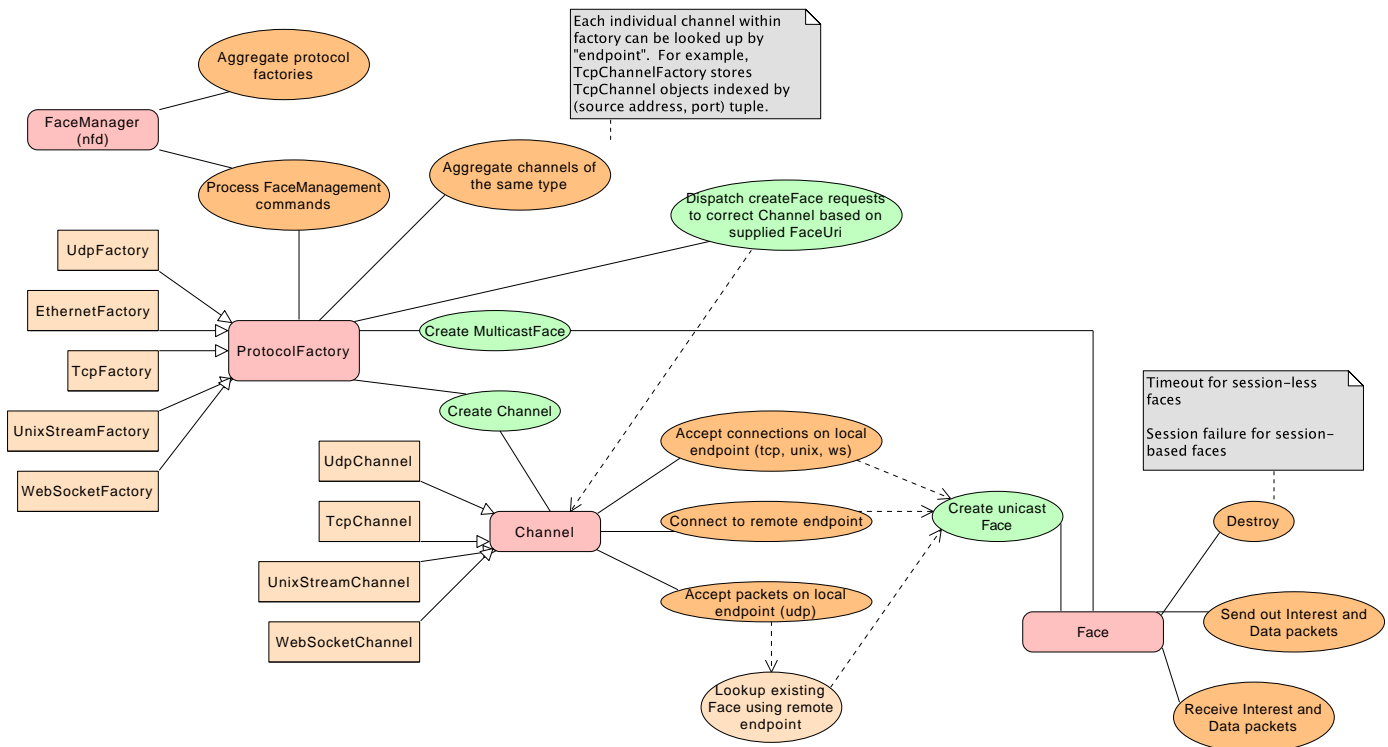


Figure 3: Face, Channel, ProtocolFactory interactions

#### 3.1 Face URI

A Face URI (`nfd::FaceUri` class) identifies a protocol endpoint and is used in Face (for local and remote endpoints) and in Channel (local endpoint) abstractions. The Face URI is similar as normal URI and is composed of the protocol name and other protocol-specific identifying information (e.g., IP address and port).

Note that some Face types will have different types of Face URI types for the local and remote endpoints. This depends on what exactly these endpoints represent. For example, UnixStream Face has UNIX Stream type of Face URI to represent the local endpoint, while FileDescriptor type used to identify the application connected to the UNIX socket. Another example

is multicast Ethernet Face, where the local endpoint represents NFD itself (MAC address of a network interface) and the remote endpoint represents the set of remote peers connected to the network interface (network device).

The following description shows examples of Face URI formats for the implemented protocols (see NFD Face Management specification for more detail [3]). When developing new Face, these examples can be used for defining Face URI format for the newly implemented protocol.

**UDP:** `udp[4|6]://<IP-or-host>[:<port>]`

- `udp4://192.0.2.1:6363` (canonical form)
- `udp6://[2001:db8::1]:6363` (canonical form)
- `udp://192.0.2.1` (remote-port defaults to 6363)
- `udp://example.net:6363`
- `udp4://example.net:6363` (resolve hostname to IPv4 address only)
- `udp6://example.net:6363` (resolve hostname to IPv6 address only)
- `udp4://224.0.23.170:56363` (multicast, canonical form)

**TCP:** `tcp[4|6]://<IP-or-host>[:<port>]`

- `tcp4://192.0.2.1:6363` (canonical form)
- `tcp6://[2001:db8::1]:6363` (canonical form)
- `tcp://192.0.2.1` (remote-port defaults to 6363)
- `tcp://example.net:6363`
- `tcp4://example.net:6363` (resolve hostname to IPv4 address only)
- `tcp6://example.net:6363` (resolve hostname to IPv6 address only)

**UNIX stream:** `unix://<path>`

- `unix:///var/run/nfd.sock` (note there are three forward-slashes after 'unix')

**File Descriptor:** `fd://<file-descriptor>`

- `fd://6`

**Ethernet** `ether://<MAC>`

- `ether://[08:00:27:01:01:01]`

**Network Device:** `dev://<ifname>`

- `dev://eth0`



### 3.2 Protocol Factory Abstraction

The protocol factory is the highest level abstraction in NFD’s face system. Each protocol factory handles a specific protocol that is natively supported by NFD. Table 1 describes the current set of support protocols.

NFD currently supports the protocols described in Table 1. Note that the protocols marked “listen only” are limited to listen and accept incoming connections and cannot initiate connections to remote endpoints.

Table 1: Supported protocols

Protocol	Unicast	Multicast	Factory Class	Channel Class	Face Class
Unix domain stream-oriented sockets	Listen only	No	UnixStreamFactory	UnixStreamChannel	UnixStreamFace
Raw Ethernet type-II frames	No	Yes	EthernetFactory	N/A	EthernetFace (multicast)
TCP	Yes	No	TcpFactory	TcpChannel	TcpFace, TcpLocalFace
UDP	Yes	Yes	UdpFactory	UdpChannel	UdpFace, UdpMulticastFace
WebSocket [8]	Listen only	No	WebSocketFactory	WebSocketChannel	WebSocketFace

The two main tasks that the protocol factory is designed to do are: (1) creation and management of channels (`createChannel`), and (2) creation and management of multicast faces (`createMulticastFace`). Most protocol factories support both operations. However, some protocols may support only the ability to create channels (e.g., `UnixStreamFactory` and `TcpFactory`) or only the ability to create multicast faces (e.g., `EthernetFactory`).

The `ProtocolFactory` abstract class defines two basic type aliases employed throughout the face system: `FaceCreatedCallback` and `FaceCreationFailedCallback`. These types are just C++ *typedefs* for function pointers used as callbacks in asynchronous operations, but they help to make the code easier to read and understand.

Moreover, `ProtocolFactory` requires subclasses to implement the pure virtual method `createFace(const FaceUri& uri, const FaceCreatedCallback& onCreated, const FaceCreationFailedCallback& onConnectFailed)`, which is a convenience wrapper responsible for automatically selecting a suitable channel (based on the `uri` parameter) and delegating the actual face creation to it. Factories that do not support unicast faces may throw an error when `createFace` is invoked.

### 3.3 Channel Abstraction

The purpose of the channel abstraction is to encapsulate the functionalities needed to accept incoming connections or to start outgoing connections, and to create a face when the connection attempt is successful.

Channels are created by `createChannel(const Endpoint& localEndpoint)` method on protocol factories. This method allocates and returns a channel that can listen and accept incoming connections on the specified local endpoint. An endpoint is a protocol-specific type that encapsulates all the information needed to uniquely identify an endpoint on a machine, for example in the TCP case the endpoint is the pair  $\langle host, port \rangle$ . Multiple channels can be created from the same protocol factory, but each channel must be instantiated on a different local endpoint. Also note that channels make sense only for protocols that support unicast faces; for instance, there is no Ethernet channel, because the Ethernet face is exclusively multicast.

Usually, when a channel is constructed, no resources are reserved; thus, in order to prepare it for accepting connections, `listen` must be called on the channel instance. This method takes care of allocating the necessary operating system resources (sockets, ports, ...), and then starts listening for incoming connections in a non-blocking fashion. This means that `listen` returns immediately and that incoming connection attempts are serviced from Boost.Asio’s event loop.

When channel receives an incoming connection on a specific local endpoint (e.g., an application connected to specific UNIX socket, remote NFD established TCP connection to specific IP/port, etc.), the designated callback of the channel bound to that endpoint is invoked. This callback function will create a Face for the corresponding protocol, which, in turn, executes the `FaceCreatedCallback` that was supplied to the initial `listen` invocation, and will start handling all subsequent communications for the connection. If any errors or timeouts are encountered during this procedure, the connection setup is aborted and `FaceCreationFailedCallback` is executed instead.

For session-less protocols such as UDP the concept of establishing a connection obviously does not apply, therefore for these protocols the `listen` method just puts the channel in an asynchronous wait for incoming packets. As soon as a datagram is received from an unknown peer (i.e., no Face is already handling the remote endpoint), a new face is instantiated and the triggering packet is handed over to it for normal processing. Upon creation, the face binds itself to the local and remote endpoints, thus all subsequent packets from that peer will be dispatched directly to the face by the OS kernel.

The process of establishing a connection to a remote peer entails calling the method `connect`, which starts an asynchronous connection attempt towards the specified endpoint. If the connection is successful, a face is instantiated and the caller-supplied `FaceCreatedCallback` is invoked; otherwise, the error is signaled via `FaceCreationFailedCallback`.



Note that UDP protocol handles slightly differently Faces that are created “on-demand” (as a response of received a datagram from a remote host) and Faces that are created through explicit “connect” operation. When Face is created on-demand, it will be automatically destroyed after the configured time interval (see Section 10.1) if there was nothing has been received or sent through this Face. The explicitly connected Faces will never be automatically destroyed and will exist until it is explicitly destroyed or some network failure happens (e.g., when system’s IP stack goes down). When connect command is issued for already created on-demand Face, the Face is simply converted to “non-demand” state and will no longer be considered for automatic destruction.

### 3.4 Face Abstraction

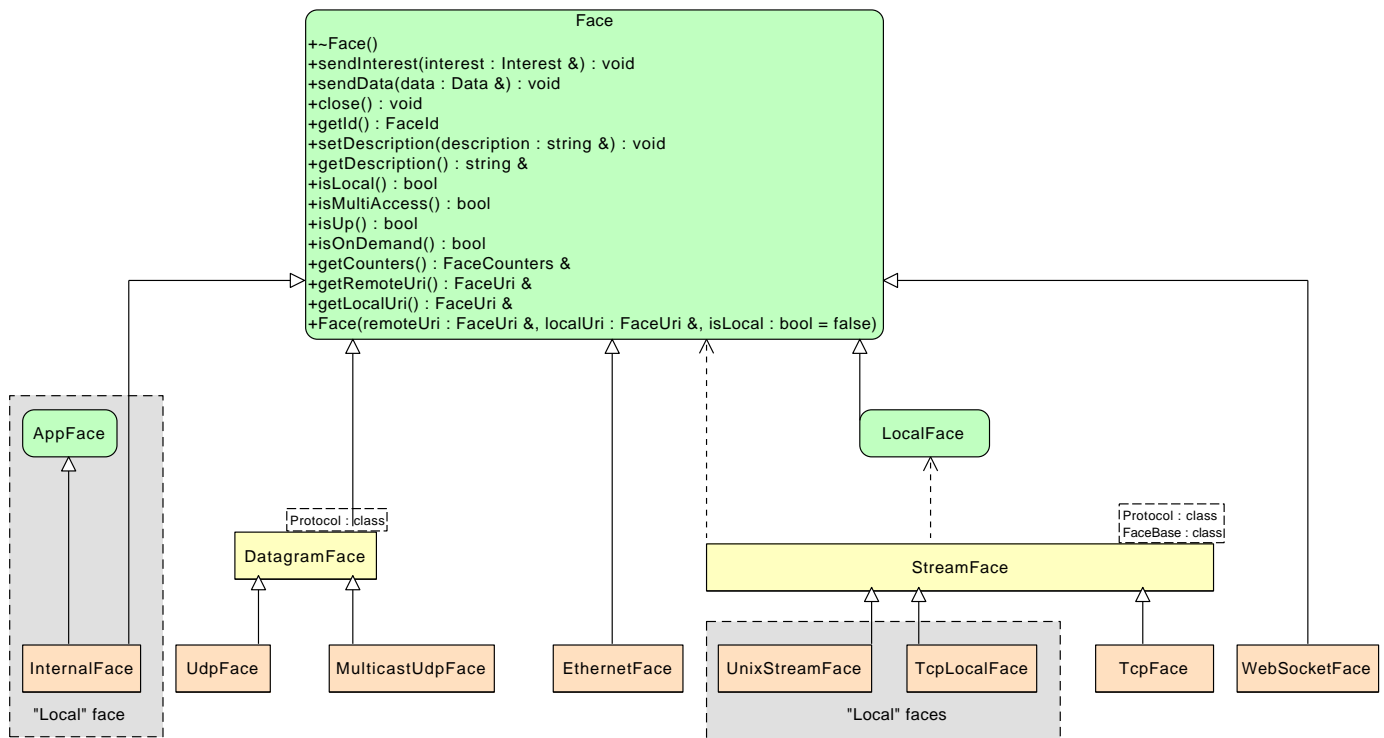


Figure 4: NFD faces

The Face abstraction contains the low-level communication primitives to send and receive Interest and Data packets. All faces derive from the common `Face` abstract base class and the various concrete subclasses can be categorized according to different criteria. For example, we can distinguish between (Figure 4):

- Local and non-local faces: the `Internal`, `TcpLocal`, and `UnixStream` faces are considered *local*, because they can communicate only with other programs running on the same machine (this restriction is enforced by NFD). Local faces are the only ones that can send to and receive from the “/localhost” namespace; they can also support the `LocalControlHeader` [10] that is used by some special applications. All other faces are considered *non-local*.
- Unicast and multicast faces: *unicast* faces, such as `TcpFace`, `UdpFace`, and `WebSocketFace`, can communicate with a single peer, i.e. packets are sent from the local endpoint to exactly one remote endpoint and vice versa. *Multicast* faces, on the other hand, are able to transmit a single packet to multiple remote endpoints, and receive from all of them, forming a set of intercommunicating peers that is usually called multicast group; the `MulticastUdp` and `Ethernet` faces are examples of multicast faces.
- Datagram and stream faces: this distinction is based on the same difference that exists between datagram-oriented and stream-oriented sockets. Therefore the `Udp` and `MulticastUdp` faces are *datagram* faces, while the `Tcp` and `UnixStream` faces are *stream* faces.

### 3.5 WebSocket Face and Encapsulation of NDN Packet

WebSocket is the protocol used by many Web applications running inside Web browsers to connect to remote hosts without using HTTP/HTTPS. It is also used by NDN.JS client library to establish connections between Web browsers and NDN forwarders. WebSocket implements a message-based protocol (on top of TCP for reliability). Therefore it is necessary to clarify how NDN packets should be encapsulated inside WebSocket frames.

The current implementation of `WebSocketFace` in NFD only accepts NDN packets encapsulated in exactly ONE WebSocket frame. Frames containing incomplete NDN packets will be dropped silently and the event will be logged by NFD. For frames containing more than one NDN packets, the packets after the first one will be ignored by NFD. Client applications (and libraries) should not send such packets to NFD. For example, a JavaScript client inside a Web browser should always feed complete NDN packets into the `WebSocket.send()` interface.

Note that this behavior is different from the old WebSocket proxy, which was used to bridge WebSocket clients to the old *ccnd/ndnd* forwarder. The WebSocket proxy simply converts TCP segments received from the forwarder into separate WebSocket frames. Since it is valid for NDN packets to span across multiple TCP segments, it is possible for the proxy to send out WebSocket frames with partial NDN packets. As a result, the NDN.JS client library had to implement the functionality of parsing and de-fragmenting partial NDN packets. This is no longer needed when the client is talking to an NFD forwarder.

### 3.6 Extending NFD Face System

To extend NFD with a new type of Face, developers need to implement the face, channel and protocol factory abstractions. The new classes will typically inherit from the Face, Channel and ProtocolFactory base classes respectively.

The new factory class interacts directly with the face manager and is created upon startup of the daemon. The configuration for the new type of face is written in the *nfd.conf* file (see Section 10.1). Developers usually need to add a new method `processSectionXYZ` into `FaceManager` to read and process the configuration section for the new face type. After parsing the configuration, this method will create new factories and add the factories into an internal hash table. Developers can follow the patterns in other `processSectionXYZ` methods when implementing this function.

The factory class usually provides two interfaces: `createChannel` and `createFace`. `createChannel` will open a local endpoint to wait for connections from other NFD instances. `createFace` will allow NFD to connect to other NFD instances. If the developer decides that the new face protocol will not be used for interconnections between forwarders (e.g., the WebSocket protocol), the body of `createFace` should throw an exception.

The channel class manages all incoming connections. It must provide a `listen` method to start listening on the local socket. In some cases the channel class may be required to perform message dispatching (e.g., demultiplexing among multiple faces on the same local endpoint), depending on how the underlying communication protocol is implemented. It also needs to handle connection close and remove that connection from the internal face table.

The face class provides the basic communication primitives such as `sendInterest`, `sendData`, `close`, and so on. When implementing the receiving function, developers only need to check the length of the incoming message to make sure that the TLV wire encoding in the received packet is complete. For stream-based protocols, this means that in most cases an internal buffer must be maintained in order to collect all incoming segments, until the entire NDN-TLV packet is available and can be parsed. Finally developers simply invoke `decodeAndDispatchInput`, which is inherited from the `Face` base class. That method will check the type field of the incoming packet (i.e., Interest or Data) and pass the packet to NFD's forwarding pipeline (see Section 5).

## 4 Tables

The tables module provides main data structures for NFD.

The Forwarding Information Base (FIB) (Section 4.1) is used to forward Interest packets toward potential source(s) of matching Data. It's almost identical to an IP FIB except it allows for a list of outgoing faces rather than a single one.

The Network Region Table (Section 4.2) contains a list of producer region names for mobility support.

The Content Store (CS) (Section 4.3) is a cache of Data packets. Arriving Data packets are placed in this cache as long as possible, in order to satisfy future Interests that request the same Data.

The Interest Table (PIT) (Section 4.4) keeps track of Interests forwarded upstream toward content source(s), so that Data can be sent downstream to its requester(s). It also contains recently satisfied Interests for loop detection and measurements purposes.

The Dead Nonce List (Section 4.5) supplements the Interest Table for loop detection.

The Strategy Choice Table (Section 4.6) contains the forwarding strategy (Section 6) chosen for each namespace.

The Measurements Table (Section 4.7) is used by forwarding strategies to store measurements information regarding a name prefix.

FIB, PIT, Strategy Choice Table, and Measurements Table have much commonality in their index structure. To improve performance and reduce memory usage, a common index, the Name Tree (Section 4.8), is designed to be shared among these four tables.

### 4.1 Forwarding Information Base (FIB)

The Forwarding Information Base (FIB) is used to forward Interest packets toward potential source(s) of matching Data [11]. For each Interest that needs to be forwarded, a longest prefix match lookup is performed on the FIB, and the list of outgoing faces stored on the found FIB entry is an important reference for forwarding.

The structure, semantics, and algorithms of FIB is outlined in Section 4.1.1. How FIB is used by rest of NFD is described in Section 4.1.2. The implementation of FIB algorithms is discussed in Section 4.8.

#### 4.1.1 Structure and Semantics

Figure 5 shows logical content and relationships between the FIB, FIB entries, and NextHop records.

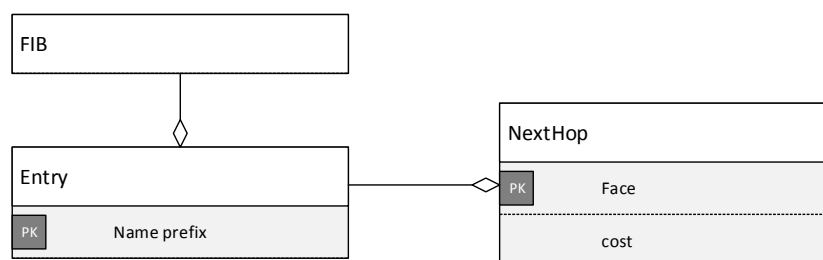


Figure 5: FIB and related entities

#### FIB entry and NextHop record

A FIB entry (`nfd::fib::Entry`) contains a name prefix and a non-empty collection of NextHop records. A FIB entry of a certain prefix means that given an Interest under this prefix, a potential source(s) of matching Data can be reached via the faces given by the NextHop record(s) in this FIB entry.

Each NextHop record (`nfd::fib::NextHop`) contains an outgoing face toward a potential content source and its routing cost. A FIB entry can contain at most one NextHop record toward the same outgoing face. Within a FIB entry, NextHop records are ordered by ascending cost. The routing cost is relative between NextHop records; the absolute value is insignificant.

Unlike the RIB (Section 8.3), there is no inheritance between FIB entries. The NextHop records within a FIB entry are the only “effective” nexthops for this FIB entry.

#### FIB

The FIB (`nfd::Fib`) is a collection of FIB entries, indexed by name prefix. The usual insertion, deletion, exact match operations are supported. FIB entries can be iterated over in a forward iterator, in unspecified order.

**Longest Prefix Match** algorithm (`Fib::findLongestPrefixMatch`) finds a FIB entry that should be used to guide the forwarding of an Interest. It takes a name as input parameter; this name should be the name field in an Interest. The return value is a FIB entry such that its name prefix is (1) a prefix of the parameter, and (2) the longest among those satisfying condition 1; NULL is returned if no FIB entry satisfy condition 1.

`Fib::removeNextHopFromAllEntries` is a convenient method that iterates over all FIB entries, and removes NextHop record of a certain face from every entry. Since a FIB entry must contain at least one FIB entry, if the last NextHop record is removed, the FIB entry is deleted. This is useful when a face is gone.

#### 4.1.2 Usage

The FIB is updated only through using FIB management protocol, which on NFD sides is operated by the FIB manager (Section 7.1.2). Typically, FIB manager takes commands from RIB Daemon (Section 8), which in turn receives static routes configured manually or registered by applications, and dynamic routes from routing protocols. Since most FIB entries ultimately come from dynamic routes, the FIB is expected to contain a small number of entries, if the network has a small number of advertised prefixes.

The FIB is expected to be relatively stable. FIB updates are triggered by RIB updates, which in turn is caused by manual configuration, application startup or shutdown, and routing updates. These events are infrequent in a stable network. However, each RIB update can cause lots of FIB updates, because changes in one RIB entry may affect its descendants due to inheritance.

The longest prefix match algorithm is used by forwarding in *incoming Interest pipeline* (Section 5.2.1). It is called at most once per incoming Interest.

## 4.2 Network Region Table

The Network Region Table (`nfd::NetworkRegionTable`) is used for mobility support (Section 5.2.3). It contains an unordered set of producer region names, taken from NFD configuration file (Section 7.2.5). If any delegation name in the Link object of an Interest is a prefix of any region name in this table, it means the Interest has reached the producer region, and should be forwarded according to its Name rather than delegation name.

## 4.3 Content Store (CS)

The Content Store (CS) is a cache of Data packets. Arriving Data packets are placed in this cache as long as possible, in order to satisfy future Interests that request same Data. As define in NDN architecture [11] and described in more detail in Section 5, the Content Store is searched before the incoming Interest is given to the forwarding strategy for further processing. This way, the cached Data, if available, can be used to satisfy the Interest without actually forwarding the Interest anywhere else.

The following Section 4.3.1 will define the semantics and algorithms of CS, while details about current implementation are discussed in Section 4.3.2.

### 4.3.1 Semantics and Usage

The Content Store (`nfd::cs::Cs`) is a cache of Data packets. Data packets are inserted to the CS (`Cs::insert`) either in the *incoming Data pipeline* (Section 5.3.1) or in the *Data unsolicited pipeline* (Section 5.3.2), after forwarding ensures eligibility of the Data packet to be processed at all (e.g., that the Data packet does not violate the name-based scope [12]).

Before storing the Data packet, the **admission policy** is evaluated. Local applications can give hints to the admission policy in *CachePolicy* field of *LocalControlHeader* [10] attached to the Data packet. Those hints are considered advisory.

After passing the admission policy, the Data packet is stored, along with the time point at which it would become stale and can no longer satisfy an Interest with *MustBeFresh* Selector.

CS is queried (`Cs::find`) with an incoming Interest before it's forwarded in *incoming Interest pipeline* (Section 5.2.1). The lookup API is asynchronous. The search algorithm gives the Data packet that best matches the Interest to *ContentStore hit pipeline* (Section 5.2.4), or informs *ContentStore miss pipeline* (Section 5.2.3) if there's no match.

CS has limited capacity, measured in number of packets. It is controlled via NFD configuration file (Section 7.2.5). Management calls `Cs::setLimit` to update the capacity. CS implementation should ensure number of cached packets does not exceed the capacity.

**Enumeration and Entry Abstraction** The Content Store can be enumerated via ForwardIterators. This feature is not directly used in NFD, but it could be useful in simulation environments.

In order to keep a stable enumeration interface, but still allow the CS implementation to be replaced, the iterator is dereferenced to `nfd::cs::Entry` type, which is an abstraction of CS entry. The public API of this type allows the caller to get the Data packet, whether it's unsolicited, and the time point at which it would become stale. An CS implementation may define its own concrete type, and convert to the abstraction type during enumeration.

### 4.3.2 Implementation

CS performance has a big impact on the overall performance of NFD, because it stores a large number of packets, and virtually every packet accesses the CS. The choice of the underlying data structure for an efficient lookup, insertion, and deletion, and cache replacement algorithm is crucial for maximizing the practical benefits of in-network caching.

The current implementation uses two separate data structures: the **Table** as a Name index, and **CachePolicy** for cache replacement.

**Table for lookup** The **Table** is an ordered container that stores concrete entries (`nfd::cs::EntryImpl`, subclass of the entry abstraction type). This container is sorted by Data Name with implicit digest.<sup>2</sup>

Lookups are performed entirely using the Table. The lookup procedure is optimized to minimize the number of entries visited in the expected case. In the worst case, a lookup would visit all entries that has Interest Name as a prefix.

Although the lookup API is asynchronous, the currently implementation does lookups synchronously.

The **Table** uses `std::set` as the underlying container because its good performance in benchmarks. A previous CS implementation uses a skip list for a similar purpose, but its performance is worse than `std::set`, probably due to algorithm complexity and code quality.

**CachePolicy for cache replacement** A **CachePolicy** is a interface used to keep track of data usage information in CS and has the same capacity limit as CS all the time. CS uses `CS::setPolicy` to specify a Cache Policy. When `CS::setLimit` is invoked to update CS's capacity. `CachePolicy::setLimit` should be invoked as well.

All CS cache policies are implemented as subclasses of `CS::CachePolicy` base class, which provides four public APIs for interaction of the implemented cache policy and CS. The implementation of the APIs are seperated from the public APIs and are declared as pure virtual method.

1. `CachePolicy::afterInsert` invoked by CS after a new entry is inserted and implemeneted in `CachePolicy::doAfterInsert`;
2. `CachePolicy::afterRefresh` invoked by CS after an existing entry is refreshed by same Data and implemented in `CachePolicy::doAfterRefresh`;
3. `CachePolicy::beforeErase` invoked by CS before an entry is erased due to management command and implemented in `CachePolicy::doBeforeErase`;
4. `CachePolicy::beforeUse` invoked by CS before an entry is used to match a lookup and implemented in `CachePolicy::doBeforeUse`.

#### `CachePolicy::doAfterInsert`

When a new entry is inserted into CS, the policy will decide whether to accept it or not. If it is accepted, it should be inserted into a cleanup index. Otherwise, `CachePolicy::evictEntries` will be called to inform CS to do cleanup.

#### `CachePolicy::doAfterRefresh`

When data is refreshed in CS, the policy may witness this refresh and thus update data usange information to make better eviction decisions in the future.

#### `CachePolicy::doBeforeErase`

When an entry is erased due to management command, the policy may not need to keep track of its usage any more. It may need to cleanup information about this entry.

<sup>2</sup>Implicit digest computation is CPU-intensive. The Table has an optimization that avoids implicit digest computation in most cases while still guarantee correct sorting order.

**CachePolicy::doBeforeUse**

When entry is looked up in CS, the policy may witness this usage and thus update data usage information to make better eviction decisions in the future.

**CachePolicy::evictEntries**

Besides the four pure method which are used to separate implementation and APIs of CachePolicy, the `CachePolicy::evictEntries` is used to help make eviction decisions when needed. It is also declared as pure virtual.

When eviction is needed because the policy is exceeding its capacity after an insertion or capacity adjustment, it will emit a `CachePolicy::beforeEvict` signal which CS connects to and after that CS will erase the entry upon signal emission. The entry which will be picked for eviction depends on how the policy is designed.

**Priority FIFO cache policy**

Priority-FIFO is the default CachePolicy. Priority-FIFO do evict upon every insertion, because its performance is more predictable; the alternate, periodical cleanup of many entries, can cause jitter in packet forwarding. Priority-FIFO uses three queues three queues to keep track of data usage in CS:

1. **unsolicited queue** contains entries with unsolicited Data;
2. **stale queue** contains entries with stale Data;
3. **FIFO queue** contains entries with fresh Data.

At any time, an entry belongs to exactly one Queue, and can appear only once in that queue. Priority-FIFO keeps which queue each entry belongs to.

These fields, along with the Table iterator stored in the Queue, establish a bidirectional relation between the Table and the Queues.

Mutation operations must maintain this relation:

- When an entry is inserted, the Entry is emplaced in the Table, and
- When an entry is evicted, its Table iterator erased from the head of its Queue, and the entry is erased from the Table.
- When a fresh entry becomes stale (which is controlled by a timer), its Table iterator is moved from the FIFO queue to the stale queue, and the Queue indicator and iterator on the entry are updated.
- When an unsolicited/stale entry is updated with a solicited Data, its Table iterator is moved from the unsolicited/stale queue to the FIFO queue, and the Queue indicator and iterator on the entry are updated

A **Queue**, despite the name, is not a real first-in-first-out queue, because an entry can move between queues (see mutation operations above). When an entry is moved, its Table iterator is detached from the old Queue, and appended to the new Queue. `std::list` is used as the underlying container; `std::queue` and `std::deque` are unsuitable because they can't efficiently detach a node.

**LRU cache policy**

LRU cache policy implements the Least Recently Used cache replacement algorithm which discards discards the least recently used items first. LRU do evict upon every insertion, because its performance is more predictable; the alternate, periodical cleanup of many entries, can cause jitter in packet forwarding.

LRU uses one queue to keep track of data usage in CS. The Table iterator is stored in Queue. At any time, when an entry is used or refreshed, its Table iterator is relocated to the tail of the queue. Also, when an entry is newly inserted, its Table iterator is pushed at the tail of the queue. When an entry is evicted, its Table iterator erased from the head of its Queue, and the entry is erased from the Table.

The **Queue** uses `boost::multi_index_container` as the underlying container because its good performance in benchmarks. `boost::multi_index_container::sequenced_index` is used for inserting, updating usage and refreshing and `boost::multi_index_container::ordered_unique_index` is used for erasing by `Table::iterator`.



## 4.4 Interest Table (PIT)

The Interest Table (PIT) keeps track of Interests forwarded upstream toward content source(s), so that Data can be sent downstream to its requester(s) [11]. It also contains recently satisfied Interests for loop detection and measurements purposes. This data structure is called “pending Interest table” in NDN literatures; however, NFD’s PIT contains both pending Interests and recently satisfied Interests, so “Interest table” is a more accurate term, but the abbreviation “PIT” is kept.

PIT is a collection of PIT entries, used only by forwarding (Section 5). The structure and semantics of PIT entry, and how it’s used by forwarding are described in Section 4.4.1. The structure and algorithms of PIT, and how it’s used by forwarding are described in Section 4.4.2. The implementation of PIT algorithms is discussed in Section 4.8.

### 4.4.1 PIT Entry

Figure 6 shows the PIT, PIT entries, in-records, out-records, and their relations.

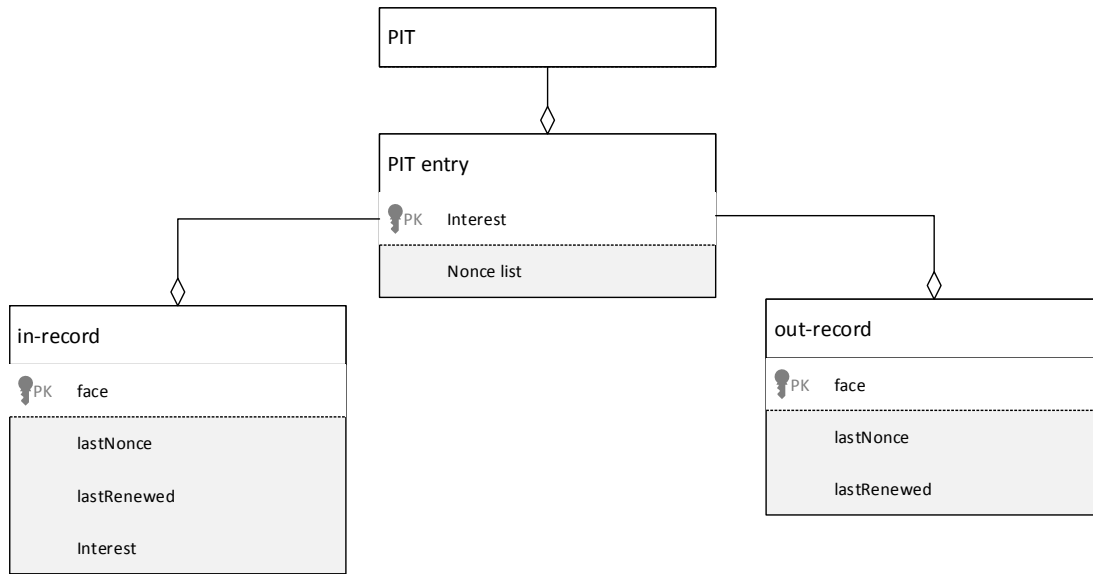


Figure 6: PIT and related entities

#### PIT entry

A PIT entry (`nfd::pit::Entry`) represents either a pending Interest or a recently satisfied Interest. Two Interest packets are *similar* if they have same Name and same Selectors [1]. Multiple similar Interests share the same PIT entry.

Each PIT entry is identified by an Interest. All fields in this Interest, except Name and Selectors, are insignificant.

Each PIT entry contains a collection of in-records, a collection of out-records, and two timers, described below. In addition, forwarding strategy is allowed to store arbitrary information on PIT entry, in-records, and out-records (Section 6.1.3).

#### In record

An **in-record** (`nfd::pit::InRecord`) represents a downstream face for the Interest. A downstream face is a requester for the content: Interest comes from downstream, and Data goes to downstream.

The in-record stores:

- a reference to the face
- the Nonce in the last Interest packet from this face
- the timestamp on which the last Interest packet from this face arrives
- the last Interest packet

An in-record is inserted or updated by *incoming Interest pipeline* (Section 5.2.1). All in-records are deleted by *incoming Data pipeline* (Section 5.3.1) when a pending Interest is satisfied.

An in-record *expires* when InterestLifetime has elapsed after the last Interest packet arrives. A PIT entry expires when all in-records expire. A PIT entry is said to be *pending* if it contains at least one unexpired in-record.

### Out record

An **out-record** (`nfd::pit::OutRecord`) represents an upstream face for the Interest. An upstream face is a potential content source: Interest is forwarded to upstream, and Data comes from upstream.

The out-record stores:

- a reference to the face
- the Nonce in the last Interest packet to this face
- the timestamp on which the last Interest packet to this face is sent

An out-record is inserted or updated by *outgoing Interest pipeline* (Section 5.2.5). An out-record is deleted by *incoming Data pipeline* (Section 5.3.1) when a pending Interest is satisfied by a Data from that face.

An out-record *expires* when InterestLifetime has elapsed after the last Interest packet is sent.

### Timers

There are two timers on a PIT entry, used by forwarding pipelines (Section 5):

- *unsatisfy timer* fires when the PIT entry expires (Section 5.2.1)
- *straggler timer* fires when the PIT entry can be deleted because it has been satisfied or rejected, and is no longer needed for loop detection and measurements purposes (Section 5.3.1)

#### 4.4.2 PIT

The PIT (`nfd::Pit`) is a table containing PIT entries, indexed by `<Name,Selectors>` tuple. The usual insert and delete operations are supported. `Pit::insert` method first looks for a PIT entry for similar Interest, and inserts one only if it does not already exist; there is no separate method for exact match, because forwarding does not need to determine the existence of a PIT entry without inserting it. The PIT is not iterable, because this is not needed by forwarding.

**Data Match** algorithm (`Pit::findAllDataMatches`) finds all Interests that a Data packet can satisfy. It takes a Data packet as input parameter. The return value is a collection of PIT entries that can be satisfied by this Data packet. This algorithm does not delete any PIT entry.

## 4.5 Dead Nonce List

The Dead Nonce List is a data structure that supplements PIT for loop detection purposes.

In August 2014, we found a persistent loop problem when InterestLifetime is short (Bug 1953). Loop detection previously only uses the Nonces stored in PIT entry. If an Interest is unsatisfied within InterestLifetime, the PIT entry is deleted at the end of InterestLifetime. When the network contains a cycle whose delay is longer than InterestLifetime, a looping Interest around this cycle cannot be detected because the PIT entry is gone before the Interest loops back.

A naive solution to this persistent loop problem is to keep the PIT entry for longer duration. However, the memory consumption of doing this is too high, because PIT entry contains many other things than the Nonce. Therefore, Dead Nonce List is introduced to store Nonces "dead" from the PIT.

The Dead Nonce List is a global container in NFD. Each entry in this container stores a tuple of Name and Nonce. The existence of an entry can be queried efficiently. Entries are kept for a duration after which the Interest is unlikely to loop back.

The structure and semantics of Dead Nonce List, and how it's used by forwarding are described in Section 4.5.1. Section 4.5.2 discusses how the capacity of Dead Nonce List is maintained.

### 4.5.1 Structure, Semantics, and Usage

A tuple of Name and Nonce is added to Dead Nonce List (`DeadNonceList::add`) in *incoming Data pipeline* (Section 5.3.1) and *Interest finalize pipeline* (Section 5.2.8) before out-records are deleted.

The Dead Nonce List is queried (`DeadNonceList::has`) in *incoming Interest pipeline* (Section 5.2.1). If an entry with same Name and Nonce exists, the incoming Interest is a looping Interest.

The Dead Nonce List is a probabilistic data structure: each entry is stored as a 64-bit hash of the Name and Nonce. This greatly reduces the memory consumption of the data structure. At the same time, there's a non-zero probability of hash collisions, which inevitably cause false positives: non-looping Interests are mistaken as looping Interests. Those false positives are recoverable: the consumer can retransmit the Interest with a fresh Nonce, which most likely would yield a different hash that doesn't collide with an existing one. We believe the gain from memory savings outweighs the harm of false positives.



### 4.5.2 Capacity Maintenance

Entries are kept in Dead Nonce List for a configurable lifetime. The entry lifetime is a trade-off between effectiveness of loop detection, memory consumption of the container, and probability of false positives. Longer entry lifetime improves the effectiveness of loop detection, because a looping Interest can be detected only if it loops back before the entry is removed, and longer lifetime allows detecting looping Interests in network cycles with longer delay. Longer entry lifetime causes more entries to be stored, and therefore increases the memory consumption of the container; having more entries also means higher probability of hash collisions and thus false positives. The default entry lifetime is set to 6 seconds.

A naive approach of entry lifetime enforcement is to keep a timestamp in each entry. This approach consumes too much memory. Given that the Dead Nonce List is a probabilistic data structure, entry lifetime doesn't need to be precise. Thus, we index the container as a first-in-first-out queue, and approximate entry lifetime to the configured lifetime by adjusting the capacity of the container.

It's infeasible to statically configure the capacity of the container, because the frequency of adding entries is correlated to Interest arrival rate, which cannot be accurately estimated by an operator. Therefore, we use the following algorithm to dynamically adjust the capacity for *expected* entry lifetime  $L$ :

- At interval  $M$ , we add a special entry called a *mark* to the container. The mark doesn't have a distinct type: it's an entry with a specific value, with the assumption that the hash function is non-invertible so that the probability of colliding with a hash value computed from Name and Nonce is low.
- At interval  $M$ , we count the number of marks in the container, and remember the count. The order between adding a mark and counting marks doesn't matter, but this shall be consistent.
- At interval  $A$ , we look at recent counts. When the capacity of the container is optimal, there should be  $L/M$  marks in the container at all times. If all recent counts are above  $L/M$ , the capacity is adjusted down. If all recent counts are below  $L/M$ , the capacity is adjusted up.

In addition, there is a hard upper bound and lower bound of the capacity, to avoid memory overflow and to ensure correct operations. When the capacity is adjusted down, to bound algorithm execution time, excess entries are not evicted all at once, but are evicted in batches during future adding operations.

## 4.6 Strategy Choice Table

The Strategy Choice Table contains the forwarding strategy (Section 6) chosen for each namespace. This table is a new addition to the NDN architecture. Theoretically, forwarding strategy is a program that is supposed to be stored in FIB entries [11]. In practice, we find that it is more convenient to save the forwarding strategy in a separate table, instead of storing it with FIB entry, for the following reasons:

- FIB entries come from RIB entries, which are managed by NFD RIB Daemon (Section 8). Storing the strategy in FIB entries would require the RIB Daemon to create/update/remove strategies when it manipulates the FIB. This increases the RIB Daemon's complexity.
- FIB entry is automatically deleted when the last NextHop record is removed, including when the last upstream face fails. However, we don't want to lose the configured strategy.
- The granularity of strategy configuration is different from the granularity of RIB entry or FIB entry. Having both in the same table makes inheritance handling more complex.

The structure, semantics, and algorithms of Strategy Choice Table is outlined in Section 4.6.1. How Strategy Choice Table is used by rest of NFD is described in Section 4.6.2. The implementation of Strategy Choice Table algorithms is discussed in Section 4.8.

### 4.6.1 Structure and Semantics

#### Strategy Choice entry

A Strategy Choice entry (`nfd::strategy_choice::Entry`) contains a Name prefix, and the Name of a forwarding strategy chosen for this namespace. Currently, there is no parameters.

At runtime, a reference to the instantiation of the strategy program is also linked from the Strategy Choice entry.

## Strategy Choice Table

The Strategy Choice Table (`nfd::StrategyChoice`) is a collection of Strategy Choice entries—associations of namespaces with specific strategies. There could be only one strategy set per namespace, but sub-namespaces can have their own choices for the strategy.

Currently, the Strategy Choice Table also maintains a collection of the available (“installed”) strategies and is consulted by the StrategyChoice manager (see Section 7.1.3) whenever a control command is received. Therefore, in order for any new custom strategy to be known to NFD and be used in the namespace-strategy association, it should be “installed” using `StrategyChoice::install` method. Note that each installed strategy should have its own unique name, otherwise a runtime error will be generated.

In order to guarantee that every namespace has a strategy, NFD always insert the root entry for `/` namespace to the Strategy Choice Table during initialization. The strategy chosen for this entry, called the *default strategy*, is defined by the hard-coded `makeDefaultStrategy` free function in `daemon/fw/available-strategies.cpp`. The default strategy can be replaced, but the root entry in Strategy Choice Table can never be deleted.

The insertion operation (`StrategyChoice::insert`) inserts a Strategy Choice entry, or updates the chosen strategy on an existing entry. The new strategy must have been installed.

The deletion operation (`StrategyChoice::erase`) deletes a Strategy Choice entry. The namespace covered by the deletes would inherit the strategy defined on the parent namespace. It is disallowed to delete the root entry.

The usual exact match operation is supported. Strategy Choice entries can be iterated over in a forward iterator, in unspecified order.

**Find Effective Strategy** algorithm (`StrategyChoice::findEffectiveStrategy`) finds a strategy that should be used to forward an Interest. The effective strategy for the namespace can be defined as follows:

- If the namespace is explicitly associated with the strategy, then this is the effective strategy
- Otherwise, the first parent namespace for which strategy was explicitly set defines the effective strategy.

The find effective strategy algorithm takes a Name, a PIT entry, or a measurements entry as input parameter.<sup>3</sup> The return value of the algorithm is a forwarding strategy that is found by longest prefix match using the supplied name. This return value is always a valid entry, because every namespace must have a strategy.

### 4.6.2 Usage

The Strategy Choice Table is updated only through management protocol. Strategy Choice manager (Section 7.1.3) is directly responsible for updating the Strategy Choice Table.

The Strategy Choice is expected to be stable, as strategies are expected to be manually chosen by the local NFD operator (either user for personal computers or system administrators for the network routers).

The effective strategy search algorithm is used by forwarding in *incoming Interest pipeline* (Section 5.2.1), *Interest unsatisfied pipeline* (Section 5.2.7), and *incoming Data pipeline* (Section 5.3.1). It is called at most twice per incoming packet.

## 4.7 Measurements Table

The Measurements Table is used by forwarding strategies to store measurements information regarding a name prefix. Strategy can store arbitrary information in PIT and in Measurements (Section 6.1.3). The Measurements Table is indexed by namespace, so it’s suitable to store information that is associated with a namespace, but not specific to an Interest.

The structure and algorithms of Measurements Table is outlined in Section 4.7.1. How Measurements Table is used by rest of NFD is described in Section 4.7.2. The implementation of Measurements Table algorithms is discussed in Section 4.8.

### 4.7.1 Structure

#### Measurements entry

A Measurements entry (`nfd::measurements::Entry`) contains a Name, and APIs for strategy to store and retrieve arbitrary information (`nfd::StrategyInfoHost`, Section 6.1.3). It’s possible to add some standard metrics that can be shared among strategies, such as round trip time, delay, jitter, etc. However, we feel that every strategy has its unique needs, and adding those standard metrics would become unnecessary overhead if the effective strategy is not making use of them. Therefore, currently the Measurements entry does not contain standard metrics.

<sup>3</sup>Since the strategy choices can change during the runtime, the last two parameters are necessary to ensure correctness of strategy-specific information stored in PIT and measurement. For more detail, see Section 6.1.3.

## Measurements Table

The Measurements Table (`nfd::Measurements`) is a collection of Measurements entries.

`Measurements::get` method finds or inserts a Measurements entry. The parameter is a Name, a FIB entry, or a PIT entry. Because of how Measurements table is implemented, it's more efficient to pass in a FIB entry or a PIT entry, than to use a Name. `Measurements::getParent` method finds or inserts a Measurements entry of the parent namespace.

Unlike other tables, there is no delete operation. Instead, each entry has limited lifetime, and is automatically deleted when its lifetime is over. Strategy must call `Measurements::extendLifetime` to request extending the lifetime of an entry.

Exact match and longest prefix match lookups are supported for retrieving existing entries.

### 4.7.2 Usage

Measurements Table is solely used by forwarding strategy. How many entries are in the Measurements Table and how often they are accessed are determined by forwarding strategies. A well-written forwarding strategy stores no more than  $O(\log(N))$  entries, and performs no more than  $O(N)$  lookups, where  $N$  is the number of incoming packets plus the number of outgoing packets.

### Measurements Accessor

Recall that NFD has per-namespace strategy choice (Section 4.6), each forwarding strategy is allowed to access the portion of Measurements Table that are under the namespaces managed by that strategy. This restriction is enforced by a Measurements Accessor.

A Measurements Accessor (`nfd::MeasurementsAccessor`) is a proxy for a strategy to access the Measurements Table. Its APIs are similar to the Measurements Table. Before returning any Measurements entry, the accessor looks up the Strategy Choice Table (Section 4.6) to confirm whether the requesting strategy owns the Measurements entry. If an access violation is detected, null is returned instead of the entry.

## 4.8 NameTree

The NameTree is a common index structure for FIB (Section 4.1), PIT (Section 4.4), Strategy Choice Table (Section 4.6), and Measurements Table (Section 4.7). It is feasible to use a common index, because there are much commonality in the index of these four tables: FIB, StrategyChoice, and Measurements are all indexed by Name, and PIT is indexed by Name and Selectors [1]. It is beneficial to use a common index, because lookups on these four tables are often related (eg. FIB longest prefix match is invoked in *incoming Interest pipeline* (Section 5.2.1) after inserting a PIT entry), and using a common index can reduce the number of index lookups during packet processing; the amount of memory used by the index(es) is also reduced.

NameTree data structure is introduced in Section 4.8.1. NameTree operations and algorithms are described in Section 4.8.2. Section 4.8.3 describes how NameTree can help reducing number of index lookups by adding shortcuts between tables.

### 4.8.1 Structure

The conceptual NameTree data structure is shown in Figure 7. The NameTree is a collection of NameTree entries, indexed by Name. FIB, PIT, Strategy Choice, and Measurements entries are attached onto NameTree entry.

#### NameTree entry

A NameTree entry (`nfd::name_tree::Entry`) contains:

- the Name prefix
- a pointer to the parent entry
- a list of pointers to child entries
- zero or one FIB entry
- zero or more PIT entries
- zero or one Strategy Choice entry
- zero or one Measurements entry

NameTree entries form a tree structure via parent and children pointers.

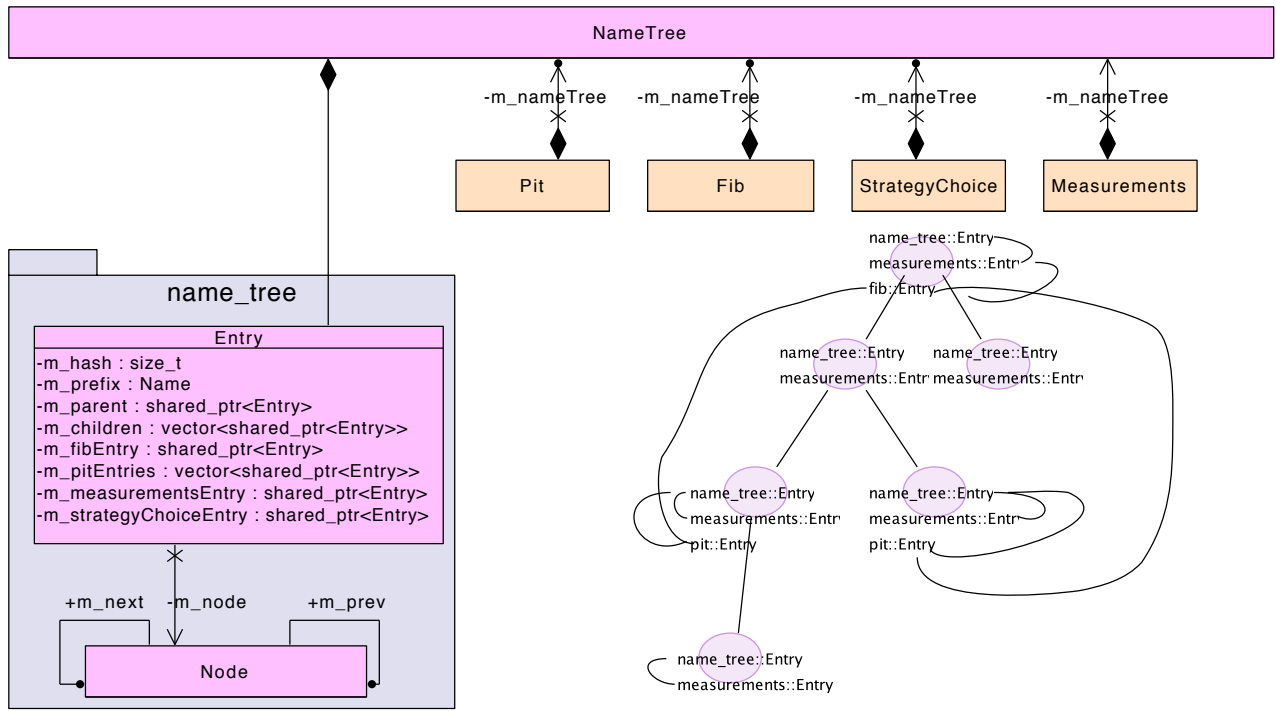


Figure 7: NameTree overview

### NameTree hash table

In addition to the tree structure, the NameTree also has a hash table to enable faster lookups.

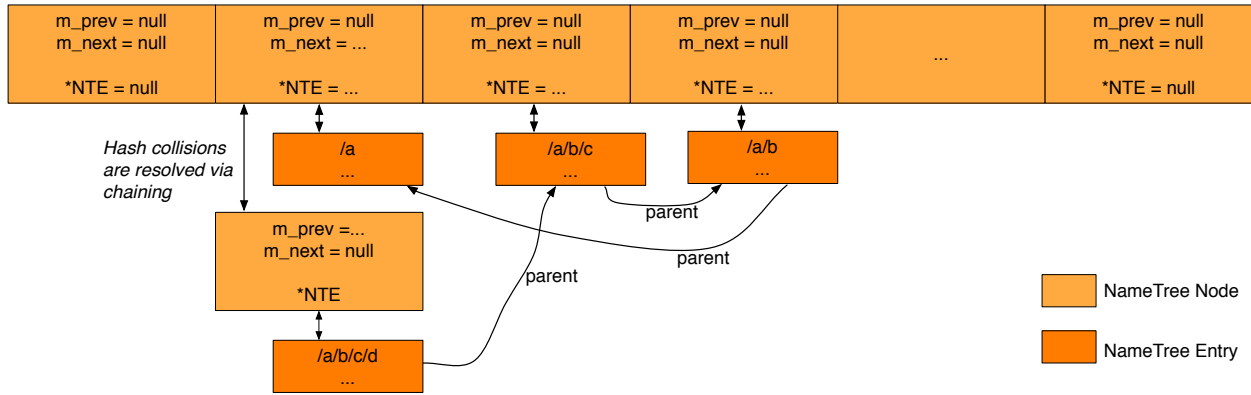


Figure 8: NameTree hash table data structure

We decide to implement the hash table from scratch, rather than using an existing library, so that we can have better control for performance tuning. The hash table data structure is shown in Figure 8.

Hash values are computed using CityHash [13]; this hash function is chosen because it is fast. For a given Name prefix, hash is computed over the TLV representation of the Name, and the hash value is mapped to one of the buckets. Hash collisions are resolved via chaining: if multiple Names are mapped to the same bucket, all these entries are chained in that bucket through a singly linked list.

As the number of stored NameTree entries changes, the hash table is automatically resized. During a resize operation, the new number of buckets is computed; this number is a trade-off between wasted memory of empty buckets and time overhead of chaining. Every NameTree entry is then rehashed and moved to a bucket in the new hashtable.

To reduce the overhead of resize operation, the hash value of a Name is stored in the NameTree entry. We also introduce a **NameTree Node** type. A Node is stored in the bucket, and contains a pointer to an entry, and a pointer to the next Node in the chain. The resize operation only needs to move Nodes (which are smaller than entries), and do not need to change

This revision is OUTDATED and preserved for historical purposes only. Check for the most recent revision at <https://named-data.net/publications/techreports/>

entries.

In Figure 8, name prefixes /a, /a/b, /a/b/c, /a/b/c/d are stored. The parent pointers shown on the figure show the relationship between these four name prefixes. As shown in the figure, there is a hash collision between /a and /a/b/c/d, and the hash collision is resolved via chaining.

## 4.8.2 Operations and Algorithms

### Insertion and Deletion operations

The **lookup/insertion** operation (`NameTree::lookup`) finds or inserts an entry for a given Name. To maintain the tree structure, ancestor entries are inserted if necessary. This operation is called when a FIB/PIT/StrategyChoice/Measurements entry is being inserted.

The **conditional deletion** operation (`NameTree::eraseEntryIfEmpty`) deletes an entry if no FIB/PIT/StrategyChoice/Measurements entry is stored on it, and it has no children; ancestors of the deleted entry are also deleted if they meet the same requirements. This operation is called when a FIB/PIT/StrategyChoice/Measurements entry is being deleted.

### Matching algorithms

The **exact match** algorithm (`NameTree::findExactMatch`) finds the entry with a specified Name, or returns null if such entry does not exist.

The **longest prefix match** algorithm (`NameTree::findLongestPrefixMatch`) finds the entry of longest prefix match of a specified Name, filtered by an optional *EntrySelector*. An *EntrySelector* is a predicate that decides whether an entry can be accepted (returned). This algorithm is implemented as: start from looking up the full Name in the hash table; if no NameTree entry exists or it's rejected by the predicate, remove the last Name component and lookup again, until an acceptable NameTree entry is found. This algorithm is called by FIB longest prefix match algorithm (Section 4.1.1), with a predicate that accepts a NameTree entry only if it contains a FIB entry. This algorithm is called by StrategyChoice find effective strategy algorithm (Section 4.6.1), with a predicate that accepts a NameTree entry only if it contains a StrategyChoice entry.

The **all match** algorithm (`NameTree::findAllMatches`) enumerates all entries that are prefixes of a given Name, filtered by an optional *EntrySelector*. This algorithm is implemented as: perform a longest prefix match first; remove the last Name component, until reaching the root entry. This algorithm is called by PIT data match algorithm (Section 4.4.2).

### Enumeration algorithms

The **full enumeration** algorithm (`NameTree::fullEnumerate`) enumerates all entries, filtered by an optional *EntrySelector*. This algorithm is used by FIB enumeration and Strategy Choice enumeration.

The **partial enumeration** algorithm (`NameTree::partialEnumerate`) enumerates all entries under a specified Name prefix, filtered by an optional *EntrySubTreeSelector*. An *EntrySelector* is a double-predicate that decides whether an entry can be accepted, and whether its children shall be visited. This algorithm is used during runtime strategy change (Section 6.1.3) to clear StrategyInfo items under a namespace changing ownership.

## 4.8.3 Shortcuts

One benefit of the NameTree is that it can reduce the number of index lookups during packet forwarding. To achieve this benefit, one method is to let forwarding pipelines perform a NameTree lookup explicitly, and use fields of the NameTree entry. However, this is not ideal because NameTree is introduced to improve the performance of four tables, and it should change the procedure of forwarding pipelines.

To reduce the number of index lookups, but still hide NameTree away from forwarding pipelines, we add shortcuts between tables. Each FIB/PIT/StrategyChoice/Measurements entry contains a pointer to the corresponding NameTree entry; the NameTree entry contains pointers to FIB/PIT/StrategyChoice/Measurements entries and the parent NameTree entry. Therefore, for example, given a PIT entry, one can retrieve the corresponding NameTree entry in constant time by following the pointer, and then retrieve or attach a Measurements entry via the NameTree entry, or find longest prefix match FIB entry by following pointers to parents.

NameTree entry is still exposed to forwarding if we take this approach. To also hide NameTree entry away, we introduce new overloads to table algorithms that take a relevant table entry in place of a Name. These overloads include:

- `Fib::findLongestPrefixMatch` can accept PIT entry or Measurements entry in place of a Name
- `StrategyChoice::findEffectiveStrategy` can accept PIT entry or Measurements entry in place of a Name
- `Measurements::get` can accept FIB entry or PIT entry in place of a Name

An overload that takes a table entry is generally more efficient than the overload taking a Name. Forwarding can take advantage of reduced index lookups by using those overloads, but does not need to deal with NameTree entry directly.

To support these overloads, NameTree provides `NameTree::get` method, which returns the NameTree entry linked from a FIB/PIT/StrategyChoice/Measurements entry. This method allows one table to retrieve the corresponding NameTree from an entry of another table, without knowing the internal structure of that entry. It also permits a table to depart from NameTree in the future without breaking other code: suppose someday PIT is no longer based on NameTree, `NameTree::get` could perform a lookup using Interest Name in the PIT entry; `Fib::findLongestPrefixMatch` can still accept PIT entries, although it's not more efficient than using a Name.



## 5 Forwarding

The packet processing in NFD consists of **forwarding pipelines** described in this section and **forwarding strategies** described in Section 6. A **forwarding pipeline** (or just pipeline) is a series of steps that operates on a packet or a PIT entry, which is triggered by the specific event: reception of the Interest, detecting that the received Interest was looped, when an Interest is ready to be forwarded out of the Face, etc. A **forwarding strategy** (or just strategy) is a decision maker about Interest forwarding, which is attached at the end or beginning of the pipelines. In other words, the strategy makes decisions whether, when, and where to forward an Interest, while the pipelines supply the strategy the Interests and supporting information to make these decisions.

Figure 9 shows the overall workflow of forwarding pipelines and strategy, where blue boxes represent pipelines and white boxes represent decision points of the strategy.

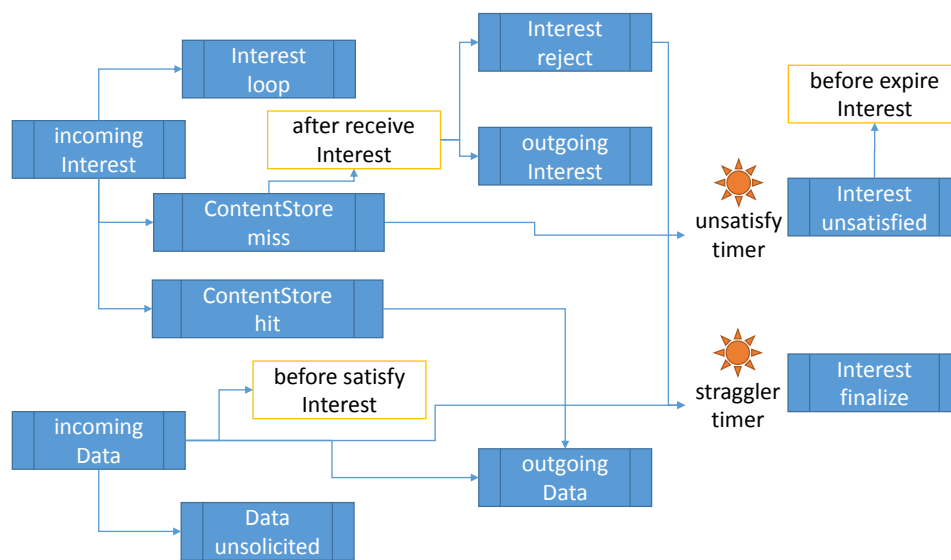


Figure 9: Pipelines and strategy: overall workflow

### 5.1 Forwarding Pipelines

Pipelines operate on network layer packets (Interest or Data) and each packet is passed from one pipeline to another (in some cases through strategy decision points) until all processing is finished. Processing within pipelines uses CS, PIT, dead Nonce list, FIB, network region table, and StrategyChoice tables, however for the last three pipelines have only read-only access (they are managed by the corresponding managers and are not directly affected by the data plane traffic).

FaceTable keeps track all active Faces in NFD. It's the entrypoint from which an incoming network layer packet is given to forwarding pipelines for processing. Pipelines are also allowed to send packets through faces.

The processing of Interest and Data packets in NDN is quite different (the one serves as a request, while other satisfies pending requests), we separate forwarding pipelines into **Interest processing path** and **Data processing path**, described in the following sections.

### 5.2 Interest Processing Path

NFD separates Interest processing into the following pipelines:

- incoming Interest: processing of incoming Interests
- Interest loop: processing incoming looped Interests
- ContentStore miss: processing of incoming Interests that cannot be satisfied by cached Data
- ContentStore hit: processing of incoming Interests that can be satisfied by cached Data
- outgoing Interest: preparation and sending out Interests

- Interest reject: processing PIT entries that are rejected by the strategy
- Interest unsatisfied: processing PIT entries that are unsatisfied before all downstreams timeout
- Interest finalize: deleting PIT entry

### 5.2.1 Incoming Interest Pipeline

The incoming Interest pipeline is implemented in `Forwarder::onIncomingInterest` method and is entered from `Forwarder::onInterest` method, which is triggered by `Face::onReceiveInterest` signal (see [14] for more detail about signals). The input parameters to the incoming interest pipeline include the newly received Interest packet and reference to the Face on which this Interest packet was received.

This pipeline includes the following steps, summarized in Figure 10:

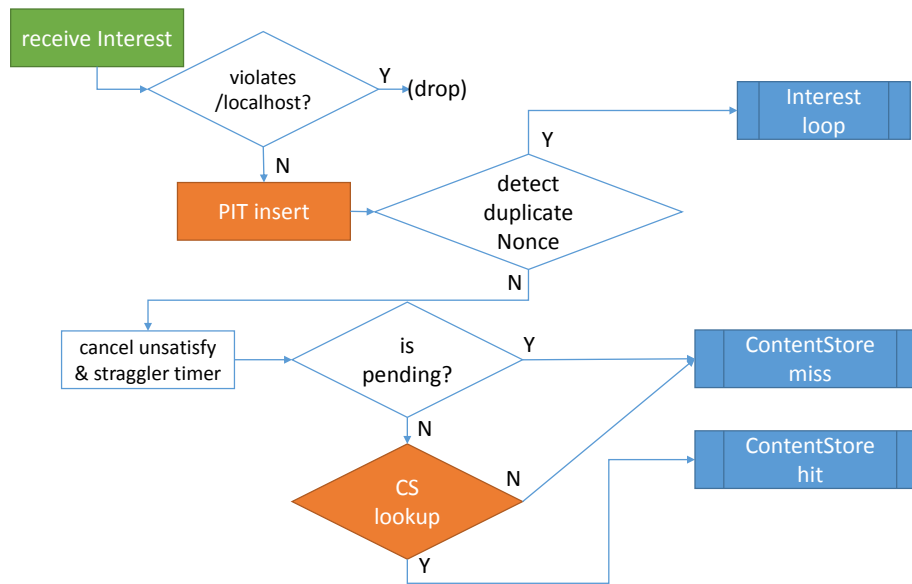


Figure 10: Incoming Interest pipeline

- The first step after entering the incoming Interest pipeline is check for `/localhost` scope [12] violation. In particular, an Interest from a non-local Face is not allowed to have a name that starts with `/localhost` prefix, as it is reserved for localhost communication. If violation is detected, such Interest is immediately dropped and no further processing on the dropped Interest is performed. This check guards against malicious senders; a compliant forwarder will never send a `/localhost` Interest to a non-local Face. Note that `/localhost` scope is not checked here, because its scope rules do not restrict incoming Interests.
- The next step is looking up existing or creating a new PIT entry, using name and selectors specified in the Interest packet. As of this moment, PIT entry becomes a processing subject of the incoming Interest and following pipelines. Note that NFD creates PIT entry before performing ContentStore lookup. The main reason for this decision is to reduce lookup overhead: ContentStore is most likely be significantly larger than PIT and can incur significant overhead, since, as described below, ContentStore lookup can be skipped in certain cases.
- Before the incoming Interest is processed any further, its Nonce is checked against the Nonces in the PIT entry and the Dead Nonce List (Section 4.5). If a match is found, the incoming Interest is considered a duplicate due to either loop or multi-path arrival, and is given to *Interest loop pipeline* for further processing (Section 5.2.2). If a match is not found, processing continues.
- Next, the *unsatisfy timer* (described below) and *straggler timer* (Section 5.2.6) on the PIT entry are cancelled, because a new valid Interest is arriving for the PIT entry, so that the lifetime of the PIT entry needs to be extended. The timers could get reset later in the Interest processing path, e.g., if ContentStore will be able to satisfy the Interest.
- The pipeline then tests whether the Interest is pending, i.e., the PIT entry has already another in-record from the same or other incoming Face. Recall that NFD's PIT entry can represent not only pending Interest but also recently satisfied



Interest (Section 4.4.1), this test is equivalent to “having a PIT entry” in CCN Node Model [11], whose PIT contains only pending Interests.

- If the Interest is not pending, the Interest is matched against the ContentStore (`Cs::find`, Section 4.3.1). Otherwise, CS lookup is unnecessary because a pending Interest implies that a previous CS returns no match. Depending on whether there’s a match in CS, Interest processing continues either in *ContentStore miss pipeline* (Section 5.2.3) or in *ContentStore hit pipeline* (Section 5.2.4).

### 5.2.2 Interest Loop Pipeline

This pipeline is implemented in `Forwarder::onInterestLoop` method and is entered from *incoming Interest pipeline* (Section 5.2.1) when an Interest loop is detected. The input parameters to this pipeline include an Interest packet, its incoming Face, and the PIT entry.

In the current implementation, this pipeline simply drops the Interest. In the future, this pipeline may generate some form of explicit notifications (e.g., Interest NACKs [15]) to the downstream to inform about the detected loop.

### 5.2.3 ContentStore Miss Pipeline

This pipeline is implemented in `Forwarder::onContentStoreMiss` method and is entered after *incoming Interest pipeline* (Section 5.2.1) performs a ContentStore lookup (Section 4.3.1) and there’s no match. The input parameters to this pipeline include an Interest packet, its incoming Face, and the PIT entry.

When this pipeline is entered, the Interest is valid and cannot be satisfied by cached Data, so it needs to be forwarded somewhere. This pipeline takes the following steps, summarized in Figure 11:

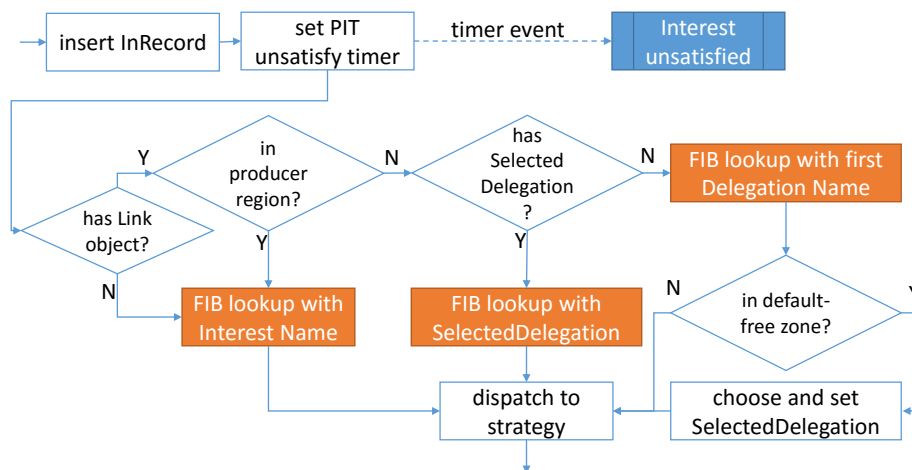


Figure 11: ContentStore miss pipeline

- An in-record for the Interest and its incoming Face is created in the PIT entry or simply gets refreshed if it was already there (e.g., the Interest is being re-expressed by the consumer). The expiration time of the in-record is directly controlled by the `InterestLifetime` field in the Interest packet. If `InterestLifetime` is omitted, the default value of 4 seconds is used.

Note that forwarding defers to the strategy the decision on whether to forward again a similar Interest (same Name and Selectors, but different Nonce). Most strategies will suppress forwarding of Interests if the previous forwarding is within a short period of time (see Section 6.1.1 for more detail).

- After PIT entry is updated, the pipeline sets the *unsatisfy timer* for the PIT entry. This timer expires when all InRecords in the PIT entry expire. Expiration of the unsatisfy timer triggers entering the *Interest unsatisfied pipeline* (Section 5.2.7).
- If the Interest does not carry a Link object and hence does not require mobility processing, FIB is looked up using Interest Name (Section 4.1.1, Longest Prefix Match algorithm). This needs to be in the pipeline, because strategy does not have direct access to the FIB. Note that FIB guarantees that Longest Prefix Match would return a valid FIB entry. However, a FIB entry may contain empty set of NextHop records, which could effectively result (but, strictly speaking, is not required to happen) in rejecting of the Interest by the strategy.

- If the Interest carries a Link object, it is processed for mobility support.

First, the pipeline determines whether the Interest has reached the producer region, by checking if any delegation name in the Link object is a prefix of any region name from the *network region table* (Section 4.2). If so, FIB lookup is performed using Interest Name, as if the Link object does not exist.

Second, the pipeline inspects whether the Interest contains the SelectedDelegation field, which indicates that a downstream forwarder has chosen which delegation to use. If so, it implies that the Interest is already in default-free zone, FIB lookup is performed using the selected delegation name.

Third, the pipeline determines whether the Interest is in the consumer region or has reached the default-free zone, by looking up the first delegation Name in FIB. If this lookup turns out the default route (i.e., the root FIB entry `ndn:/` with a non-empty nexthop list), it means the Interest is still in consumer region, and this lookup result is passed down to next step. Otherwise, the Interest has reached the default-free zone.

Last, the pipeline selects a delegation for an Interest that has reached the default free zone, by looking up every delegation name in the FIB, and choose the lowest-cost delegation that matches a non-empty FIB entry. The chosen delegation is written into the SelectedDelegation field in the Interest packet, so that upstream forwarders can follow this selection without doing multiple FIB lookups.

- The final step of this pipeline is determining which strategy is responsible to process Interests (i.e., Interest's name is checked against StrategyChoice table using Find Effective Strategy algorithm, see Section 4.6.1). The selected strategy is then triggered for the *after receive Interest* action with the PIT entry, incoming Interest packet, and FIB entry (Section 6.1.1).

#### 5.2.4 ContentStore Hit Pipeline

This pipeline is implemented in `Forwarder::onContentStoreMiss` method and is entered after *incoming Interest pipeline* (Section 5.2.1) performs a ContentStore lookup (Section 4.3.1) and there's a match. The input parameters to this pipeline include an Interest packet, its incoming Face, the PIT entry, and the matched Data packet.

This pipeline sets the *straggler timer* (Section 5.2.6) on the Interest because it's being satisfied, and then pass the matching Data to *outgoing Data pipeline* (Section 5.3.3). The Interest processing is completed.

#### 5.2.5 Outgoing Interest Pipeline

The outgoing Interest pipeline is implemented in `Forwarder::onOutgoingInterest` method and is entered from `Strategy::sendInterest` method which handles *send Interest action* for strategy (Section 6.1.2). The input parameters to this pipeline include a PIT entry, an outgoing Face, and a `wantNewNonce` flag. Note that the Interest packet is not a parameter when entering the pipeline. The pipeline steps either use PIT entry directly to perform checks, or obtain reference to an Interest stored inside the PIT entry.

This pipeline includes the following steps, summarized in Figure 12:

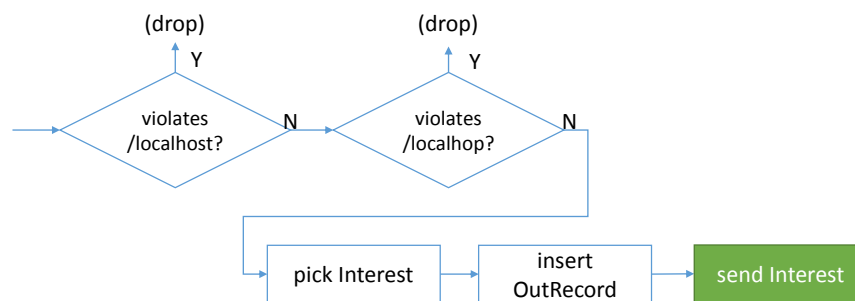


Figure 12: Outgoing Interest pipeline

- The initial step is to check for potential violations of `/localhost` and `/localhop` scopes [12]:
  - Interest packets that start with `/localhost` prefix cannot be send out to a non-local Faces
  - Interest packets that start with `/localhop` prefix can be send out to a non-local Faces only if PIT entry has at least one in-record that represents a local Face.

This check guards against a careless strategy and guarantees properties of `/localhost` and `/localhop` name-based scope control in NFD.

- On the next step an Interest packet is selected among the recorded Interests inside in-records in the PIT entry. This is necessary because Interests in different in-records can have different guiders [1] (e.g. `InterestLifetime`). The current implementation always selects the last incoming Interest. However, this simple selection criteria can change in the future releases after we understand better the effects of guiders.
- If the strategy indicates a new nonce is wanted (the `wantNewNonce` flag), the Interest is copied, and a random nonce is set onto the copy.

This flag is necessary since the strategy may want to retransmit the pending Interest. During the retransmission, the nonce must be changed, otherwise the upstream nodes may falsely detect Interest loops and prevent the retransmitted Interest from being processed.

- The next step is to create in the PIT entry an out-record for the Interest and insert entry for the specified outgoing Face. If an out-record and/or an entry for the outgoing Face already exist, it will get refreshed by the value of `InterestLifetime` in the selected Interest packet (if `InterestLifetime` in Interest packet is omitted, value of 4 seconds is used).
- Finally, the Interest is forwarded via the Face.

### 5.2.6 Interest Reject Pipeline

This pipeline is implemented in `Forwarder::onInterestReject` method and is entered from `Strategy::rejectPendingInterest` method which handles *reject pending Interest action* for strategy (Section 6.1.2). The input parameters to this pipeline include a PIT entry.

The pipeline cancels the *unsatisfy timer* on the PIT entry (set by *incoming Interest pipeline*), and then sets the *straggler timer*. After the straggler timer expires, Interest finalize pipeline (Section 5.2.8) is entered.

The purpose of the straggler timer is to keep PIT entry alive for a short period of time in order to facilitate duplicate Interest detection and to collect data plane measurements. For duplicate Interest detection this is necessary, since NFD uses the Nonces stored inside PIT entry to remember recently seen Interest Nonces. For data plane measurement it is desirable to obtain as much data points as possible, i.e., if several incoming Data packets can satisfy the pending Interest, all of these Data packets should be used to measure performance of the data plane. If PIT entry is deleted right away, NFD may fail to properly detect Interest loop and valuable measurements can be lost.

We chose 100 ms as a static value for the straggler timer, as we believe it gives good tradeoff between the functionality and memory overhead: for loop detection purposes, this time is enough for most packets to go around a cycle; for measurement purposes, a working path that is more than 100 ms slower than the best path is usually not useful. If necessary, this value can be adjusted in `daemon/fw/forwarder.cpp` file.

### 5.2.7 Interest Unsatisfied Pipeline

This pipeline is implemented in `Forwarder::onInterestUnsatisfied` method and is entered from the *unsatisfy timer* (Section 5.2.1) when `InterestLifetime` expires for all downstreams. The input parameters to this pipeline include a PIT entry.

The processing steps in the Interest unsatisfied pipeline include:

- Determining the strategy that is responsible for the PIT entry using Find Effective Strategy algorithm on the `StrategyChoice` table (see Section 4.6.1).
- Invoking *before expire Interest* action of the effective strategy with the PIT entry as the input parameter (Section 6.1.1).
- Entering Interest finalize pipeline (Section 5.2.8).

Note that at this stage there is no need to keep PIT entry alive for any time longer, as it is the case in the Interest reject pipeline (Section 5.2.6). Expiration of the unsatisfy timer implies that PIT entry was already alive for substantial period of time and all Interest loops have been already prevented and no matching Data packet has been received.

### 5.2.8 Interest Finalize Pipeline

This pipeline is implemented in `Forwarder::onInterestFinalize` method and is entered from the *straggler timer* (Section 5.2.6) or Interest unsatisfied pipeline (Section 5.2.7).

The pipeline first determines whether Nonces recorded in the PIT entry need to be inserted to the Dead Nonce List (Section 4.5). The Dead Nonce List is a global data structure designed to detect looping Interests, and we want to insert as few Nonces as possible to keep its size down. Only outgoing Nonces (in out-records) need to be inserted, because an incoming Nonce that has never been sent out won't loop back.

We can further take chances on the ContentStore: if the PIT entry is satisfied, and the ContentStore can satisfy a looping Interest (thus stop the loop) during *Dead Nonce List entry lifetime* if Data packet isn't evicted, Nonces in this PIT entry don't need to be inserted. The ContentStore is believed to be able to satisfy a looping Interest, if the Interest does not have `MustBeFresh` selector, or the cached Data's `FreshnessPeriod` is no less than *Dead Nonce List entry lifetime*.

If it's determined that Nonces in the PIT entry should be inserted to the Dead Nonce List, tuples on Name and Nonce are added to the Dead Nonce List (Section 4.5.1).

Finally, the PIT entry is removed from PIT.

## 5.3 Data Processing Path

Data processing in NFD is split into these pipelines:

- incoming Data: processing of incoming Data packets
- Data unsolicited: processing of incoming unsolicited Data packets
- outgoing Data: preparation and sending out Data packets

### 5.3.1 Incoming Data Pipeline

This pipeline is implemented in `Forwarder::onIncomingData` method and is entered from `Forwarder::onData` method, which is triggered by `Face::onReceiveData` event emitter. The input parameters to this pipeline include a Data packet and its incoming Face.

This pipeline includes the following steps, summarized in Figure 13:

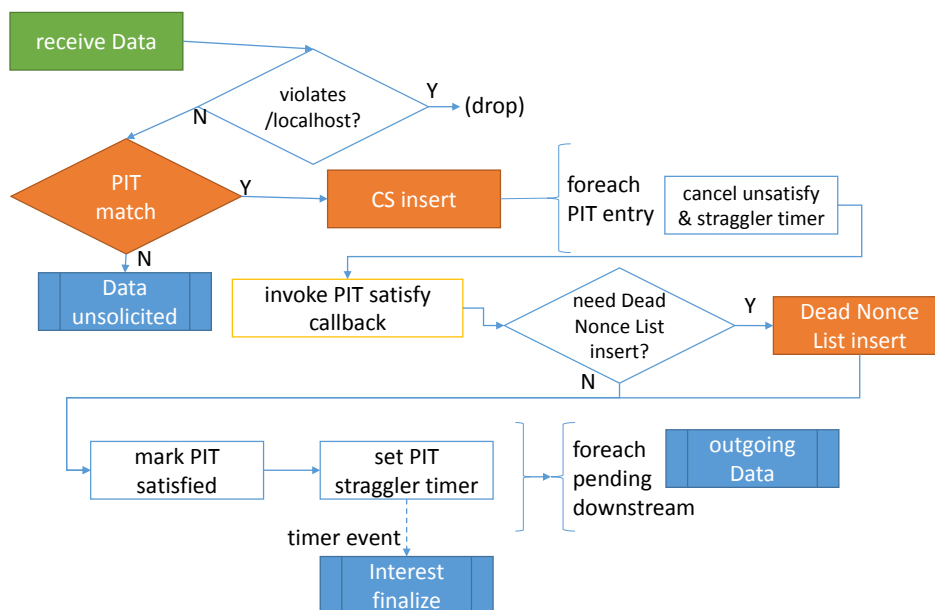


Figure 13: Incoming Data pipeline

- Similar to the incoming Interest pipeline, the first step in the incoming Data pipeline is to check the Data packet for violation of `/localhost` scope [12]. If the Data comes from a non-local Face, but the name begins with `/localhost` prefix, the scope is violated, Data packet is dropped, and further processing is stopped.

This check guards against malicious senders; a compliant forwarder will never send a `/localhost` Data to a non-local Face. Note that `/localhop` scope is not checked here, because its scope rules do not restrict incoming Data.

- After name-based scope constraint is checked, the Data packet is matched against the PIT using Data Match algorithm (Section 4.4.2). If no matching PIT entry is found, the Data is unsolicited, and is given to *Data unsolicited pipeline* (Section 5.3.2).
- If one or more matching PIT entries are found, the Data is inserted to ContentStore. Note that even if the pipeline inserts the Data to the ContentStore, whether it is stored and how long it stays in the ContentStore is determined by ContentStore admission and replacement policy.<sup>4</sup>
- The next step is to cancel the *unsatisfy timer* (Section 5.2.1) and *straggler timer* (Section 5.2.6) for each found PIT entry, because the pending Interest is now getting satisfied.
- Next, the effective strategy responsible for the PIT entry is determined using Find Effective Strategy algorithm (Section 4.6.1). The selected strategy is then triggered for the *before satisfy Interest* action with the PIT entry, the Data packet, and its incoming Face (Section 6.1.1).
- The Nonce on the PIT out-record corresponding to the incoming Face of the Data is inserted to the Dead Nonce List (Section 4.5), if it's deemed necessary (Section 5.2.8). This step is necessary because the next step would delete the out-record and the outgoing Nonce would be lost.
- The PIT entry is then marked satisfied by deleting all in-records, and the out-record corresponding to the incoming Face of the Data.
- The *straggler timer* (Section 5.2.6) is then set on the PIT entry.
- Finally, for each pending downstream except the incoming Face of this Data packet, *outgoing Data pipeline* (Section 5.3.3) is entered with the Data packet and the downstream Face. Note that this happens only once for each downstream, even if it appears in multiple PIT entries. To implement this, during the processing of matched PIT entries as described above, NFD collects their pending downstreams into an unordered set, eliminating all potential duplicates.

### 5.3.2 Data Unsolicited Pipeline

This pipeline is implemented in `Forwarder::onDataUnsolicited` method and is entered from the *incoming Data pipeline* (Section 5.3.1) when a Data packet is found to be unsolicited. The input parameters to this pipeline include a Data packet, and its incoming Face.

Generally, unsolicited Data needs to be dropped as it poses security risks to the forwarder. However, there are cases when unsolicited Data packets needs to be accepted to the ContentStore. In particular, the current implementation allows any unsolicited Data packet to be cached if this Data packet arrives from a local Face. This behavior supports a commonly used approach in NDN applications to “pre-publish” Data packets, when future Interests are anticipated (e.g., when serving segmented Data packets).

If it is desirable to cache unsolicited Data from non-local Faces, the implementation of `Forwarder::onDataUnsolicited` needs to be updated to include the desired acceptance policies

### 5.3.3 Outgoing Data Pipeline

This pipeline is implemented in `Forwarder::onOutgoingData` method and pipeline is entered from *incoming Interest pipeline* (Section 5.2.1) when a matching Data is found in ContentStore and from *incoming Data pipeline* (Section 5.3.1) when the incoming Data matches one or more PIT entries. The input parameters to this pipeline include a Data packet, and the outgoing Face.

This pipeline includes the following steps:

- The Data is first checked for `/localhost` scope [12]:
  - Data packets that start with `/localhost` prefix cannot be send out to a non-local Faces.<sup>5</sup>

`/localhop` scope is not checked here, because its scope rules do not restrict outgoing Data.

<sup>4</sup>The current implementation has fixed “admit all” admission policy and “priority FIFO” as replacement policy, see Section 4.3.

<sup>5</sup>This check is only useful in a specific scenario (see NFD Bug 1644).

- The next step is reserved for the traffic manager actions, such as to perform traffic shaping, etc. The current implementation does not include traffic manager implementation, but it is planned to be implemented in one of the next releases.
- Finally, the Data packet is sent via the outgoing Face.



## 6 Forwarding Strategy

As mentioned before, the forwarding strategy in NFD is a decision maker, deciding whether, when, and where to forward the Interests. NFD features an abstract interface (strategy API), which provides the baseline for implementation of multiplicity strategies, without the need of reimplementing full Interest processing pipeline. The main motivation for having multiple strategies is that our experience with NDN application showed that there a single fixed strategy cannot fit the needs for all applications. For example, some applications may require to multicast Interests to all available Faces to retrieve any matching copy of the Data as soon as possible, while the other may want to retrieve Data only from locations pointed by the routing system.

To provide the maximum flexibility, NFD allows per-namespace selection of the specific strategy, which is envisioned to be performed by the NFD operator. This per-namespace strategy choice is recorded in StrategyChoice table (Section 4.6), which is consulted in the forwarding pipelines when decision about Interest forwarding needs to be made. In addition to the Interest forwarding decision points, strategy can also receive notifications when the forwarded Interests are getting satisfied or timed out. Therefore, strategy presents a closed loop subsystem in NFD to control Interest forwarding.

Conceptually, a strategy can be considered a program, which is written for an abstract machine (strategy API, Section 6.1) and determines how to forward Interests. All current NFD strategies are written in C++ and are built-in into the NFD binary. However, future releases of NFD may allow custom strategies to be loaded at runtime and/or written in a scripting language against the strategy API abstract machine. Therefore, we chose to identify the forwarding strategy by NDN name, which can universally represent either a built-in strategy (Section 6.2) or, in the future, any external strategy program to be fetched from the network.

Since the objective of NFD is to provide a framework for easy experimentation, the list of the provided build-in strategies is in no way comprehensive and we encourage implementation and experimentation of new strategies. Section 6.3 provides insights to decide when implementation of a new strategy may be appropriate and give step-by-step guidelines explaining the process of developing new NFD strategies.

### 6.1 Strategy API

All NFD strategies are implemented as subclasses of `nfd::Strategy` base class, which provides the strategy API for interaction of the implemented strategy and the rest of NFD. This API is the only way a strategy can access NFD elements, therefore available functionality in the strategy API determines what NFD strategy can or cannot do.

A strategy is invoked through one of the *triggers* (Section 6.1.1). The forwarding decision is made with *actions* (Section 6.1.2). Strategies are also allowed to store information on certain table entries (Section 6.1.3).

#### 6.1.1 Triggers

Triggers are entrypoints to the strategy program. A trigger is declared as a virtual method of `nfd::Strategy` class, and is expected to be overridden by a subclass.

##### After Receive Interest Trigger

This trigger is declared as `Strategy::afterReceiveInterest` method. This method is pure virtual, which must be overridden by a subclass.

When an Interest is received, passes necessary checks, and needs to be forwarded, *Incoming Interest pipeline* (Section 5.2.1) invokes this trigger with the PIT entry, incoming Interest packet, and FIB entry. At that time, the following conditions hold for the Interest:

- The Interest does not violate `/localhost` scope.
- The Interest is not looped.
- The Interest cannot be satisfied by ContentStore.
- The Interest is under a namespace managed by this strategy.

After being triggered, the strategy should decide whether and where to forward this Interest. If the strategy decides to forward this Interest, it should invoke *send Interest* action at least once; it can do so either immediately or some time in the future using a timer.<sup>6</sup> If the strategy concludes that this Interest cannot be forwarded, it should invoke *reject pending Interest* action, so that the PIT entry will be deleted shortly.

<sup>6</sup>**Warning:** although a strategy is allowed to invoke *send Interest* action via a timer, this forwarding may never happen in special cases. For example, if while such a timer is pending an NFD operator updates the strategy on Interest's namespace, the timer event will be cancelled and new strategy may not decide to forward the Interest until after all out-records in the PIT entry expire.

### Before Satisfy Interest Trigger

This trigger is declared as `Strategy::beforeSatisfyInterest` method. The base class provides a default implementation that does nothing; a subclass can override this method if the strategy needs to be invoked for this trigger, e.g., to record data plane measurement results for the pending Interest.

When a PIT entry is satisfied, before Data is sent to downstreams (if any), *Incoming Data pipeline* (Section 5.3.1) invokes this trigger with the PIT entry, the Data packet, and its incoming face. The PIT entry may represent either a pending Interest or a recently satisfied Interest.

### Before Expire Interest Trigger

This trigger is declared as `Strategy::beforeExpirePendingInterest` method. The base class provides a default implementation that does nothing; a subclass can override this method if the strategy needs to be invoked for this trigger, e.g., to record data plane measurement results for the pending Interest.

When a PIT entry expires because it has not been satisfied before all in-records expire, before it is deleted, *Interest Unsatisfied pipeline* (Section 5.3.1) invokes this trigger with the PIT entry. The PIT entry always represents a pending Interest.

Note: this trigger will not be invoked if *reject pending Interest* action has been invoked.

#### 6.1.2 Actions

Actions are forwarding decisions made by the strategy. An action is implemented as a non-virtual protected method of `nfd::Strategy` class.

#### Send Interest action

This action is implemented as `Strategy::sendInterest` method. Parameters include a PIT entry, an outgoing face, and a `wantNewNonce` flag.

This action triggers entering the *Outgoing Interest pipeline* (Section 5.2.5).

#### Reject Pending Interest action

This action is implemented as `Strategy::rejectPendingInterest` method. Parameters include a PIT entry.

This action triggers entering the *Interest reject pipeline* (Section 5.2.6).

#### 6.1.3 Storage

Strategies are allowed to store arbitrary information on PIT entries, PIT downstream records (in-records), PIT upstream records (out-records), and Measurements entries, all of which are derived from `StrategyInfoHost` type<sup>7</sup>. Inside the triggers, the strategy already has access to PIT entry and can lookup desired in- and out-records. Measurement entry of the Measurements Table (Section 4.7) can be accessed via `Strategy::getMeasurements` method; the strategy's access is restricted to Measurements entries under the namespace(s) under its control (Section 4.7.2).

To store strategy-specific information, the strategy needs to declare a data structure(s) for the information to be stored, derived from `StrategyInfo` base class. At any point of time, the strategy can save an instance of `StrategyInfo`-derived object using `StrategyInfoHost::setStrategyInfo` method and/or retrieve it using `StrategyInfoHost::getStrategyInfo<T>` method. Note that strategy itself must ensure that the data structure used to retrieve an item is the same as the one used for storing. If there is a type mismatch, behavior is undefined and NFD will most likely crash.

Since the strategy choice for a namespace can be changed at runtime, NFD ensures that all strategy-stored items under the transitioning namespace will be destroyed. Therefore, the strategy must be prepared that some entities may not have strategy-stored items; however, if an item exists, its type is guaranteed to be correct. The destructor of stored item must also cancel all timers, so that the strategy will not be activated on an entity that is no longer under its managed namespace.

Strategy is only allowed to store information using the above mechanism. The strategy object (subclass of `nfd::Strategy`) should be otherwise stateless.

## 6.2 Built-in Strategies

Current version of NFD comes with these built-in strategies:

- best route strategy (`/localhost/nfd/strategy/best-route`, Section 6.2.1) sends Interest to lowest cost upstream.

<sup>7</sup>“Host” is in the sense of holding strategy information, not an endpoint/network entity.



- multicast strategy (`/localhost/nfd/strategy/multicast`, Section 6.2.2) sends every Interest to every upstream.
- client control strategy (`/localhost/nfd/strategy/client-control`, Section 6.2.3) allows the consumer to control where an Interest goes.
- NCC strategy (`/localhost/nfd/strategy/ncc`, Section 6.2.4) is similar to CCNx 0.7.2 default strategy.
- access router strategy (`/localhost/nfd/strategy/access`, Section 6.2.5) is designed for local site prefix on an access/edge router.

### 6.2.1 Best Route Strategy

The best route strategy forwards an Interest to the upstream with lowest routing cost. This strategy is implemented as `nfd::BestRouteStrategy2` class.

The strategy forwards a new Interest to the lowest-cost nexthop (except downstream). After that, if the consumer retransmits the Interest (with a new Nonce) or a similar Interest arrives from another downstream, the Interest is suppressed if it's within `MIN_RETRANSMISSION_INTERVAL`, otherwise it's forwarded again. A retransmitted Interest is forwarded to the lowest-cost nexthop (except downstream) that is not previously used; if all nexthops have been used, it is forwarded to a nexthop that is used earliest.

`MIN_RETRANSMISSION_INTERVAL` is currently set to a fixed value of 100 milliseconds. We are exploring better ways to set this parameter, such as RTT estimation or exponential back-off.

### 6.2.2 Multicast Strategy

The multicast strategy forwards every Interest to all upstreams, indicated by the supplied FIB entry. This strategy is implemented as `nfd::MulticastStrategy` class.

After receiving an Interest to be forwarded, the strategy iterates over the list of nexthop records in the FIB entry, and determines which ones are eligible. A nexthop face is *eligible* as an upstream if this face is not already an upstream (unexpired out-record exists in PIT entry), it is not the sole downstream (another in-record exists in PIT entry), and scope is not violated; `pit::Entry::canForwardTo` method is convenient for evaluating these rules. The strategy then multicasts the Interest to all eligible upstreams. If there is no eligible upstream, the Interest is rejected.

### 6.2.3 Client Control Strategy

The client control strategy allows a local consumer application to choose the outgoing face of each Interest. This strategy is implemented as `nfd::ClientControlStrategy` class.

If an Interest is received from a `LocalFace` (Section 2.1) that enables `NextHopFaceId` feature in `LocalControlHeader` [10], and the Interest packet carries a `LocalControlHeader` that contains a `NextHopFaceId` field, the Interest is forwarded to the outgoing face specified in the `NextHopFaceId` field if that face exists, or dropped if that face does not exist. Otherwise, the Interest is forwarded in the same manner as the best route strategy (Section 6.2.1).

### 6.2.4 NCC Strategy

The NCC strategy<sup>8</sup> is an reimplementation of CCNx 0.7.2 default strategy [16]. It has similar algorithm but is not guaranteed to be equivalent. This strategy is implemented as `nfd::NccStrategy` class.

### 6.2.5 Access Router Strategy

The access router strategy is specifically designed for local site prefix on an access/edge router. It is suitable for a namespace where producers are single-homed and are one hop away. This strategy is implemented as `nfd::AccessStrategy` class.

The strategy is able to make use of multiple paths in the FIB entry, and remember which path can lead to contents. It is most efficient when FIB nexthops are accurate, but can tolerate imprecise nexthops, and still be able to find the correct paths.

The strategy is able to recover from a packet loss in the last-hop link. It retries Interests retransmitted by consumer in the same manner as best route strategy (Section 6.2.1); The same mechanism also allows the strategy to deal with producer mobility.

<sup>8</sup>NCC does not stand for anything; it is just CCN backwards.

## 6.3 How to Develop a New Strategy

Before starting development of a new forwarding strategy, it is necessary to assess necessity of the new strategy, as well strategy capabilities and limitations (Section 6.3.1). The procedure of developing a new built-in strategy is outlined in Section 6.3.2.

### 6.3.1 Should I Develop a New Strategy?

In many network environments, it may be sufficient to use one of the existing strategies: best-route, multicast, ncc, or access. In cases when an application wants a fine-grain control of Interest forwarding, it can use the special client control strategy (Section 6.2.3) and specify an outgoing face for every Interest. However, this could control the outgoing face of local forwarder only. In other cases, a new strategy development could be warranted, provided that the desired behavior can fit within the strategy API framework.

When developing a new strategy, one needs to remember that the strategy choice is local to a forwarder and only one strategy can be effective for the namespace. Choosing the new strategy on a local forwarder will not affect the forwarding decisions on other forwarders. Therefore, developing a new strategy may require reconfiguration of all network nodes.

The only purpose of the strategy is to decide how to forward Interests and cannot override any processing steps in the forwarding pipelines. If it is desired to support a new packet type (other than Interest and Data), a new field in Interest or Data packets, or override some actions in the pipelines (e.g., disable ContentStore lookup), it can be only accomplished by modification of the forwarding pipelines.

Even with the mentioned limitations, the strategy can provide a powerful mechanism to control how Data is retrieved in the network. For example, by using a precise control of how and where Interests are forwarded and re-transmitted, a strategy can adapt Data retrieval for a specific network environment. Another example would be an application of limits on how much Interests can be forwarded to which Faces. This way a strategy can implement various congestion control and DDoS protections schemes [15, 17].

### 6.3.2 Develop a New Built-in Strategy

The initial step in creating a new strategy is to create a class, say `MyStrategy` that is derived from `nfd::Strategy`. This subclass must at least override the *triggers* that are marked pure virtual and may override other available *triggers* that are marked just virtual.

If the strategy needs to store information, it is needed to decide whether the information is related to a namespace or an Interest. Information related to a namespace but not specific to an Interest should be stored in Measurements entries; information related to an Interest should be stored in PIT entries, PIT downstream records, or PIT upstream records. After this decision is made, a data structure derived from `StrategyInfo` class needs to be declared. In the existing implementation, such data structures are declared as nested classes as it provides natural grouping and scope protection of the strategy-specific entity, but it is not required to follow the same model. If timers (Section 10.4) are needed, `EventId` fields needs to be added to such data structure(s).

The final step is to implement the *triggers* with the desired strategy logic. When implementing strategy logic, refer to Section 6.1.1 describing when each trigger is invoked and what is it expected to do.

Notes and common pitfalls during strategy development:

- When retrieving a stored item from an entity, you should always check whether the retrieved element is not NULL (Section 6.1.3). Otherwise, even the strategy logic guarantees that item will always be present on an entity, because NFD allows dynamic per-namespace strategy change, the expected item could not be there.
- Timers must be cancelled in the destructor of the stored item (Section 6.1.3). This is necessary to ensure that the strategy will not be accidentally triggered on an entity that is no longer being managed by the strategy.
- Measurements entries are cleaned up automatically. If Measurements entries are used, you need to call `this->getMeasurements()->extendLifetime` to avoid an entry from being cleaned up prematurely.
- *Before satisfy Interest trigger* (Section 6.1.1) may be invoked with either pending Interest or recently satisfied Interest.
- The strategy is allowed to retry, but retries should not be attempted after the PIT entry expires. It is also not allowed to send the same Interest via the same outgoing face before the previous out-record expires.
- The strategy should not violate scope. If the scope is violated, the *outgoing Interest pipeline* (Section 5.2.5) will not send the Interest and the strategy may incorrectly gauge data plane performance.
- The strategy is responsible for performing congestion control.

Before the strategy can be actually used, it is necessary to modify `daemon/fw/available-strategies.cpp` and install the new strategy to the list of existing built-in strategies. If the strategy is installed as non-default, the strategy needs to be activated on desired namespaces via a StrategyChoice management command (Section 7.1.3), e.g., using `nfdc` command-line tool.

## 7 Management

Management modules, referred to as *managers*, provide an Interest/Data API for controlling NFD. In particular, users can:

- create, destroy Faces, and enable/disable local control features on local faces (Face Manager)
- add and remove FIB entries (FIB Manager)
- manipulate selection of the forwarding strategy for namespaces (Strategy Choice Manager)

Each manager is an interface for some part of the lower layers of NFD. For example, the Face Manager handles Face creation/destruction. The current set of managers are independent and do not interact with one another. Consequently, adding a new manager is a fairly straightforward task; one only needs to determine what part(s) of NFD should be exported to an Interest/Data API and create an appropriate command Interest interpreter.

In general, NFD managers do not need to offer much functionality through a programmatic API. Most managers only need to register request handlers or notification streams to the *ndn::Dispatcher*, such that the corresponding requests will be routed to them and the produced notifications could be post out. Some managers may also require to hook into the configuration file parser. All managerial tasks to control NFD internals should be performed via the defined Interest/Data management protocol.

All management actions that change NFD state require the use of *control commands* [4]; a form of signed Interests. These allow NFD to determine whether the issuer is authorized to perform the specified action. Management modules respond with *control responses* to inform the user of the commands success or failure. Control responses have status codes similar to HTTP and describe the action that was performed or any errors that occurred.

Management actions that just query the current state of NFD do not need to be authenticated. Therefore, these actions are defined in NFD Management Protocol [3] as *status dataset*, and are currently implemented in NFD as a simple Interest/Data exchange. In the future if data access control is desired, some data can be encrypted.

Currently, as shown in figure 14, all managers utilize the *ndn::Dispatcher* as an agent to deal with high-level Interest / Data exchange, so that they can focus on low-layer operations toward the NFD.

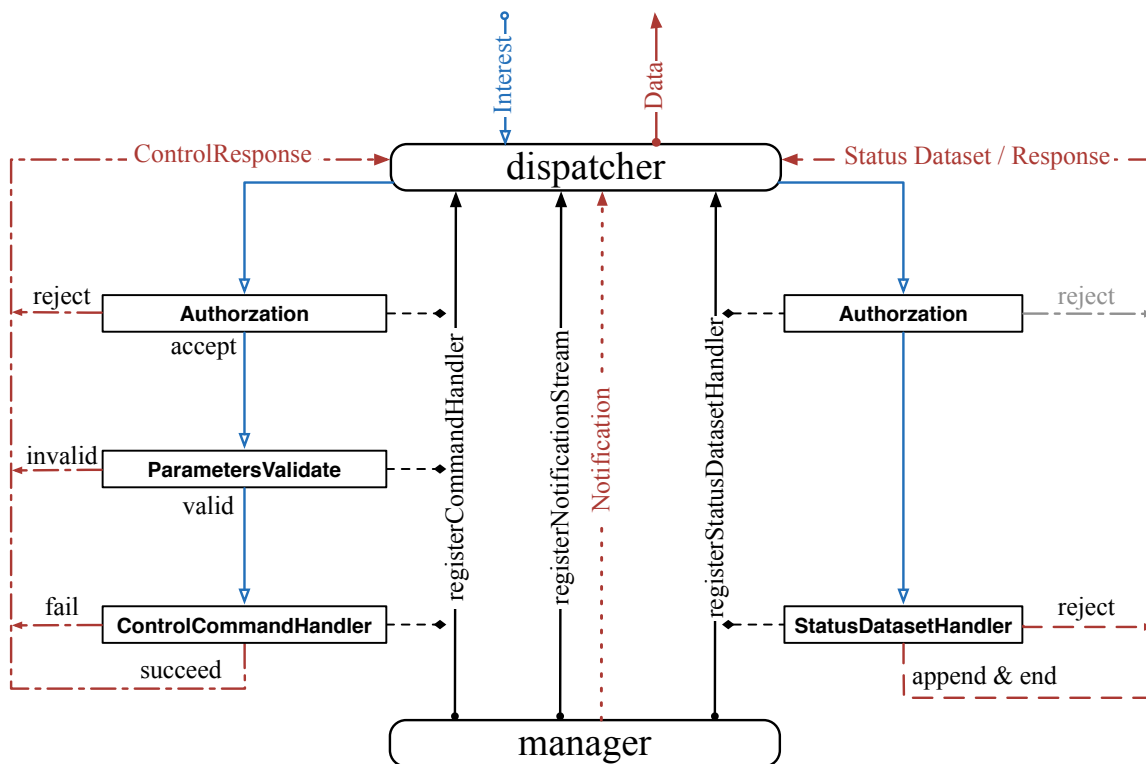


Figure 14: Overview of the manager Interest / Data exchange via the dispatcher.

More specifically, a manager always consists of a series of handlers, each of which is responsible to deal with a *control command* request or a *status dataset* request. Per management protocol, those requests [3], follow a namespace pattern of `/localhost/nfd/<manager-name>/<verb>`. Here, *verb* describes the action that the *manager-name* manager should

perform. For example, `/localhost/nfd/fib/add-next-hop` directs the FIB Manager to add a next hop (command arguments follow the verb).

A handler is registered, by some manager, to the dispatcher with a partial name that is composed of the manager's name and a specified verb. Then, after all managers have registered their handlers, the dispatcher will make full prefixes with those partial names and each registered top prefix (i.e., `/localhost/nfd`), and then sets interest filters for them. When a request arrives the dispatcher, it will finally be directed to some management handler after processed by interest filters. On the other hand, all types of responses, such as of *control responses*, *status datasets*, etc., will be back to the dispatcher, for concatenating, segmenting and signing when required.

For *control command*, a handler is always registered along with an *authorization* method and a *parametersValidate* method. Then, a request can be directed to the handler, if and only if it is accepted by the *authorization*, and its *control parameters* can be validated by the *parametersValidate*. While for *status dataset*, there is no need to validate parameters, and, currently, the *authorization* is made to accept all requests<sup>9</sup>.

Besides, some managers (such as the FaceManger), that produces notifications, will also register *notification streams* to the dispatcher. And this type of registration returns a *postNotification*, through which the manager will only need to generate the notification content, leaving other parts of work (making packet, signing, etc.) to the dispatcher.

The remainder of this section will describe the managers and their support modules in greater detail. Note, however, that we intentionally omit many of the details of the management protocols themselves and refer interested readers to the NFD Management Protocol specification [3].

## 7.1 Managers

### 7.1.1 Face Manager

The Face Manager (`nfd::FaceManager`) creates and destroys Faces for its configured channels. Local control headers [10] can also be enabled/disabled to learn over which Face a Data packet arrived or to direct Interest out specific Faces when used in conjunction with the *client control* forwarding strategy (Section 6.2.3).

#### Configuration

The NFD startup process registers the Face Manager as the `face_system` configuration file section handler via `setConfigFile`. This will cause `onConfig` to be called by the configuration file parser (`ConfigFile`, Section 10.1).

The Face Manager relies heavily on the NFD configuration file's `face_system` section. In particular, this section is used to determine which Face protocols should be enabled, which protocol channels should be constructed for future Face creation, and whether multicast Faces for the protocol need to be created.

The `onConfig` method performs dispatching for `face.system` subsections (methods beginning with `processSection-`). All subsection processors are given the `ConfigSection` instance representing their subsection (a typedef around the boost property tree node [18]) and a flag indicating whether or not a dry run is currently being performed. This allows NFD to test the sanity of the configuration file before performing any modifications.

Some subsection processors take a list of `NetworkInterfaceInfo` pointers as one of the input parameters. `onConfig` gets this list from the `listNetworkInterfaces` free function defined in `core/network-interface.hpp` file. The list describes all available network interfaces available on the machine. In particular, `processSectionUdp` and `processSectionEther` use the list for detecting multicast-capable interfaces for creating multicast faces.

The Face Manager maintains a protocol (`string`) to `shared_ptr<ProtocolFactory>` mapping (`m_factories`) to facilitate Face creation tasks. The mapping is initialized during configuration by the `processSection-` methods. Each subsection processor creates a factory of the appropriate type and stores it in the mapping. For example, the TCP processor creates a `shared_ptr<TcpFactory>` and adds it to the map with `"tcp4"` and `"tcp6"` keys. When the FaceManager receives command to create a Face that specifies FaceURI that starts with `"tcp4://"`, `"tcp6://"`, it will use this factory to properly dispatch the request. The factory will use the protocol specified in the FaceURI to further dispatch the request to the appropriate IPv4 or IPv6 channel (Note that `"tcp://"` protocol is no longer supported). Refer to Section 2 for more details on the workings and interactions of the `ProtocolFactory`, `Channel`, and `Face` classes.

#### Command Processing

On creation, the Face Manger registers four command handlers, `createFace`, `destroyFace`, `enableLocalControl`, `disableLocalControl`, to the dispatcher with names `faces/create`, `faces/destroy`, `faces/enable-local-control` and `faces/disable-local-control` respectively.

- `createFace`: create unicast TCP/UDP Faces

<sup>9</sup>This may be changed whenever data access control is desired.

- `destroyFace`: destroy Faces
- `enableLocalControl`: enable local control feature on requesting Face
- `disableLocalControl`: disable local control feature on requesting Face

While NFD supports a range of different protocols, the Face management protocol currently only supports the creation of unicast TCP and UDP Faces during runtime. That said, the Face Manager may also be configured to have other channel types to listen for incoming connections and create Faces on demand.

`createFace` uses `FaceUri` to parse the incoming URI in order to determine the type of Face to make. The URI must be canonical. A canonical URI for UDP and TCP tunnels should specify either IPv4 or IPv6, have IP address instead of hostname, and contain port number (e.g., “`udp4://192.0.2.1:6363`” is canonical, but “`udp://192.0.2.1`” and “`udp://example.net:6363`” are not). Non-canonical URI results in a code 400 “Non-canonical URI” control response. The URI’s scheme (e.g., “`tcp4`”, “`tcp6`”, etc.) is used to lookup the appropriate `ProtocolFactory` in `m_factories`. Failure to find a factory results in a code 501 “unsupported protocol” control response. Otherwise, Face Managers calls `ProtocolFactory::createFace` method to initiate asynchronous process of face creation (DNS resolution, connection to remote host, etc.), supplying `afterCreateFaceSuccess` and `afterCreateFaceFailure` callbacks. These callbacks will be called by the face system after the Face is either successfully created or failed to be created, respectively.

After Face has been created (from `afterCreateFaceSuccess` callback), the Face Manager adds the new Face to the Face Table<sup>10</sup> and responds with a code 200 “success” control response to the original control command. Unauthorized, improperly-formatted requests and requests when Face is failed to be created will be responded with appropriate failure codes and failure reasons. Refere to the NFD Management protocol specification [3] for the list of possible error codes.

`destroyFace` attempts to close the specified Face. The Face Manager responds with code 200 “Success” if the Face is successfully destroyed or it cannot be found in the Face Table, but no errors occurred. The Face Manager does not directly remove the Face from the Face Table, but it is a side effect of calling `Face::close`.

`LocalControlHeader` can be enabled on local Faces (`UnixStreamFace` and `TcpLocalFace`) in order to expose some internal NFD state to the application or to give the application some control over packet processing. Currently `LocalControlHeader` specification [10] defines the following *local control features*:

- `IncomingFaceId`: provide the `FaceId` that Data packets arrive from
- `NextHopFaceId`: forward Interests out the Face with a given `FaceId` (requires the `client-control` forwarding strategy, Section 6.2.3)

As their names imply, the `(enable|disable)LocalControl` methods enable and disable the specified local control features on the Face sending the control command. Both methods utilize `extractLocalControlParameters` method to perform common functionality of option validation and ensuring that the requesting Face is local. When incorrectly formatted, unauthorized request or request from a non-local Face is received, the Face Manager responds with an appropriate error code. Command success, as defined by Control Command specification [4], is always responded with code 200 “OK” response.

### Datasets and Event Notification

The Face Manager provides two datasets: Channel Status and Face Status. The Channel Status dataset lists all channels (in the form of their local URI) that this NFD has created and can be accessed under the `/localhost/nfd/faces/channels` namespace. Face Status, similarly, lists all created Faces, but provides much more detailed information, such as flags and incoming/outgoing Interest/Data counts. The Face Status dataset can be retrieved from the `/localhost/nfd/faces/list` namespace. These datasets are supplied when `listFaces` and `listChannels` methods are invoked. Besides, another method `queryFaces` is provided to supply the status of face with a specified name. When the Face Manger is constructed, it will register these three handlers to the dispatcher with names `faces/list`, `faces/channels` and `faces/query` respectively.

In addition to these datasets, the Face Manager also publishes notifications when Faces are created and destroyed. This is done using the `postNotification` function returns after registering a notification stream to the dispatcher with the name `/faces/events`. Two methods, `afterFaceAdded` and `afterFaceRemoved`, that take the function `postNotification` as a argument, are set as connections to the Face Table’s `onAdd` and `onRemove` Signals [14]. Once these two signals are emitted, the connected methods will be invoked immediately, where the `postNotification` will be used to publish notifications through the dispatcher.

<sup>10</sup>The Face Table is a table of Faces that is managed by the Forwarder. Using this table, the Forwarder assigns each Face a unique ID, manage active Faces, and perform lookup for a Face object by ID when requested by other modules.



### 7.1.2 FIB Manager

The FIB Manager (`nfd::FibManager`) allows authorized users (normally, it is only RIB Manager daemon, see Section 8) to modify NFD's FIB and publishes a dataset of all FIB entries and their next hops. At a high-level, authorized users can request the FIB Manager to:

1. add a next hop to a prefix
2. update the routing cost of reaching a next hop
3. remove a next hop from a prefix

The first two capabilities correspond to the `add-nexthop` verb, while removing a next hop falls under `remove-nexthop`. These two verbs are used along with the manager name `fib` to register the following handlers of control commands:

- `addNextHop`: add next hop or update existing hop's cost
- `removeNextHop`: remove specified next hop

Note that `addNextHop` will create a new FIB entry if the requested entry does not already exist. Similarly, `removeNextHop` will remove the FIB entry after removing the last next hop.

**FIB Dataset** One status dataset handler, `listEntries` is registered, when the FIB Manger is constructed, to the dispatcher with the name `fib/list`, to publish FIB entries according to the FIB dataset specification.

On invocation, the whole FIB is serialized in the form of a collection of `FibEntry` and nested `NextHopList` TLVs, which are appended to a `StatusDatasetContext` of the dispatcher. After all parts are appended, that context is ended and will process all received status data.

### 7.1.3 Strategy Choice Manager

The Strategy Choice Manager (`nfd::StrategyChoiceManager`) is responsible for setting and unsetting forwarding strategies for the namespaces via the Strategy Choice table. Note that setting/unsetting the strategy applies only to the local NFD. Also, the current implementation requires that the selected strategy must have been added to a pool of known strategies in NFD at compile time (see Section 6). Attempting to change to an unknown strategy will result in a code 504 “unsupported strategy” response. By default, there is at least the root prefix (“/”) available for strategy changes, which defaults to the “best route” strategy. However, it is an error to attempt to unset the strategy for root (code 403).

Similar to the FIB and Face Managers, the Strategy Choice Manager registers, when constructed, two command handlers, `setStrategy` and `unsetStrategy`, as well as a status dataset handler `listChoices` to the dispatcher, with names `strategy-choice/set`, `strategy-choice/unset`, and `strategy-choice/list` respectively.

On invocation, `setStrategy` and `unsetStrategy` will set / unset the specified strategy, while `listChoices` will serialize the Strategy Choice table into `StrategyChoice` TLVs, and publish them as the dataset.

### 7.1.4 Forwarder Status Manager

The Forwarder Status Manager (`nfd::StatusServer`) provides information about the NFD and basic statistics about NFD by the method `listStatus`, which is registered to the dispatcher with the name `status` (no verb). The supplied information includes the NFD's version, startup time, Interest/Data packet counts, and various table entry counts, and is published with a 5 second freshness time to avoid excessive processing.

## 7.2 Management Support Classes

This section describes several support classes that are used exclusively by NFD's management modules.

### 7.2.1 Manager Base

`ManagerBase` is the base class for all managers. This class holds the manager's shared `Dispatcher` and `CommandValidator`, and provides a number of commonly used methods. In particular, `ManagerBase` provides the methods to register command handler, to register status dataset handler, and to register notification streams. Besides, `ManagerBase` also provides convenience methods for authorizing *control commands* (`authorize`), for validating *control parameters* (`validateParameters`), and also for extracting the name of the requester from the Interest (`extractRequester`).

On construction, `ManagerBase` obtains a reference to a `Dispatcher` that is responsible to dispatch requests to the target management handler, and a `CommandValidator` that will be used for control command authorization later. Derived manager



classes provide the `ManagerBase` constructor with the name of their `privilege` (e.g., `faces`, `fib`, or `strategy-choice`). This privilege is used to specify the set of authorized capabilities for a given NDN identity certificate in the configuration file.

### 7.2.2 Internal Face and Internal Client Face

To initialize the `Dispatcher`, a reference to a `Internal Client Face` derived from the `ndn::Face` class, is provided, which is connected to a `Internal Face` derived from the `nfd::Face` for internal use in NFD. Consequently, the dispatcher is granted to perform Interest / Data exchange bidirectionally with the `Forwarder`.

### 7.2.3 Command Validator

The `CommandValidator` validates control commands based on privileges specified in the NFD configuration file. The NFD startup process registers `CommandValidator` as the processor of the `authorizations` section using `setConfigFile`, which will in turn invoke the `onConfig` method. `onConfig` supports several privileges:

- `faces` (Face Manager)
- `fib` (FIB Manager)
- `strategy-choice` (Strategy Choice Manager)

These privileges are associated with a specified NDN identity certificate that will then be authorized to issue control commands to the listed management modules. The `CommandValidator` learns about which privileges to expect in the configuration file via the `addSupportedPrivilege` method. This method is invoked by each manager's `ManagerBase` constructor with the appropriate privilege name.

`CommandValidator` also supports the notion of a “wildcard” identity certificate for demonstration purposes to remove the “burden” of configuring certificates and privileges. Note, however, that this feature is security risk and should not be used in production. See Section 10.1 for more detail about `CommandValidator` configuration.

### 7.2.4 General Configuration File Section Parser

The `general` namespace provides parsing for the identically named `general` configuration file section. The NFD startup process invokes `setConfigSection` to trigger the corresponding localized (static) `onConfig` method for parsing.

At present, this section is limited to specifying an optional user and group name to drop the effective `userid` and `groupid` for safer operation. The `general` section parser initializes a global `PrivilegeHelper` instance to perform the actual (de-)escalation work.

### 7.2.5 Tables Configuration File Section Parser

`TablesConfigSection` provides parsing for the `tables` configuration file section. This class can then configure the various NFD tables (CS, PIT, FIB, Strategy Choice, Measurements, and Network Region) appropriately. Currently, the `tables` section supports changing the default maximum number of Data packets that the Content Store can hold, per-prefix strategy choices, and network region names. Like other configuration file parsers, `TablesConfigSection` is registered as the processor of its corresponding section by the NFD startup process via `setConfigFile` method, which invokes `onConfig`.

## 8 RIB Management

The RIB manager manages the routing information base (RIB) and updates the forwarding information base (FIB) when needed. RIB manager is implemented as an independent module of NFD, which runs as a separate thread and communicates with NFD through Face abstraction only.<sup>11</sup> This separation was done in order to keep packet forwarding logic lightweight and simple, as RIB operations can require complex manipulations. Figure 15 shows the high-level interaction of the RIB Manager with NFD and other applications. A more detailed interaction is shown in Figure 16.

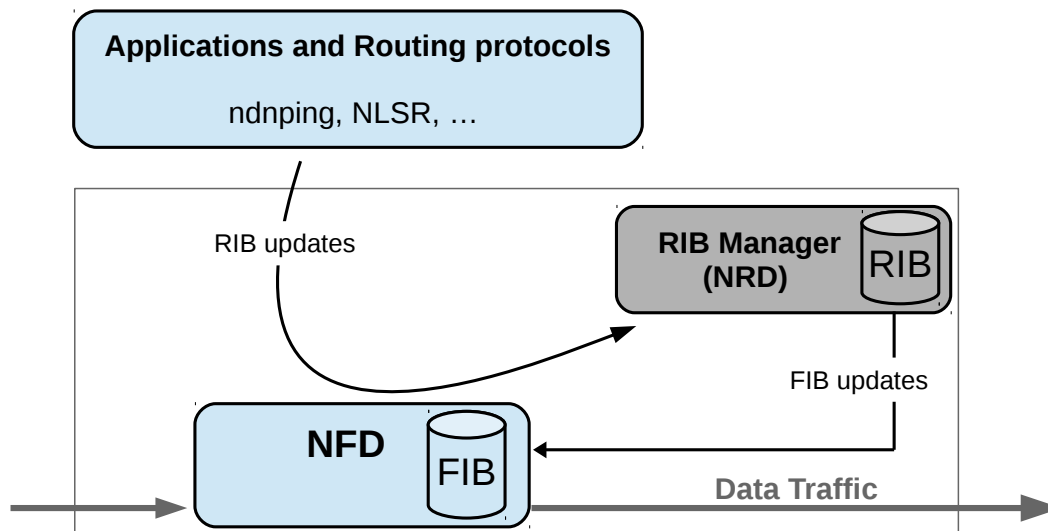


Figure 15: RIB Manager—system diagram

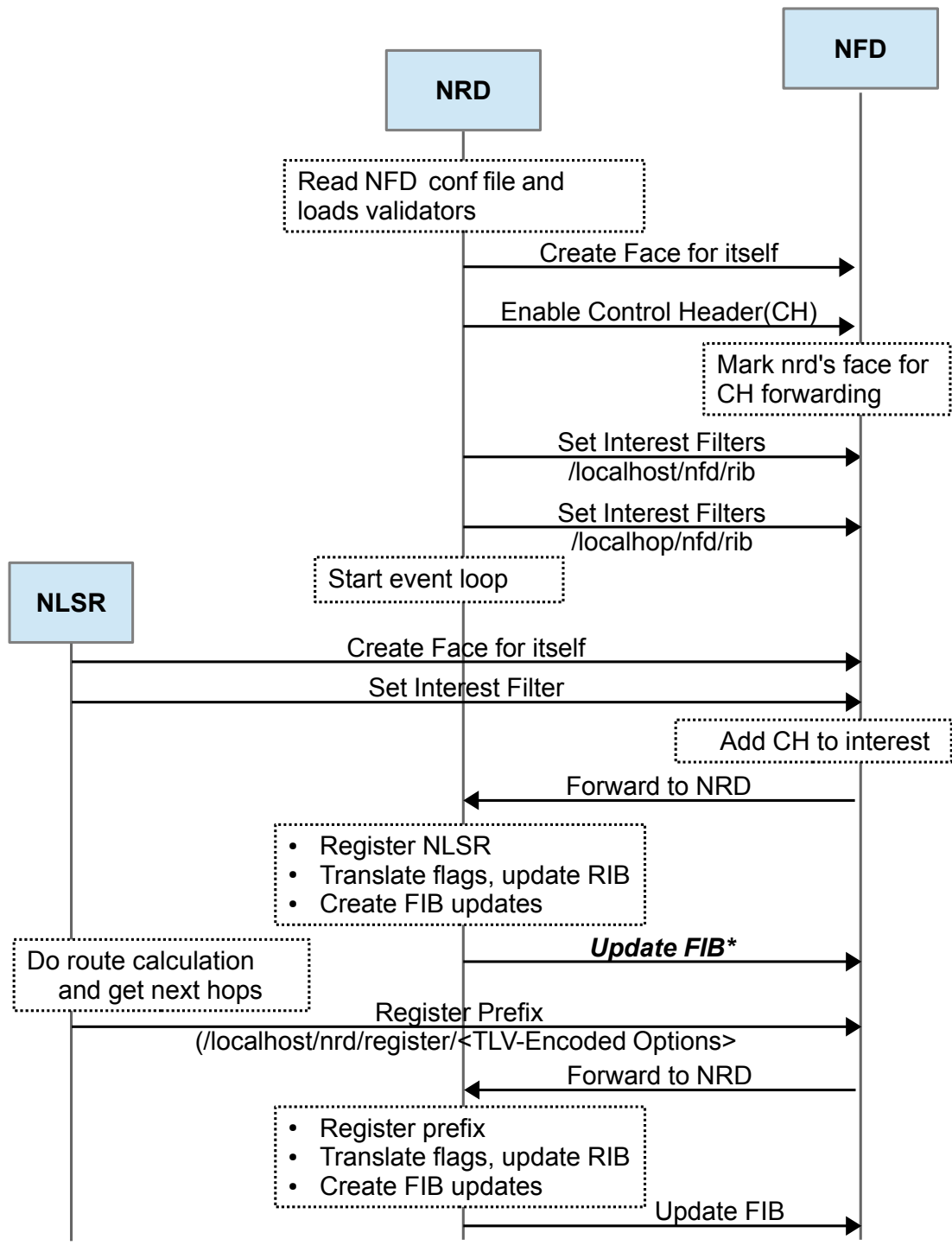
NFD provides various flags for prefix registration that allow a fine grained control and features such as hole-punching in a namespace. Depending on the flag, a single registration request may result in multiple FIB entry changes. The RIB manager takes the responsibility of processing these flags in place of the FIB Manager. In other words, the RIB manager receives all registration requests, processes the included flags, and creates FIB updates as needed, which makes the forwarder leaner and faster. As the RIB can be updated by different parties in different ways, including various routing protocols, application’s prefix registrations, and command-line manipulation by sysadmins, the RIB manager also provides a common abstraction to all these processes and generates a consistent forwarding table. Therefore, applications should use the RIB management interface to manipulate the RIB, and only RIB manager should use the FIB management interface to directly manipulate NFD’s FIB.

### 8.1 Initializing RIB Manager

When an instance of the RIB Manager is created, the following operations are performed:

- localhost and localhop validation rules are loaded from the the *rib* block of the NFD configuration file;
- the control command prefixes */localhost/nfd/rib* and */localhop/nfd/rib* (if enabled) are “self-registered” in NFD’s FIB. This allows RIB manager to receive RIB-related control commands (registration/unregistration requests), requests for RIB management datasets, and notifications.
- LocalControlHeader [10] is requested on the Face between RIB manager and NFD. This allows RIB manager to get FaceId from where the prefix registration/unregistration commands were received (for “self-registrations” from NDN applications).
- RIB manager subscribes to the Face status notifications using the FaceMonitor class [14] to receive notifications whenever a Face is created or destroyed, so the corresponding RIB entries can be updated.

<sup>11</sup>On NFD-android platform (<https://github.com/named-data/NFD-android>), RIB manager runs within the same thread as NFD, but is still independent.



\* Self-registration of NLSR is complete here. Similar steps are followed by other applications for self-registration. Please note that the steps beyond this point are only required by a routing protocol.

Figure 16: RIB Manager—timing diagram

This revision is **OUTDATED** and preserved for historical purposes only. Check for the most recent revision at <https://named-data.net/publications/techreports/>

## 8.2 Communicating with RIB Manager

Applications, including routing protocols, may register or unregister routes through RIB manager by using control commands [4] sent to the RIB management module.

### 8.2.1 Registering a Route

A route may be registered with the command-verb: **register** and the following **ControlParameters**:

- **Name**: Associated name prefix (**required**)
- **FaceId**: The Face ID returned from the Face Management module after Face creation.  
If the FaceId is set to zero or not set, the requesting Face is used (self-registration).
- **Origin**: The producer of the command; defaults to 0 (Local producer application)
- **Cost**: Route preference; defaults to 0
- **Flags**: inclusive OR of route inheritance flags (Section 8.4)
- **ExpirationPeriod**: The duration for which the route is active (in milliseconds)

### 8.2.2 Unregistering a Route

A route may be unregistered with the command-verb: **unregister** and the following **ControlParameters**:

- **Name**: Associated name prefix (**required**)
- **FaceId**: The Face ID returned from the Face Management module after Face creation.  
If the FaceId is set to zero or not set, the requesting Face is used (self-deregistration).
- **Origin**: The producer of the command; defaults to 0 (Local producer application)

## 8.3 RIB Entry

The RIB contains a list of **RIB entries** each of which holds the following information for a **route**:

- **name**: Associated name prefix
- **FaceId**: The nexthop Face
- **origin**: The producer of the announcement. A prefix registration request can be sent by different parties. This field is used to differentiate between them. If one of the routing protocols or applications quits, this field helps to remove RIB entries added by that protocol or application.
  - 0: Local producer application
  - 128: NLSR
  - 255: Static route
- **flags**: inclusive OR of route inheritance flags (Section 8.4)
- **cost**: Route preference
  - When there are multiple routes for the same name prefix, a lower cost indicates a more preferred nexthop.
- **expires**: The duration for which the route is active (in milliseconds)

## 8.4 Prefix Registration Flags

The prefix registration flags allow a fine grained control over prefix registration. The currently defined flags are:

- **CHILD\_INHERIT**: Longer matching name prefixes may use this route; true by default. Use `ndn::nfd::ROUTE_FLAG_CHILD_INHERIT` to set the flag.
- **CAPTURE**: No shorter name prefix route may be used for this prefix. If specified, this flag overrides the `CHILD_INHERIT` flag. Note that if any route in a RIB entry has the `CAPTURE` flag set, the entire RIB entry's namespace is blocked from using shorter prefix routes. Use `ndn::nfd::ROUTE_FLAG_CAPTURE` to set the flag.

### 8.4.1 Examples

Table 2: Example RIB

Name prefix	Nexthop FaceId	CHILD_INHERIT	CAPTURE	Effective Nexthops*
/	1	true	false	1
/	2	false	false	2
/A	3	true	false	3, 1
/A/B/C	4	true	false	4, 3, 1
/D	5	true	true	5
/D	6	true	false	6
/D/E	7	false	false	7, 6, 5

\* "Effective Nexthops" column shows the registered and inherited nexthops (it is not a part of the actual RIB).

For the RIB example shown in Table 2, the following forwarding decisions can be taken:

1. Interest /S can go through Faces 1 and 2.
2. Interest /A/P can go through Faces 1 and 3, but cannot go through Face 2 since that route has `CHILD_INHERIT=false`.
3. Interest /A/B/C/Q can through Faces 1, 3, and 4 because route /A/B/C inherits Face 1 from / and Face 3 from /A.
4. Interest /D/R can go through Faces 5 and 6, but cannot go through Face 1 since one of the routes on /D sets `CAPTURE=true`.
5. Interest /D/E/F can go through Faces 5, 6, and 7 due to the `CHILD_INHERIT=true` routes on /D, but cannot go through Face 1 since one of the routes on /D sets `CAPTURE=true`.

### 8.4.2 Cost Inheritance

Version 0.2 of NFD implements the following logic to assign costs to nexthops in the FIB when `CHILD_INHERIT` is set. When the flag is set on the route of a RIB entry, that route's Face and cost are applied to the longer prefixes (children) under the RIB entry's namespace. If a child already has a route with the Face that would be inherited, the inherited route's Face and cost are **not** applied to that child's nexthops in the FIB. Also, if a child already has a route with the Face that would be inherited and the child's route has its `CHILD_INHERIT` flag set, the inherited route's Face and cost are **not** applied to the nexthops of the child nor the children of the child namespace. If a RIB entry has neither the `CHILD_INHERIT` nor the `CAPTURE` flag set on any of its routes, that RIB entry can inherit routes from longer prefixes which do not have the same Face ID as one of the RIB entry's routes.

**Future versions will assign the lowest available cost to a nexthop Face based on all inherited RIB entries not blocked by a CAPTURE flag.**

Table 3: Version 0.2 Nexthop cost calculation. The nexthops and their costs for each RIB entry are calculated using the RIB (the table on the left) and are installed into the FIB (the table on the right).

Name prefix	Nexthop FaceId	CHILD_INHERIT	CAPTURE	Cost	Name prefix	(Nexthop FaceId, Cost)
/	1	true	false	75	/	(1, 75)
/a	2	false	false	50	/a	(2, 50), (1, 75)
/a/b	1	false	false	65	/a/b	(1, 65)
/b	1	true	false	100	/b	(1, 100)
/b/c	3	true	true	40	/b/c	(3, 40)
/b/c/e	1	false	false	15	/b/c/e	(1, 15), (3, 40)
/b/d	4	false	false	30	/b/d	(4, 30), (1, 100)

## 8.5 FIB Updater

The FIB Updater is used by the RIB Manager to process route flags, generate FIB updates, and send FIB updates to NFD. The RIB is only modified after FIB updates generated by a RIB update are applied by the FIB Updater successfully. The RIB Manager takes parameters from incoming requests and creates a RIB update that is passed to the RIB using the `Rib::beginApplyUpdate` method. The RIB creates a `RibUpdateBatch`, a collection of RIB updates for the same `FaceId`, and adds the batch to a queue. The RIB will advance the queue and hand off each batch to the FIB Updater for processing. Only one batch can be handled by the FIB Updater at a time, so the RIB will only advance the queue when the FIB Updater is not busy. The FIB Updater processes the flags of the update in the received `RibUpdateBatch`, generates the necessary FIB updates, and sends the FIB updates to the FIB as needed. If the FIB updates are applied successfully, the RIB update is applied to the RIB. Otherwise, the FIB update process fails, and the RIB update is not applied to the RIB.

If a FIB update fails:

1. in the case of a non-recoverable error (eg. signing key of RIB Daemon is not trusted, more than 10 timeouts for the same command), NFD will terminate with error;
2. in the case of a non-existent face error with the same face as the `RibUpdateBatch`, the `RibUpdateBatch` is abandoned;
3. in the case of a non-existent face error with a different face than the `RibUpdateBatch`, the FIB update that failed is skipped;
4. in other cases (eg. timeout), the FIB update is retried.

## 8.6 On Request

When RIB manager receives a request, it first validates it. If the validation fails, it returns a control response with `error code 403`. If the validation is successful, it confirms the passed command is valid and if it is, executes one of the following commands:

- **Register Entry:** The RIB Manager takes the passed parameters from the incoming request, creates a RIB update, and passes it to the RIB to begin the FIB update process. If the FIB update process is successful, the RIB searches for an entry that matches the name, `FaceId`, and `Origin` of the incoming request. If the `FaceId` is 0 in the incoming request then it means that an application is trying to register itself with NFD (self-registration). For self-registration requests, the RIB Manager fetches the `FaceId` of the application from the Control Header and uses it for registration. If no match is found, the passed parameters are inserted as a new entry. Otherwise, the matching entry is updated.
- **Unregister Entry:** The RIB Manager takes the passed parameters from the incoming request, creates a RIB update, and passes it to the RIB to begin the FIB update process. If the FIB update process is successful, the Route with the same name, `FaceId`, and `origin` is removed from the RIB.

In both cases, the RIB Manager returns a control response with `code 200` if the command is received successfully. Because one RIB update may produce multiple FIB updates, the RIB Manager does not return the result of the command execution since the application of multiple FIB updates may take longer than the `InterestLifetime` associated with the RIB update command. Figure 17 shows the verb dispatch workflow of the RIB manager.

## 8.7 Auto Prefix Propagator

The Auto Prefix Propagator could be enabled by the RIB manager to propagate necessary knowledges of local prefix registrations to the connected gateway router.

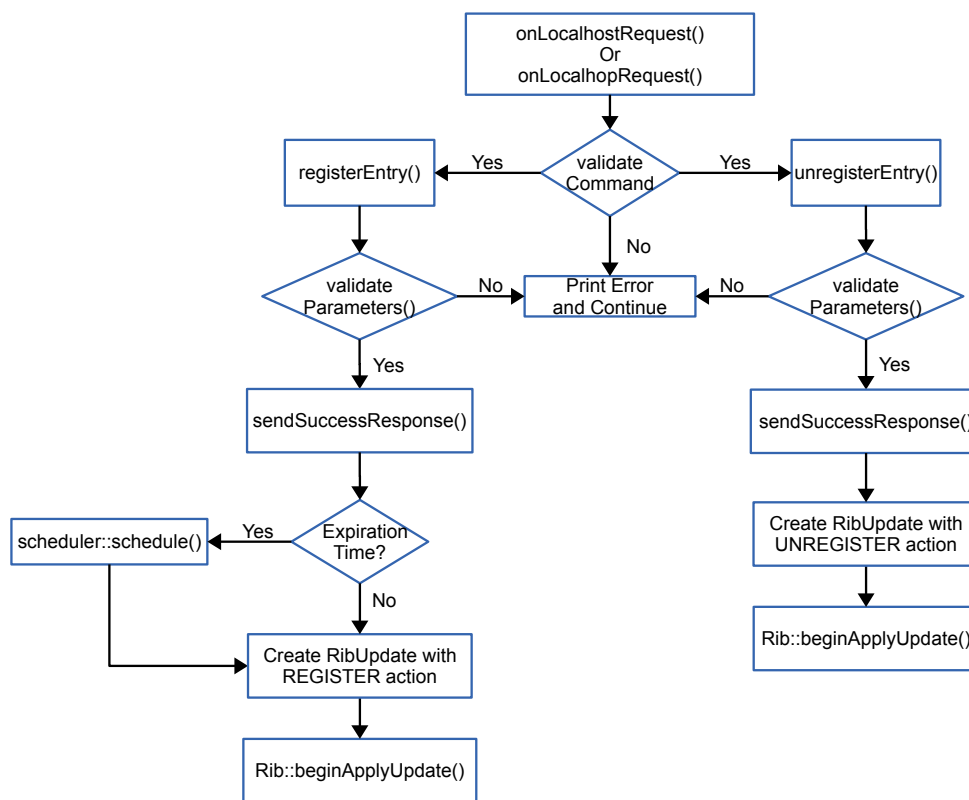


Figure 17: Verb dispatch workflow of the RIB Manager

### 8.7.1 What Prefix to Propagate

To reduce not only the cost of propagations but also increments of the router’s RIB, the propagated prefixes should be aggregated whenever is feasible. Actually, local NRD owns a key-chain consisting of a set of identities, each of which defines a namespace and can cover one or more local RIB entries. Given a RIB entry, the Auto Prefix Propagator queries the local key-chain for signing identities that is authorized to sign a prefix registration command of a prefix of the RIB prefix. If one or more signing identities are found, the identity that can sign a prefix registration command of the shortest prefix is chosen, and the Auto Prefix Propagator will attempt to propagate that shortest prefix a prefix to the router. Figure 18 presents a high level example of prefix propagation.

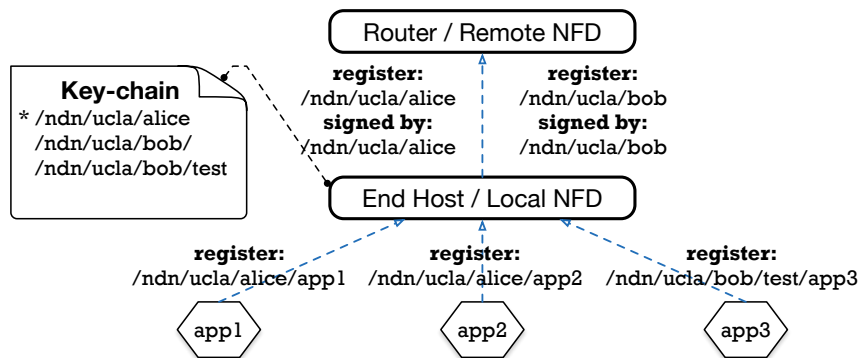


Figure 18: An example of prefix propagation.

If the propagation succeeds, an event is scheduled to propagate the same prefix to refresh this propagation, after a pre-defined duration. By contrast, if the propagation fails, another event is scheduled to propagate the same prefix to retry this propagation, after some waiting period that is calculated based on an exponential back-off strategy.

This revision is **OUTDATED** and preserved for historical purposes only. Check for the most recent revision at <https://named-data.net/publications/techreports/>



### 8.7.2 When to Propagate

The Auto Prefix Propagator monitors RIB insertions and deletions by subscribing two signals, `Rib:afterInsertEntry` and `Rib:afterEraseEntry` respectively. Once those two signals are emitted, two connecting (the connections are established when Auto Prefix Propagator is enabled) methods, `AutoPrefixPropagator::afterInsertRibEntry` and `AutoPrefixPropagator::afterEraseRibEntry` are invoked to process the attempts of propagation and revocation respectively. But not all attempts are necessary and could be performed on the gateway router actually. For propagation, only the RIB prefix, that is not scoped for local use (i.e., starts with `/localhost`) and does not indicate a connectivity to the router (i.e., is the **link local nfd prefix**, `/localhost/nfd`), will lead to a necessary propagation if it has not been processed for another RIB prefix. Further, a necessary propagation could be actually complete on the router only when there is an active connectivity to the router. Similarly, for revocation, only a qualified RIB prefix (not start with `/localhost` and not be the **link local nfd prefix**) can lead to a necessary revocation, as long as there is no other RIB prefix causing the propagation of the same prefix. A actual remote operation will be triggered only when the propagation to be revoked has already succeed and there is active connectivity currently.

Figure 19 demonstrates a simplified workflow of a successful propagation. The process for revocation is similar.

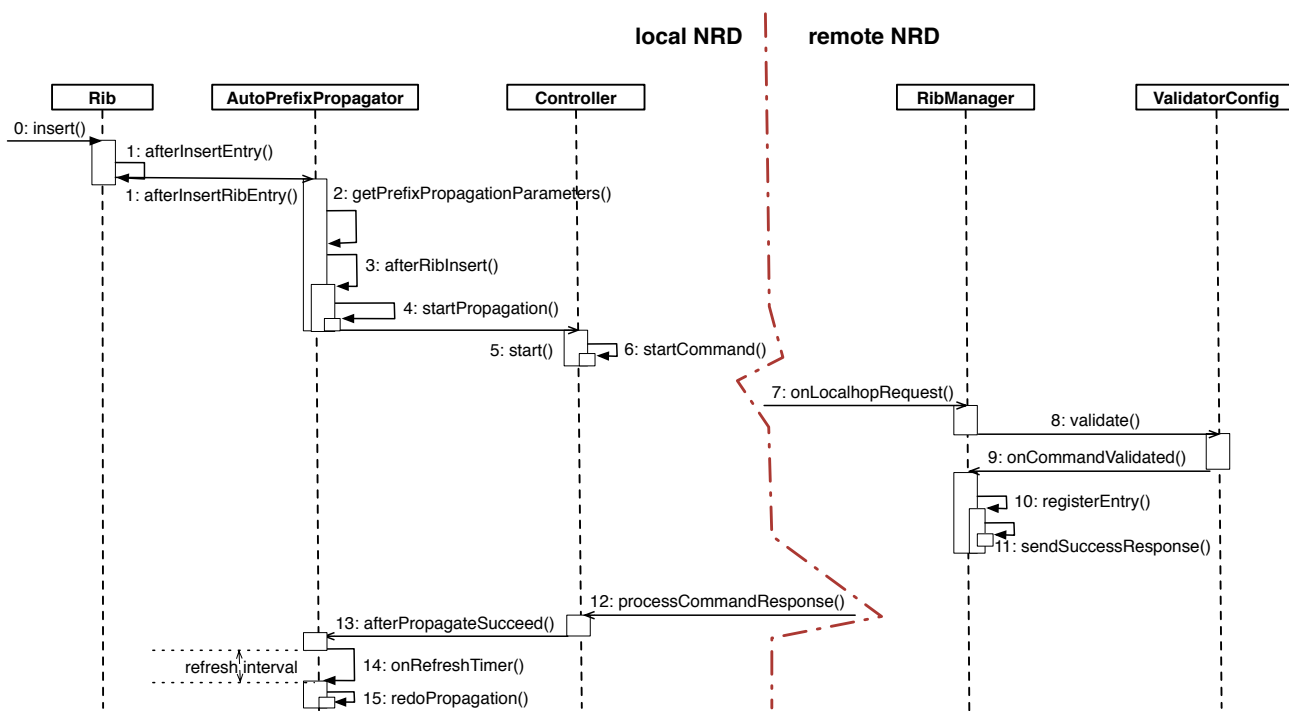


Figure 19: The simplified workflow of a successful propagation.

### 8.7.3 Secure Propagations

To enable gateway router processes remote registrations, the `rib.localhop-security` section must be configured on the router. A series of policies and rules can be defined to validate registration / unregistration commands for propagations / revocations in format.

Besides, according to the trust schema, a command for propagation / revocation can not pass the validation unless its signing identity can be verified by the configured trust anchor on the gateway router.

### 8.7.4 Propagated-entry State Machine

A propagated entry consists of a `PropagationStatus` indicates current state of this entry, as well as an event could be scheduled for either refresh or retry. Besides, it stores a copy of the signing identity for this entry. All propagated entries are maintained as a un-ordered map (`AutoPrefixPropagator::m_propagatedEntries`), where the propagated prefix is used as the key to retrieve the corresponding entry.

More specifically, a propagated entry will stay in one of the following five states in logic.

- **NEW**, the initial state.

- **PROPAGATING**, the state when the corresponding propagation is being processed but the response is not back yet.
- **PROPAGATED**, the state when the corresponding propagation has succeeded.
- **PROPAGATE\_FAIL**, the state when the corresponding propagation has failed.
- **RELEASED**, indicates this entry has been released. It's noteworthy that this state is not required to be explicitly implemented, because it can be easily determined by checking whether an existing entry can still be accessed. Thus, any entry to be released is directly erased from the list of propagated entries.

Given a propagated entry, there are a series of events that can lead to a transition with a state switch from one to another, or some triggered actions, or even both of them. All related input events are listed below.

- **rib insert**, corresponds to `AutoPrefixPropagator::afterRibInsert`, which happens when the insertion of a RIB entry triggers a necessary propagation.
- **rib erase**, corresponds to `AutoPrefixPropagator::afterRibErase`, which happens when the deletion of a RIB entry triggers a necessary revocation.
- **hub connect**, corresponds to `AutoPrefixPropagator::afterHubConnect`, which happens when the connectivity to a router is established (or recovered).
- **hub disconnect**, corresponds to `AutoPrefixPropagator::afterHubDisconnect`, which happens when the connectivity to the router is lost.
- **propagate succeed**, corresponds to `AutoPrefixPropagator::afterPropagateSucceed`, which happens when the propagation succeeds on the router.
- **propagate fail**, corresponds to `AutoPrefixPropagator::afterPropagateFail`, which happens when a failure is reported in response to the registration command for propagation.
- **revoke succeed**, corresponds to `AutoPrefixPropagator::afterRevokeSucceed`, which happens when the revocation of some propagation succeeds on the router.
- **revoke fail**, corresponds to `AutoPrefixPropagator::afterRevokeFail`, which happens when a failure is reported in response to the unregistration command for revocation.
- **refresh timer**, corresponds to `AutoPrefixPropagator::onRefreshTimer`, which happens when the timer scheduled to refresh some propagation is fired.
- **retry timer**, corresponds to `AutoPrefixPropagator::onRetryTimer`, which happens when the timer scheduled to retry some propagation is fired.

A state machine is implemented to maintain and direct transitions according to the input events. Figure 20 lists all related events and the corresponding transitions.

### 8.7.5 Configure Auto Prefix Propagator

When the RIB manager loading configurations from the NFD config file at the section `rib`, the Auto Prefix Propagator will loading its configurations from the sub section `rib.auto_prefix_propagte`. A series of parameters can be configured.

Table 4 presents the common parameters shared by all propagations, some of which are configurable and are actually set or modified in this method when loading configurations.

## 8.8 Extending RIB Manager

The RIB Manager currently supports only two commands: `register` and `unregister`. However, the functionality of the RIB Manager can be extended by introducing more commands. For example, in the current implementation, if a node wants to announce a prefix, it needs to communicate with a specific routing protocol. After RIB manager will define and an interface for that, e.g., `advertise` and `withdraw` commands, the process of announcing and withdrawal of prefixes in routing protocol could become more uniform and simple.

	NEW	PROPAGATING	PROPAGATED	PROPAGATE_FAIL	RELEASED
<b>rib insert</b>	logically IMPOSSIBLE	logically IMPOSSIBLE	logically IMPOSSIBLE	logically IMPOSSIBLE	-> NEW
<b>rib erase</b>	-> RELEASED	-> RELEASED	-> RELEASED start revocation cancel refresh timer	-> RELEASED cancel retry timer	logically IMPOSSIBLE
<b>hub connect</b>	-> PROPAGATING start propagation	logically IMPOSSIBLE	logically IMPOSSIBLE	logically IMPOSSIBLE	RELEASED
<b>hub disconnect</b>	logically IMPOSSIBLE	-> NEW	-> NEW	-> NEW	RELEASED
<b>propagate succeed</b>	logically IMPOSSIBLE	-> PROPAGATED set refresh timer	logically IMPOSSIBLE	logically IMPOSSIBLE	RELEASED start revocation
<b>propagate fail</b>	logically IMPOSSIBLE	-> PROPAGATE_FAIL set retry timer	logically IMPOSSIBLE	logically IMPOSSIBLE	RELEASED
<b>revoke succeed</b>	logically IMPOSSIBLE	PROPAGATING start propagation	-> PROPAGATING start propagation	PROPAGATE_FAIL	RELEASED
<b>revoke fail</b>	logically IMPOSSIBLE	PROPAGATING	PROPAGATED	PROPAGATE_FAIL	RELEASED
<b>refresh timer</b>	logically IMPOSSIBLE	logically IMPOSSIBLE	-> PROPAGATING start propagation	logically IMPOSSIBLE	logically IMPOSSIBLE
<b>retry timer</b>	logically IMPOSSIBLE	logically IMPOSSIBLE	logically IMPOSSIBLE	-> PROPAGATING start propagation	logically IMPOSSIBLE

Figure 20: The transition table of propagated-entry state machine.

Table 4: Shared parameters for prefix propagation.

member variable <sup>a</sup>	default setting	configurable <sup>b</sup>
m_controlParameters	Cost	15
	Origin	ndn::nfd::ROUTE_ORIGIN_CLIENT
	FaceId	0
m_commandOptions	Prefix	/localhop/nfd
	Timeout	10,000 (milliseconds)
m_refreshInterval <sup>c</sup>	25 (seconds)	YES
m_baseRetryWait	50 (seconds)	YES
m_maxRetryWait	3600 (seconds)	YES

<sup>a</sup>these parameters are maintained in some member variables of **AutoPrefixPropagator**.<sup>b</sup>indicates whether this parameter can be configured in the NFD config file.<sup>c</sup>this setting must be less than the idle time of UDP faces whose default setting is 3,600 seconds.

## 9 Security

Security consideration of NFD involves two parts: interface control and trust models.

### 9.1 Interface Control

The default NFD configuration requires superuser privileges to access raw ethernet interfaces and the Unix socket location. Due to the research nature of NFD, users should be aware of the security risks and consequences of running as the superuser.

It is also possible to configure NFD to run without elevated privileges, but this requires disabling ethernet faces and changing the default Unix socket location<sup>12</sup> (both in the NFD configuration file, see Section 10.1). However, such measures may be undesirable (e.g. performing ethernet-related development). As a middle ground, users can also configure an alternate effective user and group id for NFD to drop privileges to when they are not needed. This does not provide any real security benefit over running exclusively as the superuser, but it could potentially buggy code from damaging the system (see Section 10.1).

### 9.2 Trust Model

Different trust models are used to validate command Interests depending on the recipient. Among the four types of commands in NFD, the commands of `faces`, `fib`, and `strategy-choice` are sent to NFD, while `rib` commands are sent to the RIB Manager.

#### 9.2.1 Command Interest

Command Interests are a mechanism for issuing authenticated control commands. Signed commands are expressed in terms of a command Interest's name. These commands are defined to have five additional components after the management namespace: command name, timestamp, random-value, SignatureInfo, and SignatureValue.

```
/signed/interest/name/<timestamp>/<nonce>/<signatureInfo>/<signatureValue>
```

The command Interest components have the following usages:

- `timestamp` is used to protect against replay attack.
- `nonce` is a random value (32 bits) which adds additional assurances that the command Interest will be unique.
- `signatureInfo` encodes a SignatureInfo TLV block.
- `signatureValue` encodes the a SignatureBlock TLV block.

A command interest will be treated as invalid in the following four cases:

- one of the four components above (SignatureValue, SignatureInfo, nonce, and Timestamp) is missing or cannot be parsed correctly;
- the key, according to corresponding trust model, is not trusted for signing the control command;
- the signature cannot be verified with the public key pointed to by the KeyLocator in SignatureInfo;
- the producer has already received a valid signed Interest whose timestamp is equal or later than the timestamp of the received one.

Note that in order to detect the fourth case, the producer needs to maintain a latest timestamp state for each trusted public key<sup>13</sup>. For each trusted public key, the state is initialized as the timestamp of the first valid Interest signed by the key. Afterwards, the state will be updated each time the producer receives a valid command Interest.

Note that there is no state for the first command Interest. To handle this special situation, the producer should check the Interest's timestamp against a proper interval (e.g., 120 seconds):

$$[current\_timestamp - interval/2, current\_timestamp + interval/2].$$

The first Interest is invalid if its timestamp is outside of the interval.

<sup>12</sup>libndn-cxx expects the default Unix socket location, but this can be changed in the library's client.conf configuration file.

<sup>13</sup>Since public key cryptography is used, sharing private keys is not recommended. If private key sharing is inevitable, it is the key owner's responsibility to keep clock synchronized.

### 9.2.2 NFD Trust Model

With the exception of the RIB Manager, NFD uses a simple trust model of associating privileges with NDN identity certificates. There are currently three privileges that can be directly granted to identities: `faces`, `fib`, and `strategy-choice`. New managers can add additional privileges via the `ManagerBase` constructor.

A command Interest is unauthorized if the signer's identity certificate is not associated with the command type. Note that key retrievals are not permitted/performed by NFD for this trust model; an identity certificate is either associated with a privilege (authorized) or not (unauthorized). For details about how to set privileges for each user, please see Section 10 and Section 7.

### 9.2.3 NFD RIB manager Trust Model

RIB manager uses its own trust model to authenticate `rib` type command Interests. Applications that want to register a prefix in NFD (i.e., receive Interests under a prefix) may need to send an appropriate `rib` command Interest. After RIB manager authenticates the `rib` command Interest, RIB manager will issue `fib` command Interests to NFD to set up FIB entries.

Trust model for NFD RIB manager defines the conditions for keys to be trusted to sign `rib` commands. Namely, the trust model must answer two questions:

1. Who are trusted signers for `rib` command Interests?
2. How do we authenticate signers?

Trusted signers are identified by expressing the name of the signing key with a [NDN Regular Expression](#) [19]. If the signing key's name does not match the regular expression, the command Interest is considered to be invalid. Signers are authenticated by a rule set that explicitly specifies how a signing key can be validated via a chain of trust back to a trust anchor. Both Signer identification and authentication can be specified in a configuration file that follows the [Validator Configuration File Format specification](#) [20].

RIB manager supports two modes of prefix registration: `localhost` and `localhop`. In `localhop` mode, RIB manager expects prefix registration requests from applications running on remote machines, (i.e., NFD is running on an access router). When `localhop` mode is enabled, `rib` command Interests are accepted if the signing key can be authenticated along the naming hierarchy back to a (configurable) trust anchor. For example, the trust anchor could be the root key of the NDN testbed, so that any user in the testbed can register prefixes through the RIB manager. Alternatively, the trust anchor could be the key of a testbed site or institution, thus limiting RIB manager's prefix registration to users at that site/institution.

In `localhost` mode, RIB manager expects to receive prefix registration requests from local applications. By default, RIB manager allows any local application to register prefixes. However, the NFD administrator may also define their own access control rules using the same configuration format as the trust model configuration for `localhop` mode.

## 9.3 Local Key Management

NFD runs as a user level application. Therefore, NFD will try to access the keys owned by the user who runs NFD. The information about a user's key storage can be found in a configuration file `client.conf`. NFD will search the configuration file at three places (in order): user home directory (`~/.ndn/client.conf`), `/usr/local/etc/ndn/client.conf`, and `/etc/ndn/client.conf`. Configuration file specify the locator of two modules: Public-key Information Base (PIB) and Trusted Platform Module (TPM). The two modules are paired up. TPM is a secure storage for private keys, while PIB is provide public information about signing keys in the corresponding TPM. NFD will lookup available keys in the database pointed by PIB locator, and send packet signing request to the TPM pointed by TPM locator.

## 10 Common Services

NFD contains several common services to support forwarding and management operations. These services are an essential part of the source code, but are logically separated and placed into the `core/` folder.

In addition to core services, NFD also relies extensively on `libndn-cxx` support, which provides many basic functions such as: packet format encoding/decoding, data structures for management protocol, and security framework. The latter, within the context of NFD, is described in more detail in Section 9.

### 10.1 Configuration File

Many aspects of NFD are configurable through a configuration file, which adopts the Boost INFO format [18]. This format is very flexible and allows any combination of nested configuration structures.

#### 10.1.1 User Info

Currently, NFD defines 6 top level configuration sections: *general*, *tables*, *log*, *face\_system*, *security*, and *rib*.

- **general:** The general section defines various parameters affecting the overall behavior of NFD. Currently, the implementation only allows `user` and `group` parameter settings. These parameters define the effective user and effective group that NFD will run as. Note that using an effective user and/or group is different from just dropping privileges. Namely, it allows NFD to regain superuser privileges at any time. By default, NFD must be initially run with and be allowed to regain superuser privileges in order to access raw ethernet interfaces (Ethernet face support) and create a socket file in the system folder (Unix face support). Temporarily dropping privileges by setting the effective user and group id provides minimal security risk mitigation, but it can also prevent well intentioned, but buggy, code from harming the underlying system. It is also possible to run NFD without superuser privileges, but it requires the disabling of ethernet faces (or proper configuration to allow non-root users to perform privileged operations on sockets) and modification of the Unix socket path for NFD and all applications (see your installed `nfd.conf` configuration file or `nfd.conf.sample` for more details). When applications are built using the `ndn-cxx` library, the Unix socket path for the application can be changed using the `client.conf` file. The library will search for `client.conf` in three specific locations and in the following order:

- `~/ndn/client.conf`
- `/SYSCONFDIR/ndn/client.conf` (by default, `SYSCONFDIR` is `/usr/local/etc`)
- `/etc/ndn/client.conf`

- **tables:** The tables section configures NFD's tables: Content Store, PIT, FIB, Strategy Choice, Measurements, and Network Region. NFD currently supports configuring the maximum Content Store size, per-prefix strategy choices, and network region names:

- `cs_max_packets`: Content Store size limit in number of packets. Default is 65536, which corresponds to about 500 MB, assuming maximum size if 8 KB per Data packet.
- `strategy_choice`: This subsection selects the initial forwarding strategy for each specified prefix. Entries are listed as `<namespace> <strategy-name>` pairs.
- `network_region`: This subsection contains a set of network regions used by the forwarder to determine if an Interest carrying a Link object has reached the producer region. Entries are a list of `<names>`.

- **log:** The log section defines the logger configuration such as the default log level and individual NFD component log level overrides. The log section is described in more detail in the Section 10.2.

- **face\_system:** The face system section fully controls allowed face protocols, channels and channel creation parameters, and enabling multicast faces. Specific protocols may be disabled by commenting out or removing the corresponding nested block in its entirety. Empty sections will result in enabling the corresponding protocol with its default parameters.

Version 0.2.0 of NFD contains the following face protocols:

- **unix:** Unix protocol

This section can contain the following parameter:

- \* `path`: sets the path for Unix socket (default is `/var/run/nfd.sock`)



Note that if the **unix** section is present, the created Unix channel will always be in a “listening” state. Commenting out the **unix** section disables Unix channel creation.

– **udp**: UDP protocol

This section can contain the following parameters:

- \* **port**: sets UDP unicast port number (default is 6363)
- \* **enable\_v4**: controls whether IPv4 UDP channels are enabled (enabled by default)
- \* **enable\_v6**: controls whether IPv6 UDP channels are enabled (enabled by default)
- \* **idle\_timeout**: sets the idle time in seconds before closing a UDP unicast face (default is 600 seconds)
- \* **keep\_alive\_timeout**: sets the interval (seconds) between keep-alive refreshes (default is 25 seconds)
- \* **mcast**: controls whether UDP multicast faces need to be created (enabled by default)
- \* **mcast\_port**: sets UDP multicast port number (default is 56363)
- \* **mcast\_group**: UDP IPv4 multicast group (default is 224.0.23.170)

Note that if the **udp** section is present, the created UDP channel will always be in a “listening” state as UDP is a session-less protocol and “listening” is necessary for all types of face operations.

– **tcp**: TCP protocol

This section can contain the following parameters:

- \* **listen**: controls whether the created TCP channel is in listening mode and creates TCP faces when an incoming connection is received (enabled by default)
- \* **port**: sets the TCP listener port number (default is 6363)
- \* **enable\_v4**: controls whether IPv4 TCP channels are enabled (enabled by default)
- \* **enable\_v6**: controls whether IPv6 TCP channels are enabled (enabled by default)

– **ether**: Ethernet protocol (NDN directly on top of Ethernet, without requiring IP protocol)

This section can contain the following parameters:

- \* **mcast**: controls whether Ethernet multicast faces need to be created (enabled by default)
- \* **mcast\_group**: sets the Ethernet multicast group (default is 01:00:5E:00:17:AA)

Note that the Ethernet protocol only supports multicast mode at this time. Unicast mode will be implemented in future versions of NFD.

– **websocket**: The WebSocket protocol (tunnels to connect from JavaScript applications running in a web browser)

This section can contain the following parameters:

- \* **listen**: controls whether the created WebSocket channel is in listening mode and creates WebSocket faces when incoming connections are received (enabled by default)
- \* **port 9696** ; WebSocket listener port number
- \* **enable\_v4**: controls whether IPv4 WebSocket channels are enabled (enabled by default)
- \* **enable\_v6**: controls whether IPv6 WebSocket channels are enabled (enabled by default)

- **authorizations**: The **authorizations** section provides a fine-grained control for management operations. As described in Section 7, NFD has several managers, the use of which can be authorized to specific NDN users. For example, the creation and destruction of faces can be authorized to one user, management of FIB to another, and control over strategy choice to a third user.

To simplify the initial bootstrapping of NFD, the sample configuration file does not restrict local NFD management operations: any user can send management commands to NFD and NFD will authorize them. However, such configuration should not be used in a production environment and only designated users should be authorized to perform specific management operations.

The basic syntax for the **authorizations** section is as follows. It consists of zero or more **authorize** blocks. Each **authorize** block associates a single NDN identity certificate, specified by the **certfile** parameter, with **privileges** blocks. The **privileges** block defines a list of permissions/managers (one permission per line) that are granted to the user identified by **certfile** defines a file name (relative to the configuration file format) of the NDN certificate. As a special case, primarily for demo purposes, **certfile** accepts value “any”, which denotes any certificate possessed by any user. Note that all managers controlled by the **authorizations** section are local. In other words, all commands start with **/localhost**, which are possible only through local faces (Unix face and TCP face to 127.0.0.1).



**Note for developers:**

The `privileges` block can be extended to support additional permissions with the creation of new managers (see Section 7). This is achieved by deriving the new manager from the `ManagerBase` class. The second argument to the `ManagerBase` constructor specifies the desired permission name.

- **rib:** The `rib` section controls behavior and security parameters for NFD RIB manager. This section can contain two subsections: `localhost_security` and `localhop_security`. The former controls authorizations for registering and unregistering prefixes in RIB from local users (through local faces: Unix socket or TCP tunnel to 127.0.0.1). `localhop_security` defines authorization rules for so called localhop prefix registrations: registration of prefixes on the next hop routers.

Unlike the main `authorizations` section, the `rib` security section uses a more advanced validator configuration, thus allowing a greater level of flexibility in specifying authorizations. In particular, it is possible to specify not only specific authorized certificates, but also indirectly authorized certificates. For more details about validator configuration and its capabilities, refer to Section 9 and [Validator Configuration File Format specification](#) [20].

Similar to the `authorizations` section, the sample configuration file, allows any local user to send register and unregister commands (`localhost_security`) and prohibits remote users from sending registration commands (the `localhop_security` section is disabled). On NDN Testbed hubs, the latter is configured in a way to authorize any valid NDN Testbed user (i.e., a user possessing valid NDN certificate obtained through [ndncert website](#) [21]) to send registration requests for user namespace. For example, a user Alice with a valid certificate `/ndn/site/alice/KEY/.../ID-CERT/...` would be allowed to register any prefixes started with `/ndn/site/alice` on NDN hub.

**10.1.2 Developer Info**

When creating a new management module, it is very easy to make use of the NFD configuration file framework. Most heavy lifting is performed using the `Boost.PropertyTree` [18] library and NFD implements an additional wrapper (`ConfigFile`) to simplify configuration file operations.

1. Define the format of the new configuration section. Reusing an existing configuration section could be problematic, since a diagnostic error will be generated any time an unknown parameter is encountered.
2. The new module should define a callback with prototype `void*(*) (ConfigSection..., bool isDryRun)` that implements the actual processing of the newly defined section. The best guidance for this step is to take a look at the existing source code of one of the managers and implement the processing in a similar manner. The callback can support two modes: dry-run to check validity of the specified parameters, and actual run to apply the specified parameters.

As a general guideline, the callback should be able to process the same section multiple times in actual run mode without causing problems. This feature is necessary in order to provide functionality of reloading configuration file during run-time. In some cases, this requirement may result in cleaning up data structures created during the run. If it is hard or impossible to support configuration file reloading, the callback must detect the reloading event and stop processing it.

3. Update NFD initialization in `daemon/main.cpp` file. In particular, the new management module needs to be created somewhere around `initializeManagement` call, once created the second step callback needs to be added to `ConfigFile` class dispatch. Similar updates should be made to `reload` call in `main.cpp`.

As another general recommendation, do not forget to create proper test cases to check correctness of the new config section processing. This is vital for providing longevity support for the implemented module, as it ensures that parsing follows the specification, even after NFD or the supporting libraries are changed.

**10.2 Basic Logger**

One of the most important core services is the logger. NFD's logger provides support for multiple log levels, which can be configured in the configuration file individually for each module. The configuration file also includes a setting for the default log level that applies to all modules, except explicitly listed.

**10.2.1 User Info**

Log level is configured in the `log` section of the configure file. The format for each configuration setting is a key-value pair, where key is name of the specific module and value is the desired log level. Valid values for log level are:

- **NONE**: no messages
- **ERROR**: show only error messages
- **WARN**: show also warning messages
- **INFO**: show also informational messages (default)
- **DEBUG**: show also debugging messages
- **TRACE**: show also trace messages
- **ALL**: all messages for all log levels (most verbose)

Individual module names can be found in the source code by looking for `NFD_LOG_INIT(<module name>)` statements in `.cpp` files, or using `--modules` command-line option for the `nfd` program. There is also a special `default_level` key, which defines log level for all modules, except explicitly specified (if not specified, `INFO` log level is used).

### 10.2.2 Developer Info

To enable NFD logging in a new module, very few actions are required from the developer:

- include `core/logger.hpp` header file
- declare logging module using `NFD_LOG_INIT(<module name>)` macros
- use `NFD_LOG_<LEVEL>(statement to log)` in the source code

The effective log level for unit testing is defined in `unit-tests.conf` (see sample `unit-tests.conf.sample` file) rather than the normal `nfd.conf`. `unit-tests.conf` is expected under the top level NFD directory (i.e. same directory as the sample file).

## 10.3 Hash Computation Routines

Common services also include several hash functions, based on city hash algorithm [13], to support fast name-based operations. Since efficient hash table index size depends on the platform, NFD includes several versions, for 16-bit, 32-bit, 64-bit, and 128-bit hashing.<sup>14</sup>

Name tree implementation generalizes the platform-dependent use of hash functions using a template-based helper (see `computeHash` function in `daemon/tables/name-tree.cpp`). Depending on the size of `size_t` type on the platform, the compiler will automatically select the correct version of the hash function.

Other hash functions may be included in the future to provide tailored implementations for specific usage patterns. In other words, since the quality of the hash function is usually not the sole property of the algorithm, but also relies on the hashed source (hash functions need to hash uniformly into the hash space), depending on which Interest and Data names are used, other hash functions may be more appropriate. Cryptographic hash functions are also an option, however they are usually prohibitively expensive.

### 10.4 Global Scheduler

The `ndn-cxx` library includes a scheduler class that provides a simple way to schedule arbitrary events (callbacks) at arbitrary time points. Normally, each module/class creates its own scheduler object. An implication of this is that a scheduled object, when necessary, must be cancelled in a specific scheduler, otherwise the behavior is undefined.

NFD packet forwarding has a number of events with shared ownership of events. To simplify this and other event operations, common services include a global scheduler. To use this scheduler, one needs to include `core/scheduler.hpp`, after which new events can be scheduled using the `scheduler::schedule` free function. The scheduled event can then be cancelled at any time by calling the `scheduler::cancel` function with the event id that was originally returned by `scheduler::schedule`.

<sup>14</sup>Even though the performance is not a primary goal for the current implementation, we tried to be as much efficient as possible within the developed framework.

## 10.5 Global IO Service

The NFD packet forwarding implementation is based on Boost.Asio [9], which provides efficient asynchronous operations. The main feature of this is the `io_service` abstraction. `io_service` implements the dispatch of any scheduled events in an asynchronous manner, such as sending packets through Berkeley sockets, processing received packets and connections, and many others including arbitrary function calls (e.g., scheduler class in `ndn-cxx` library is fully based on `io_service`).

Logically, `io_service` is just a queue of callbacks (explicitly or implicitly added). In order to actually execute any of these callback functions, at least one processing thread should be created. This is accomplished by calling the `io_service::run` method. The execution thread that called the `run` method then becomes such an execution thread and starts processing enqueued callbacks in an application-defined manner. Note that any exceptions that will be thrown inside the enqueued callbacks can be intercepted in the processing thread that called the `run` method on `io_service` object.

The current implementation of NFD uses a single global instance of `io_service` object with a single processing thread. This thread is initiated from the main function (i.e., main function calls `run` method on the global `io_service` instance).

In some implementations of new NFD services, it may be required to specify a `io_service` object. For example, when implementing TCP face, it is necessary to provide an `io_service` object as a constructor parameter to `boost::asio::ip::tcp::socket`. In such cases, it is enough to include `core/global-io.hpp` header file and supply `getGlobalIoService()` as the argument. The remainder will be handled by the existing NFD framework.

## 10.6 Privilege Helper

When NFD is run as a super user (may be necessary to support Ethernet faces, enabling TCP/UDP/WebSocket faces on privileged ports, or enabling Unix socket face in a root-only-writeable location), it is possible to run most of the operations in unprivileged mode. In order to do so, NFD includes a PrivilegeHelper that can be configured through configuration file to drop privileges as soon as NFD initialization finishes. When necessary, NFD can temporarily regain privileges to do additional tasks, e.g., (re-)create multicast faces.

## 11 Testing

In general, software testing consists of multiple testing levels. The levels that have been majorly supported during the NFD development process include unit and integration tests.

At the *unit test level*, individual units of source code (usually defined in a single `.cpp` file) are tested for functional correctness. Some of the developed unit tests in NFD involve multiple modules and interaction between modules (e.g., testing of forwarding strategies). However, even in these cases, all testing is performed internally to the module, without any external interaction.

NFD also employs *integration testing*, where NFD software and its components is evaluated as a whole in a controlled networking environment.

### 11.1 Unit Tests

NFD uses Boost Test Library [22] to support development and execution of unit testing.

#### 11.1.1 Test Structure

Each unit test consists of one or more test suites, which may contain one or more locally defined classes and/or attributes and a number of test cases. A specific test case should test a number of the functionalities implemented by a NFD module. In order to reuse common functions among the test suites of a test file or even among multiple test files, one can make use of the fixture model provided by the Boost Test Library.

For example, the `tests/daemon/face/face.t.cpp` file, which tests the correctness of the `daemon/face/face.hpp` file, contains a single test suite with a number of test cases and a class that extends the `DummyFace` class. The test cases check the correctness of the functionalities implemented by the `daemon/face/face.hpp` file (e.g., Local Control Header feature and face connection to a local application).

#### 11.1.2 Running Tests

In order to run the unit tests, one has to configure and compile NFD with the `--with-tests` parameter. Once the compilation is done, one can run all the unit tests by typing:

```
./build/unit-tests
```

One can run a specific test case of a specific test suite by typing:

```
./build/unit-tests -t <TestSuite>/<TestCase>
```

#### 11.1.3 Test Helpers

`ndn-cxx` library provides a number of helper tools to facilitate development of unit tests:

- **DummyClientFace** (`<ndn-cxx/util/dummy-client-face.hpp>`): a socket-independent `Face` abstraction to be used during the unit testing process;
- **UnitTestSystemClock** and **UnitTestSteadyClock**: abstractions to mock system and steady clocks used inside `ndn-cxx` and NFD implementation.

In addition to library tools, NFD unit test environment also includes a few NFD-specific common testing elements:

- **LimitedIo** (`tests/limited-io.hpp`): class to start/stop IO operations, including operation count and/or time limit for unit testing;
- **IdentityManagementFixture**: a test suite level fixture that can be used fixture to create temporary identities. Identities added via `IdentityManagementFixture::addIdentity` method are automatically deleted during test teardown.
- **StrategyTester** (`tests/daemon/fw/strategy-tester.hpp`): a framework to test a forwarding strategy. This helper extends the tested strategy to offer recording of its invoked actions, without passing them to the actual forwarder implementation.
- **TopologyTester** (`tests/daemon/fw/topology-tester.hpp`): a framework to construct a virtual mock topology and implement various network events (e.g., failing and recovering links). The purpose is to test the forwarder and an implemented forwarding strategy across this virtual topology.

### 11.1.4 Test Code Guidelines and Naming Conventions

The current implementation of NFD unit tests uses the following naming convention:

- A test suite for the `folder/module-name.hpp` file should be placed in `tests/folder/module-name.t.cpp`. For example, the test suite for the `daemon/fw/forwarder.hpp` file should be placed in `tests/daemon/fw/forwarder.t.cpp`.
- A test suite should be named as `FolderModuleName`. For example, the test suite for the `daemon/fw/forwarder.hpp` file should be named `FwForwarder`. The only exception is a test suite for a header in a top-level folder. This test suite should be named as `TestModuleName`, so that the unqualified `ModuleName` still refers to the tested type. In this way, the test suite for the `src/name.hpp` file should be named `TestName`.
- A test suite should be declared in the same namespace of the tested type plus additional tests namespace. For example, a test suite for the `nfd::Forwarder` class is declared in the namespace `nfd::tests`, while a test suite for the `nfd::cs::Cs` class is declared in the `nfd::cs::tests` namespace. We should note that if needed, the sub-namespace of a parent's tests can be imported with the "using namespace" directive.
- A test suite should use the `nfd::tests::BaseFixture` fixture to get automatic setup and teardown of global `io_service`. If a custom fixture is defined for a test suite or a test case, this custom fixture should derive from the `BaseFixture`. Moreover, the `IdentityManagementFixture` should be used if a test case needs to create identities in `KeyChain`. Last but not least, the `UnitTestFixture` should be used if mocked clocks are feasible/desirable.

## 11.2 Integration Tests

NFD team has developed a number of integrated test cases (<http://gerrit.named-data.net/#/admin/projects/NFD/integration-tests>) that can be run in a dedicated test environment. We are currently refactoring the way integrated tests are developed and executed. Detailed description of the new process will be available in the next versions of the current guide.

## References

- [1] NDN Project Team, “NDN packet format specification (version 0.1),” <http://named-data.net/doc/ndn-tlv/>, 2014.
- [2] —, “NFD - Named Data Networking Forwarding Daemon,” <http://named-data.net/doc/NFD/current/>, 2014.
- [3] —, “NFD management protocol,” <http://redmine.named-data.net/projects/nfd/wiki/Management>, 2014.
- [4] —, “Control command,” <http://redmine.named-data.net/projects/nfd/wiki/ControlCommand>, 2014.
- [5] J. Shi, “Ndnlpv2,” <http://redmine.named-data.net/projects/nfd/wiki/NDNLPv2>.
- [6] D. Katz and D. Ward, “Bidirectional Forwarding Detection (BFD),” RFC 5880 (Proposed Standard), Internet Engineering Task Force, Jun. 2010. [Online]. Available: <http://www.ietf.org/rfc/rfc5880.txt>
- [7] B. Adamson, C. Bormann, M. Handley, and J. Macker, “Multicast Negative-Acknowledgment (NACK) Building Blocks,” RFC 5401 (Proposed Standard), Internet Engineering Task Force, Nov. 2008. [Online]. Available: <http://www.ietf.org/rfc/rfc5401.txt>
- [8] I. Fette and A. Melnikov, “The WebSocket protocol,” RFC 6455 <http://tools.ietf.org/html/rfc6455>, December 2011.
- [9] C. Kohlhoff, “Boost.Asio,” [http://www.boost.org/doc/libs/1\\_48\\_0/doc/html/boost\\_asio.html](http://www.boost.org/doc/libs/1_48_0/doc/html/boost_asio.html), 2003–2013.
- [10] NDN Project Team, “NFD Local Control Header,” <http://redmine.named-data.net/projects/nfd/wiki/LocalControlHeader>, 2014.
- [11] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, “Networking named content,” in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT ’09. New York, NY, USA: ACM, 2009, pp. 1–12. [Online]. Available: <http://doi.acm.org/10.1145/1658939.1658941>
- [12] J. Shi, “Namespace-based scope control,” <http://redmine.named-data.net/projects/nfd/wiki/ScopeControl>.
- [13] Google, “The CityHash family of hash functions,” <https://code.google.com/p/cityhash/>, 2011.
- [14] A. Afanasyev, J. Shi, Y. Yu, and S. DiBenedetto, *ndn-cxx: NDN C++ library with eXperimental eXtensions: Library and Applications Developer’s Guide*. NDN Project (named-data.net), 2015.
- [15] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, “A case for stateful forwarding plane,” *Computer Communications*, vol. 36, no. 7, pp. 779–791, 2013, iSSN 0140-3664. [Online]. Available: <http://dx.doi.org/10.1016/j.comcom.2013.01.005>
- [16] J. Shi, “ccnd 0.7.2 forwarding strategy,” <http://redmine.named-data.net/projects/nfd/wiki/CcndStrategy>, University of Arizona, Tech. Rep., 2014.
- [17] A. Afanasyev, P. Mahadevan, I. Moiseenko, E. Uzun, and L. Zhang, “Interest flooding attack and countermeasures in Named Data Networking,” in *Proc. of IFIP Networking 2013*, May 2013. [Online]. Available: <http://networking2013.poly.edu/program-2/>
- [18] M. Kalicinski, “Boost.PropertyTree,” [http://www.boost.org/doc/libs/1\\_48\\_0/doc/html/property\\_tree.html](http://www.boost.org/doc/libs/1_48_0/doc/html/property_tree.html), 2008.
- [19] Y. Yu, “NDN regular expression,” <http://redmine.named-data.net/projects/ndn-cxx/wiki/Regex>, 2014.
- [20] —, “Validator configuration file format,” <http://redmine.named-data.net/projects/ndn-cxx/wiki/CommandValidatorConf>, 2014.
- [21] NDN Project Team, “NDN-Cert,” <https://github.com/named-data/ndncert>, 2014.
- [22] G. Rozental, “Boost test library,” [http://www.boost.org/doc/libs/1\\_48\\_0/libs/test/doc/html/index.html](http://www.boost.org/doc/libs/1_48_0/libs/test/doc/html/index.html), 2007.