NFD Developer's Guide

Alexander Afanasyev[1], Junxiao Shi[2], Beichuan Zhang[2], Lixia Zhang[1], Ilya Moiseenko[1], Yingdi Yu[1], Wentao Shang[1], Yanbiao Li[1], Spyridon Mastorakis[1], Yi Huang[2], Jerald Paul Abraham[2], Eric Newberry[2], Teng Liang[2], Klaus Schneider[2], Steve DiBenedetto[3], Chengyu Fan[3], Susmit Shannigrahi[3], Christos Papadopoulos[3], Davide Pesavento[4], Giulio Grassi[4], Giovanni Pau[4], Hang Zhang[5], Tian Song[5], Haowei Yuan[6], Hila Ben Abraham[6], Patrick Crowley[6], Syed Obaid Amin[7], Vince Lehman[7], Muktadir Chowdhury[7], Ashlesh Gawande[7], Lan Wang[7], and Nicholas Gordon[7]

[1]University of California, Los Angeles
[2]The University of Arizona
[3]Colorado State University
[4]University Pierre & Marie Curie, Sorbonne University
[5]Beijing Institute of Technology
[6]Washington University in St. Louis
[7]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 10 (July 4, 2018)**:
  - Description of EthernetUnicast transport
  - Updates regarding Transport MTU handling
- **Revision 9 (May 4, 2018)**:
  - Updated description of congestion control
  - Updates in forwarding pipelines and strategy description
- **Revision 8 (February 19, 2018)**:
  - Updated description of face system
  - Interface whitelist and blacklist for multicast faces
  - TCP permanent face
  - IPv6 support in MulticastUdpTransport
  - New *ad hoc* link type
  - Content Store policy configuration and policy API
  - Unsolicited data policy
  - Forwarding pipeline updates, including semantics of removing Link from Interest when it reaches producer region
  - Description of new semantics of NextHopFaceId
  - Scope control in strategies
  - Strategy parameters
  - Updated description of multicast strategy
  - Command Authenticator

  - Updated face management to match current NFD implementation
  - RIB-to-NLSR readvertise
  - New section on Congestion Control

- **Revision 7 (October 4, 2016)**:
  - Added brief description and reference to the new Adaptive SRTT-based (ASF) forwarding strategy
  - Update description of Strategy API to reflect latest changes
  - Miscellaneous updates

- **Revision 6 (March 25, 2016)**:
  - Added description of refactored Face system (Face, LinkService, Transport)
  - Added description of WebSocket transport
  - Updated description of RIB management
  - Added description of Nack processing
  - Added introductory description of NDNLP
  - Added description of best-route retransmission suppression
  - Other updates to synchronize description with current NFD implementation

- **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 encapsulationg of NDN packets inside WebSocket messages
  - Revised FaceManager description, requiring canonical FaceUri in create operations
  - Added description of the new access router strategy

- **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

1 Introduction **6**
1.1 NFD Modules . . . 6
1.2 How Packets are Processed in NFD . . . 7
1.3 How Management Interests are Processed in NFD . . . 8

2 Face System **9**
2.1 Face . . . 9
2.2 Transport . . . 10
2.2.1 Internal Transport . . . 11
2.2.2 Unix Stream Transport . . . 11
2.2.3 Ethernet Unicast Transport . . . 11
2.2.4 Ethernet Multicast Transport . . . 12
2.2.5 UDP Unicast Transport . . . 13
2.2.6 UDP Multicast Transport . . . 13
2.2.7 TCP Transport . . . 13
2.2.8 WebSocket Transport . . . 14
2.2.9 Developing a New Transport . . . 14
2.3 Link Service . . . 14
2.3.1 Generic Link Service . . . 15
2.3.2 Vehicular Link Service . . . 16
2.3.3 Developing a New Link Service . . . 16
2.4 `FaceSystem`, `ProtocolFactory`, and `Channel` Classes . . . 16
2.4.1 Initialization . . . 18
2.4.2 Inside a ProtocolFactory . . . 18
2.4.3 Outgoing Unicast Face Creation . . . 18
2.4.4 Multicast Face Creation . . . 18
2.5 NIC-associated Permanent Faces . . . 19
2.6 NDNLP . . . 19

3 Tables **21**
3.1 Forwarding Information Base (FIB) . . . 21
3.1.1 Structure and Semantics . . . 21
3.1.2 Usage . . . 22
3.2 Network Region Table . . . 22
3.3 Content Store (CS) . . . 22
3.3.1 Semantics and Usage . . . 22
3.3.2 Lookup Table . . . 23
3.3.3 Cache Replacement Policy . . . 23
3.4 Interest Table (PIT) . . . 24
3.4.1 PIT Entry . . . 24
3.4.2 PIT . . . 26
3.5 Dead Nonce List . . . 26
3.5.1 Structure, Semantics, and Usage . . . 26
3.5.2 Capacity Maintenance . . . 26
3.6 Strategy Choice Table . . . 27
3.6.1 Structure and Semantics . . . 27
3.6.2 Usage . . . 28
3.7 Measurements Table . . . 28
3.7.1 Structure . . . 28
3.7.2 Usage . . . 28
3.8 NameTree . . . 29
3.8.1 Structure . . . 29
3.8.2 Operations and Algorithms . . . 30
3.8.3 Shortcuts . . . 30

4 Forwarding **32**
4.1 Forwarding Pipelines . . . 32
4.2 Interest Processing Path . . . 32
4.2.1 Incoming Interest Pipeline . . . 33
4.2.2 Interest Loop Pipeline . . . 34
4.2.3 ContentStore Hit Pipeline . . . 34
4.2.4 ContentStore Miss Pipeline . . . 34
4.2.5 Outgoing Interest Pipeline . . . 35
4.2.6 Interest Finalize Pipeline . . . 35
4.3 Data Processing Path . . . 35
4.3.1 Incoming Data Pipeline . . . 36
4.3.2 Data Unsolicited Pipeline . . . 37
4.3.3 Outgoing Data Pipeline . . . 37
4.4 Nack Processing Path . . . 37
4.4.1 Incoming Nack Pipeline . . . 37
4.4.2 Outgoing Nack Pipeline . . . 38
4.5 Helper Algorithms . . . 38
4.5.1 FIB lookup . . . 38

5 Forwarding Strategy **39**
5.1 Strategy API . . . 39
5.1.1 Triggers . . . 39
5.1.2 Actions . . . 41
5.1.3 Storage . . . 41
5.2 List of Strategies . . . 41
5.2.1 Best Route Strategy . . . 42
5.2.2 Multicast Strategy . . . 43
5.2.3 NCC Strategy . . . 43
5.2.4 Access Router Strategy . . . 43
5.2.5 ASF Strategy . . . 45
5.3 How to Develop a New Strategy . . . 45
5.3.1 Should I Develop a New Strategy? . . . 45
5.3.2 Develop a New Strategy . . . 46

6 Management **48**
6.1 Protocol Overview . . . 48
6.2 Dispatcher and Authenticator . . . 48
6.2.1 Manager Base . . . 49
6.2.2 Command Authenticator . . . 49
6.3 Forwarder Status . . . 50
6.4 Face Management . . . 50
6.4.1 Command Processing . . . 50
6.4.2 Datasets and Event Notification . . . 51
6.5 FIB Management . . . 51
6.6 Strategy Choice Management . . . 52
6.7 Configuration Handlers . . . 52
6.7.1 General Configuration File Section Parser . . . 52
6.7.2 Tables Configuration File Section Parser . . . 52
6.8 How to Extend NFD Management . . . 52

7 RIB Management **53**
7.1 Initializing the RIB Manager . . . 55
7.2 Command Processing . . . 55
7.3 FIB Updater . . . 56
7.3.1 Route Inheritance Flags . . . 57
7.3.2 Cost Inheritance . . . 57
7.4 RIB Status Dataset . . . 57

7.5 Readvertise . . . 57
7.5.1 ReadvertisePolicy . . . 58
7.5.2 ReadvertiseDestination . . . 58
7.6 Auto Prefix Propagator . . . 58
7.6.1 What Prefix to Propagate . . . 59
7.6.2 When to Propagate . . . 59
7.6.3 Secure Propagations . . . 59
7.6.4 Propagated-entry State Machine . . . 59
7.6.5 Configure Auto Prefix Propagator . . . 61
7.7 Extending RIB Manager . . . 61

8 Congestion Control **63**
8.1 Congestion Detection . . . 63
8.2 Congestion Signaling . . . 63
8.3 Consumer Adaptation . . . 63

9 Security **64**
9.1 Interface Control . . . 64
9.2 Trust Model . . . 64
9.2.1 Command Interest . . . 64
9.2.2 NFD Trust Model . . . 65
9.2.3 NFD RIB manager Trust Model . . . 65
9.3 Local Key Management . . . 65

10 Common Services **66**
10.1 Configuration File . . . 66
10.1.1 User Info . . . 66
10.1.2 Developer Info . . . 68
10.1.3 Configuration Reload . . . 69
10.2 Basic Logger . . . 69
10.2.1 User Info . . . 69
10.2.2 Developer Info . . . 69
10.3 Hash Computation Routines . . . 70
10.4 Global Scheduler . . . 70
10.5 Global IO Service . . . 70
10.6 Privilege Helper . . . 70

11 Testing **71**
11.1 Unit Tests . . . 71
11.1.1 Test Structure . . . 71
11.1.2 Running Tests . . . 71
11.1.3 Test Helpers . . . 71
11.1.4 Test Code Guidelines and Naming Conventions . . . 72
11.2 Integration Tests . . . 72

References **73**

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.

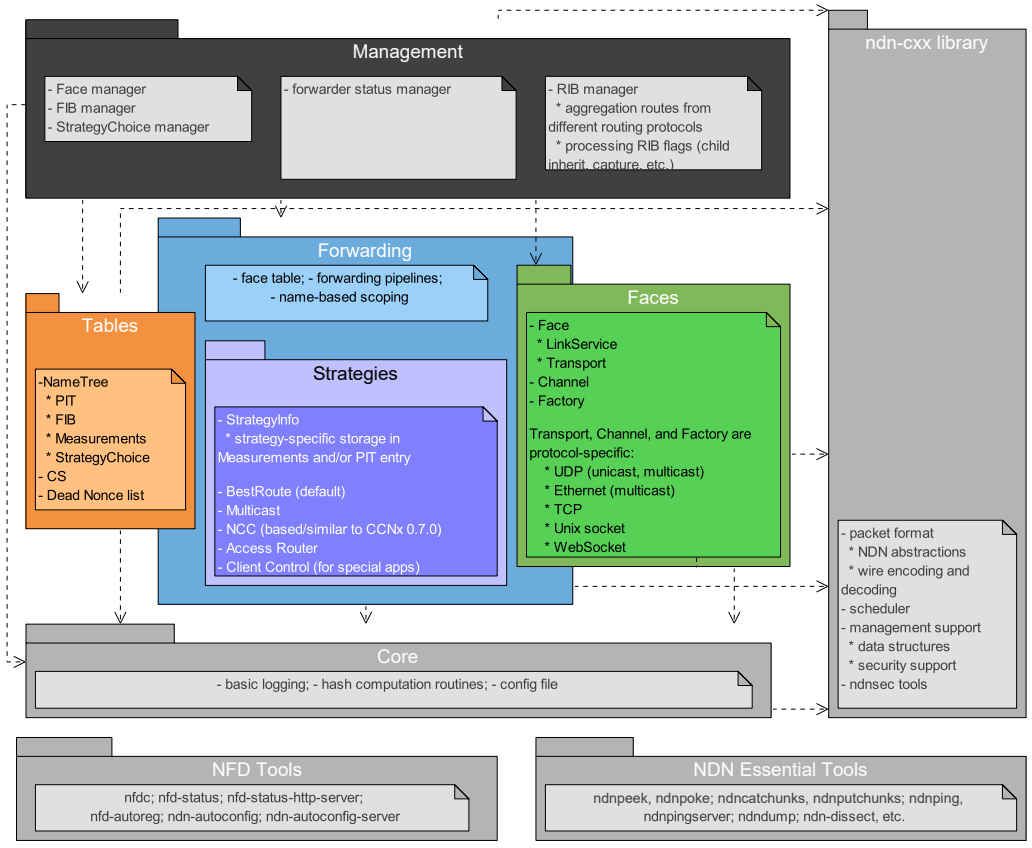


Figure 1: Overview of NFD modules

- **Tables** (Section 3)

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

- **Forwarding** (Section 4)

  Implements basic packet processing pathways, which interact with Faces, Tables, and Strategies (Section 5).

  Strategies are a major part of the forwarding module. The forwarding module implements a framework to support different forwarding strategies in the form of forwarding pipelines, described in detail in Section 4.

- **Management** (Section 6)

  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 7)

  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 an overlay tunnel (where NDN operates as an overlay above TCP, UDP, or WebSocket). In addition, communication between NFD and a local application can be done via a Unix-domain socket, which is also a Face. A Face is composed of a *LinkService* and a *Transport*. The *LinkService* provides high-level services for the Face, like fragmentation and reassembly, network-layer counters, and failure detection, while the *Transport* acts as an wrapper over an underlying network transmission protocol (TCP, UDP, Ethernet, etc.) and provides services like link-layer counters. The Face reads incoming stream or datagrams via the operating system API, extracts network-layer packets from link protocol packets, and delivers these network-layer packets (NDN packet format Interests, Datas, or Nacks) to the forwarding.

A network-layer packet (Interest, Data, or Nack) 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*.

When the forwarder receives an Interest packet, the incoming Interest is first inserted into the *Interest table* (PIT), where each entry represents a pending Interest or a recently satisfied Interest. A lookup for a matching Data 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, fragments the Interest if necessary, encapsulates the network-layer packet(s) in one or more link-layer packets, and sends the link-layer packets as an outgoing stream or datagrams via the operating system APIs.

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. Otherwise, the Data is added to the Content Store. Forwarding strategy that is responsible for each of the matched 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.

When the forwarder receives a Nack, the processing varies depending upon the *forwarding strategy* in use.

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 a control command request, it directs that request to a special Face, called the *Internal Face*.[1] When a request 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 6). The manager then looks at the request name to decide which action is requested. If the name refers to a valid control command, the dispatcher validates the command (checks the signature and validates whether the requester is authorized to send this command), and the manager 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 7), which is performed in a separate thread. All RIB Management control commands, instead of Internal Face, are forwarded toward the RIB thread using the same methods as forwarding to any local application (the RIB thread "registers" itself with NFD for the RIB management prefix when it starts).

A **status dataset** is a dataset containing an internal NFD status that is generated either periodically or on-demand (e.g., NFD general status or NFD Face 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 in that they can be accessed by anybody using unsigned Interests, but operate differently. Subscribers that want to receive notification streams still send Interests directed towards the designated manager. However, these Interests are dropped by the dispatcher and are not forwarded to the manager. Instead, whenever a notification is generated, the manager puts a Data packet into the forwarding, satisfying all outstanding notification stream Interests, and the notification is delivered to all subscribers. 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.

[1] There is always a FIB entry for the management protocol prefix that points to the Internal Face.

2 Face System

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. Section 2.6 introduces NDN Link Protocol (NDNLPv2), a link adaptation protocol implemented in face system.

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 *point-to-point*, *multi-access*, or *ad hoc*. 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** provide statistics about the count and size of Interests, Data, Nacks, and lower layer packets sent and received on the face.

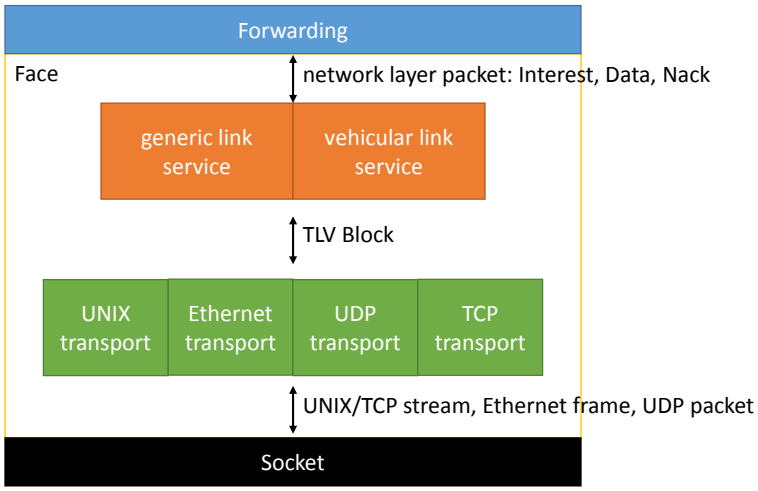


Figure 2: Face = LinkService + Transport

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 link service contains a fragmenter and a reassembler to allow it to perform fragmentation and reassembly.

The face is implemented as `nfd::face::Face` class. 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.

Received and sent packets pass through both the transport and the link service before they are passed to forwarding or are sent on the link, respectively. When packets are received by the transport, they are passed to the link service by calling the `LinkService::receivePacket` function. When a packet is sent through the face, it is first passed to the link service through a function specific to the packet type (`Face::sendInterest`, `Face::sendData`, or `Face::sendNack`). Once the packet has been processed, it is passed (or, if it has been fragmented, its fragments are passed) to the transport by calling the `Transport::send` function. Within the link service and transport, remote endpoints are identified using a remote endpoint id (`Transport::EndpointId`), which is a 64-bit unsigned integer that contains a protocol-specific unique identifier for each remote host.

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. Currently, this attribute can only be set during creation of the face and read-only afterwards. The transport may change the value of this attribute at any time, the link service realization 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 Internal Transport

Internal transport (`nfd::face::InternalForwarderTransport`) is a transport that pairs with an internal client-side transport (`nfd::face::InternalClientTransport`). Packets transmitted on the internal forwarder-side transport is received on the paired client-side transport, and vice versa. This is mainly used to communicate with NFD management; this is also used to implement `TopologyTester` (Section 11.1.3) for unit testing.

2.2.2 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 (for all relevant face commands, NFD returns actual MTU)

`UnixStreamTransport` is derived from `StreamTransport`, a transport which is used for all stream-based transports, including Unix stream and TCP. Most of the functionality of `UnixStreamTransport` is handled by `StreamTransport`. As such, when a `UnixStreamTransport` receives a packet that exceeds the maximum packet size or is not valid, the transport enters a failed state and closes.

Received data is stored in a buffer. The current amount of data in the buffer is stored. Upon every receive, the transport processes all valid packets in the buffer. Then, the transport copies any remaining octets to the beginning of the buffer and waits for more octets, appending the new octets to the end of the existing octets.

2.2.3 Ethernet Unicast Transport

Ethernet unicast transport (`nfd::face::UnicastEthernetTransport`) is a transport that communicates directly between two endpoints over Ethernet.

NFD listens for incoming Ethernet frames via `EthernetChannel` on each Ethernet-compatible physical interface (i.e., excluding loopback interface, point-to-point links, GRE tunnels, etc.). A `UnicastEthernetTransport` is created for each new remote endpoint: either explicitly using `face/create` command from the management interface or implicitly when receiving a packet from a new remote endpoint.

The static attributes of an Ethernet unicast transport are:

- **LocalUri** `dev://`*ifname* where *ifname* is the network interface name; e.g., `dev://eth0`
- **RemoteUri** `ether://[`*ethernet-addr*`]` where *ethernet-addr* is the remote unicast endpoint; e.g., `ether://[01:00:5e:00:17:`
- **Scope** non-local
- **Persistency** on-demand for transports created from incoming connections, persistent or permanent for transports created for outgoing connections (can be changed using `face/update` management command)
- **LinkType** point-to-point
- **Mtu** MTU of the network interface (currently, can only be set during the face creation)

`UnicastEthernetTransport` uses a libpcap handle to transmit and receive on an Ethernet link. The handle is initialized by activating onto an interface. Then, the link-layer header format is set to EN10MB and libpcap is set to only capture incoming packets. After the handle is initialized, the transport sets a packet filter to only capture packets with the NDN type (0x8624) that are from the remote endpoint address and destined for the local interface address.

The libpcap handle is integrated with Boost.Asio through the `async_read_some` function by calling the pcap read functions in the read handler. If an oversized or invalid packet is received, it is dropped.

The Ethernet unicast transport relies on NDNLPv2 fragmentation and reassembly to fragment packets to fit the MTU of the link (if needed) and reassemble any received fragmented packets, respectively.

Outgoing frames that are too small will be padded with zeros to meet the minimum Ethernet packet size. Ethernet unicast transports with an on-demand persistency will time out if a non-zero idle timeout is set.

2.2.4 Ethernet Multicast Transport

Multicast Ethernet transport (`nfd::face::MulticastEthernetTransport`) is a transport that communicates via multicast directly on Ethernet.

NFD automatically creates multicast Ethernet transports on multicast-capable network interfaces during initialization.

In `face_system.ether` config section, a whitelist+blacklist can be specified to create multicast Ethernet transports on a subset of available network interfaces. The whitelist and the blacklist can each contain, in no particular order:

- network interface names, e.g., `ifname eth0`
- Ethernet MAC addresses, e.g., `ether 85:3b:4d:d3:5f:c2`
- IPv4 subnets, e.g., `subnet 192.168.2.0/24`
- a single asterisk (`*`), that matches all network interfaces on the system

IPv6 address/prefix declarations are not yet supported. Interface names can also be expressed as Unix shell-style wildcard patterns, also known as glob patterns. An example of such a wildcard pattern would be: `ifname enp?48*`. Note that these patterns are different from regular expressions. All interfaces are whitelisted by default.

Setting `face_system.ether.mcast` to "no" disables Ethernet multicast transports altogether.

The multicast group is specified on `face_system.ether.mcast_group` option in NFD configuration file. All NDN hosts on the same Ethernet segment must be configured with the same multicast group in order to communicate with each other; therefore, it's recommended to keep the default multicast group setting.

Ethernet multicast transport also supports *ad hoc* networks. This can be configured via the `face_system.ether.mcast_ad_hoc` option in NFD configuration file. Setting this option to "yes" changes the link type of the transport to *ad hoc*, which allows Interest and Data to be forwarded to the incoming face. By default, the option is set to "no", meaning that created transports will have a multi-access link type.

The static attributes of an Ethernet multicast transport are:

- **LocalUri** `dev://`*ifname* where *ifname* is network interface name; e.g., `dev://eth0`
- **RemoteUri** `ether://[`*ethernet-addr*`]` where *ethernet-addr* is the multicast group; e.g., `ether://[01:00:5e:00:17:aa]`
- **Scope** non-local
- **Persistency** permanent; other persistency settings are forbidden
- **LinkType** multi-access or ad hoc
- **Mtu** MTU of the network interface (for all relevant face commands, NFD returns actual MTU)

`MulticastEthernetTransport` uses a libpcap handle to transmit and receive on an Ethernet link. The handle is initialized by activating onto an interface. Then, the link-layer header format is set to EN10MB and libpcap is set to only capture incoming packets. After the handle is initialized, the transport sets a packet filter to only capture packets with the NDN type (0x8624) that are sent to the multicast address. The use of promiscuous mode is avoided by directly adding the address to the link-layer multicast filters using SIOCADDMULTI. However, if this fails, the interface can fall back to promiscuous mode.

The libpcap handle is integrated with Boost.Asio through the `async_read_some` function by calling the pcap read functions in the read handler. If an oversized or invalid packet is received, it is dropped.

2.2.5 UDP Unicast Transport

UDP unicast transport (`nfd::face::UnicastUdpTransport`) is a transport that communicates on UDP tunnels over IPv4 or IPv6.

NFD listens for incoming datagrams via `UdpChannel` at a port number specified by the `face_system.udp.port` configuration option. A `UnicastUdpTransport` is created for each new remote endpoint. NFD can also create outgoing UDP unicast transports.

The static attributes of a UDP unicast transport are:

- **LocalUri** and **RemoteUri**
  - IPv4 `udp4://`*ip*`:`*port* ; e.g., `udp4://192.0.2.1:6363`
  - IPv6 `udp6://`*ip*`:`*port* where *ip* is in lower case and enclosed by square brackets; e.g., `udp6://[2001:db8::1]:6363`
- **Scope** non-local
- **Persistency** on-demand for transport created from accepted socket, persistent or permanent for transport created for outgoing connection (can be changed using `face/update` management command)
- **LinkType** point-to-point
- **Mtu** maximum IP length minus IP and UDP header (for all relevant face commands, NFD returns actual MTU)

`UnicastUdpTransport` is derived from `DatagramTransport`. As such, it is created by adding an existing UDP socket to the transport.

The UDP unicast transport relies on IP fragmentation instead of fitting packets to the MTU of the underlying link. This allows packets to traverse links with lower MTUs, as intermediate routers are able to fragment the packet as needed. IP fragmentation is enabled by preventing the Don't Fragment (DF) flag from being set on outgoing packets. On Linux, this is done by disabling PMTU discovery.

When the transport receives a packet that is too large or is incomplete, the packet is dropped. UDP unicast transports with an on-demand persistency will time out if a non-zero idle timeout is set.

UDP unicast transports will fail on an ICMP error, unless they have a permanent persistency. However, when choosing permanent persistency, note that there are no UP/DOWN transitions, requiring the use of a strategy that tries multiple faces.

2.2.6 UDP Multicast Transport

UDP multicast transport (`nfd::face::MulticastUdpTransport`) is a transport that communicates on a UDP multicast group.

NFD automatically creates UDP multicast transports on multicast-capable network interfaces during initialization. In `face_system.udp` config section, a whitelist+blacklist can be specified to create UDP multicast transports on a subset of available network interfaces. Setting `face_system.udp.mcast` to "no" disables UDP multicast transports altogether.

The multicast group and port number can be specified in NFD's configuration file: the `face_system.udp.mcast_group` and `face_system.udp.mcast_port` options are for IPv4, while `face_system.udp.mcast_group_v6` and `face_system.udp.mcast_port_v6` are for IPv6. All NDN hosts on the same IP subnet must be configured with the same multicast group and port number in order to communicate with each other; therefore, it is recommended to keep the default settings.

UDP multicast transport also supports *ad hoc* networks. This can be configured via the `face_system.udp.mcast_ad_hoc` option in NFD configuration file. Setting this option to "yes" changes the link type of the transport to *ad hoc*, which allows Interest and Data to be forwarded to the incoming face. By default, the option is set to "no", meaning that created transports will have a multi-access link type.

The static attributes of a UDP multicast transport are:

- **LocalUri** and **RemoteUri**
  - IPv4 `udp4://`*ip*`:`*port* ; e.g., `udp4://224.0.23.170:56363`
  - IPv6 `udp6://`*ip*`:`*port* where *ip* is in lower case and enclosed by square brackets; e.g., `udp6://[ff02::114%eth0]:56363`
- **Scope** non-local
- **Persistency** permanent
- **LinkType** multi-access or ad hoc
- **Mtu** maximum IP length minus IP and UDP header (for all relevant face commands, NFD returns actual MTU)

`MulticastUdpTransport` is derived from `DatagramTransport`. The transport uses two separate sockets, one for sending and one for receiving. These functions are split between the sockets to prevent sent packets from being looped back to the sending socket.

2.2.7 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 or permanent for transport created for outgoing connection (can be changed using `face/update` management command)
- **LinkType** point-to-point
- **Mtu** unlimited (for all relevant face commands, NFD returns actual MTU)

Like `UnixStreamTransport`, `TcpTransport` is derived from `StreamTransport`, and thus its other specifics can be found in the `UnixStreamTransport` section (section 2.2.2).

2.2.8 WebSocket Transport

WebSocket implements a message-based protocol on top of TCP for reliability. WebSocket is the protocol used by many web applications to maintain a long connection to remote hosts. It is used by NDN.JS client library to establish connections between browsers and NDN forwarders.

NFD listens for incoming WebSocket connections via `WebSocketChannel` at a port number specified by `face_system.websocket.po` configuration option. The channel listens over unencrypted HTTP and at the root path (i.e. `ws://<ip>:<port>/`); you may deploy a frontend proxy to enable TLS encryption or change the listener path (`wss://<ip>:<port>/<path>`). A `WebSocketTransport` is created for each incoming connection. NFD does not support outgoing WebSocket connections.

The static attributes of a WebSocket transport are:

- **LocalUri** `ws://`*ip*`:`*port* ; e.g., `ws://192.0.2.1:9696`, `ws://[2001:db8::1]:6363`
- **RemoteUri** `wsclient://`*ip*`:`*port* ; e.g., `ws://192.0.2.2:54420`, `ws://[2001:db8::2]:54420`
- **Scope** local if remote endpoint has loopback IP, non-local otherwise
- **Persistency** on-demand
- **LinkType** point-to-point
- **Mtu** unlimited (for all relevant face commands, NFD returns actual MTU)

WebSocket encapsulation of NDN packets `WebSocketTransport` expects exactly one NDN packet or LpPacket in each WebSocket frame. Frames containing incomplete or multiple packets will be dropped and the event will be logged 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.

`WebSocketTransport` is implemented using the websocketpp library.

The relationship between `WebSocketTransport` and `WebSocketChannel` is tighter than most Transport–Channel relationships. This is because messages are delivered through the channel.

2.2.9 Developing a New Transport

A new transport type can be created by first creating a new transport class that either specializes one of the transport template classes (`StreamTransport` and `DatagramTransport`) or inherits from the `Transport` base class. If the new transport type is inheriting directly from the `Transport` base class, then you will need to implement some pure virtual functions, including `doClose` and `doSend`. In addition, you will need to set the static properties (`LocalUri`, `RemoteUri`, `Scope`, `Persistency`, `LinkType`, and `Mtu`) in the constructor. When necessary, you can set the `State` of the transport and the `ExpirationTime`.

When specializing a transport class template, some of the aforementioned tasks may be handled by the generic implementation in the template definition. Depending on the template, all you may need to implement is the constructor, in addition to any needed helper functions. However, note that you will still need to set the static properties of the transport in the constructor.

If you want the new transport to support persistency changes, you will need to override the `canChangePersistencyToImpl` and `afterChangePersistency` virtual member functions.

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 0.6.1, the following features are implemented:

1. encoding and decoding of Interest, Data, and Nack

   Interests, Data, and Nack are encapsulated in LpPackets (the Generic Link Service only supports one network layer packet or fragment per LpPacket). LpPackets contain header fields and a fragment. This allows hop-by-hop information to be separated from ICN information.

2. fragmentation and reassembly (indexed fragmentation)

   Interests and Data can be fragmented and reassembled hop-by-hop to allow for the traversal of links with different MTUs.

3. consumer controlled forwarding (NextHopFaceId field)

   The NextHopFaceId field enables the consumer to specify the face that an Interest should be sent out of on a connected forwarder.

4. local cache policy (CachePolicy field)

   The CachePolicy field enables a producer to specify the policy under which a Data should be cached (or not cached, depending upon the policy).

5. incoming face indication (IncomingFaceId field)

   The IncomingFaceId field can be attached to an LpPacket to inform local applications about the face on which the packet was received.

6. congestion signaling (CongestionMark field)

   The CongestionMark field can be used to signal consumers and downstream routers of current congestion levels (see Section 8 for details).

Other planned features include:

1. failure detection (similar to BFD [6])
2. link reliability improvement (repeat request, similar to ARQ [7])

What services are enabled depends upon the type of transport:

	Fragmentation	Local Fields (NextHopFaceId, CachePolicy, IncomingFaceId)
Internal	No	Yes
UnixStream	No	No*
UnicastEthernet	Yes	No
MulticastEthernet	Yes	No
UnicastUdp	Yes	No
MulticastUdp	Yes	No
Tcp	No	No*
WebSocket	No	No

* Local fields can be enabled on these transport types when they have a local scope. They can be enabled through the `enableLocalControl` management command (see Section 6.4).

If fragmentation is enabled, the link service submits the network layer packet encapsulated in a link layer packet to the fragmenter. The specific implementation of the fragmenter is discussed in a separate section below. The link service hands each fragment off to the transport for transmission. If fragmentation is not enabled, a sequence is assigned to the packet and it is passed to the transport for transmission.

When a link layer packet is received on the other end, it is passed from the transport to the link service. If fragmentation is not enabled on the receiving link service, the received packet is checked for `FragIndex` and `FragCount` fields and dropped if it contains them. The packet is then given to the reassembler, which returns a reassembled packet, but only if the received fragment completed it. The reassembled packet is then decoded and passed up to the forwarding. Otherwise, the received fragment is not processed further.

Packet Fragmentation and Reassembly in the Generic Link Service The Generic Link Service uses indexed fragmentation (described in further detail in Section 2.6). The sending link service has a fragmenter. The fragmenter returns a vector of fragments encapsulted in link layer packets. If the size of the packet is less than the MTU, the fragmenter returns a vector containing only one packet. The link service assigns each fragment a consecutive Sequence number and, if there is more than one fragment, inserts a `FragIndex` and `FragCount` field into each. The `FragIndex` is the 0-based index of the fragment in relation to the network layer packet and the `FragCount` is the total number of fragments produced from the packet.

The receiving link service has a reassembler. The reassembler keeps track of received fragments in a map with a key based on the remote endpoint (see Section 2.1) and the Sequence of the first fragment in the packet. It returns the reassembled packet if it is complete. The reassembler also manages timeouts of incomplete packets, setting the drop timer when the first fragment is received. Upon receiving a new fragment for a packet, the drop timer for that packet is reset.

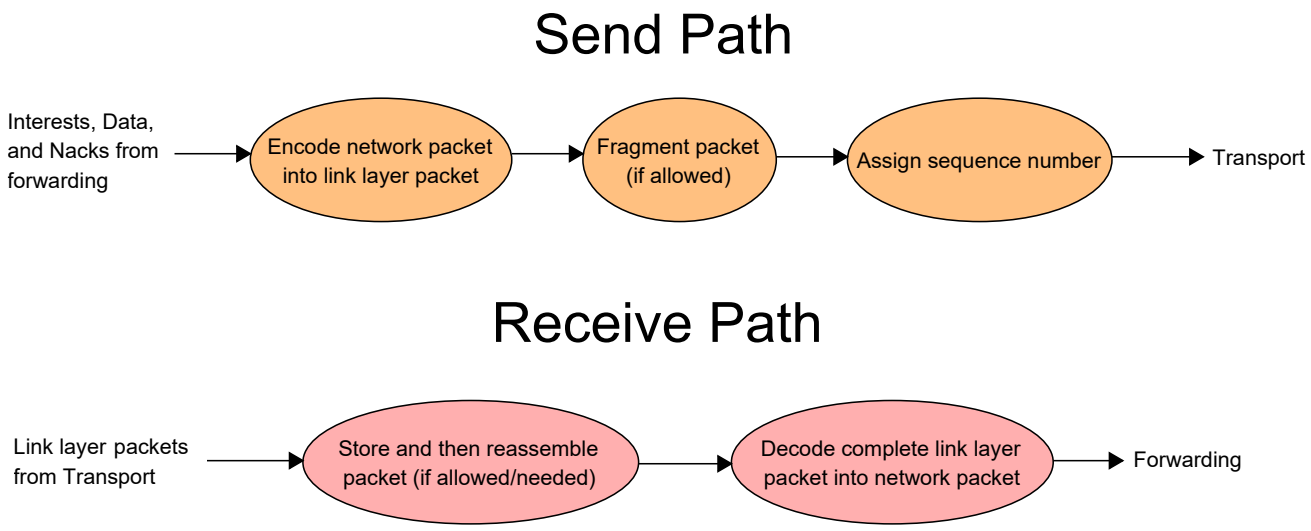


Figure 3: GenericLinkService Internal Structure

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

The link service can provide many services to the face, so a new link service needs to handle a number of tasks. At the minimum, a link service must encode outgoing and decode incoming Interests, Data, and Nacks. Depending on the intended use of the new link service, it may also be necessary to implement services like fragmentation and reassembly, local fields, and sequence number assignment, in addition to any other needed services.

2.4 `FaceSystem`, `ProtocolFactory`, and `Channel` Classes

The face system is organized as a `FaceSystem`—`ProtocolFactory`—`Channel`—`Face` hierarchy. The `FaceSystem` class is the entry point of the face system, which owns `ProtocolFactory` instances. Each `ProtocolFactory` subclass owns unicast `Channel`s and multicast faces of a particular underlying protocol. The `Channel` represents a local endpoint of a particular underlying protocol, and owns unicast faces bound to this local endpoint.

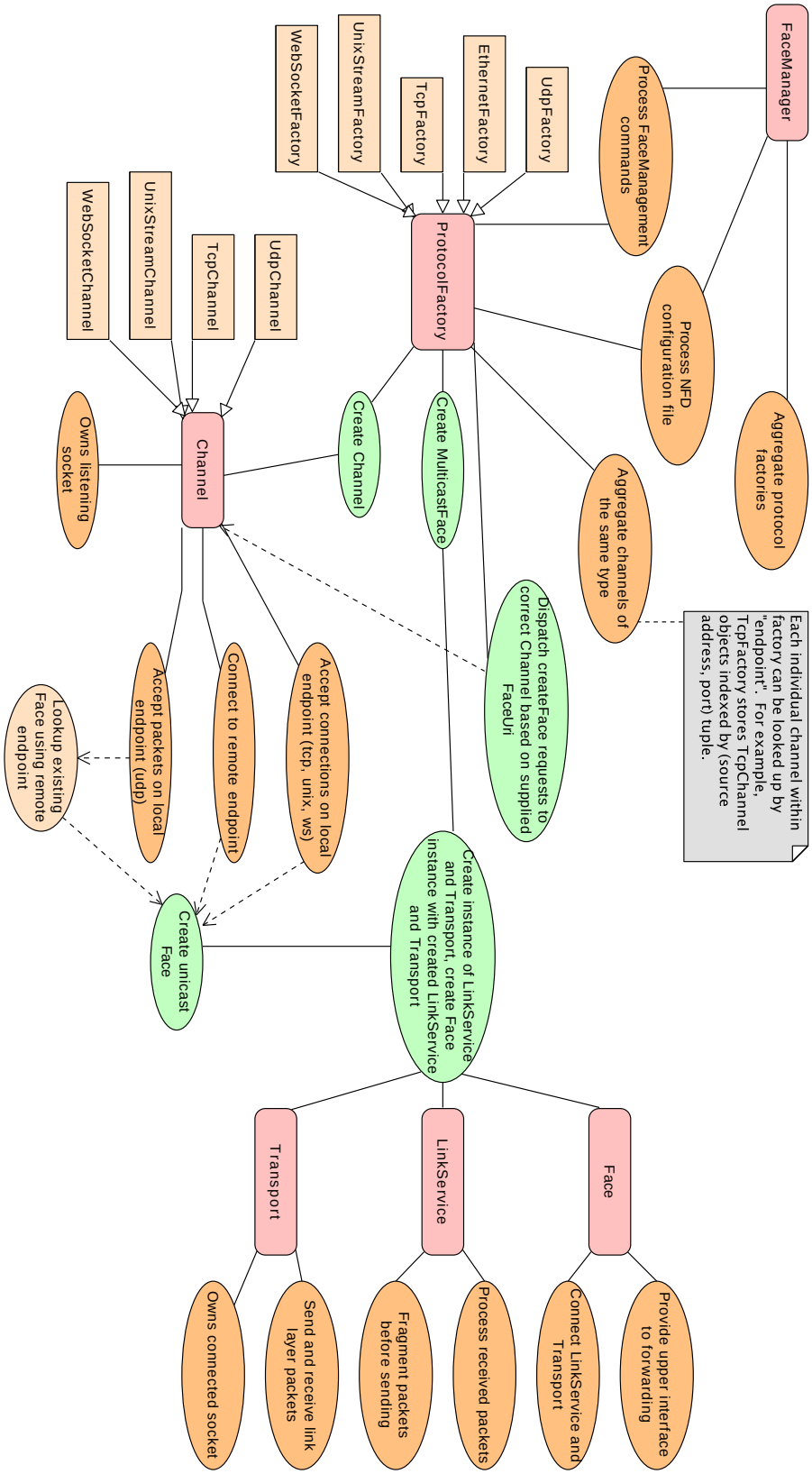


Figure 4: FaceManager, Channel, ProtocolFactory, and Face interactions

2.4.1 Initialization

The face system has a two-step initialization process. First, all available `ProtocolFactory` types make themselves known to the `FaceSystem` class using `NFD_REGISTER_PROTOCOL_FACTORY` macro, so that they are instantiated by `FaceSystem` constructor. Second, `FaceSystem` receives the `face_system` config section from the configuration parser (Section 10.1), and passes the necessary parameters to the available protocol factories, which in turn can initialize channels and faces.

Each `ProtocolFactory` subclass has a "factory id", and must override the `processConfig` method which will receive `face_system.`*factory-id* config section. `FaceSystem::processConfig` loops through all `ProtocolFactory` instances, and invokes `processConfig` with the relevant subsection.

Each `ProtocolFactory` subclass can define its own config options under `face_system.`*factory-id* section. Depending on the capability of the protocol factory and the characteristics of the underlying protocol, this may include enable flags, port numbers, timeout values, and so on. Additionally, a `ProtocolFactory` subclass can define the behavior when its config section is omitted; typically, the corresponding protocol would be disabled in this situation.

`ProtocolFactory::processConfig` is not only used during the first initialization, but also invoked during a configuration reload (Section 10.1.3). In order to give the protocol factory an opportunity to de-initialize itself, `ProtocolFactory::processConfig` is invoked even if `face_system.`*factory-id* config section is omitted. However, in current implementation, not all protocol factories fully support configuration reloads: they might keep the initial configuration, or become only partially disabled, in case the config section is omitted.

2.4.2 Inside a ProtocolFactory

A protocol factory manages the channels (unicast) and multicast faces of a particular protocol type. Subclasses of **ProtocolFactory** need to implement the `createFace` and `getChannels` virtual functions. Optionally, the `createChannel`, `createMulticastFace`, and `getMulticastFaces` functions can be implemented, in addition to any protocol-specific functions.

A channel listens for and handles incoming unicast connections and starts outgoing unicast connections for a specific local endpoint. Faces are created upon the success of either of these actions. Channels are created by protocol factories when the `createChannel` function is called. Upon creation of a new face, either incoming or outgoing, the `FaceCreatedCallback` specified to the `listen` function is called. If creation of the face failed, then the `FaceCreationFailedCallback` (also specified to `listen`) is called. Ownership of the listening socket (or in the case of WebSocket, the WebSocket server handle) lies with the individual channel. Sockets connected to remote endpoints are owned by the transport associated with the relevant face, except in the case of WebSocket, where all faces use the same server handle. Note that there are no Ethernet channels, as Ethernet links in NFD are multicast only.

2.4.3 Outgoing Unicast Face Creation

Most outgoing faces are created by the `FaceManager` (Section 6.4) in response to a `faces/create` command. The parameters to that command includes the remote FaceUri (Section 2.2).

When a `ProtocolFactory` instance is initialized, its `ProtocolFactory::processConfig` method should declare which FaceUri schemes, if any, can be used for outgoing face creation on this `ProtocolFactory` instance, by populating the `ProtocolFactory::providedSchemes` container. When the `FaceManager` receives a `faces/create` command, the scheme portion of the remote FaceUri is used to find an appropriate `ProtocolFactory` instance. `ProtocolFactory::createFace` is then invoked on the instance.

Each `ProtocolFactory` subclass can define its own requirements on what FaceUris are acceptable. Typically, the FaceUri is required to be in *canonical* form. This eliminates the need to perform DNS resolution inside the NFD process, so as to reduce overhead.

Most `ProtocolFactory` subclasses organize unicast faces into `Channel`s. When there are multiple channels in a protocol factory, it is the factory that decides which channel is to be used to create a new face. Typically, a channel with a compatible underlying protocol and endpoint is chosen.

2.4.4 Multicast Face Creation

`EthernetFactory` and `UdpFactory` can create multicast faces. In `face_system.ether` and `face_system.udp` config sections, there are options to enable or disable multicast faces, and a whitelist+blacklist to choose which network interfaces should have multicast faces.

When one of these config section is processed in `EthernetFactory::processConfig` and `UdpFactory::processConfig`, multicast-related config options are stored in the factory instance. Multicast faces are then created or destroyed based on configuration and the current list of available network interfaces, as provided by the `ndn::NetworkMonitor` class. Whenever

a network interface is added, removed, or otherwise changed, a signal from `NetworkMonitor` triggers `EthernetFactory` and `UdpFactory` to reevaluate the stored configuration, and create or destroy multicast faces as necessary.

2.5 NIC-associated Permanent Faces

This is an upcoming feature.

NIC-associated permanent faces are permanent faces automatically created on, and bound to, an OS-level network interface (physical or virtual).

The `NicFaces` class coordinates all NIC-associated permanent faces. It is configured by `face_system.nicfaces` config section, which contains zero or more "rules"; each rule contains a whitelist+blacklist to select network interfaces, and a remote FaceUri to which an outgoing unicast face should be created from each matching network interface.

A `ProtocolFactory` subclass can declare itself as supporting NIC-associated permanent face by adding `scheme+dev` into `ProtocolFactory::providedSchemes` container during initialization, and overriding `ProtocolFactory::doCreateNicFace`.

2.6 NDNLP

The NDN Link Protocol (version 2) provides a link protocol between the forwarding and the underlying network transmission protocols and systems (like TCP, UDP, Unix sockets, and Ethernet). It allows for a uniform interface to the forwarding link services and provides a bridge between these services and the underlying network layers. Through this method, the specific features and mechanisms of these underlying layers can be overlooked by the upper layers. In addition, the link protocol provides services common to every type of link, the specific implementation of which may vary from link type to link type. The link service also specifies a common TLV link-layer packet format. The services currently provided by NDNLP include fragmentation and reassembly, Negative Acknowledgement (Nack), consumer-controlled forwarding, cache policy control, and providing information about the incoming face of a packet to applications. Features planned for the future include link failure detection (BFD) and ARQ. These features can be individually enabled and disabled.

In NFD, the link protocol is implemented in the **LinkService** and its subclasses. This link protocol replaces the previous version of the NDN Link Protocol (NDNLPv1) [8].

A description of each NDNLPv2 feature follows.

Fragmentation and reassembly

Fragmentation and reassembly is done hop-by-hop using indexed fragmentation. Packets are fragmented and are assigned a **FragIndex** field, which indiciates their zero-based index in the fragmented packet, and a **FragCount** field, which is the total number of fragments of the packet. All link-layer headers associated with the network-layer packet are only attached to the first fragment. Other, unrelated link-layer headers may be attached to any fragment by "piggybacking" onto it. The recipient uses the **FragIndex** and **FragCount** fields from each fragment to reassemble the complete packet.

Negative Acknowledgement (Nack)

Negative Acknowledgements are messages sent downstream to indicate that a forwarder was unable to satisfy a particular Interest. The relevant Interest is included with the Nack in the **Fragment** field. The Nack itself is indicated with the **Nack** header field in the packet.

The Nack can optionally include a **NackReason** field (under the **Nack** field) to indicate why the forwarder was unable to satisfy the Interest. These reasons include congestion on the link, a duplicate Nonce being detected, and no route matching the Interest.

Consumer-controlled forwarding

Consumer-controlled forwarding allows an application to specify which face an outgoing Interest should be sent on. It is indicated with the **NextHopFaceId** header, which includes the ID of the face on the local forwarder that the Interest should be sent out of.

Cache policy control

Through cache policy control, a producer can indicate how its Data should be cached (or not cached). This is done using the **CachePolicy** header, which contains the **CachePolicyType** field. The non-negative integer contained in this inner field is the indication of which cache policy this application wishes downstream forwarders to follow.

Incoming face indication

A forwarder can inform an application of the face on which a particular packet was received by attaching the **IncomingFaceId** header to it. This field contains the face ID of the face on the forwarder that the packet was received on.

Congestion signaling

A host can signal the current congestion state to other hosts using the CongestionMark field. A value of 0 indicates no congestion; a value greater than 0 indicates some level of congestion. The exact meaning of the bits in this field is left up to the congestion control strategy in use.

3 Tables

The tables module provides main data structures for NFD.

The Forwarding Information Base (FIB) (Section 3.1) 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 3.2) contains a list of producer region names for mobility support.

The Content Store (CS) (Section 3.3) 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 3.4) 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 3.5 supplements the Interest Table for loop detection.

The Strategy Choice Table (Section 3.6) contains the forwarding strategy (Section 5) chosen for each namespace.

The Measurements Table (Section 3.7) 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 3.8), is designed to be shared among these four tables.

3.1 Forwarding Information Base (FIB)

The Forwarding Information Base (FIB) is used to forward Interest packets toward potential source(s) of matching Data [9]. 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 3.1.1. How FIB is used by rest of NFD is described in Section 3.1.2. The implementation of FIB algorithms is discussed in Section 3.8.

3.1.1 Structure and Semantics

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

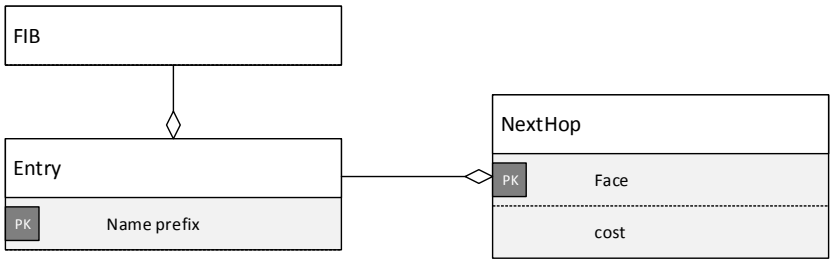


Figure 5: FIB and related entities

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 7.3.1), 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.

3.1.2 Usage

The FIB is updated only through using FIB management protocol, which on NFD sides is operated by the FIB manager (Section 6.5). Typically, FIB manager takes commands from RIB Management (Section 7), 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 4.2.1). It is called at most once per incoming Interest.

`cleanupOnFaceRemoval` function (declared in `daemon/table/cleanup.hpp`) is a joint FIB-PIT cleanup function upon face removal. This function is invoked when a face is destroyed. It visits all FIB and PIT entries, and deletes FIB nexthop records, PIT in-records, and PIT out-records pointing to the removed face. A FIB entry losing its last nexthop record is also deleted. Conversely, a PIT entry losing all its in-records and out-records is kept in the table[2]. Cleanup for FIB and PIT is performed in the same function because both tables are stored on the NameTree (Section 3.8), so that this function can cleanup both tables in one pass of NameTree enumeration.

3.2 Network Region Table

The Network Region Table (`nfd::NetworkRegionTable`) is used for mobility support (Section 4.2.4). It contains an unordered set of producer region names, taken from NFD configuration file (Section 6.7.2). If any delegation name in the forwarding hint 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.

3.3 Content Store (CS)

The Content Store (CS) is a cache of Data packets. Forwarding pipelines (Section 4) places arriving Data packets in the CS, so that future Interests requesting the same Data can be satisfied without forwarding further.

The CS offers procedures to insert a Data packet, find cached Data that matches an Interest, and enumeration of cached Data. Section 3.3.1 describes the semantics of those procedures and their usage.

The CS is implemented in (`nfd::cs::Cs`) class. The implementation consists of two parts: a lookup table, and a cache replacement policy. The lookup table (Section 3.3.2) is a name-based index of CS entries, in which cached Data packets are stored. The cache replacement policy (Section 3.3.3) is responsible for keeping the CS under capacity limits. It separately maintains a cleanup index in order to determine which entry to evict when the CS is full. NFD offers multiple cache replacement policies including a priority FIFO policy and a LRU policy, which can be selected in NFD configuration file.

3.3.1 Semantics and Usage

Insertion Data packets are inserted to the CS (`Cs::insert`) either in the *incoming Data pipeline* (Section 4.3.1) or in the *Data unsolicited pipeline* (Section 4.3.2), 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 control [10]).

Before storing the Data packet, the **admission policy** is evaluated. Local applications can give hints to the admission policy via NDNLPv2 [5] *CachePolicy* field attached to the Data packet. These 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.

The CS is configured with a capacity limit, which is checked at this point. If the insertion of this new Data packet causes the CS to exceed the capacity limit, the **cache replacement policy** evicts excessive entries to bring the CS under capacity limit.

[2]This is pending discussion in https://redmine.named-data.net/issues/3685#note-7.

Lookup The CS offers an asynchronous lookup API (`Cs::find`). *incoming Interest pipeline* (Section 4.2.1) invokes this API with an incoming Interest. The search algorithm gives the Data packet that best matches the Interest to *ContentStore hit pipeline* (Section 4.2.3), or informs *ContentStore miss pipeline* (Section 4.2.4) if there's no match.

Enumeration and Entry Abstraction The Content Store can be enumerated via forward iterators. 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 a 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. A Content Store implementation may define its own concrete type, and convert to the abstraction type during enumeration.

3.3.2 Lookup Table

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.[3]

Lookups are performed entirely using the Table. The lookup procedure (`Cs::find*`) 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 current implementation does lookups synchronously.

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

3.3.3 Cache Replacement Policy

The cache replacement policy keeps the CS under its capacity limit. The main capacity limit is measured in terms of number of cached Data packets. This metric is chosen over Data packet size because the memory overhead of table index can be significant for small packets so that a packet size metric would be inaccurate. This capacity limit can be configured in NFD configuration file `tables.cs\_max\_packets` key, or via `Cs::setLimit` API in a simulation environment. Runtime changes of the capacity limit are allowed.

NFD offers multiple policy implementations as subclasses of `nfd::cs::Policy`. The effective policy can be configured in NFD configuration file `tables.cs\_policy` key, or via `Cs::setPolicy` API in a simulation environment. This configuration can only be applied during initialization; runtime changes are disallowed.

`Policy` class API `nfd::cs::Policy` is the base class for all cache replacement policy implementations.

The capacity limit is stored on the `Policy` class, and can be changed via `Policy::setLimit` public method. A pure virtual method `evictEntries` must be provided by a policy implementation to handle the lowering of the capacity limit by evicting enough entries to bring the CS back under the new capacity limit. In addition to the main capacity limit (referred to as the "hard limit"), a policy may offer additional capacity limits (such as a separate limit for certificates, or a packet size based limit).

In order for a policy to maintain the capacity limit, it needs to know what Data packets have been added to the cache and their access patterns. Therefore, the policy class exposes four public methods to receive those information:

- **Policy::afterInsert** is invoked after a new entry is inserted;
- **Policy::afterRefresh** is invoked after an existing entry is refreshed by same Data;
- **Policy::beforeErase** is invoked before an entry is erased via management command (currently unused);
- **Policy::beforeUse** is invoked when an entry is found by a lookup and before it's used in forwarding.

These public methods call into corresponding pure virtual methods: `doAfterInsert`, `doAfterRefresh`, `doBeforeErase`, `doBeforeUse`. These pure virtual methods, as well as `evictEntries`, should be overridden in subclasses.

Based on information received via the above APIs, a policy maintains an internal cleanup index, which is used to determine which Data packet should be evicted when the CS exceeds the capacity limit. The structure of this internal cleanup index is defined in each policy implementation. It should reference CS entries (stored in the *Table*) via iterators (`nfd::cs::iterator`). When a policy decides to evict an entry, it should emit `beforeEvict` signal to inform the CS to erase the entry from the *Table*, and then delete the corresponding item in the policy's internal cleanup index. Note that `beforeErase` will not be invoked for entries evicted via `beforeEvict` signal.

The recommended procedures for each function are:

[3] 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.

- In **doAfterInsert**, the policy decides whether to accept the new entry. If it is accepted, the iterator of the new entry should be inserted into the internal cleanup index; otherwise, `cs::Policy::evictEntries` will be called to inform CS to do cleanup. Then, the policy should check whether CS size exceeds the capacity limit, and evict an entry (probably by calling `evictEntries` function) if so.
- In **doAfterRefresh**, the policy may update its cleanup index to note that the same Data has arrived again.
- In **doBeforeErase**, the policy should delete the corresponding item in its cleanup index.
- In **doBeforeUse**, the policy may update its cleanup index to note that the indicated entry is accessed to satisfy an incoming Interest.
- In **evictEntries**, the policy should evict enough entries so that the CS does not exceed capacity limit.

Priority FIFO cache policy Priority-FIFO is the default cs::Policy. Priority-FIFO evicts upon every insertion, because its performance is more predictable; the alternative, periodic cleanup of a batch of entries, can cause jitter in packet forwarding. Priority-FIFO uses three queues to keep track of Data packets in CS:

- **unsolicited queue** contains entries with unsolicited Data;
- **stale queue** contains entries with stale Data;
- **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 the least recently used items first. LRU evicts upon every insertion, because its performance is more predictable; the alternative, periodic cleanup of a batch of 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 the 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 needs to be evicted, its Table iterator is erased from the head of its queue, and the entry is erased from the Table.

The **queue** uses `boost::multi_index_container` [11] as the underlying container due to 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`.

3.4 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) [9]. 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 4). The structure and semantics of PIT entry, and how it's used by forwarding are described in Section 3.4.1. The structure and algorithms of PIT, and how it's used by forwarding are described in Section 3.4.2. The implementation of PIT algorithms is discussed in Section 3.8.

3.4.1 PIT Entry

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

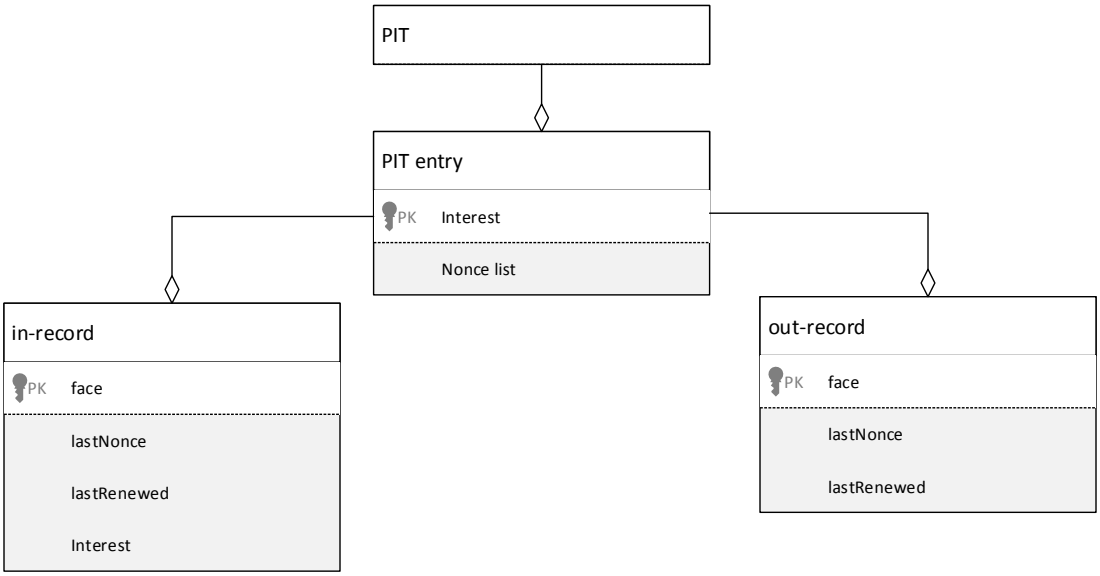


Figure 6: PIT and related entities

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 a timer, described below. In addition, forwarding strategy is allowed to store arbitrary information on PIT entry, in-records, and out-records (Section 5.1.3).

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 4.2.1). All in-records are deleted by *incoming Data pipeline* (Section 4.3.1) 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
- Nacked field: indicates the last outgoing Interest has been Nacked; this field also records the Nack reason

An out-record is inserted or updated by *outgoing Interest pipeline* (Section 4.2.5). An out-record is deleted by *incoming Data pipeline* (Section 4.3.1) 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.

Timer

Each PIT entry has one timer, the *expiry timer*. This timer is used by the forwarding pipelines (Section 4) and it fires when the PIT entry expires (Section 4.2.1).

3.4.2 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.

`cleanupOnFaceRemoval` function is a joint FIB-PIT cleanup function upon face removal. See Section 3.1.1 for more information.

3.5 Dead Nonce List

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

In August 2014, we found a persistent loop problem when InterestLifetime is short (Bug 1953). Loop detection previously used only the Nonces stored in PIT entries. If an Interest is not satisfied within InterestLifetime, the PIT entry is deleted. 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 a longer duration. However, the memory consumption of doing so would be too high, because a PIT entry contains many other things than the Nonce. Therefore, the 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 the Dead Nonce List, and how it's used by forwarding are described in Section 3.5.1. Section 3.5.2 discusses how the capacity of Dead Nonce List is maintained.

3.5.1 Structure, Semantics, and Usage

A tuple of Name and Nonce is added to Dead Nonce List (`DeadNonceList::add`) in *incoming Data pipeline* (Section 4.3.1) and *Interest finalize pipeline* (Section 4.2.6) before out-records are deleted.

The Dead Nonce List is queried (`DeadNonceList::has`) in *incoming Interest pipeline* (Section 4.2.1). 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.

3.5.2 Capacity Maintenance

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

A naive approach to 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 we approximate the 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 *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 it 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 decreased. If all recent counts are below L/M, the capacity is increased.

In addition, there is a hard upper and lower bound for 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 insertion operations.

3.6 Strategy Choice Table

The Strategy Choice Table contains the forwarding strategy (Section 5) 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 [9]. 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 the NFD-RIB service (Section 7). Storing the strategy in FIB entries would require the RIB service to create/update/remove strategies when it manipulates the FIB. Therefore, this would increase the complexity of the RIB service.
- 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 3.6.1. How Strategy Choice Table is used by rest of NFD is described in Section 3.6.2. The implementation of Strategy Choice Table algorithms is discussed in Section 3.8.

3.6.1 Structure and Semantics

Strategy Choice entry

A Strategy Choice entry (`nfd::strategy_choice::Entry`) contains a name prefix and a forwarding strategy chosen for that namespace. A *nfd::fw::Strategy* subclass instance is created at runtime, and stored inside each Strategy Choice entry.

Strategy Choice Table

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

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 `getDefaultStrategyName` free function in `daemon/fw/forwarder.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. This operation accepts a name prefix on which the strategy choice is applied, and a strategy instance name. The strategy instance name starts with a name prefix that indicates the strategy "program", such as `ndn:/localhost/nfd/strategy/best-route`; then, it may contain an optional version number to choose a specific version of the strategy program, and additional name components as "parameters" which are passed to the strategy constructor. The strategy instance name is used to locate a strategy type in the strategy registry (within `nfd::fw::Strategy` class); if a strategy type is found, it is instantiated and stored into the Strategy Choice entry.

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. 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.

3.6.2 Usage

The Strategy Choice Table is updated only through management protocol. Strategy Choice manager (Section 6.6) 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 *ContentStore miss pipeline* (Section 4.2.4), *incoming Data pipeline* (Section 4.3.1), and *incoming Nack pipeline* (Section 4.4.1). It is called at most twice per incoming packet.

3.7 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 5.1.3). 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 3.7.1. How Measurements Table is used by rest of NFD is described in Section 3.7.2. The implementation of Measurements Table algorithms is discussed in Section 3.8.

3.7.1 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 5.1.3). 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.

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.

3.7.2 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 3.6), 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 3.6) to confirm whether the requesting strategy owns the Measurements entry. If an access violation is detected, null is returned instead of the entry.

3.8 NameTree

The NameTree is a common index structure for FIB (Section 3.1), PIT (Section 3.4, Strategy Choice table (Section 3.6, and Measurements table (Section 3.7). It is feasible to use a common index, because there are much commonality in the index of these four tables: FIB, Strategy Choice, and Measurements are all indexed by Name, while 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 4.2.1) 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 3.8.1. NameTree operations and algorithms are described in Section 3.8.2. Section 3.8.3 describes how NameTree can help reducing number of index lookups by adding shortcuts between tables.

3.8.1 Structure

The NameTree is a collection of NameTree entries, indexed by Name and organized in a tree structure. 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
- 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. The tree structure follows name hierarchy: the name prefix of a parent entry is the name prefix of its child minus the last component.

The FIB, Strategy Choice, and Measurements entries attached to a NameTree entry have the same name as the NameTree entry. In most cases, PIT entries attached to a NameTree entry can have the same name as the NameTree entry and differ only in Selectors; as a special case, a PIT entry whose Interest name ends with an implicit digest component is attached to the NameTree entry that corresponds to the Interest name minus the implicit digest component, so that the *all match* algorithm (Section 3.8.2) with an incoming Data name (without computing its implicit digest) can find this PIT entry.

NameTree hash table

NameTree entries are organized into a hash table, in addition to the tree structure, to enable faster name-based lookups. Specifically, the hash table allows us to locate a deep entry without going from the root of the tree. We decide to implement the hash table from scratch (`nfd::name_tree::Hashtable`), rather than using an existing library, so that we can have better control for performance tuning.

The hash table contains a number of *buckets*. To insert an entry, we use CityHash [12] to compute the hash value of its name prefix; this hash function is chosen due to its fast speed. Specifically, hash value is computed over the TLV representation of the name components, but does not cover outer `NAME-TYPE TLV-LENGTH` fields; this allows us to compute hash values of all prefixes of a name together when needed. The entry is then mapped into a bucket chosen by the hash value. In case multiple names are mapped to the same bucket, we resolve hash collision by chaining the entries in a doubly linked list.

The hash table is resized automatically as the number of stored NameTree entries changes. Resize operations are controlled by parameters in `nfd::name_tree::HashtableOptions`. The parameter setting is a trade-off between wasted memory of

empty buckets and time overhead of chaining. When the load factor (number of entries divided by number of buckets) goes above the expand threshold or below the shrink threshold, a resize operation is triggerred, in which every NameTree entry is moved to the appropriate bucket in the new hash table.

We introduce a `nfd::name_tree::Node` to store fields used in hash table implementation, including:

- the hash value, so that resize operation does not need to re-compute it
- pointer to previous node in doubly linked list of the bucket
- pointer to next node in doubly linked list of the bucket
- the NameTree entry (Node owns Entry; Entry has a pointer back to the Node)

3.8.2 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/Strategy Choice/Measurements entry is being inserted.

The **conditional deletion** operation (`NameTree::eraseEntryIfEmpty`) deletes an entry if no FIB/PIT/Strategy Choice/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/Strategy Choice/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** algoritm (`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 3.1.1), with a predicate that accepts a NameTree entry only if it contains a FIB entry. This algorithm is called by Strategy Choice find effective strategy algorithm (Section 3.6.1), with a predicate that accepts a NameTree entry only if it contains a Strategy Choice 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 3.4.2).

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 5.1.3) to clear StrategyInfo items under a namespace changing ownership.

3.8.3 Shortcuts

One benefit of the NameTree is that it can reduce the number of index lookups during packet forwarding. To achieve this goal, 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 not 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/Strategy Choice/Measurements entry contains a pointer to the corresponding NameTree entry; the NameTree entry contains pointers to FIB/PIT/Strategy Choice/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 [4], 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:

[4] This applies only if the PIT entry's Interest Name does not end with an implicit digest; otherwise, a regular lookup would be performed.

- `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::getEntry` function template, which returns the NameTree entry linked from a FIB/PIT/Strategy Choice/Measurements entry. `NameTree::getEntry` allows one table to retrieve the corresponding NameTree from an entry of another table, without knowning the internal structure of that entry. It also permits a table to depart from NameTree in the future without breaking other code: suppose someday Measurements is no longer based on NameTree, `NameTree::getEntry` could perform a lookup using Interest Name in the Measurements entry; `Fib::findLongestPrefixMatch` can still accept Measurements entries, although it's not more efficient than using a Name.

4 Forwarding

NFD has a smart forwarding plane, which consists of **forwarding pipelines** (Section 4) and **forwarding strategies** (Section 5). A forwarding pipeline (or pipeline) consists of a series of processing steps on a packet. In addition, a pipeline is entered when an event is triggered and a condition is matched, such as on receiving an Interest, when detecting the received Interest was looped, when an Interest is ready to be forwarded out of a face, etc. A forwarding strategy (or strategy) makes desicions on packet forwarding, including whether, when, and where to forward a packet. NFD can have multiple strategies serving different namespaces, and pipelines will give packets to strategies accordingly.

Figure 7 shows the overall workflow of forwarding pipelines and strategy, where blue boxes represent pipelines and white boxes represent decision points of the strategy.

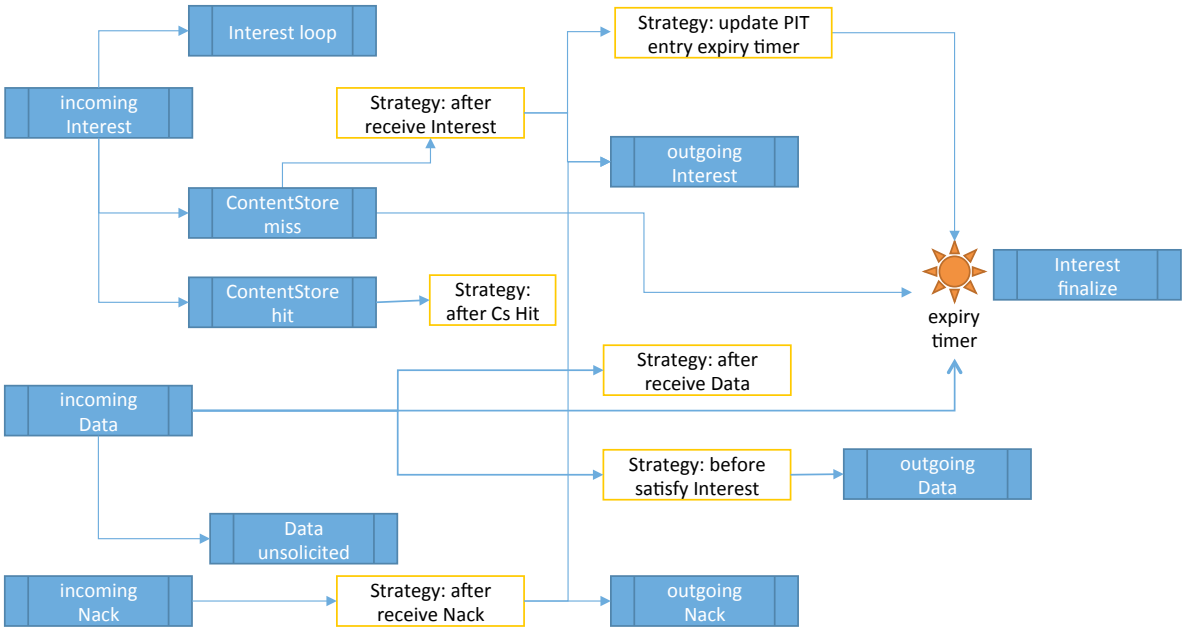


Figure 7: Pipelines and strategy: overall workflow

4.1 Forwarding Pipelines

The pipelines operate on network layer packets (Interest, Data, or Nack) 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 Strategy Choice table, however for the last three pipelines have only read-only access, as those tables are managed by the corresponding managers and are not directly affected by data plane traffic.

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

The processing of Interest, Data, and Nack packets in NDN is quite different. We separate forwarding pipelines into **Interest processing path**, **Data processing path**, and **Nack processing path**, described in the following sections.

4.2 Interest Processing Path

NFD separates Interest processing into the following pipelines:

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

- Interest finalize: deleting PIT entries

4.2.1 Incoming Interest Pipeline

The incoming Interest pipeline is implemented in `Forwarder::onIncomingInterest` method and is entered from `Forwarder::startProcessInterest` method, which is triggered by `Face::afterReceiveInterest` signal. 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 8:

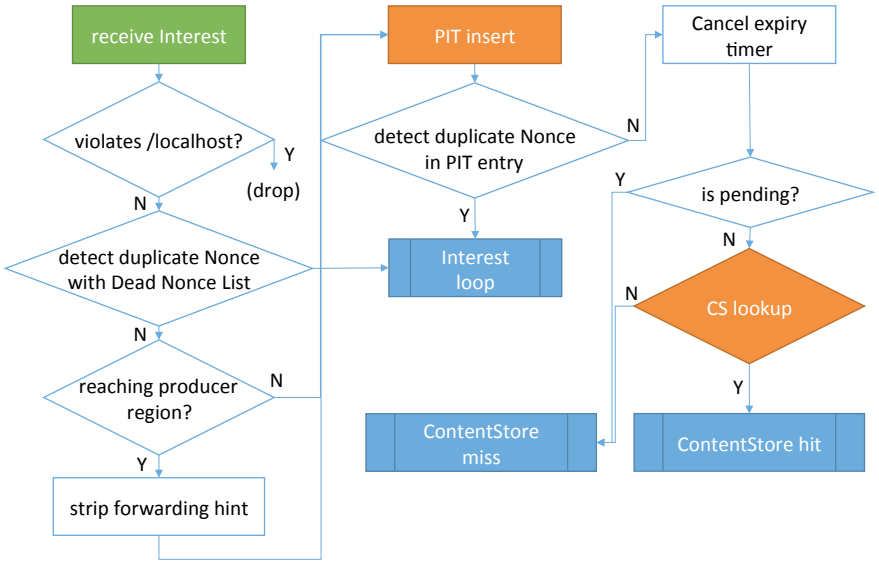


Figure 8: Incoming Interest pipeline

1. The first step is to check for `/localhost` scope [10] 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 a violation is detected, the Interest is immediately dropped and no further processing is performed. This check guards against malicious senders; a compliant forwarder will never send a `/localhost` Interest to a non-local face. Note that `/localhop` scope is not checked here, because its scope rules do not restrict incoming Interests.

2. The Name and Nonce of the incoming Interest are checked against the Dead Nonce List (Section 3.5). If a match is found, the incoming Interest is suspected of a loop, and is given to the *Interest loop pipeline* for further processing (Section 4.2.2). If a match is not found, processing continues onto the next step. Note that, unlike a duplicate Nonce detected with PIT entry (described below), a duplicate detected by the Dead Nonce List does not cause the creation of a PIT entry, because creating an in-record for this incoming Interest would cause matching Data, if any, to be returned to the downstream, which is incorrect; on the other hand, creating a PIT entry without an in-record is not helpful for future duplicate Nonce detection.

3. If the Interest carries a forwarding hint, the procedure determines whether the Interest has reached the producer region, by checking if any delegation name in the forwarding hint object is a prefix of any region name from the *network region table* (Section 3.2). If so, the forwarding hint is stripped away, as it has completed its mission of bringing the Interest into the producer region, and are no longer necessary.

4. 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, the PIT entry becomes a processing subject of the incoming Interest and following pipelines. Note that NFD creates the PIT entry before performing ContentStore lookup. The main reason for this decision is to reduce the lookup overhead, on the assumption that the ContentStore is likely to be significantly larger than the PIT, because in some cases described below CS lookup can be skipped.

5. Before the incoming Interest is processed any further, its Nonce is checked against the Nonces among PIT in-records. If a match is found in an in-record of a different face, 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 4.2.2). Otherwise, processing

continues. Note that if the Nonce duplicates a Nonce previously received on the face, the Interest is considered a legit retransmission because there is no risk of persistent loop in this case.

6. Next, the *expiry timer* on the PIT entry is cancelled, because a new valid Interest has arrived, so the lifetime of the PIT entry needs to be extended. The timer may be reset later on in the Interest processing path, for example if a matching Data is found in ContentStore.

7. The pipeline then tests whether the Interest is pending, i.e., if the PIT entry already has another in-record from the same or another incoming Face. Recall that NFD's PIT entry can represent not only a pending Interest but also a recently satisfied Interest (Section 3.4.1). This test is equivalent to "having a PIT entry" in CCN Node Model [9], whose PIT contains only pending Interests.

8. If the Interest is not pending, the Interest is matched against the ContentStore (`Cs::find`, Section 3.3.1). 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 4.2.4) or in *ContentStore hit pipeline* (Section 4.2.3).

4.2.2 Interest Loop Pipeline

This pipeline is implemented in `Forwarder::onInterestLoop` method and is entered from *incoming Interest pipeline* (Section 4.2.1) when an Interest loop is detected. The input parameters to this pipeline include an Interest packet, and its incoming Face.

This pipeline sends a Nack with reason code Duplicate to the Interest incoming face, if it's a point-to-point face. Since the semantics of Nack is undefined on a multi-access link, if the incoming face is multi-access, the looping Interest is simply dropped.

4.2.3 ContentStore Hit Pipeline

This pipeline is implemented in `Forwarder::onContentStoreHit` method and is entered in the incoming Interest pipeline (Section 4.2.1), after performing a ContentStore lookup (Section 3.3.1) and finding a match. The input parameters to this pipeline include an Interest packet, its incoming Face, the PIT entry, and the matched Data packet.

As illustrated in Figure 9, this pipeline first sets the expiry timer to now on the Interest, then invokes *after Content Store hit* trigger of the chosen strategy.

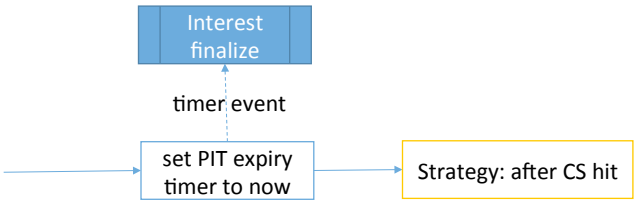


Figure 9: ContentStore Hit pipeline

4.2.4 ContentStore Miss Pipeline

This pipeline is implemented in `Forwarder::onContentStoreMiss` method and is entered in the incoming Interest pipeline (Section 4.2.1), after performing a ContentStore lookup (Section 3.3.1) and finding no match. The input parameters to this pipeline include an Interest packet, its incoming Face, and the PIT entry.

As illustrated in Figure 10, this pipeline takes the following steps:

1. An in-record for the Interest and its incoming face is inserted into the PIT entry. In case an in-record for the same incoming face already exists (i.e., the Interest is being retransmitted by the same downstream), it's simply refreshed with the newly observed Interest Nonce and expiration time. The expiration time of this in-record is controlled by the `InterestLifetime` field in the Interest packet; if `InterestLifetime` is omitted, the default 4 seconds is used.

2. The *expiry timer* on the PIT entry is set to the time that the last PIT in-record expires. When the expiry timer expires, the *Interest Finalize pipeline* (Section 4.2.6) is executed.

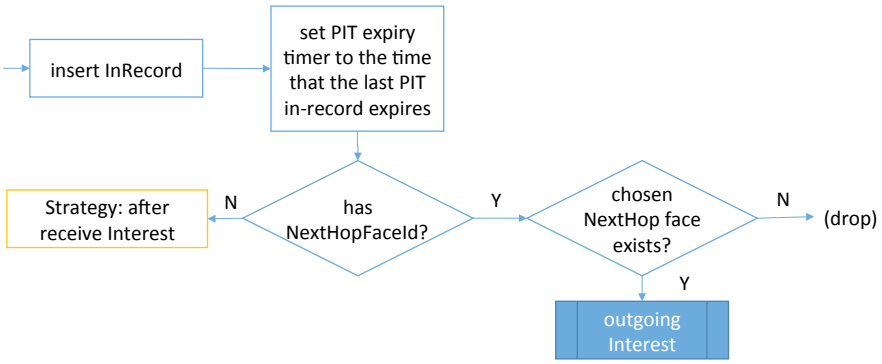


Figure 10: ContentStore Miss pipeline

3. If the Interest carries a `NextHopFaceId` field in its NDNLPv2 header, the pipeline honors this field. The chosen next hop face is looked up in the FaceTable. If a face is found, the *outgoing Interest pipeline* (Section 4.2.5) is executed; if the face does not exist, the Interest is dropped.

4. Without a `NextHopFaceId` field, a forwarding strategy is responsible for making forwarding decision on the Interest. Therefore, the pipeline invokes Find Effective Strategy algorithm (Section 3.6.1) to determine which strategy to use, and invokes the *after receive Interest* trigger of the chosen strategy with the Interest packet, its incoming face, and the PIT entry (Section 5.1.1).

Note that forwarding defers to the strategy the decision on whether, when, and where to forward an Interest. Most strategies forward a new Interest immediately to one or more upstreams found through a FIB lookup. For a retransmitted Interest, most strategies will suppress it if the previous forwarding happened recently (see Section 5.1.1 for more details), and forward it otherwise.

4.2.5 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 5.1.2). The input parameters to this pipeline include a PIT entry, an outgoing Face, and the Interest packet. Note that the Interest packet is not a parameter when entering the pipeline. The pipeline steps either use the PIT entry directly to perform checks, or obtain a reference to an Interest stored inside the PIT entry.

This pipeline first inserts an out-record in the PIT entry for the specified outgoing face, or update an existing out-record for the same face; in either case, the PIT out-record remembers the Nonce of the last outgoing Interest packet, which is useful for matching incoming Nacks, as well as an expiration timestamp which is the current time plus InterestLifetime. Finally, the Interest is sent to the outgoing face.

4.2.6 Interest Finalize Pipeline

This pipeline is implemented in `Forwarder::onInterestFinalize` method and is entered from the *expiry timer*.

The pipeline first determines whether any Nonces recorded in the PIT entry need to be inserted into the Dead Nonce List (Section 3.5). 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 cannot loop back.

We can take further 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 is determined that one or more Nonces should be inserted into the Dead Nonce List, tuples of Name and Nonce are added to the Dead Nonce List (Section 3.5.1).

Finally, the PIT entry is removed from the PIT.

4.3 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 of Data packets

4.3.1 Incoming Data Pipeline

The incoming Data pipeline is implemented in `Forwarder::onIncomingData` method and is entered from `Forwarder::startProcessData` method, which is triggered by `Face::afterReceiveData` signal. The input parameters to this pipeline include a Data packet and its incoming Face.

As illustrated in Figure 11, this pipeline includes the following steps:

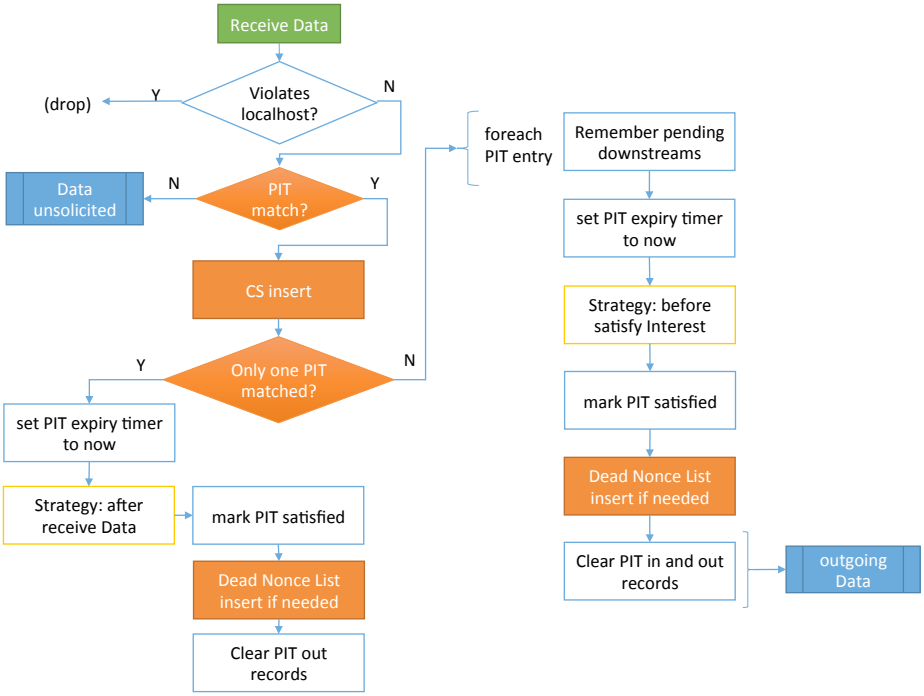


Figure 11: Incoming Data pipeline

1. The first step is to check if the Data violates `/localhost` scope [10]. If the Data comes from a non-local Face but its name starts with `/localhost` prefix, the scope is violated and the Data is dropped.

   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.

2. Then, the pipeline checks if the Data matches PIT entries, by using Data Match algorithm (Section 3.4.2). If no matching PIT entry is found, the Data is given to *Data unsolicited pipeline* (Section 4.3.2); if matching PIT entries are found, the Data is inserted into the ContentStore. Note that even if the pipeline inserts the Data to the ContentStore, whether the Data is stored and how long it stays in the ContentStore is determined by ContentStore admission and replacement policy.[5]

3. Next, the pipeline checks if only one matching PIT entry is found or more than one matching PIT entry is found. This check decides if forwarding strategy is cabable of manipulating Data forwarding. Normally, only one matching PIT entry would be found. More than one matching PIT entry means more than one forwarding strategy is possible to manipulate Data forwarding, which is forbidden to avoid potential conflicts among strategies.

4. If only one matching PIT entry is found, meaning that only one forwarding strategy is controlling Data forwarding, the pipeline will set the PIT expiry timer to now, invoke *after receive Data* trigger of the strategy, mark PIT satisfied, insert dead nonce list if needed, and clear PIT entry's out records.

[5]The current implementation has a fixed "admit all" admission policy, and "priority FIFO" as replacement policy, see Section 3.3.

5. If more than one matching PIT entry is found, for each matching PIT entry, the pipeline will remember its pending downstreams, set the PIT expiry timer to now, invoke *before satisfy Interest* trigger of the strategy, mark PIT satisfied, insert dead nonce list if needed, and clear PIT entry's in and out records. Finally, the pipeline will forward the Data to each pending downstream, except that if the pending downstream Face is the same as the incoming Face of the Data, and the Face is not ad-hoc.

4.3.2 Data Unsolicited Pipeline

This pipeline is implemented in `Forwarder::onDataUnsolicited` method and is entered from the *incoming Data pipeline* (Section 4.3.1) when a Data packet is found to be unsolicited. The input parameters to this pipeline include a Data packet, and its incoming Face.

The current *unsolicited Data policy* is consulted to decide whether to drop the Data, or add it to the ContentStore. By default, NFD forwarding is configured with a "drop-all" policy which drops all unsolicited Data, as they pose a security risk to the forwarder.

There can be cases where unsolicited Data packets need to be accepted. The policy can be changed in NFD configuration file at `tables.cs_unsolicited_policy` key.

4.3.3 Outgoing Data Pipeline

This pipeline is implemented in `Forwarder::onOutgoingData` method and pipeline is entered from *incoming Interest pipeline* (Section 4.2.1) when a matching Data is found in ContentStore and from *incoming Data pipeline* (Section 4.3.1) when the incoming Data matches more than one PIT entry. The input parameters to this pipeline include a Data packet, and the outgoing Face.

This pipeline contains the following steps:

1. The Data is first checked for `/localhost` scope [10]: Data packets with a `/localhost` prefix cannot be sent to a non-local face.[6] `/localhop` scope is not checked here, because its scope rules do not restrict outgoing Data.

2. The next step is reseved for the traffic manager actions, such as to perform traffic shaping, etc. The current version does not include any traffic management, but it is planned to be implemented in a future release.

3. Finally, the Data packet is sent via the outgoing Face.

4.4 Nack Processing Path

Nack processing in NFD is split into these pipelines:

- Incoming Nack: processing of incoming Nacks
- Outgoing Nack: preparation and sending out of Nacks

4.4.1 Incoming Nack Pipeline

The incoming Nack pipeline is implemented in `Forwarder::onIncomingNack` method and is entered from `Forwarder::startProcessNack` method, which is triggered by `Face::afterReceiveNack` signal. The input parameters to this pipeline include a Nack packet and its incoming Face.

First, if the incoming face is not a point-to-point face, the Nack is dropped without further processing, because the semantics of Nack is only defined on a point-to-point link.

The Interest carried in the Nack and its incoming face is used to locate a PIT out-record for the same face where the Interest has been forwarded to, and the last outgoing Nonce was same as the Nonce carried in the Nack. If such an out-record is found, it's marked *Nacked* with the Nack reason. Otherwise, the Nack is dropped because it's no longer relevant.

The effective strategy responsible for the PIT entry is determined using Find Effective Strategy algorithm (Section 3.6.1). The selected strategy is then triggered for the *after receive Nack* procedure with the Nack packet, its incoming Face, and the PIT entry (Section 5.1.1).

[6] This check is only useful in a specific scenario (see NFD Bug 1644).

4.4.2 Outgoing Nack Pipeline

The outgoing Nack pipeline is implemented in `Forwarder::onOutgoingNack` method and is entered from `Strategy::sendNack` method which handles *send Nack action* for strategy (Section 5.1.2). The input parameters to this pipeline include a PIT entry, an outgoing Face, and the Nack header.

First, the PIT entry is queried for an in-record of the specified outgoing face (downstream). This in-record is necessary because protocol requires the last Interest received from the downstream, including its Nonce, to be carried in the Nack packet. If no in-record is found, abort this procedure, because the Nack cannot be sent without this Interest.

Second, if the downstream is not a point-to-point face, abort this procedure, because the semantics of Nack is only defined on a point-to-point link.

After both checks are passing, a Nack packet is constructed with the provided Nack header and the Interest from the in-record, and sent through the face. The in-record is erased as it has been "satisfied" by the Nack, and no further Nack or Data should be sent to the same downstream unless there's a retransmission.

4.5 Helper Algorithms

Several algorithms used in forwarding pipelines and multiple strategies are implemented as helper functions. As we identify more reusable algorithms, they will be implemented as helper functions as well, rather than repeating the code in several places.

`nfd::fw::wouldViolateScope` determines whether forwarding an Interest out of a face would violate namespace-based scope control.

`nfd::fw::findDuplicateNonce` searches a PIT entry to see if there's a duplicate Nonce in any in-record or out-record.

`nfd::fw::hasPendingOutRecords` determines whether a PIT entry has an out-record that is still pending, i.e. neither Data nor Nack has come back.

4.5.1 FIB lookup

`Strategy::lookupFib` implements a FIB lookup procedure with consideration of forwarding hint. The procedure is:

1. If the Interest does not carry a forwarding hint and hence does not require mobility processing, FIB is looked up using Interest Name (Section 3.1.1, Longest Prefix Match algorithm). FIB guarantees that Longest Prefix Match returns 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 the strategy rejecting the Interest.
2. If the Interest carries a forwarding hint, it is processed for mobility support. [7]
3. The procedure looks up the FIB using each delegation name contained in the forwarding hint, and returns the first matched FIB entry that has at least one nexthop. It does not distinguish whether the Interest is in consumer region or default-free zone.
4. In case none of the delegation names match a FIB entry with at least one nexthop, an empty FIB entry is returned.

A limitation of current implementation is that, when an Interest reaches the first default-free router, which delegation to use is solely determined by this FIB lookup procedure according to the routing cost in the FIB. Ideally, this choice should be made by the strategy which can take current performance of different upstreams into consideration. We are exploring a better design in this aspect.

[7] Presence of a forwarding hint at this point indicates the Interest has not reached the producer region, because the forwarding hint should have been stripped in *incoming Interest pipeline* when entering the producer region.

5 Forwarding Strategy

In NFD forwarding, forwarding strategies provide the intelligence to make decision on whether, when, and where to forwarding Interests. Forwarding strategies, together with forwarding pipelines (Section 4), make up the packet processing logic in NFD. The forwarding strategy is triggered from the forwarding pipelines when a decision about Interest forwarding needs to be made. In addition, the strategy can receive notifications on the outcome of its forwarding decisions, e.g., when a forwarded Interest is satisfied, timed out, or brings back a Nack.

Our experience with NDN applications has shown that different applications need different forwarding behaviors. For example, a file retrieval application wants to retrieve contents from a content source with highest bandwidth, an audio chat application wants the lowest delay, and a dataset synchronization library (such as ChronoSync) wants to multicast Interests to all available faces in order to reach its peers. The need for different forwarding behaviors motivates us to have multiple forwarding strategies in NFD.

Despite having multiple strategies in NFD, the forwarding decision of an individual Interest must be made by a single strategy. NFD implements per-namespace strategy choice. An operator may configure a specific strategy for a name prefix, and Interests under this name prefix will be handled by this strategy. This configuration is recorded in the Strategy Choice table (Section 3.6), and is consulted by the forwarding pipelines.

Currently, the choice of forwarding strategy is a local configuration. The same Interest may be handled by completely different strategies on different nodes. Alternatives to this approach are being discussed as of Jan 2016. One notable idea is *routing annotations* where the routing announcement carries an indication about the preferred strategy.

5.1 Strategy API

Conceptually, a strategy is a program written for an abstract machine, the strategy API. The "language" of this abstract machine contains standard arithmetic and logical operations, as well as interactions with the rest of NFD. The state of this abstract machine is stored in NFD tables.

Each NFD strategy is implemented as a subclass of `nfd::fw::Strategy` class, which provides the API for the interaction between the implemented strategy and the rest of NFD. This API is the only way a strategy can access NFD components, therefore available functionality in the strategy API determines what NFD strategies can or cannot do.

A strategy is invoked through one of the *triggers* (Section 5.1.1). The forwarding decision is made with *actions* (Section 5.1.2). Strategies are also allowed to store information on certain table entries (Section 5.1.3).

5.1.1 Triggers

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

When an Interest is received, passes necessary checks, and needs to be forwarded, *Incoming Interest pipeline* (Section 4.2.1) invokes this trigger with the Interest packet, its incoming face, and the PIT 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. Most strategies need a FIB entry to make this decision, which can be obtained by calling `Strategy::lookupFib` accessor function. 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.[8] If the strategy concludes that this Interest cannot be forwarded, it should invoke the `Strategy::setExpiryTimer` action and set the timer to expire immediately, so that the PIT entry can eventually be deleted.

[8] **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.

After Content Store Hit Trigger

This trigger is declared as `Strategy::afterContentStoreHit` method. The base class provides a default implementation that sends the matching Data to the downstream.

After Receive Data Trigger

This trigger is declared as `Strategy::afterReceiveData` method. This trigger is invoked when an incoming Data satisfies exactly one PIT entry, and gives the strategy full control over Data forwarding. When this trigger is invoked, the Data has been verified to satisfy the PIT entry, and the PIT entry expiry timer is set to fire immediately. The base class implementation invokes the `Strategy::beforeSatisfyInterest` trigger and then returns the Data to all downstream faces.

Inside this trigger:

- A strategy should return Data to downstream nodes via the methods `Strategy::sendData` or `Strategy::sendDataToAll`.
- A strategy can modify the Data as long as it still satisfies the PIT entry, such as adding or removing congestion marks.
- A strategy can delay Data forwarding by prolonging the PIT entry lifetime via `Strategy::setExpiryTimer`, and forward Data before the PIT entry is erased.
- A strategy can collect measurements about the upstream.
- A strategy can collect responses from additional upstream nodes by prolonging the PIT entry lifetime every time a Data is received. Note that only one Data should be returned to each downstream node.

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 4.3.1) 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 measurements for the expiring 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 4.3.1) invokes this trigger with the PIT entry. The PIT entry always represents a pending Interest.

After Receive Nack Trigger

This trigger is declared as `Strategy::afterReceiveNack` method. The base class provides a default implementation that does nothing, which means all incoming Nacks will be dropped, and will not be passed to downstreams. A subclass can override this method if the strategy needs to be invoked for this trigger.

When an Interest is received, and passes necessary checks, *Incoming Nack pipeline* (Section 4.4.1) invokes this trigger with the Nack packet, its incoming face, and the PIT entry. At that time, the following conditions hold:

- The Nack is received in response an forwarded Interest.
- The Nack has been confirmed to be a response to the last Interest forwarded to that upstream, i.e. the PIT out-record exists and has a matching Nonce.
- The PIT entry is under a namespace managed by this strategy. [9]
- The NackHeader has been recorded in the *Nacked* field of the PIT out-record.

After being triggered, the strategy could typically do one of the following:

- Retry the Interest by forwarding it to the same or different upstream(s), by invoking *send Interest* action. Most strategies need a FIB entry to find out potential upstreams, which can be obtained by calling `Strategy::lookupFib` accessor function.
- Give up and return the Nack to downstream(s), by invoking *send Nack* action.
- Do nothing for this Nack. If some but not all upstreams have Nacked, the strategy may want to wait for Data or Nack from more upstreams. In this case, it's unnecessary for the strategy to record the Nack in its own StrategyInfo, because the Nack header is available on the PIT out-record in *Nacked* field.

[9] Note: The Interest is not necessarily forwarded by this strategy. In case the effective strategy is changed after an Interest forwarded, and then a Nack comes back, the new effective strategy would be triggered.

5.1.2 Actions

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

Send Interest action

This action is implemented as `Strategy::sendInterest` method. Parameters include a PIT entry, an outgoing face, and the Interest packet which must match the PIT entry. This action enters the *Outgoing Interest pipeline* (Section 4.2.5).

The strategy is responsible for checking that forwarding of the Interest does not violate namespace-based scope control [10]. Typically, the strategy should invoke this action with one of the incoming Interest packets found in the PIT in-records, although the strategy may make a copy of an Interest and modify its guider fields, as long as the Interest still match the PIT entry.

Send Data action

This action is implemented as `Strategy::sendData` method. Parameters include a PIT entry, a Data, and a downstream face.

This action deletes the PIT entry's in-record and enters the *outgoing Data pipeline* (Section 4.3.3).

In many cases the strategy may want to send Data to every downstream. `Strategy::sendDataToAll` method is a helper for this purpose, which accepts a PIT entry, a Data packet, and the face where the packet came from. Note that `sendDataToAll` will send the Data to each pending downstream, unless the pending downstream face is the same as the incoming face of the Data and the face is not *ad hoc*.

Send Nack action

This action is implemented as `Strategy::sendNack` method. Parameters include a PIT entry, a downstream face, and a Nack header.

This action enters the *outgoing Nack pipeline* (Section 4.4.2). An in-record for the downstream face should exist in the PIT entry, and a Nack packet will be constructed by taking the last incoming Interest from the PIT in-record and adding the specified Nack header. If the PIT in-record is missing, this action has no effect.

In many cases the strategy may want to send Nacks to every downstream (that still has an in-record). `Strategy::sendNacks` method is a helper for this purpose, which accepts a PIT entry and a Nack header. Calling this helper method is equivalent to invoking *send Nack* action for every downstream.

5.1.3 Storage

Strategies are allowed to store arbitrary information on PIT entries, PIT in-records, PIT out-records, and Measurements entries, all of which are derived from `StrategyInfoHost` type[10] . Inside the triggers, the strategy already has access to PIT entry and can lookup desired in-records and out-records. Measurement entries (Section 3.7) can be accessed via `Strategy::getMeasurements` method; a strategy's access is restricted to Measurements entries under the namespace(s) under its control (Section 3.7.2).

Strategy-specific information should be contained in a subclass of `StrategyInfo`. At anytime, the strategy may call `getStrategyInfo`, `insertStrategyInfo`, and `eraseStrategyInfo` on a `StrategyInfoHost` to store and retrieve the information. Note that the strategy must ensure that each `StrategyInfo` has a distinct TypeId; if the same TypeId is assigned to multiple types, 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 control.

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

5.2 List of Strategies

NFD comes with these strategies:

- best route strategy (`/localhost/nfd/strategy/best-route`, Section 5.2.1) sends Interest to lowest cost upstream.
- multicast strategy (`/localhost/nfd/strategy/multicast`, Section 5.2.2) sends every Interest to every upstream.
- NCC strategy (`/localhost/nfd/strategy/ncc`, Section 5.2.3) is similar to CCNx 0.7.2 default strategy.

[10] "Host" is in the sense of holding strategy information, not an endpoint/network entity.

- access router strategy (`/localhost/nfd/strategy/access`, Section 5.2.4) is designed for local site prefix on an access/edge router.
- Adaptive SRTT-based Forwarding (ASF) strategy (`/localhost/nfd/strategy/asf`, Section 5.2.5) sends Interests to the upstream with the lowest measured SRTT and periodically probes alternative upstreams.

Since the objective of NFD is to provide a framework for easy experimentation, the list of the provided strategies is in no way comprehensive and we encourage implementation and experimentation of new strategies. Section 5.3 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.

5.2.1 Best Route Strategy

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

Interest forwarding The strategy forwards a new Interest to the lowest-cost eligible Face. After the new Interest is forwarded, a similar Interest with same Name, Selectors, and Link but different Nonce would be suppressed if it's received during a retransmission suppression interval. However, a similar Interest received after the suppression interval is called a "retransmission", and is forwarded to the lowest-cost eligible Face that is not previously used; if all Faces have been used, it is forwarded to an eligible Face that was used earliest.

Note that applying the suppression and retransmission mechanisms does not distinguish whether the Interest come from the same downstream or a different downstream. Although the former is typically a retransmission from the same consumer and the latter is typically from a different consumer making use of NDN's built-in Data multicast, there's no prominent difference in terms of forwarding, so they are processed alike.

Retransmission suppression interval Instead of forwarding every incoming Interest, the retransmission suppression interval is imposed to prevent a malicious or misbehaving downstream from sending too many Interests end-to-end. The retransmission suppression interval should be chosen so that it permits reasonable consumer retransmissions, while prevents DDoS attacks by overly frequent retransmissions.

We have identified three design options for setting the retransmission suppression interval:

- A **fixed interval** is the simplest, but it's hard to find a balance between reasonable consumer retransmissions and DDoS prevention.
- Doing **RTT estimation** would allow a retransmission after the strategy believes the previous forwarded Interest is lost or otherwise won't be answered, but RTT estimations aren't reliable, and in case the consumer applications are also using RTT estimation to schedule their retransmissions, this results in double control loop and potentially unstable behavior.
- Using **exponential back-off** gives consumer applications control over the retransmission, and also effectively prevents DDoS. Starting with a short interval, the consumer can retransmit quickly in low RTT communication scenario; the interval goes up after each accepted retransmission, so an attacker cannot abuse the mechanism by retransmitting too frequently.

We finally settled with the exponential back-off algorithm. The initial interval is set to 10 milliseconds. After each retransmission being forwarded, the interval is doubled (multiplied by 2.0), until it reaches a maximum of 250 milliseconds.

Nack generation The best route strategy uses Nack to improve its performance. When an Interest can not be forwarded to an eligible Face, a Nack will be returned to the incoming Face to notify the downstream.

The eligibility of an Face depends on its type. An ad hoc Face is *eligible* if forwarding an Interest to it does not violate scope [10]. Moreover, for multicast and point-to-point Face, besides the scope control, the outgoing Face should not be the incoming Face.

TODO#3420 add "face is UP" condition. If there's no eligible Face available, the strategy rejects the Interest, and returns a Nack to the downstream with reason *no route*.

Currently, the Nack packet does not indicate which prefix that the node has no route to reach, because it's non-trivial to compute this prefix. Also, Nacks will not be returned to multicast or ad hoc Faces.

Nack processing Upon receiving an incoming Nack, the strategy itself does not retry the Interest with other nexthops (because "best route" forwards to only one nexthop for each incoming Interest), but informs the downstream(s) as quickly as possible. If the downstream/consumer wants, it can retransmit the Interest, and the strategy would retry it with another nexthop.

Specifically, depending on the situation of other upstreams, the strategy takes one of these actions:

- If all pending upstreams have Nacked, a Nack is sent to all downstreams.
- If all but one pending upstream have Nacked, and that upstream is also a downstream, a Nack is sent to that downstream.

- Otherwise, the strategy continues waiting for the arrival of more Nacks or Data.

To determine what situation a PIT entry is in, the strategy makes use of the *Nacked* field (Section 3.4.1) on PIT out-records, and does not require extra measurement storage.

The second situation, "all but one pending upstream have Nacked and that upstream is also a downstream", is introduced to address a specific "live deadlock" scenario, where two hosts are waiting for each other to return the Nack. More details about this scenario can be found at http://redmine.named-data.net/issues/3033#note-7.

In the first situation, the Nack returned to downstreams need to indicate a reason. In the easy case where there's only one upstream, the reason from this upstream is passed along to downstream(s). When there are multiple upstreams, the **least severe** reason is passed to downstream(s), where the severity of Nack reasons are defined as: Congestion <Duplicate <NoRoute. For example, one upstream has returned Nack-NoRoute and the other has returned Nack-Congestion. This forwarder choose to tell downstream(s) "congestion" so that they can retry with this path after reducing their Interest sending rate, and this forwarder can forward the retransmitted Interests to the second upstream at a slower rate and hope it's no longer congested. If we instead tell downstream(s) "no route", it would make downstreams believe that this forwarder cannot reach the content source at all, which is inaccurate.

5.2.2 Multicast Strategy

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

Interest forwarding The strategy forwards (multicasts) a new Interest to all upstream faces from the list of nexthop records in the FIB entry that do not violate scope. Similar to Best Route Strategy (5.2.1), exponential retransmission suppression is used to prevent too many Interests. A same Interest (same Name, Selectors, and Link) with different Nonce is suppressed if it's received during a retransmission suppression interval. A same Interest received after the suppression interval is treated as a new Interest. However, unlike Best Route Strategy the suppression is done per upstream rather than per pit entry. For example: if NFD gets an interest on a face and the application wants to send the same interest back (with a new Nonce) within the suppression period, then it will not be suppressed for that face because upstream of the interest is different and no suppression has been calculated on it yet.

Nack generation If there is no eligible upstream, the Interest is rejected and a Nack with reason *no route* is returned to the downstream.

Nack processing Nack processing is exactly the same as Best Route Strategy (5.2.1).

5.2.3 NCC Strategy

The NCC strategy [11] is an reimplementation of CCNx 0.7.2 default strategy [13]. It has similar algorithm but is not guaranteed to be equivalent. This strategy is implemented as `nfd::fw::NccStrategy` class.

5.2.4 Access Router Strategy

The access router strategy (aka access 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::fw::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 5.2.1); The same mechanism also allows the strategy to deal with producer mobility.

Motivation and Use Case One of NDN's benefits is that it does not require precise routing: a route indicates that contents under a certain prefix is *probably* available from a nexthop. The property brings a challenge to forwarding strategy design: if an Interest matches multiple routes, which nexthop should we forward it to?

[11] NCC does not stand for anything; it is just CCN backwards.

• One option is to forward the Interest to all the nexthops. This is implemented in the multicast strategy (Section 5.2.2). The Data, if available from any of these nexthops, can be retrieved with shortest delay. However, since every Interest is forwarded to many nexthops, it has significant bandwidth overhead.

• Another option is to forward the Interest to only one nexthop, and if it doesn't work, try another nexthop. This is implemented in the best route strategy (Section 5.2.1). While bandwidth overhead on the upstream side is reduced, since each incoming Interest can be forwarded to only one nexthop, the consumer has to retransmit an Interest multiple times in order to reach the correct nexthop. It's even worse when an upstream does not return a Nack or the Nack is lost; in this case, the consumer has to wait for a timeout (either based on *InterestLifetime* or RTO) before it can decide to retransmit, causing further delays. In addition, repeated consumer retransmissions increase bandwidth overhead on the downstream side.

• Between these two extremes, we want to design a strategy with a trade-off between delay and bandwidth overhead.

On gateway/access/edge routers of the NDN testbed, despite the availability of *automatic prefix propagation* (Section 7.6), the majority of Interest forwarding from an access router to end hosts are relying on routes installed by the `nfd-autoreg` tool: when an end host connects to the router, this tool installs a route for the *local site prefix* toward this end host. Since the local site prefix is statically configured, these routes will have the same prefix, and an Interest under this prefix will match all these routes. This is an extreme case of imprecise routing: every end host is a nexthop of a very broad prefix, and they are many end hosts. The multicast strategy would forward every Interest toward all end hosts, even if only a small subset of them can serve the content. The best route strategy would require the consumer to retransmit, in the worst case, as many times as the number of connected end hosts.

The access strategy is designed for this use case. We want to find contents available on end hosts without forwarding every Interest to all end hosts, and require minimal consumer retransmissions.

How Access Router Strategy Works The basic idea of this strategy is to multicast the first Interest, learn which nexthop can serve the contents, and forward the bulk of subsequent Interests toward the chosen nexthop; if it does not respond, the strategy starts multicasting again to find an alternate path.

This idea is somewhat similar to Ethernet self-learning, but there are two challenges:

• Ethernet switch learns the mapping from an **exact** address to a switch port, but NDN router needs to learn the mapping from a **prefix** of the Interest name to a nexthop in order to be useful for subsequent Interests. What prefix can we learn from Interest-Data exchanges?

• In Ethernet, each address is reachable via only one path, and a moved host floods an ARP packet to inform the network about its new location. In NDN, each prefix can be reachable via multiple paths, and it's the strategy's responsibility to detect the chosen nexthop is no longer working so it can start finding an alternate path; a moved producer won't actively inform the network of its new location, and a failed producer has no way to ask other producers to flood an announcement.

Granularity Problem and Solution The first challenge, what prefix can we learn from Interest-Data exchanges, is called the **granularity problem**. There are a few differnet approaches to solve this problem, but we pick a simple solution in the access strategy: learned nexthop is associated with the Data name minus the last component. A commonly adopted NDN naming convention [14] puts the version number and segment number as the last two components of Data names. Under this naming convention, the Data name minus the last component covers all segments of a versioned object. We believe it's safe to assume that all segments of a version is available on the same upstream, and thus choose this solution.

As a consequence of choosing this simple solution, the access strategy could perform badly if the application does not follow the above naming convention. Most notably, NDN-RTC [15] realtime conference library (version 1.3 when we did this analysis in Sep 2015 [12]) adopts a naming scheme which expresses Interests similar to `/ndn/edu/ucla/remap/ndnrtc/user/remap/streams/camera\_1469c/mid/key/2991/data/\%00\%00` and generates Data names similar to `/ndn/edu/ucla/remap/ndnrtc/user/remap/streams/camera\_1469c/mid/key/2991/data/\%00\%00/23/89730/86739/5/27576`. In this name, the component before `data` is the frame number (`2991` in the example), and the component after `data` is the segment number (`%00%00` in the example); every Data name has 5 additional components than the Interest name, which carries additional signaling information for application use.

Admittedly, this is an example of bad naming design because although NDN supports in-network name discovery, the majority of Interests should carry complete names [16]; appending application-layer metadata onto the Data name violates this principle. This naming design makes the access strategy multicast every Interest, because the chosen nexthop of an Interest is recorded on the Data name minus the last component (`/ndn/edu/ucla/remap/ndnrtc/user/remap/streams/camera\_1469c/mid/key/2991/data/\%00\%00/23/89730/86739/5`), but the next Interest is `/ndn/edu/ucla/remap/ndnrtc/user/remap/streams/camera\_1469c/mid/key/2991/data/\%00\%01` which does not fall under the prefix where the chosen nexthop is recorded. As a result, the next Interest is still treated as an "initial Interest" and multicast.

[12] More details of this analysis can be found at http://redmine.named-data.net/issues/3219.

However, even if we change NDN-RTC's naming scheme so that the Data name is same as the Interest name (i.e. ends with the segment number), AccessStrategy would only perform slightly better. The chosen nexthop would be recorded on `/ndn/edu/ucla/remap/ndnrtc/user/remap/streams/camera\_1469c/mid/key/2991/data`, which matches subsequent Interests for other segments within the same frame, but cannot match Interests for other frames (such as `/ndn/edu/ucla/remap/ndnrtc/user/remap/streams/camera\_1469c/mid/key/2992/data/\%00\%00`). Within a 700Kbps video stream, the frame number changes more than 50 times per second, and there are about 25 segment numbers per frame. This means, the access strategy would multicast about 1250 times per second with NDN-RTC 1.3's naming scheme; each of those multicast Interests would reach every end host that has a nexthop added by `nfd-autoreg`, increase their bandwidth usage and CPU overhead. Changing the Data name to be same as the Interest name would reduce multicasts to 50 times per second, which is still inefficient.

The fundamental reason of access strategy's inefficiency with NDN-RTC 1.3 is the mismatch between the assumption of naming convention behind our granularity solution and the application naming scheme: the chosen nexthop is recorded at a longer prefix than the actual prefix. It's also possible for the chosen nexthop to be recorded at a too-short prefix, but no known application can suffer from the problem; however, if we change the access strategy to record the chosen nexthop at shorter prefixes (such as dropping last 3 components of the Data name, which would accommodate NDN-RTC's naming scheme after modification), this problem would happen.

It's our future work to explore other solutions to the granularity problem.

Failure Detection The second challenge, how to detect the chosen nexthop is no longer working, is currently solved with a combination of RTT-based timeout and consumer retransmission.

RTT-based timeout We maintain RTT estimations following TCP's algorithm. If a nexthop does not return Data within the Retransmission Timeout (RTO) computed from the RTT estimator, we consider the chosen nexthop to have failed, and multicast the Interest toward all nexthops (except the chosen nexthop, because otherwise the upstream would see a duplicate Nonce).

There are two kinds of RTT estimators: per-prefix RTT estimators, and per-face RTT estimators. On each prefix where we learn a chosen nexthop (i.e. the Data name minus the last component), we maintain a per-prefix RTT estimator; if a subsequent Interest is not satisified within the RTO computed from this per-prefix RTT estimator, the strategy would multicast the Interest. A per-prefix RTT estimator reflects the RTT of the current chosen nexthop; if a different nexthop is chosen, this RTT estimator shall be reset.

In order to have a good enough initial estimation, when we reset a per-prefix RTT estimator, instead of starting with a set of default values, we copy the state from the per-face RTT estimator, which is maintained for each upstream face and not associated with name prefixes. This design assumes different prefixes served by the same upstream face have similar RTT; this assumption is one reason that this strategy design is limited for use on one hop, and is unfit for usage over multiple hops.

consumer retransmission Some consumer applications have application-layer means to detect a non-working path. If the consumer believes the current path is not working, it could retransmit the Interest with a new Nonce. Unless the retransmissions are too frequent, the access strategy would take an retransmission as a signal that the chosen nexthop has stopped working, and multicast the Interest right away.

5.2.5 ASF Strategy

The ASF strategy is designed to prioritize upstreams based on their performance in Data retrieval delay. The strategy sends Interests to the upstream with the lowest measured SRTT and periodically probes alternative upstreams to gather SRTT measurements for unused upstreams [17]. This strategy is implemented as `nfd::fw::AsfStrategy` class.

5.3 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 5.3.1). The procedure of developing a new built-in strategy is outlined in Section 5.3.2.

5.3.1 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 NDNLPv2's `NextHopFaceId` field to 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 decides 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 [18, 19].

5.3.2 Develop a New Strategy

The initial step in creating a new strategy is to create a class, say `MyStrategy` that is derived from `nfd::fw::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. The class should be placed in `daemon/fw` directory of NFD codebase, and needs to be compiled into NFD binary; dynamic loading of external strategy binary may be available in the future.

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 existing strategy classes, such data structures are declared as nested classes as it provides natural grouping and scope protection of the strategy-specific entity, but your strategy 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).

After creating the data structures, you may implement the *triggers* with the desired strategy logic. When implementing strategy logic, refer to Section 5.1.1 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 `nullptr` (Section 5.1.3). 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.
- 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 5.1.1) 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.
- If implementing a timer, do not retain shared pointers of `Face`, `pit::Entry`, and other items. Instead, retain a weak pointer. In the timer callback, `.lock()` the weak pointer into a shared pointer, check it's not `nullptr` prior to use.
- Timers must be cancelled in the destructor of the stored item (Section 5.1.3). 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.
- The strategy is responsible for implementing namespace-based scope control.
- The strategy is responsible for performing congestion control.

Finally, make your strategy available for selection:

1. Choose an NDN name to identify the strategy. It is recommended to use a name like `ndn:/localhost/nfd/strategy/my-strategy` and append a version number. A version number component is required. The version number should be incremented whenever there is a non-trivial change to the strategy behavior.
2. Optionally, the strategy may accept parameters (Section 3.6). Parameters are used for things like "minimum suppression timer", "ceiling of exponential back-off", and so on. They are passed to the strategy constructor, as part of the strategy instance name after the version component.

3. Register the strategy class with `NFD_REGISTER_STRATEGY` macro.

After that, the strategy is ready to use and can be activated by a Strategy Choice management command (Section 6.6), such as `nfdc set-strategy`.

6 Management

NFD management offers the capability to monitor and control NFD through a configuration file and an Interest-based API.

NFD management is divided into several management modules. Each management module is responsible for a subsystem of NFD. It can initialize the subsystem from a section in the NFD configuration file, or offer an Interest-based API to allow others to monitor and control the subsystem.

Section 6.1 gives an overview of the NFD configuration file and of the basic mechanisms in NFD Management protocol [3]. Section 6.2 describes the dispatcher and authenticator used in the protocol implementation. Section 6.3, 6.4, 6.5, 6.6 explain the details of the four major management modules. Section 6.7 introduces two additional management modules that are used in configuration file parsing. Section 6.8 offers ideas on how to extend NFD management.

6.1 Protocol Overview

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 command 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. These actions are defined in NFD Management Protocol [3] as *status datasets*, and are currently implemented in NFD as a simple Interest/Data exchange.

6.2 Dispatcher and Authenticator

As shown in figure 12, managers utilize `ndn::Dispatcher` as an abstraction to deal with common Interest/Data processing, so that they can focus on performing the required low-level operations on the module of NFD that they manage.

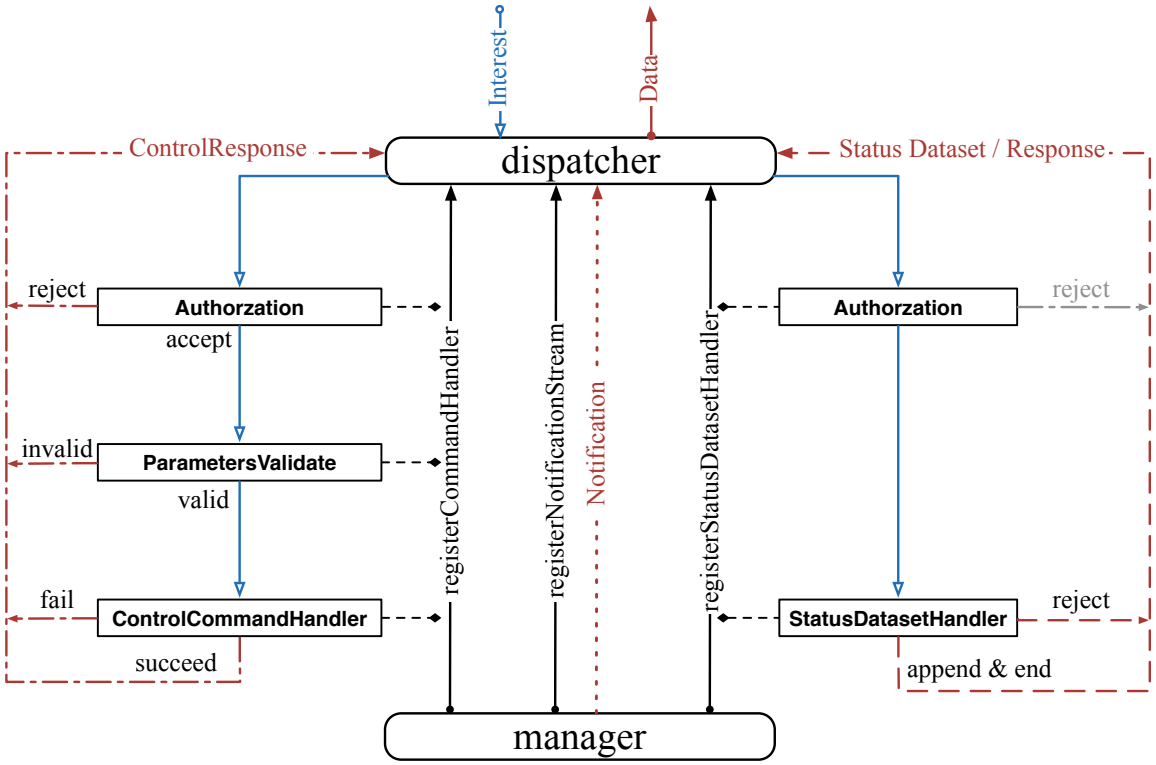


Figure 12: Overview of the manager Interest/Data processing via the dispatcher.

More specifically, a manager always consists of a series of handlers, each of which is responsible for dealing with a *control command* request or a *status dataset* request. Per the management protocol, these requests 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-nexthop` directs the FIB Manager to add a next hop (command arguments follow the verb).

A manager registers a handler with the dispatcher with a partial name that is composed of the manager's name and a verb. After all managers have registered their handlers, the dispatcher creates full prefixes using those partial names plus each registered top-level prefix (e.g., `/localhost/nfd`), and then sets up interest filters for them. When a request arrives to the dispatcher, it is directed to some management handler after being processed by interest filters. On the other hand, all types of responses, such as of *control responses*, *status datasets*, etc., go through the dispatcher, for concatenation, segmentation, and signing.

For *control commands*, a handler is always registered along with an `authorization` method and a `parametersValidate` method. A request can be directed to the handler if and only if it is accepted by `authorization` and its *control parameters* can be validated by `parametersValidate`. For *status datasets*, there is no need to validate any parameters, therefore the `authorization` is made to accept all requests[13].

Furthermore, managers that produce notifications, such as the FaceManager, will also register *notification streams* to the dispatcher. This type of registration returns a `postNotification`, through which the manager will only need to generate the notification content, leaving the rest of the work (constructing the packet, signing, etc.) to the dispatcher.

6.2.1 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 a command handler, to register a status dataset handler, and to register notification streams. Moreover, `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`, `strategy-choice`). This privilege is used when specifying the set of authorized capabilities for a given NDN identity certificate in the configuration file.

6.2.2 Command Authenticator

The `CommandAuthenticator` provides authentication and authorization of control commands.

Rules for authentication and authorization are specified in the "`authorizations`" section of the NFD configuration file. In that section, each `authorize` subsection lists a single NDN certificate and a set of privileges. An example of an `authorize` subsection is:

```
authorize
{
  certfile keys/operator.ndncert
  privileges
  {
    faces
    strategy-choice
  }
}
```

Unlike most other validators, `CommandAuthenticator` does not attempt to retrieve signing certificates during the authentication step. This is because the authenticator needs to operate before routes are established. Thus, every certificate used to sign control commands must be explicitly specified in the configuration. The value in the "`certfile`" key is a filename that refers to an NDN certificate (version 1); it can be either an absolute path, or a path relative to the configuration file. For convenience in non-production systems, a special value "`any`" can be used in "`certfile`", granting any signing certificate the privileges listed in the "`privileges`" subsection. Under the "`privileges`" subsection, a set of privileges can be specified, where each privilege is the management module name. Supported privileges are: `faces`, `fib`, `strategy-choice`.

The NFD initialization procedure creates a `CommandAuthenticator` instance and invokes `setConfigFile`, in which the authenticator registers itself as a processor of the "`authorizations`" section of NFD configuration file. The authenticator instance is then given to the constructor of each manager that accepts control commands. The manager invokes `CommandAuthenticator::makeAuthorization` method for each accepted command verb, which returns an `ndn::mgmt::Authorization` callback function that can be used with the dispatcher; this `makeAuthorization` invocation also tells the

[13] This may be changed whenever data access control is desired.

authenticator which management module names are supported. Finally, the initial configuration is parsed, which populates the internal data structures of the authenticator.

When a control command arrives, the dispatcher passes the command Interest to the `ndn::mgmt::Authorization` callback function returned by `CommandAuthenticator::makeAuthorization`. This authorization callback has four steps:

1. Based on management module and command verb, find out what signing certificates are allowed. Although the dispatcher does not directly pass management module and command verb to the authorization callback, this information is captured when the authorization callback was initially constructed, because the manager invokes `CommandAuthenticator::makeAuthorization` once for every distinct command verb and thus each gets a different authorization callback instance.
2. If "`certfile any`" applies to this management module and command verb, the control command is authorized, and the remaining steps are skipped.
3. Check whether the Interest is signed by one of the listed certificates by inspecting the KeyLocator field. If not, the control command is rejected.
4. Pass the Interest to the `ndn::security::CommandInterestValidator` which checks the timestamps on command Interests to prevent replay attacks. A single command Interest validator instance is shared among all authorization callbacks. If the timestamp is unacceptable, the control command is rejected.
5. Finally, the signature is verified against the public key using `Validator::verifySignature` function, and the control command is either accepted or rejected based on the result of the signature verification.

The benefit of this design is that it takes advantage of the management module and command verb information that is already known from the dispatcher, so that name lookups are not repeated to determine permissible signing certificates.

A weakness in this procedure is the risk of denial-of-service attack: signature verification is performed after timestamp checking, therefore an attacker can forge command Interests with the KeyLocator of an authorized signing certificate in order to manipulate the internal state of the command Interest validator, and cause subsequent Interests signed by that certificate to be rejected[14].

6.3 Forwarder Status

The Forwarder Status Manager (`nfd::ForwarderStatusManager`) provides information about NFD and basic statistics about the forwarder through the method `listGeneralStatus`, which is registered to the dispatcher with the name `status/general`. The supplied information include NFD's version, startup time, Interest/Data/Nack packet counts, and various table entry counts, and is published with a 5 second freshness time to avoid excessive processing.

6.4 Face Management

The Face Manager (`nfd::FaceManager`) creates and destroys faces for all configured protocol types. Some attributes of an existing face can also be changed via this manager. Currently, these attributes are the face persistency (Section 2.1) and, on local faces, whether local fields are allowed or not (Section 2.3.1).

6.4.1 Command Processing

On creation, the Face Manager registers three command handlers: `createFace`, `updateFace`, and `destroyFace`, with names `faces/create`, `faces/update` and `faces/destroy` respectively.

`createFace` creates a unicast face, for protocols that support it.

`FaceUri` is used to parse the incoming URI in order to determine the type of face to create. 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 scheme (e.g., "`tcp4`", "`tcp6`", etc.) is used to lookup the appropriate `ProtocolFactory` via `FaceSystem::getFactoryByScheme`. Failure to find a factory results in a code 406 "Unsupported protocol" control response. Otherwise, the Face Manager calls `ProtocolFactory::createFace` method to initiate the potentially 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. Unauthorized, improperly-formatted, conflicting, or otherwise failing requests will be responded with appropriate failure codes and failure reasons. Refer to the NFD Management protocol specification [3] for the list of possible error codes. After a new Face has been created, the Face Manager

[14] See https://redmine.named-data.net/issues/3821

adds the new Face to the Face Table[15] and responds with a code 200 "OK" control response to the original control command.

`updateFace` can change the non-readonly properties of a face.
The target face can be determined by the FaceId extracted from the ControlParameters supplied to this command. The FaceId could be set to zero, implying an update of the same face on which this command was received. If the target face does not exist or the `IncomingFaceIdTag` is missing on the command Interest, this attempt of update fails with code 404. All changes on the target face will be performed after validation, leading to a code 200 "OK" response whose body contains ControlParameters describing the updated properties. On the contrary, if any of the requested changes is invalid, such as enabling local fields on a non-local face or changing the persistency to an unsupported value, the command will fail with a code 409 response whose body contains the invalid fields as ControlParameters.

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

6.4.2 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/Nack counts. The Face Status dataset can be retrieved from the `/localhost/nfd/faces/list` namespace.

These datasets are returned when `listFaces` and `listChannels` methods are invoked. A third method, `queryFaces`, serves to obtain the status of only those faces that satisfy a set of constraints, provided as a parameter. When the Face Manger is constructed, it registers these three handlers to the dispatcher with the names `faces/list`, `faces/channels` and `faces/query` respectively.

In addition to these datasets, the Face Manager also publishes notifications when Faces are created, destroyed, go up, or go down. This is done using the `postNotification` function returned after registering a notification stream to the dispatcher with the name `faces/events`. Two methods, `afterFaceAdded` and `afterFaceRemoved`, that take the function `postNotification` as argument, are connected to the FaceTable's `onAdd` and `onRemove` signals [20]. Whenever these two signals are emitted, the connected methods will be invoked immediately, where the `postNotification` will be used to publish notifications through the dispatcher.

6.5 FIB Management

The FIB Manager (`nfd::FibManager`) allows authorized users (normally, it is only the RIB Management, see Section 7) to modify NFD's FIB. It also 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 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 cost of existing next hop
- `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 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.

[15] 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, manages active Faces, and performs lookups for a Face object by FaceId when requested by other modules.

6.6 Strategy Choice Management

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 has been added to a pool of known strategies in NFD at compile time (see Section 5). Attempting to change to an unknown strategy will result in a code 404 "strategy not found" 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.

6.7 Configuration Handlers

6.7.1 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 `onConfig` method for the actual parsing.

At present, this section is limited to specifying an optional user and group name to drop the effective `uid` and `gid` of the running NFD process, if supported by the operating system, for safer operations. The `general` section parser initializes a global `PrivilegeHelper` instance to perform the actual (de-)escalation work.

6.7.2 Tables Configuration File Section Parser

`TablesConfigSection` provides parsing for the `tables` configuration file section. This class can then configure the various NFD tables (Section 3) appropriately. Currently, the `tables` section supports changing Content Store capacity and cache replacement policy, 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`.

6.8 How to Extend NFD Management

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 through 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 `ndn::Dispatcher`, so that the corresponding requests are routed to them and the produced notifications can be sent 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.

7 RIB Management

In NFD, the routing information base (RIB) stores static or dynamic routing information registered by applications, operators, and NFD itself. Each piece of routing information is represented by a route and is associated with a particular namespace. The RIB contains a list of **RIB entries** [21] each of which maintains a list of **Routes** [21] associated with the same namespace. RIB entries form a RIB tree structure via a parent pointer and children pointers.

Routing information in the RIB is used to calculate next hops for FIB entries in the FIB (Section 3.1). The RIB manager manages the RIB and updates the FIB when needed. While a FIB entry can contain at most one NextHop record toward the same outgoing face, a RIB entry can contain multiple Routes toward the same outgoing face since different applications may create routes to the same outgoing face each with different parameters. These multiple Routes are processed by the RIB manager to calculate a single NextHop record toward an outgoing face for the FIB entry (Section 7.3.1).

The RIB manager is implemented as an independent module of NFD, which runs as a separate thread and communicates with NFD through a face.[16] This separation was done in order to keep packet forwarding logic lightweight and simple, as RIB operations can require complex manipulations. Figure 13 shows the high-level interaction of the RIB Manager with NFD and other applications. A more detailed interaction is shown in Figure 14.

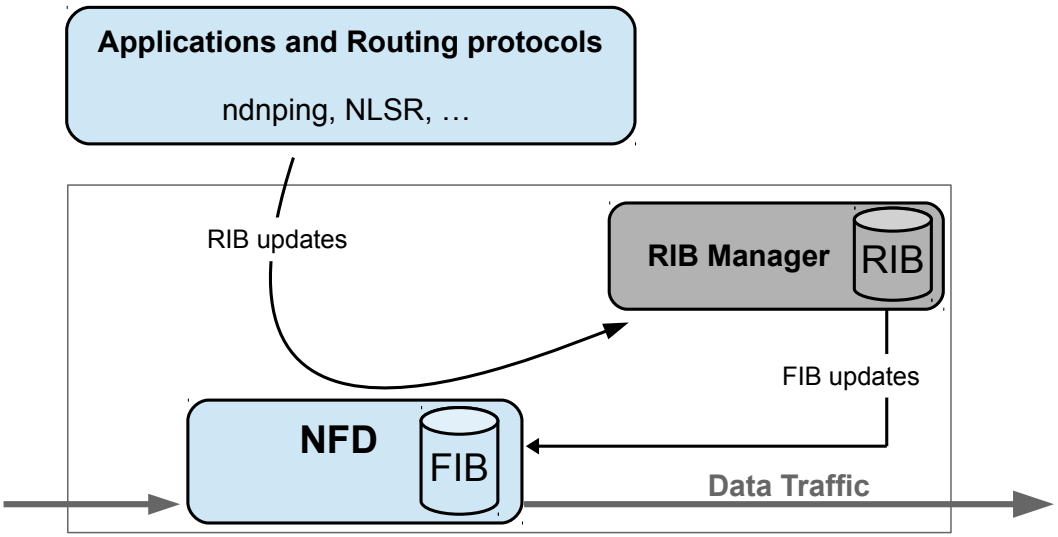


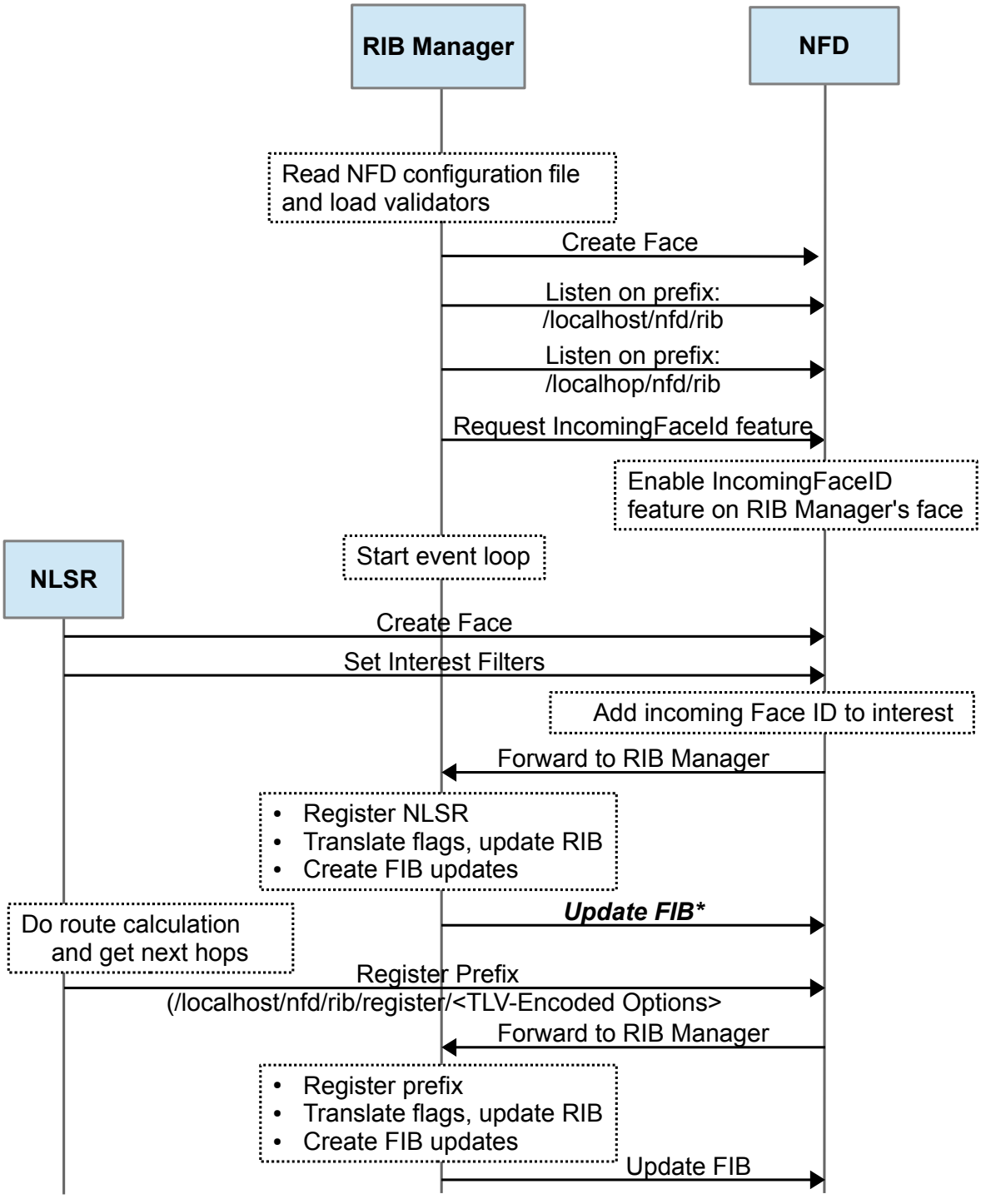
Figure 13: RIB Manager—system diagram

NFD provides various route inheritance flags for prefix registration that allow 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 route inheritance flags in place of the FIB Manager. In other words, the RIB manager receives all registration requests, processes the included route inheritance flags, and creates FIB updates as needed, which makes the forwarder leaner and faster. To facilitate this FIB update calculation, a RIB entry stores a list of inherited routes that have been calculated by the FIB Updater (Section 7.3) and applied to the entry's namespace. Each RIB entry also keeps a count of the number of its routes which have the `CAPTURE` flag set (Section 7.3.1). This list of inherited routes and the count of routes with their `CAPTURE` flag set are used by the FIB Updater to help calculate FIB updates.

The RIB manager provides an automatic prefix propagation feature which can be used to propagate knowledge of locally registered prefixes to a remote NFD. When enabled, the Auto Prefix Propagator component handles these propagations (Section 7.6).

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 provides a common interface for these processes and generates a consistent forwarding table. These different parties use the RIB Manager's interface through

[16] 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 14: RIB Manager—timing diagram

control commands [4], which need to be validated for authentication and authorization. The RIB manager listens for control commands with the command verbs *register* and *unregister* under the prefixes */localhost/nfd/rib* and */localhop/nfd/rib* [21]. The steps taken in processing these control commands are described in detail in Section 7.2. Applications should use the RIB management interface to manipulate the RIB, and only the RIB manager should use the FIB management interface to directly manipulate NFD's FIB.

7.1 Initializing the RIB Manager

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

1. Command validation rules are loaded from the the *rib* block of the NFD configuration file from the *localhost_security* and *localhop_security* subsections.
2. The Auto Prefix Propagator is initialized from the *auto_prefix_propagate* subsection with values to set the forwarding cost of remotely registered prefixes, the timeout interval of a remote prefix registration command, how often propagations are refreshed, and the minimum and maximum wait time before retrying a propagation.
3. The control command prefixes */localhost/nfd/rib* and */localhop/nfd/rib* (if enabled) are "self-registered" in NFD's FIB. This allows the RIB manager to receive RIB-related control commands (registration/unregistration requests) and requests for RIB management datasets.
4. The IncomingFaceId feature [5] is requested on the face between the RIB manager and NFD. This allows the RIB manager to get the FaceId from where prefix registration/unregistration commands were received (for "self-registrations" from NDN applications).
5. The RIB manager subscribes to the face status notifications using the FaceMonitor class [20] to receive notifications whenever a face is created or destroyed so that when a face is destroyed, Routes associated with that face can be deleted.

7.2 Command Processing

When the RIB manager receives a control command request, it first validates the Interest. The RIB manager uses the command validation rules defined in the *localhost_security* and *localhop_security* sections of the *rib* block in the NFD configuration file to determine if the signature on the command request is valid. 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 Route`: The RIB Manager takes the passed parameters from the incoming request and uses them to create a RIB update. If the FaceId is 0 or omitted in the command parameters, it means the requester wants to register a route to itself. This is called a self-registration. In this case, the RIB manager creates the RIB update with the requester's FaceId from the IncomingFaceId field. The RIB Manager then passes the RIB update to the RIB to begin the FIB update process (Section 7.3). The FIB Updater is invoked and if the FIB update process is successful, the RIB searches for a route that matches the name, FaceId, and Origin of the incoming request. If no match is found, the passed parameters are inserted as a new route. Otherwise, the matching route is updated. In both cases, the route is scheduled to expire after the passed expiration period and if the expiration period is being updated, the old expiration time is cancelled. If the registration request creates a new RIB entry and automatic prefix propagation is enabled, the Auto Prefix Propagator's `afterInsertRibEntry` method is invoked.
- `Unregister Route`: The RIB Manager takes the passed parameters from the incoming request and creates a RIB update. If the FaceId is 0 or omitted in the command parameters, it means the requester wants to unregister a route to itself. This is called a self-unregistration. In this case, the RIB manager creates the RIB update with the requester's FaceId from the IncomingFaceId field. The RIB manager then passes the RIB update to the RIB to begin the FIB update process (Section 7.3). The FIB Updater is invoked and if the FIB update process is successful, the Route with the same name, FaceId, and origin is removed from the RIB. If automatic prefix propagation is enabled, the Auto Prefix Propagator's `afterEraseRibEntry` method is invoked.

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 15 shows the verb dispatch workflow of the RIB manager.

If the expiration period in the *register* command parameters is not set to infinity, the route will be scheduled to expire after the expiration period. When the route expires, steps similar to an unregistration command are performed to remove the route, as well as any corresponding inherited routes, from the FIB and RIB.

7.3 FIB Updater

The FIB Updater is used by the RIB Manager to process route inheritance flags (Section 7.3.1), 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 route inheritance 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 case of a non-recoverable error (eg. signing key of the RIB Manager is not trusted, more than 10 timeouts for the same command), NFD will terminate with an error;
2. in case of a non-existent face error with the same face as the RibUpdateBatch, the RibUpdateBatch is abandoned;
3. in 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.

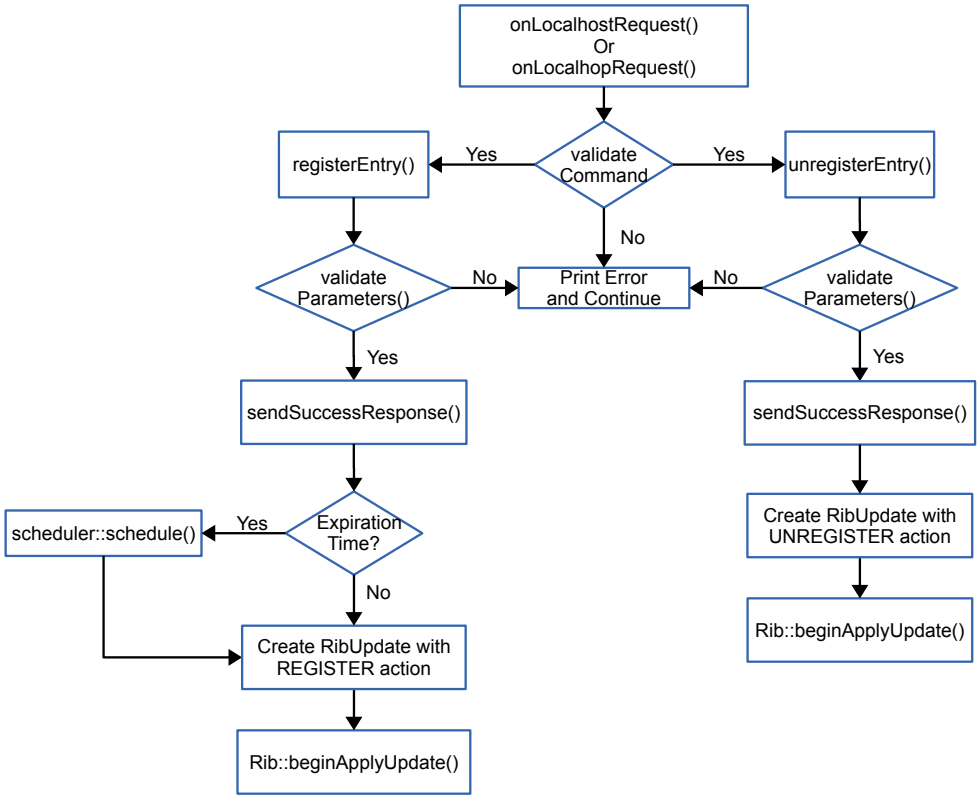


Figure 15: Verb dispatch workflow of the RIB Manager

7.3.1 Route Inheritance Flags

The route inheritance flags allow fine grained control over prefix registration. The currently defined flags and examples of route inheritance can be found at [21].

7.3.2 Cost Inheritance

Currently, NFD implements the following logic to assign costs to next hops 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 next hops 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 next hops 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 FaceId as one of the RIB entry's routes. Examples of the currently implemented logic are shown in Table 1.

Future versions will assign the lowest available cost to a next hop face based on all inherited RIB entries not blocked by a `CAPTURE` flag. Examples of the future logic are shown at [21].

Table 1: 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
/	1	`true`	`false`	75
/a	2	`false`	`false`	50
/a/b	1	`false`	`false`	65
/b	1	`true`	`false`	100
/b/c	3	`true`	`true`	40
/b/c/e	1	`false`	`false`	15
/b/d	4	`false`	`false`	30

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

7.4 RIB Status Dataset

The RIB manager publishes a status dataset under the prefix *ndn:/localhost/nfd/rib/list* which contains the current contents of the RIB. When the RIB Manager receives an Interest under the dataset prefix, the `listEntries` method is called. The RIB is serialized in the form of a collection of RibEntry and nested Route TLVs [21] and the dataset is published.

7.5 Readvertise

The Readvertise module provides a system where some policy can decide whether a route from the local RIB should be readvertised to some destination. It is triggered whenever the contents of the RIB change. Each given `Readvertise` instance has exactly one `ReadvertisePolicy` instance and one `ReadvertiseDestination` instance. Since Readvertise is a platform that can model several kinds of prefix propagation scenarios, a separate Readvertise instance must be maintained for each scenario. Currently NFD contains only one instance, used to communicate changes in the contents of the RIB to NLSR.

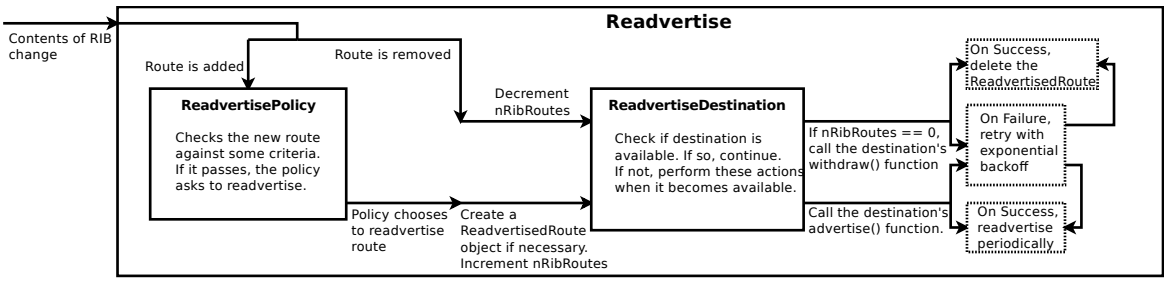


Figure 16: The simplified structure an operation of the Readvertise module.

The Readvertise module subscribes to `Signals` that the RIB maintains, which are signaled to after a RIB route is added and before one is removed. On route addition, Readvertise consults its `ReadvertisePolicy` instance. If the policy chooses to

readvertise a prefix, which may not be exactly the prefix associated with the RIB route, it will ask Readvertise to advertise the prefix to the destination, which will create a `ReadvertisedRoute` object if one does not exist, and increment the count of how many RIB routes advertise the prefix. How the destination actually models this is an implementation detail. For example, the Readvertise-to-NLSR policy and destination pair uses RibMgmt commands sent to a prefix that the local NLSR is listening on to communicate advertisements and withdrawals of routes. On route removal, Readvertise will check to see if any other RIB routes advertise this prefix. If not, it will call the destination's withdraw function. On successful withdrawal, the `ReadvertisedRoute` will be deleted. A failure from either an advertisement or withdrawal will prompt retries with exponential backoff.

If at any time the destination becomes unavailable, Readvertise will wait until next the destination is next available to advertise or withdraw any `ReadvertisedRoute`s as needed. Destination availability is left as an implementation detail, to be decided case-by-case by the destination implementer.

7.5.1 ReadvertisePolicy

When Readvertise receives a `Signal` that a RIB route has been added, it will call the `handleNewRoute()` function. The `ReadvertisePolicy` defines what kinds of prefixes should be advertised, specifically which prefix to advertise and what signing credentials to sign those advertised prefixes with. For example, in the Readvertise-to-NLSR case, the advertised prefix is just the prefix associated with the RIB route that triggered the advertisement. However, a policy could choose to advertise any prefix, perhaps the shortest sub-prefix that it can sign with some keychain.

Each policy must also define a refresh interval that specifies how often successfully advertised prefixes should be refreshed at the destination.

7.5.2 ReadvertiseDestination

As stated above, if the `ReadvertisePolicy` chooses to advertise a prefix, it will provide Readvertise with a prefix and signing credentials to advertise. Then Readvertise will call the `advertise()` function. How the function works is left as an implementation detail per-destination. Additionally, each destination must define the `withdraw()` function and should have some mechanism to manage its availability. The semantics of a destination's availability are left for the implementer to decide, keeping in mind that Readvertise will not attempt to advertise or withdraw prefixes while the destination is unavailable. Whenever a destination's availability changes, that destination should emit a `Signal` to alert Readvertise.

7.6 Auto Prefix Propagator

The Auto Prefix Propagator can be enabled by the RIB manager to propagate necessary knowledge of local prefix registrations to a single connected gateway router. Since gateway routers are currently configured to accept prefix registration commands under */localhop/nfd/rib*, the Auto Prefix Propagator cannot handle connections to multiple gateway routers; the */localhop/nfd/rib* prefix does not allow the Auto Prefix Propagator to distinguish which gateway router the propagations should and should not be forwarded to. Figure 17 provides an example of a locally registered prefix being propagated to a connected gateway router. The Auto Prefix Propagator *propagates* prefixes by registering prefixes with a remote NFD and *revokes* prefixes by unregistering prefixes from a remote NFD.

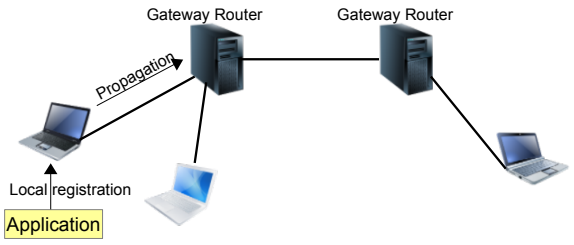


Figure 17: A local prefix registration is propagated to a connected gateway router

7.6.1 What Prefix to Propagate

To reduce not only the cost of propagations but also changes to the router's RIB, the propagated prefixes should be aggregated whenever possible. The local RIB Manager 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 are authorized to sign a prefix registration command for a prefix of the RIB prefix. If one or more signing identities are found, the identity with the shortest prefix that can sign a prefix registration command is chosen, and the Auto Prefix Propagator will then attempt to propagate that shortest prefix of the prefix to the router. Figure 18 presents a high level example of prefix propagation.

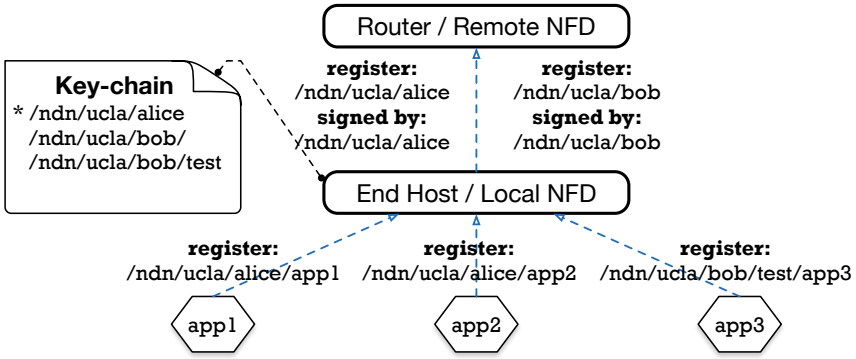


Figure 18: An example of prefix propagation

If the propagation succeeds, an event is scheduled to propagate the same prefix to refresh this propagation after a predefined 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.

7.6.2 When to Propagate

The Auto Prefix Propagator monitors RIB insertions and deletions by subscribing to 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 insertion or deletion, respectively.

When an insertion is processed, the Auto Prefix Propagator will not attempt to propagate prefixes scoped for local use (i.e., starts with `/localhost`) or that indicate connectivity to a router (i.e., the **link local NFD prefix**). The Auto Prefix Propagator also requires an active connection to the gateway router before attempting a propagation. The Auto Prefix Propagator considers the connection to the router as active if the local RIB has the *link local NFD prefix* registered and inactive if the local RIB does not have the *link local NFD prefix* registered. If the prefix has not been propagated previously, a propagation attempt will be made for the prefix.

Similarly, a revocation after deletion requires a qualified RIB prefix (not starting with `/localhost` or the **link local NFD prefix**), an active connection to a gateway router, that the prefix has been propagated previously, and that no other existing RIB prefix caused a propagation for the same prefix.

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

7.6.3 Secure Propagations

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

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

7.6.4 Propagated-entry State Machine

A propagated entry consists of a `PropagationStatus` which indicates the current state of the entry, as well as a scheduled event for either a refresh or retry. In addition, it stores a copy of the signing identity for the entry. All propagated entries are

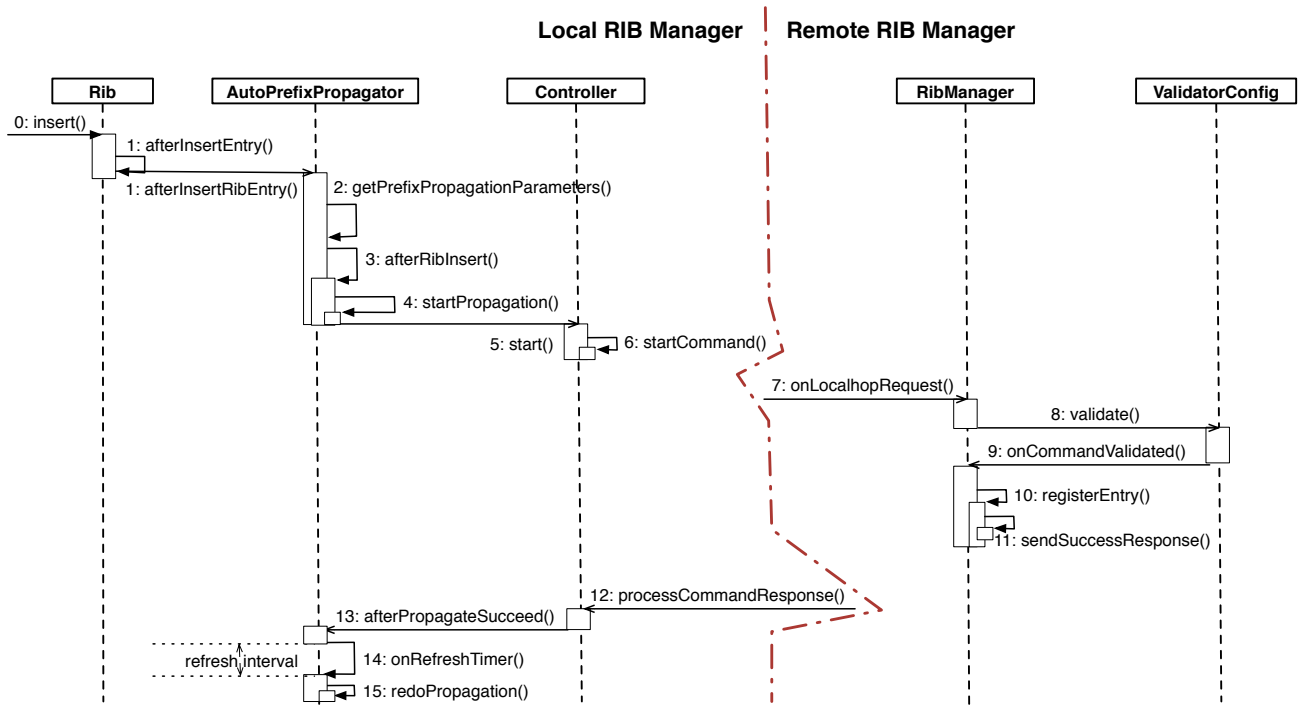


Figure 19: The simplified workflow of a successful propagation

maintained as an unordered 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. 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.

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

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

7.6.5 Configure Auto Prefix Propagator

When the RIB manager loads configurations from the `rib` section in the NFD configuration file, the Auto Prefix Propagator will load its configurations from the sub section `rib.auto_prefix_propagate`.

Table 2 presents the common parameters shared by all propagations and which parameters are configurable.

7.7 Extending RIB Manager

The RIB Manager currently supports two commands (register and unregister), RIB dataset publishing, and the automatic prefix propagation feature. However, the functionality of the RIB Manager can be extended by introducing more commands

Table 2: Shared parameters for prefix propagation

member variable[a]		**default setting**	**configurable**[b]
`m_controlParameters`	Cost	15	YES
	Origin	ndn::nfd::ROUTE_ORIGIN_CLIENT	NO
	FaceId	0	NO
`m_commandOptions`	Prefix	`/localhop/nfd`	NO
	Timeout	10,000 (milliseconds)	YES
`m_refreshInterval`[c]		25 (seconds)	YES
`m_baseRetryWait`		50 (seconds)	YES
`m_maxRetryWait`		3600 (seconds)	YES

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

and features. For example, in the current implementation, if a node wants to announce a prefix, it needs to communicate with a specific routing protocol. Once the RIB manager defines an interface for prefix announcement, e.g., advertise and withdraw commands, the process of announcing and withdrawing prefixes in routing protocols could become more uniform and simple.

8 Congestion Control

NFD implements a simplified version of the congestion control scheme in "A Practical Congestion Control Scheme for NDN" [22]. The current approach consists of three parts:

1. **Congestion Detection:** Each node detects congestion locally by monitoring its outgoing queues.
2. **Congestion Signaling:** After detecting congestion, NFD marks Data packets to inform consumers (and potentially downstream routers).
3. **Consumer Rate Adjustment:** End consumers react to congestion signals by adjusting their Interest sending rate.

8.1 Congestion Detection

Every NFD face monitors its *outgoing queue size* to decide whether the link is congested. More specifically, each face applies a simplified version of the CoDel Active Queue Management scheme [23].

A face enters the congested state (and marks the first packet) when the queue size first exceeds a threshold (default: 64KiB). As long as the queue size stays above this threshold, a new packet will be marked every interval (default: 100 ms), and this interval will be decreased with every marked packet. Thus, the congestion signal becomes stronger if the queue persists. Once the queue size falls below the threshold, the face leaves the congested state, and the marking interval is reset to the default value.

The congestion detection can be enabled in multiple ways:

- The default is configured in NFD's configuration file by setting `face_system.general.enable_congestion_marking` to `yes` or `no`.
- For a specific face, congestion detection can be enabled/disabled via `nfdc face create` by using the `congestion-marking <on|off>` parameter. Using the `nfdc` command overrides the value set in NFD's configuration file.

Congestion detection is currently not supported on Ethernet and WebSocket faces.

8.2 Congestion Signaling

Once a face enters the congested state, congestion is signaled with a single bit in the NDNLPv2 header.[17]. If using the ndn-cxx library, this congestion bit can be read and written by both consumer and producer applications:

- `uint64_t PacketBase::getCongestionMark()`
- `void PacketBase::setCongestionMark(uint64_t mark)`

Where `PacketBase` means Interest, Data, or NACK, and a mark value of 0 indicates that the mark is unset.

8.3 Consumer Adaptation

The consumer can use congestion marks as an early indication of congestion, and thus reduce its sending rate before the queue buffer overflows. Hence, a consumer should treat congestion marks just like timeouts (trigger a window decrease). The CoDel scheme ensures that congestion marks are infrequent (initially only once per 100ms) unless the congestion reaches an extreme level.

Currently, the command-line application `ndncatchunks`[18] supports congestion reaction by default. For testing, it can be disabled with the option `--aimd-ignore-cong-marks`.

[17] The CongestionMark header field is defined as `nonNegativeInteger` to allow future extensions for more fine-grained congestion signaling.
[18] https://github.com/named-data/ndn-tools/tree/master/tools/chunks

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[19] (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[20] 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.

[19] libndn-cxx expects the default Unix socket location, but this can be changed in the library's client.conf configuration file.

[20] 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 6.

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 [24]. 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 [25].

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 manchines, (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 instituion, 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 [26]. 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 orde 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 forwarding hint 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.

  NFD supports 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 6, 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 6). 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 three subsections: `localhost_security`, `localhop_security`, and `auto_prefix_propagate`. `localhost_security` 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. `auto_prefix_propagate` configures the behavior of the Auto Prefix Propagator feature of the RIB manager (Section 7.6).

  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 [25].

  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 [27]) 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.

  The `auto_prefix_propagate` subsection supports configuring the forwarding cost of remotely registered prefixes, the timeout interval of a remote prefix registration command, how often propagations are refreshed, and the minimum and maximum wait time before retrying a propagation:

  - `cost`: The forwarding cost for prefixes registered on a remote router (default is 15).
  - `timeout`: The timeout (in milliseconds) of prefix registration commands for propagation (default is 10000).
  - `refresh_interval`: The interval (in seconds) before refreshing the propagation (default is 300). This setting should be less than `face_system.udp.idle_time`, so that the face is kept alive on the remote router.
  - `base_retry_wait`: The base wait time (in seconds) before retrying propagation (default is 50).
  - `max_retry_wait`: maximum wait time (in seconds) before retrying propagation between consecutive retries (default is 3600). The wait time before each retry is calculated based on the following back-off policy: initially, the wait time is set to `base_retry_wait`. The wait time is then doubled for each retry unless it is greater than `max_retry_wait`, in which case the wait time is set to `max_retry_wait`.

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 [26] 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/nfd.hpp` and `daemon/nfd.cpp` files. In particular, an instance of the new management module needs to be created inside the `initializeManagement` method. Once module is created, it should be added to ConfigFile class dispatch. Similar updates should be made to `reloadConfigFile` method.

   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.

10.1.3 Configuration Reload

NFD reloads the configuration upon **SIGHUP** signal.

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 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 [12], 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.[21]

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`.

10.5 Global IO Service

The NFD packet forwarding implementation is based on Boost.Asio [28], 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.

[21] 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.

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 [29] 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.

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** (`<util/time-unit-test-clock.hpp>`): 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;
- **UnitTestTimeFixture** (`tests/test-common.hpp`): a base test fixture that overrides steady clock and system clock;
- **IdentityManagementFixture** (`tests/identity-management-fixture.hpp`): 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.
- **DummyTransport** (`tests/daemon/face/dummy-transport.hpp`): a dummy Transport used in testing Faces and LinkServices
- **DummyReceiveLinkService** (`tests/daemon/face/dummy-receive-link-service.hpp`): a dummy LinkService that logs all received packets. Note that this LinkService does not allow sending of packets.

- **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 `TestModuleName` and nested under test suites named after directories. For example, the test suite for the `daemon/fw/forwarder.hpp` file should be named `Fw/TestForwarder` ("daemon" part is not needed, as daemon-related unit tests are separated into a separate unit tests module `unit-tests-daemon`).
- Test suite should be declared inside 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` and a test suite for the `nfd::cs::Cs` class is declared in the `nfd::cs::tests` namespace. If needed, parent tests sub-namespace can be imported with 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`. If a test case needs to sign data, it should use `IdentityManagementFixture` or fixture that extends it. When feasible, `UnitTestTimeFixture` should be used to mock clocks.

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.

The easiest way to run the integration tests is to use Vagrant (https://www.vagrantup.com/) to automate creation and running of VirtualBox virtual machines. Note that the test machine will need at least 9Gb of RAM and at least 5 CPU threads.

- Install VirtualBox and Vagrant on the test system
- Clone integration tests repository:

```
git clone http://gerrit.named-data.net/NFD/integration-tests
cd integration-tests
```

- Run `run-vagrant-tests.sh` script that will create necessary virtual machines and run all the integration tests.
- Resulting logs can be collected using `collect-logs.sh` script.

References

[1] NDN Project Team, "NDN Packet Format Specification," http://named-data.net/doc/ndn-tlv/.

[2] ——, "NFD: Named Data Networking Forwarding Daemon," http://named-data.net/doc/NFD/current/.

[3] ——, "NFD management protocol," https://redmine.named-data.net/projects/nfd/wiki/Management.

[4] ——, "Control command," https://redmine.named-data.net/projects/nfd/wiki/ControlCommand.

[5] ——, "NDNLPv2," https://redmine.named-data.net/projects/nfd/wiki/NDNLPv2.

[6] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–49, Jun. 2010, updated by RFCs 7419, 7880. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5880.txt

[7] B. Adamson, C. Bormann, M. Handley, and J. Macker, "Multicast Negative-Acknowledgment (NACK) Building Blocks," RFC 5401 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–42, Nov. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5401.txt

[8] J. Shi and B. Zhang, "NDNLP: A link protocol for NDN," NDN, NDN Technical Report NDN-0006, Jul 2012.

[9] 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

[10] J. Shi, "Namespace-based scope control," https://redmine.named-data.net/projects/nfd/wiki/ScopeControl.

[11] J. M. L. Muñoz, "Boost multi-index containers library," http://www.boost.org/doc/libs/1_54_0/libs/multi_index/doc/index.html, 2003–2006.

[12] Google, "The CityHash family of hash functions," https://github.com/google/cityhash, 2013.

[13] J. Shi, "ccnd 0.7.2 forwarding strategy," https://redmine.named-data.net/projects/nfd/wiki/CcndStrategy, University of Arizona, Tech. Rep.

[14] Y. Yu, A. Afanasyev, Z. Zhu, and L. Zhang, "NDN technical memo: Naming conventions," NDN, NDN Memo, Technical Report NDN-0023, Jul 2014.

[15] P. Gusev and J. Burke, "NDN-RTC: Real-time videoconferencing over Named Data Networking," NDN, NDN Technical Report NDN-0033, Jul 2015.

[16] NDN Project Team, "NDN protocol design principles," http://named-data.net/project/ndn-design-principles/.

[17] V. Lehman, A. Gawande, B. Zhang, L. Zhang, R. Aldecoa, D. Krioukov, and L. Wang, "An experimental investigation of hyperbolic routing with a smart forwarding plane in NDN," NDN, NDN Technical Report NDN-0042, Jul 2016.

[18] 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

[19] 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/

[20] 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.

[21] NDN Project Team, "RIB management," https://redmine.named-data.net/projects/nfd/wiki/RibMgmt.

[22] K. Schneider, C. Yi, B. Zhang, and L. Zhang, "A practical congestion control scheme for named data networking," in *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. ACM, 2016, pp. 21–30.

[23] D. K. M. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, "Controlled Delay Active Queue Management," RFC 8289, Jan. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8289.txt

[24] Y. Yu, "NDN regular expression," https://redmine.named-data.net/projects/ndn-cxx/wiki/Regex.

[25] ——, "Validator configuration file format," https://redmine.named-data.net/projects/ndn-cxx/wiki/CommandValidatorConf.

[26] M. Kalicinski, "Boost.PropertyTree," http://www.boost.org/doc/libs/1_54_0/doc/html/property_tree.html, 2008.

[27] NDN Project Team, "NDN-Cert," https://github.com/named-data/ndncert.

[28] C. Kohlhoff, "Boost.Asio," http://www.boost.org/doc/libs/1_54_0/doc/html/boost_asio.html, 2003–2013.

[29] G. Rozental, "Boost test library," http://www.boost.org/doc/libs/1_54_0/libs/test/doc/html/index.html, 2001–2007.