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], Yi Huang[2], Jerald Paul Abraham[2], Steve DiBenedetto[3], Chengyu Fan[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], and Lan Wang[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

This document explains the internals of the Named Data Networking Forwarding Daemon (NFD). This document is intended for developers who are interested in extending and improving NFD.

Revision history

Revision	Revision date	Description
1		Initial release

Contents

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. It 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 page [2].

The main design goal of NFD is to support diverse experimentation of NDN technology. 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 spec.

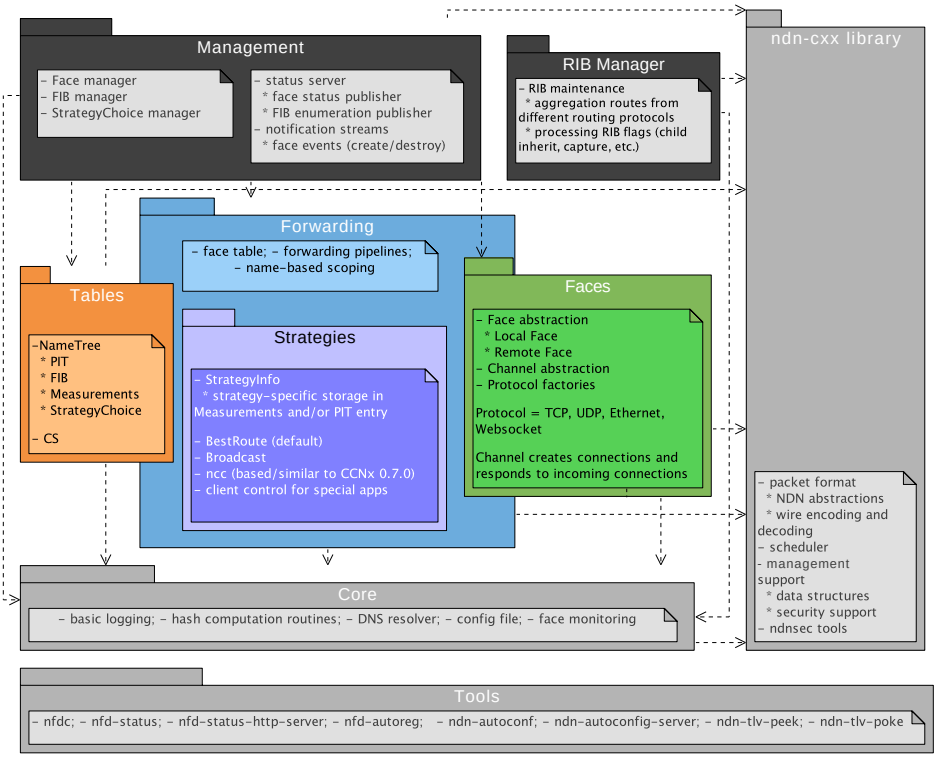

Figure 1: Overview of NFD modules

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 9)

  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 different lower level transport mechanisms.

- **Tables** (Section 3)

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

- **Forwarding** (Section 4)

  Implements basic packet processing pathways, which interact with Faces, Tables, and Strategies.

  Strategies is a major part of the forwarding module. It implements a framework to support different forwarding strategies in the form of forwarding pipelines, described in detail in Section 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. Strictly speaking RIB management is part of the NFD management module, but due to its importance to the overall operations and its more complex processing, we implement it as a separate module.

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

2 Face System

The face system in NFD is separated into three logical abstractions: protocol factories, channels, and faces. A **protocol factory** understands one underlying protocol (such as TCP), and can create channels or faces of this underlying protocol. A **channel** represents a local endpoint for unicast communication. A **face** represents either a connection between a local endpoint and a peer (an application or another forwarder), or a multi-access media between a local endpoint and zero or more peers.

The overall interaction between these abstractions is illustrated in Figure 2 and each abstraction is described in detail in the following sections. In short, these interactions can be summarized as: protocol factories create channels, channels create faces, and faces are actually responsible for sending and receiving Interest and Data packets through the protocol-specific tunnel.

The current implementation is heavily based on the Boost.Asio library [4] and uses asynchronous operations as much as possible to avoid blocking the rest of the daemon while performing potentially lengthy network operations.

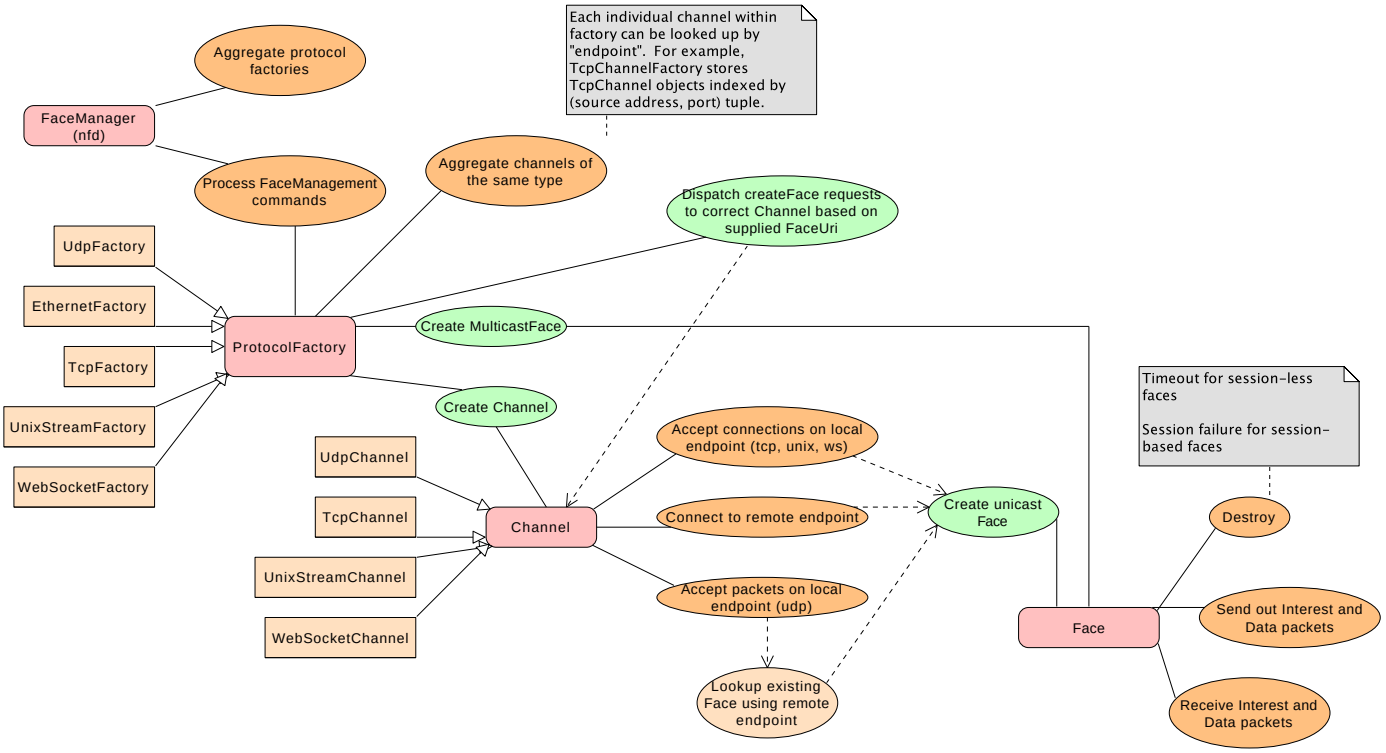

Figure 2: Face, Channel, ProtocolFactory interactions

2.1 Protocol Factory Abstraction

The protocol factory is the highest level abstraction in NFD's face system and it is the starting point for inter- and intra-node communication in NFD. Each protocol factory handles a specific protocol that is natively supported by NFD.

Currently supported protocols include:

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

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

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

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

2.2 Channel abstraction

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

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

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

When a peer makes a connection to a local endpoint, the callback of the channel bound to that endpoint is invoked. The callback function in turn creates a face for the corresponding protocol, which will handle all subsequent communications with that peer, and executes the `FaceCreatedCallback` that was supplied to the initial `listen` invocation. If any errors or timeouts are encountered during this procedure, the connection setup is aborted and `ConnectFailedCallback` is executed instead.

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

The process of establishing a connection to a remote peer entails calling the method `connect`, which starts an asynchronous connection attempt towards the specified endpoint. Non-blocking host name resolution is automatically performed if needed (see Section 9.4). If the connection is successful, a face is instantiated and the caller-supplied `FaceCreatedCallback` is invoked; otherwise, the error is signaled via `ConnectFailedCallback`.

2.3 Face abstraction

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

- Local and non-local faces: the Internal, TcpLocal, and UnixStream faces are considered *local*, because they can communicate only with other programs running on the same machine (this restriction is enforced by NFD). Local faces are the only ones that can send to and receive from the "`/localhost`" namespace; they can also support the LocalControlHeader that is used by some special applications. All other faces are considered *non-local*.

- Unicast and multicast faces: *unicast* faces, such as TcpFace, UdpFace, and WebSocketFace, can communicate with a single peer, i.e. packets are sent from the local endpoint to exactly one remote endpoint and vice versa. *Multicast* faces, on the other hand, are able to transmit a single packet to multiple remote endpoints, and receive from all of them, forming a set of intercommunicating peers that is usually called multicast group; the MulticastUdp and Ethernet faces are examples of multicast faces.

- Datagram and stream faces: this distinction is based on the same difference that exists between datagram-oriented and stream-oriented sockets. Therefore the Udp and MulticastUdp faces are *datagram* faces, while the Tcp and UnixStream faces are *stream* faces.

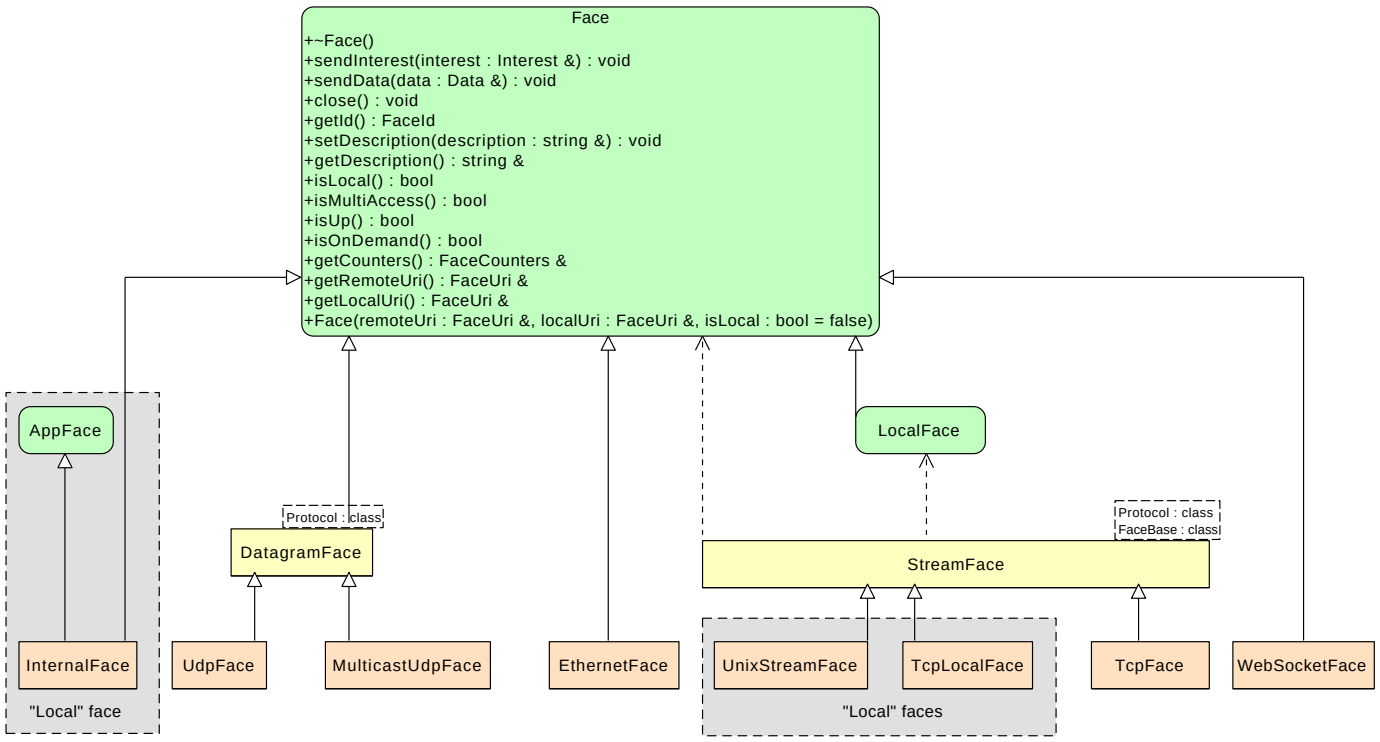

Figure 3: NFD faces

2.4 Extending NFD Face System

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

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

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

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

The face class provides the basic communication primitives such as `sendInterest`, `sendData`, `close`, and so on. When implementing the receiving function, developers only need to check the length of the incoming message to make sure that the TLV wire encoding in the received packet is complete. For stream-based protocols, this means that in most cases an internal buffer must be maintained in order to collect all incoming segments, until the entire NDN-TLV packet is available and can be parsed. Finally developers simply invoke `decodeAndDispatchInput`, which is inherited from the `Face` base class. That method will check the type field of the incoming packet (i.e., Interest or Data) and pass the packet to NFD's forwarding pipeline (see Section 4).

3 Tables: FIB, PIT, and CS

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 Content Store (CS) (Section 3.2) 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 3.3) 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 Strategy Choice Table (Section 3.4) contains the forwarding strategy (Section 5) chosen for each namespace.

The Measurements Table (Section 3.5) 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 3.6), 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 [5]. 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 3.6.

3.1.1 Structure and Semantics

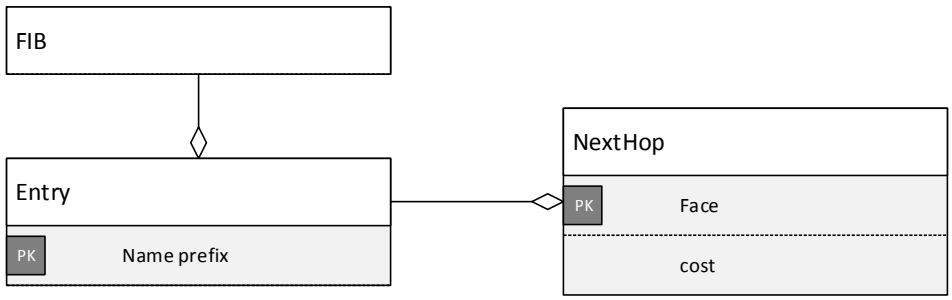

Figure 4: FIB and related entities

Figure 4 shows the FIB, FIB entries, NextHop records, and their relations.

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: given an Interest under this prefix, potential source(s) of matching Data can be reached via faces given by 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), 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 that 1. its Name prefix is a prefix of the parameter, and 2. its Name prefix is the longest among those satisfying condition 1 ; null is returned if no FIB entry satisfy condition 1.

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

3.1.2 Usage

The FIB is updated only through management. FIB manager (Section 6.1.1) is directly responsible for updating the FIB. Typically, FIB manager takes commands from RIB Daemon (Section 7), which in turn receives static routes configured by operation 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.

3.2 Content Store (CS)

The Content Store (CS) is a cache of Data packets. Arriving Data packets are placed in this cache as long as possible, in order to satisfy future Interests that request same Data. The Content Store is searched before an Interest is forwarded, so that cached Data, if available, can be used to satisfy the Interest.

The semantics and algorithms of CS, and how it's used by forwarding is outlined in Section 3.2.1. The implementation of CS is discussed in Section 3.2.2.

3.2.1 Semantics and Usage

The Content Store (`nfd::Cs`) is a cache of Data packets.

Data packets are inserted to the CS (`Cs::insert`) from forwarding in *incoming Data pipeline* (Section 4.3.1) or *Data unsolicited pipeline* (Section 4.3.2). Forwarding pipelines should determine that the Data packet is eligible to be cached (eg. does not violate scope), before giving it to CS. When a Data packet is inserted, the current timestamp is stored along with the cached packet, so that CS could later determine if the Data packet becomes stale and cannot be used to match an Interest with MustBeFresh selector.

CS is queried (`Cs::find`) with an incoming Interest before it's forwarded in *incoming Interest pipeline* (Section 4.2.1). The search algorithm should return the Data packet that best matches the Interest, or return null if no Data packet matches the Interest.

CS has limited capacity, measured in number of packets. It is controlled via NFD configuration file (Section 6.2.6). Management calls `Cs::setLimit` to update the capacity. CS implementation should ensure number of cached packets does not exceed the capacity.

3.2.2 Implementation

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

Current implementation of Content Store has a skip list [6] as its underlying data structure. Skip lists are a probabilistic alternative to balanced trees. Skip lists are balanced by virtue of a random number generator. Its average insertion and lookup complexity is *O(log n)* (Figure 5). Content Store entries are placed in the Skip List in ascending order (by Name) [1].

The assumption behind Content Store design is that it operates at its maximum capacity all the time. Therefore, it must have an efficient cache replacement strategy. The current implementation evicts Content Store entries based on prioritized FIFO (First In First Out) strategy. The entries that get removed first are *unsolicited* Data packets, which are the Data packets that got cached opportunistically without preceding forwarding of the corresponding Interest packet. Next, Data packets with expired freshness are removed from Content Store. Lastly, Data packets are removed from the Content Store on a pure FIFO basis. This cache replacement policy is currently hard-coded; we intend to make it replaceable in the future (NFD Task 1207).

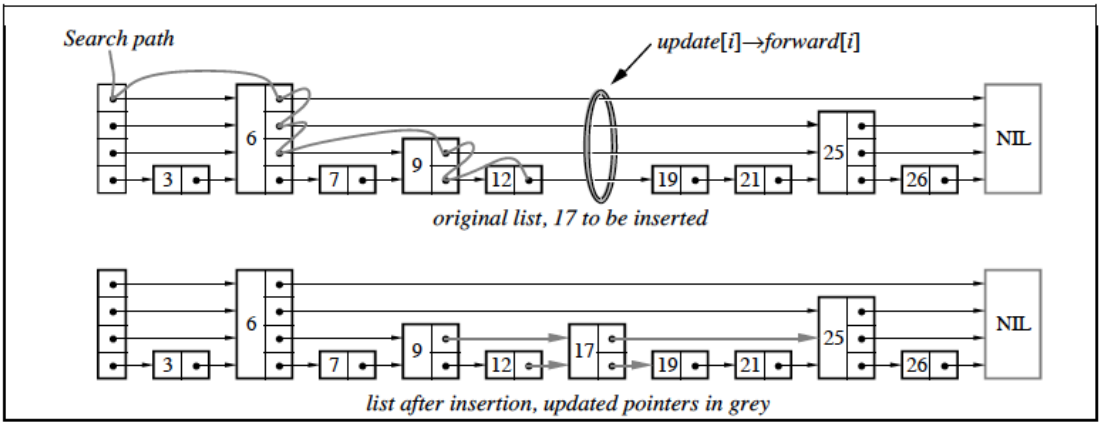

Figure 5: Insertion of an item into a SkipList

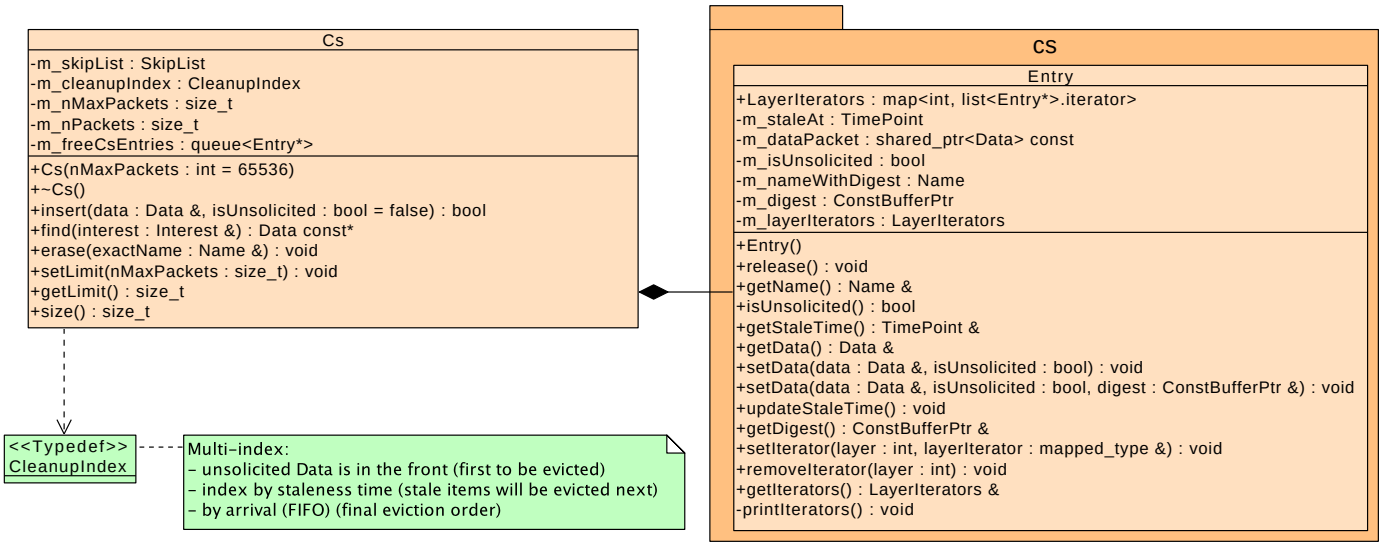

Figure 6: CS data structure

Current CS implementation is illustrated in Figure 6.

CS entry

The Data packet, along with other necessary fields, are stored in a **CS entry**.
Each entry contains:

- the Data packet
- whether the Data packet is unsolicited
- the timestamp at which the cached Data becomes stale

CS

To support the prioritized FIFO cache replacement policy, the CS maintains a multi-index container in order to keep pointers to the Data packets in a particular order.

Eviction is performed during insertion if the CS is full, and when the capacity is decreased by management. We decide not to perform periodical cleanups, because its CPU overhead would cause jitter in packet forwarding.

3.3 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) [5]. 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 3.3.1. The structure and algorithms of PIT, and how it's used by forwarding are described in Section 3.3.2. The implementation of PIT algorithms is discussed in Section 3.6.

3.3.1 PIT entry

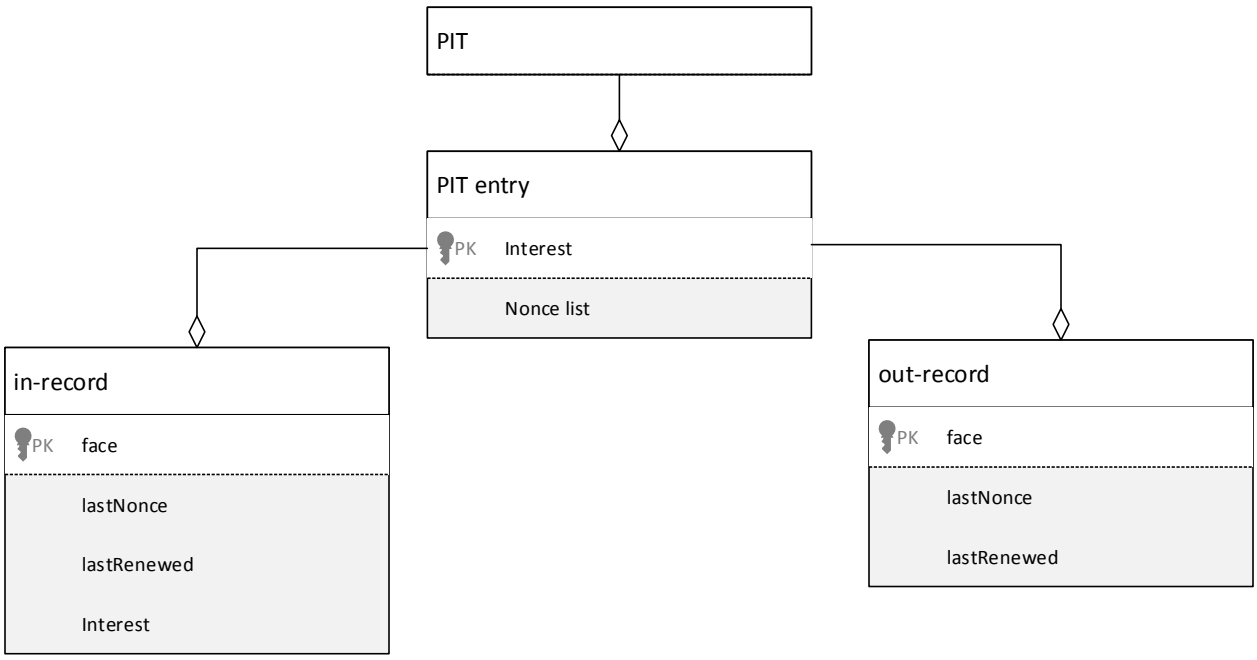

Figure 7: PIT and related entities

Figure 7 shows the PIT, PIT entries, in-records, out-records, and their relations.

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. 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 **Nonce list**. The Nonce list is an unordered set of Nonces seen so far in incoming Interest packets that match this PIT entry. *Incoming Interest pipeline* (Section 4.2.1) adds an incoming Interest's Nonce to this Nonce list, after checking whether it already exists, which indicates the incoming Interest is a duplicate.

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

An out-record is inserted or updated by *outgoing Interest pipeline* (Section 4.2.3). 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.

Timers

There are two timers on a PIT entry, used by forwarding pipelines (Section 4):

- *unsatisfy timer* fires when the PIT expires (Section 4.2.3)
- *straggler timer* fires when the PIT entry can be deleted because it has been satisfied or rejected, and is no longer needed for loop detection and measurements purposes (Section 4.3.1)

3.3.2 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 which that can be satisfied by this Data packet. This algorithm does not delete any PIT entry.

3.4 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 [5]. In practice, we find that it's more convenient to save the forwarding strategy in a separate table, instead of storing it with FIB entry, based on the following reasons:

- FIB entries come from RIB entries which are managed by NFD RIB Daemon (Section 7). Storing the strategy in FIB entries forces the RIB Daemon to understand strategy, and increases its complexity.
- FIB entry is automatically deleted when the last NextHop record is removed, including when the last upstream face fails. However, we don't want to lose the configured strategy.
- The granularity of strategy configuration is different from the granularity of RIB entry or FIB entry. Having both in the same table makes inheritance handling more complex.

The structure, semantics, and algorithms of Strategy Choice Table is outlined in Section 3.4.1. How Strategy Choice Table is used by rest of NFD is described in Section 3.4.2. The implementation of Strategy Choice Table algorithms is discussed in Section 3.6.

3.4.1 Structure and Semantics

Strategy Choice entry

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

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

Strategy Choice Table

The Strategy Choice Table (`nfd::StrategyChoice`) is a collection of Strategy Choice entries. It also contains a collection of installed strategies.

Before a strategy can be chosen for any namespace, it must be installed (`StrategyChoice::install`). The Name for each installed strategy must be unique.

To enforce every namespace to have a strategy, the root entry is inserted when the Strategy Choice Table is initialized. The initial strategy chosen on the root entry is also installed during initialization. The strategy chosen for root namespace, called the *default strategy*, can be replaced; but the root entry cannot be deleted.

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

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

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

Find Effective Strategy algorithm (`StrategyChoice::findEffectiveStrategy`) finds a strategy that should be used to forward an Interest. It takes a Name, a PIT entry, or a Measurements entry as input parameter. The return value is a forwarding strategy that is found by longest prefix match using the Name; this return value is never null because every namespace must have a strategy.

3.4.2 Usage

The Strategy Choice Table is updated only through management. Strategy Choice manager (Section 6.1.3) is directly responsible for updating the Strategy Choice Table.

The Strategy Choice is expected to be extremely stable. Strategies are manually chosen by the local operator, so updates are rare.

The find effective strategy algorithm is used by forwarding in *incoming Interest pipeline* (Section 4.2.1), *Interest unsatisfied pipeline* (Section 4.2.5), and *incoming Data pipeline* (Section 4.3.1). It is called at most twice per incoming packet.

3.5 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 3.5.1. How Measurements Table is used by rest of NFD is described in Section 3.5.2. The implementation of Measurements Table algorithms is discussed in Section 3.6.

3.5.1 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.

3.5.2 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 3.4), 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 3.4) to confirm whether the requesting strategy owns the Measurements entry. If an access violation is detected, null is returned instead of the entry.

3.6 NameTree

The NameTree is a common index structure for FIB (Section 3.1), PIT (Section 3.3, Strategy Choice Table (Section 3.4, and Measurements Table (Section 3.5). It is feasible to use a common index, because there are much commonality in the index of these four tables: FIB, StrategyChoice, and Measurements are all indexed by Name, and PIT is indexed by Name and Selectors. 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 3.6.1. NameTree operations and algorithms are described in Section 3.6.2. Section 3.6.3 describes how NameTree can help reducing number of index lookups by adding shortcuts between tables.

3.6.1 Structure

The conceptual NameTree data structure is shown in Figure 8. The NameTree is a collection of NameTree entries, indexed by Name. FIB, PIT, Strategy Choice, and Measurements entries are attached onto NameTree entry.

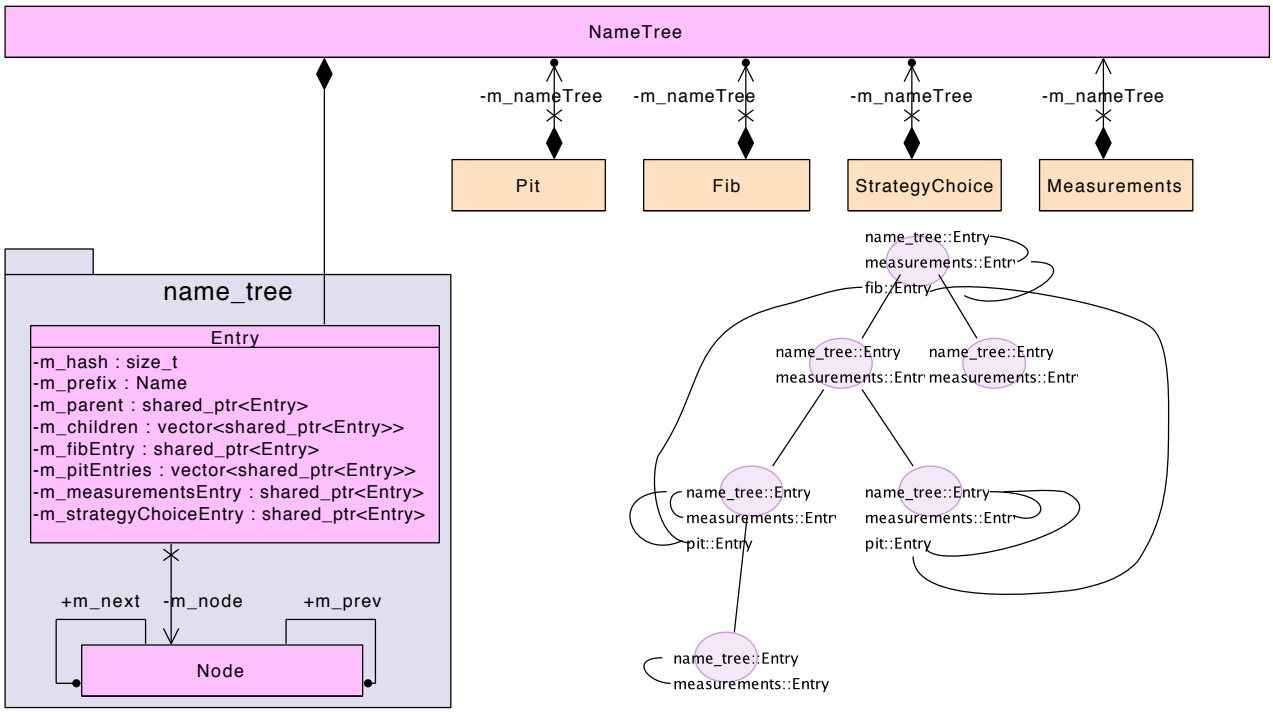

Figure 8: NameTree overview

NameTree entry

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

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

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

NameTree hash table

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

We decide to implement the hash table from scratch, rather than using an existing library, so that we can have better control for performance tuning. The hash table data structure is shown in Figure 9.

Hash values are computed using CityHash [7]; this hash function is chosen because it is fast. For a given Name prefix, hash is computed over the TLV representation of the Name, and the hash value is mapped to one of the *buckets*. Hash collisions are resolved via chaining: if multiple Names are mapped to the same bucket, all these entries are chained in that bucket through a singly linked list.

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

To reduce the overhead of resize operation, the hash value of a Name is stored in the NameTree entry. We also introduce a **NameTree Node** type. A Node is stored in the bucket, and contains a pointer to an entry, and a pointer to the next Node

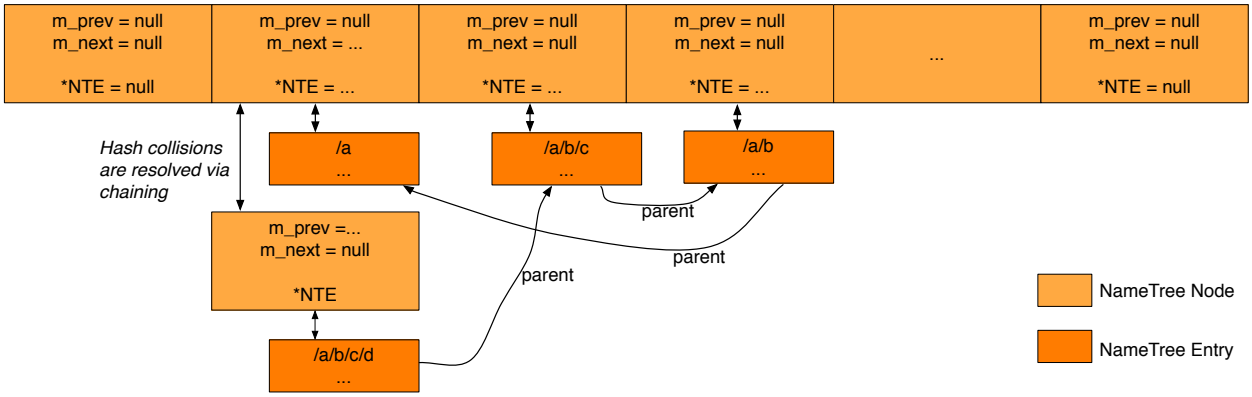

Figure 9: NameTree hash table data structure

in the chain. The resize operation only needs to move Nodes (which are smaller than entries), and do not need to change entries.

In Figure 9, name prefixes /a, /a/b, /a/b/c, /a/b/c/d are stored. The parent pointers shown on the figure show the relationship between these four name prefixes. As shown in the figure, there is a hash collision between /a and /a/b/c/d, and the hash collision is resolved via chaining.

3.6.2 Operations and Algorithms

Insertion and Deletion operations

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

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

Matching algorithms

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

The **longest prefix match** 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 StrategyChoice find effective strategy algorithm (Section 3.4.1), with a predicate that accepts a NameTree entry only if it contains a StrategyChoice entry.

The **all match** algorithm (`NameTree::findAllMatches`) enumerates all entries that are prefixes of a given Name, filtered by an optional *EntrySelector*. This algorithm is implemented as: perform a longest prefix match first; remove the last Name component, until reaching the root entry. This algorithm is called by PIT data match algorithm (Section 3.3.2).

Enumeration algorithms

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

The **partial enumerate** 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.

3.6.3 Shortcuts

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

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

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

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

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

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

4 Forwarding

The packet processing in NFD consists of **forwarding pipelines** described in this section and **forwarding strategies** described in Section 5. A **forwarding pipeline** (or just pipeline) is a series of steps that operates on a packet or a PIT entry, which is triggered by the specific event: reception of the Interest, detecting that the received Interest was looped, when an Interest is ready to be forwarded out of the Face, etc. A **forwarding strategy** (or just strategy) is a decision maker about Interest forwarding, which is attached at the end or beginning of the pipelines. In other words, the strategy makes decisions whether, when, and where to forward an Interest, while the pipelines supply the strategy the Interests and supporting information to make these decisions.

Figure 10 shows the overall workflow of forwarding pipelines and strategy, where blue boxes represent pipelines and white boxes represent decision points of the strategy.

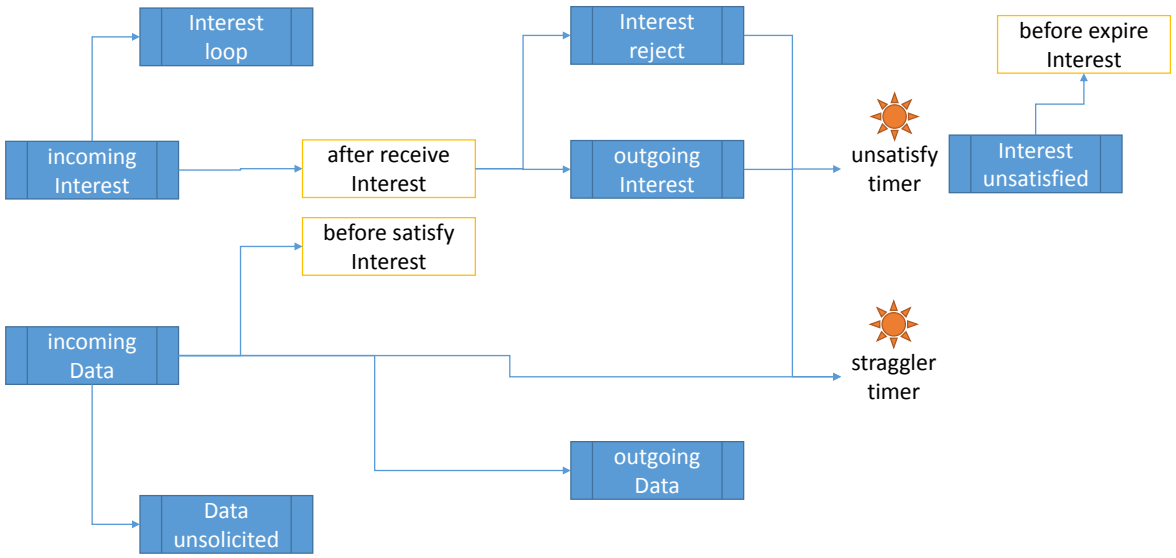

Figure 10: Pipelines and strategy: overall workflow

4.1 Forwarding Pipelines

Pipelines operate on network layer packets (Interest or Data) and each packet is passed from one pipeline to another (in some cases through strategy decision points) until all processing is finished. Processing within pipelines uses PIT, ContentStore, FIB, and StrategyChoice tables, however for the last two pipelines have only read-only access (FIB and StrategyChoice are managed by the corresponding managers and are not directly affected by the data plane traffic). In addition to that, pipelines have read access to FaceTable (the table that keeps track all active Faces in the forwarder) and are allowed to actually send packets through Faces.

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

4.2 Interest Processing Path

NFD separates Interest processing into the following pipelines:

- incoming Interest: processing of incoming Interests
- Interest loop: processing incoming looped Interests
- outgoing Interest: preparation and sending out Interests
- Interest reject: processing PIT entries that are rejected by the strategy
- Interest unsatisfied: processing PIT entries that are unsatisfied before all downstreams timeout

4.2.1 Incoming Interest Pipeline

The incoming Interest pipeline is implemented in `Forwarder::onIncomingInterest` method and is entered from `Forwarder::onInterest` method, which is triggered by `Face::onReceiveInterest` event emitter (see Section 9.5 for more detail about `EventEmitter`). 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 11:

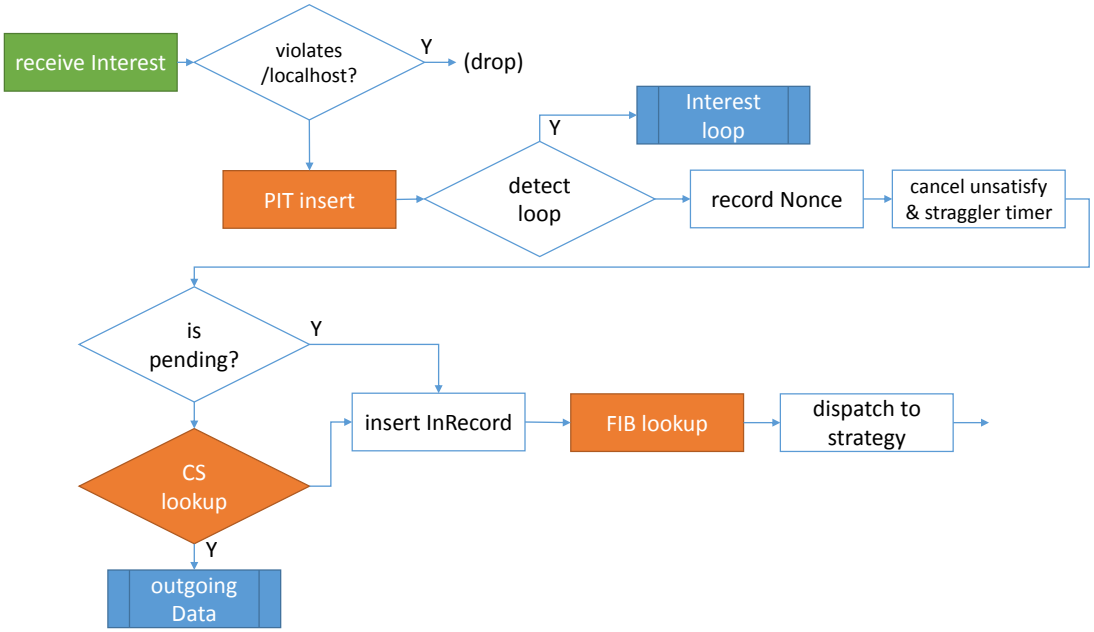

Figure 11: Incoming Interest pipeline

- The first step after entring the incoming Interest pipeline is check for `/localhost` scope [8] violation. In particular, an Interest from a non-local Face is not allowed to have a name that starts with `/localhost` prefix, as it is reserved for localhost communication. If violation is detected, such Interest is immediately dropped and no further processing on the dropped Interest is performed. This check guards against malicious senders; a compliant forwarder will never send a `/localhost` Interest to a non-local Face. Note that `/localhop` scope is not checked here, because its scope rules do not restrict incoming Interests.

- The next step is looking up existing or creating a new PIT entry, using name and selectors specified in the Interest packet. As of this moment, PIT entry becomes a processing subject of the incoming Interest and following pipelines. Note that NFD creates PIT entry before performing ContentStore lookup. The main reason for this decision is to reduce lookup overhead: ContentStore is most likely be significantly larger than PIT and can incur significant overhead, since, as described below, ContentStore lookup can be skipped in certain cases.

- Before the incoming Interest is processed any further, its nonce is checked against the Nonce list in the PIT entry. If a match is found, the incoming Interest is considered a duplicate due to loop, and is given to *Interest loop pipeline* for further processing (Section 4.2.2). If a match is not found, its Nonce is added to the Nonce list and processing continues.

- Next, the *unsatisfy timer* (Section 4.2.3) and *straggler timer* (Section 4.2.4) on the PIT entry are cancelled, because a new valid Interest is arriving for the PIT entry, so that the lifetime of the PIT entry needs to be extended. The timers could get reset later in the Interest processing path, e.g., if ContentStore will be able to satisfy the Interest.

- The pipeline then tests whether the Interest is pending, i.e., the PIT entry has already another in-record from the same or other incoming Face. Recall that NFD's PIT entry can represent not only pending Interest but also recently satisfied Interest (Section 3.3.1), this test is equivalent to "having a PIT entry" in CCN Node Model [5], whose PIT contains only pending Interests.

- If the Interest is not pending, the Interest is matched against the ContentStore (Section 3.2). Otherwise, CS lookup is unnecessary because a pending Interest implies that a previous CS returns no match. If a match is found, the best matching Data is passed to *outgoing Data pipeline* (Section 4.3.3), and the Interest processing is completed.

- At this point, the Interest is valid and cannot be satisified by cached Data, so it needs to be forwarded somewhere else. Therefore, an in-record for the Interest and its incoming Face is created in the PIT entry or simply gets refreshed if it was already there (e.g., the Interest is being re-expressed by the consumer). The expiration value of the in-record is directly controlled by the `InterestLifetime` fields in the Interest packet. If `InterestLifetime` is omitted, the default value of 4 seconds is used.

  Note that NFD defers to the strategy the decision on whether to forward again a similar Interest (same name and selectors, but different noce). All currently implemented strategies will suppress forwarding of Interests if there is at least one active out-record (see Section 5.1.1 for more detail).

- FIB is looked up using Interest Name (Section 3.1.1, Longest Prefix Match algorithm). This needs to be in the pipeline, because strategy does not have direct access to the FIB. Note that FIB guarantees that Longest Prefix Match would return a valid FIB entry. However, a FIB entry may contain empty set of NextHop records, which could effectively result (but, strictly speaking, is not required to happen) in rejecting of the Interest by the strategy.

- The final step of this pipeline is determining which strategy is responsible to process Interests (i.e., Interest's name is checked against StrategyChoice table using Find Effective Strategy algorithm, see Section 3.4.1). The selected strategy is then triggered for the *after receive Interest* action with the PIT entry, incoming Interest packet, and FIB entry (Section 5.1.1).

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, its incoming Face, and the PIT entry.

In the current implementation, this pipeline simply drops the Interest. In the future, this pipeline may generate some form of explicit notifications (e.g., Interest NACKs [9]) to the downstream to inform about the detected loop.

4.2.3 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 a `wantNewNonce` flag. Note that the Interest packet is not a parameter when entering the pipeline. The pipeline steps either use PIT entry directly to perform checks, or obtain reference to an Interest stored inside the PIT entry.

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

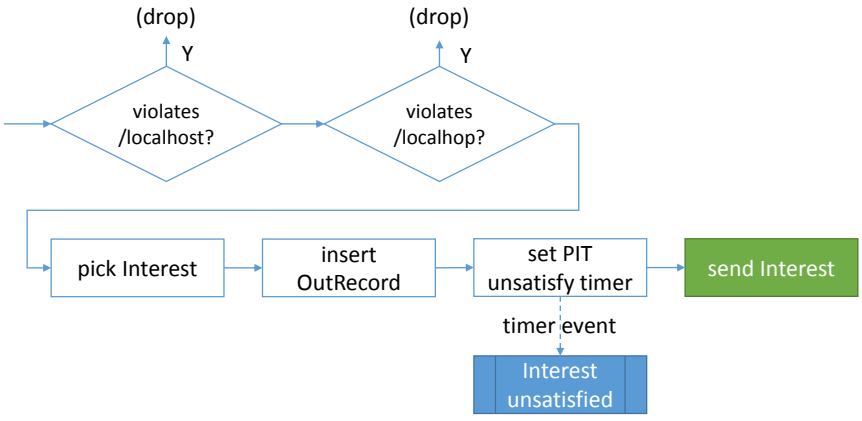

Figure 12: Outgoing Interest pipeline

- The initial step is to check for potential violations of `/localhost` and `/localhop` scopes [8]:

  - Interest packets that start with `/localhost` prefix cannot be send out to a non-local Faces
  - Interest packets that start with `/localhop` prefix can be send out to a non-local Faces only if PIT entry has at least one in-record that represents a local Face.

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

- On the next step an Interest packet is selected among the recorded Interests inside in-records in the PIT entry. This is necessary because Interests in different in-records can have different guiders (e.g., `InterestLifetime`). The current implementation always selects the last incoming Interest. However, this simple selection criteria can change in the future releases after we understand better the effects of guiders.

- If the strategy indicates a new nonce is wanted (the `wantNewNonce` flag), the Interest is copied, and a random nonce is set onto the copy.

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

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

- After PIT entry is updated, the outgoing Interest pipeline sets the *unsatisfy timer* for the PIT entry. This timer expires when all InRecords in the PIT entry expire. Expiration of the unsatisfy timer triggers entering the *Interest unsatisfied pipeline* (Section 4.2.5).

- Finally, the Interest is forwarded via the Face.

4.2.4 Interest Reject Pipeline

This pipeline is implemented in `Forwarder::onInterestReject` method and is entered from `Strategy::rejectPendingInterest` method which handles *reject pending Interest action* for strategy (Section 5.1.2). The input parameters to this pipeline include a PIT entry.

The only action currently defined in this pipeline is to set the *straggler timer*. After the straggler timer expires, the PIT entry is getting removed.

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

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

4.2.5 Interest Unsatisfied Pipeline

This pipeline is implemented in `Forwarder::onInterestUnsatisfied` method and is entered from the *unsatisfy timer* (Section 4.2.3) when InterestLifetime expires for all downstreams. The input parameters to this pipeline include a PIT entry.

The processing steps in the Interest unsatisfied pipeline include:

- Determining the strategy that is responsible for the PIT entry using Find Effective Strategy algorithm on the StrategyChoice table (see Section 3.4.1).

- Invoking *before expire Interest* action of the effective strategy with the PIT entry as the input parameter (Section 5.1.1).

- Removing PIT entry.

  Note that at this stage there is no need to keep PIT entry alive for any time longer, as it is the case in the Interest reject pipeline (Section 4.2.4). Expiration of the unsatisfy timer implies that PIT entry was already alive for substantial period of time and all Interest loops have been already prevented and no matching Data packet has been received.

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 Data packets

4.3.1 Incoming Data Pipeline

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

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

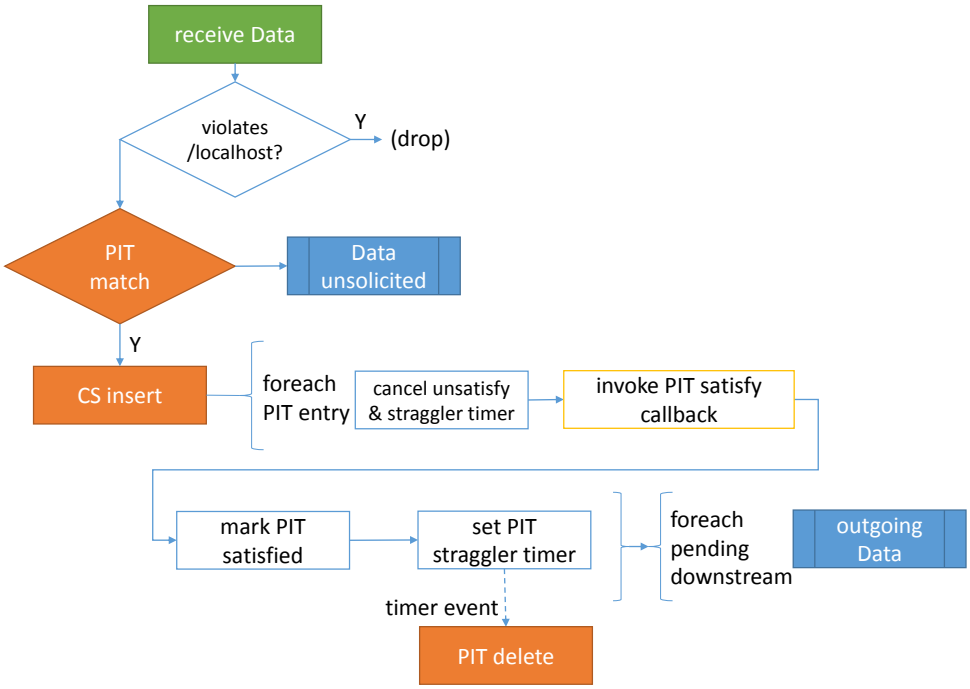

Figure 13: Incoming Data pipeline

- Similar to the incoming Interest pipeline, the first step in the incoming Data pipeline is to check the Data packet for violation of `/localhost` scope [8]. If the Data comes from a non-local Face, but the name begins with `/localhost` prefix, the scope is violated, Data packet is dropped, and further processing is stopped.

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

- After name-based scope constraint is checked, the Data packet is matched against the PIT using Data Match algorithm (Section 3.3.2). If no matching PIT entry is found, the Data is unsolicited, and is given to *Data unsolicited pipeline* (Section 4.3.2).

- If one or more matching PIT entries are found, the Data is inserted to ContentStore. Note that even if the pipeline inserts the Data to the ContentStore, whether it is stored and how long it stays in the ContentStore is determined by ContentStore admission and replacement policy.[1]
- The next step is to cancel the *unsatisfy timer* (Section 4.2.3) and *straggler timer* (Section 4.2.4) for each found PIT entry, because the pending Interest is now getting satisfied.
- Next, the effective strategy responsible for the PIT entry is determined using Find Effective Strategy algorithm (Section 3.4.1). The selected strategy is then triggered for the *before satisfy Interest* action with the PIT entry, the Data packet, and its incoming Face (Section 5.1.1).
- The PIT entry is then marked satisfied by deleting all in-records, and the out-record corresponding to the incoming Face of the Data.
- The *straggler timer* (Section 4.2.4) is then set on the PIT entry.
- Finally, for each pending downstream except the incoming Face of this Data packet, *outgoing Data pipeline* (Section 4.3.3) is entered with the Data packet and the downstream Face. Note that this happens only once for each downstream, even if it appears in multiple PIT entries. To implement this, during the processing of matched PIT entries as described above, NFD collects their pending downstreams into an unordered set, eliminating all potential duplicates.

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.

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

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

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 one or more PIT entries. The input parameters to this pipeline include a Data packet, and the outgoing Face.

This pipeline includes the following steps:

- The Data is first checked for `/localhost` scope [8]:
  - Data packets that start with `/localhost` prefix cannot be send out to a non-local Faces.[2]

  `/localhop` scope is not checked here, because its scope rules do not restrict outgoing Data.
- The next step is reseved for the traffic manager actions, such as to perform traffic shaping, etc. The current implementation does not include traffic manager implementation, but it is planned to be implemented in one of the next releases.
- Finally, the Data packet is sent via the outgoing Face.

[1] The current implementation has fixed "admit all" admission policy and "priority FIFO" as replacement policy, see Section 3.2.
[2] This check is only useful in a specific scenario (see NFD Bug 1644).

5 Forwarding Strategy

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

To provide the maximum flexibility, NFD allows per-namespace selection of the specific strategy, which is envisioned to be performed by the NFD operator. This per-namespace strategy choice is recorded in StrategyChoice table (Section 3.4), which is consulted in the forwarding pipelines when decision about Interest forwarding needs to be made. In addition to the Interest forwarding decision points, strategy can also receive notifications when the forwarded Interests are getting satisfied or timed out. Therefore, strategy presents a closed loop subsystem in NFD to control Interest forwarding.

Conceptually, a strategy can be considered a program, which is written for an abstract machine (strategy API, Section 5.1) and determines how to forward Interests. All current NFD strategies are written in C++ and are built-in into the NFD binary. However, future releases of NFD may allow custom strategies to be loaded at runtime and/or written in a scripting language against the strategy API abstract machine. Therefore, we chose to identify the forwarding strategy by NDN name, which can universally represent either a built-in strategy (Section 5.2) or, in the future, any external strategy program to be fetched from the network.

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

5.1 Strategy API

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

A strategy is invoked through one of the *triggers* (Section 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::Strategy` class, and is expected to be overridden by a subclass.

After Receive Interest Trigger

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

When an Interest is received, passes necessary checks, and needs to be forwarded, *Incoming Interest pipeline* (Section 4.2.1) invokes this trigger with the PIT entry, incoming Interest packet, and FIB entry. At that time, the following conditions hold for the Interest:

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

After being triggered, the strategy should decide whether and where to forward this Interest. If the strategy decides to forward this Interest, it should invoke *send Interest* action at least once; it can do so either immediately or some time in the future using a timer.[3] If the strategy concludes that this Interest cannot be forwarded, it should invoke *reject pending Interest* action, so that the PIT entry will be deleted shortly.

[3] **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 even will be cancelled and new strategy may not decide to forward the Interest until after all out-records in the PIT entry expire.

Before Satisfy Interest Trigger

This trigger is declared as `Strategy::beforeSatisfyPendingInterest` 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 measurement results for the pending Interest.

When a PIT entry expires because it has not been satisfied before all in-records expire, before it is deleted, *Interest Unsatisfied pipeline* (Section 4.3.1) invokes this trigger with the PIT entry.

5.1.2 Actions

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

Send Interest action

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

This action triggers entering the *Outgoing Interest pipeline* (Section 4.2.3).

Reject Pending Interest action

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

This action triggers entering the *Interest reject pipeline* (Section 4.2.4).

5.1.3 Storage

Strategies are allowed to store arbitrary information on PIT entries, PIT downstream records (in-records), PIT upstream records (out-records), and Measurements entries, all of which are derived from `StrategyInfoHost` type. Inside the triggers, the strategy already has access to PIT entry and can lookup desired in- and out-records. Measurement entry of the Measurements Table (Section 3.5) can be accessed via `Strategy::getMeasurements` method; the strategy's access is restricted to Measurements entries under the namespace(s) under its control (Section 3.5.2).

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

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

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

5.2 Built-in Strategies

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

- broadcast strategy (`/localhost/nfd/strategy/broadcast`, Section 5.2.1) sends every Interest to every upstream.
- best route strategy (`/localhost/nfd/strategy/best-route`, Section 5.2.2) sends Interest to lowest cost upstream.
- client control strategy (`/localhost/nfd/strategy/client-control`, Section 5.2.3) allows the consumer to control where an Interest goes.

- NCC strategy (`/localhost/nfd/strategy/ncc`, Section 5.2.4) is similar to CCNx 0.7.2 default strategy.

5.2.1 Broadcast Strategy

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

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

5.2.2 Best Route Strategy

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

The strategy forwards new Interests only; if an Interest is not new (unexpired out-record exists), it is not forwarded. For a new Interest, the list of nexthop records is consulted to find an *eligible* (Section 5.2.1) upstream with lowest routing cost, and the Interest is forwarded to that face. If there is no eligible upstream, the Interest is rejected.

5.2.3 Client Control Strategy

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

If an Interest is received from a LocalFace (Section 2.3) that enables NextHopFaceId feature in LocalControlHeader, and the Interest packet carries a LocalControlHeader that contains a NextHopFaceId field, the Interest is forwarded to the outgoing face specified in the NextHopFaceId field if that face exists, or dropped if that face does not exist. Otherwise, the Interest is forwarded in the same manner as the best route strategy (Section 5.2.2).

5.2.4 NCC Strategy

The NCC strategy is an reimplementation of CCNx 0.7.2 default strategy [10]. It has similar algorithm but is not guaranteed to be equivalent. This strategy is implemented as `nfd::NccStrategy` 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?

For the most applications and the most network environments, it is sufficient to use one of the existing strategies. If an application wants a fine-grain control of Interest forwarding, it can use the client control strategy (Section 5.2.3) and specify an outgoing face for every Interest; however this could control the outgoing face of local forwarder only.

When developing a new strategy, one needs to remember that the strategy choice is local to a forwarder. 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 [9, 11].

5.3.2 Develop a New Built-in Strategy

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

If the strategy needs to store information, it is needed to decide whether the information is related to a namespace or an Interest. Information related to a namespace but not specific to an Interest should be stored in Measurements entries; information related to an Interest should be stored in PIT entries, PIT downstream records, or PIT upstream records. After this decision is made, a data structure derived from `StrategyInfo` class needs to be declared. In the existing implementation, such data structures are declared as nested classes as it provides natural grouping and scope protection of the strategy-specific entity, but it is not required to follow the same model. If timers (Section 9.7) are needed, `EventId` fields needs to be added to such data structure(s).

The final step is to implement the *triggers* with the desired strategy logic. When implementing strategy logic, refer to Section 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 not NULL (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.
- 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.
- 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. It is also not allowed to send the same Interest via the same outgoing face before the previous out-record expires.
- The strategy should not violate scope. If the scope is violated, the *outgoing Interest pipeline* (Section 4.2.3) will not send the Interest and the strategy may incorrectly gauge data plane performance.
- The strategy is responsible for performing congestion control.

Before the strategy can be actually used, it is necessary to modify `daemon/fw/available-strategies.cpp` and install the new strategy to the list of existing built-in strategies. If the strategy is installed as non-default, the strategy needs to be activated on desired namespaces via a StrategyChoice management command (Section 6.1.3), e.g., using `nfdc` command-line tool.

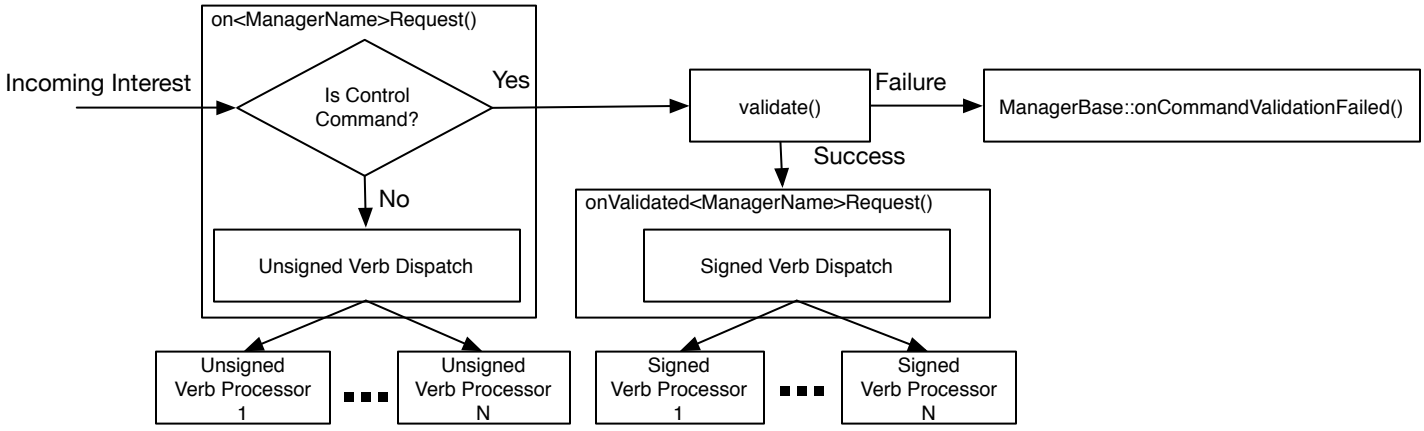

Figure 14: Overview of the manager verb dispatch workflow

6 Management

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

1. Add and remove FIB entries (FIB Manager)
2. Face creation, destruction and enable/disable local control features (Face Manager)
3. Outbound Interest next hop and inbound Data previous hop (Face Manager)
4. Forwarding strategy (Strategy Choice Manager)

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

In general, NFD managers do not offer much functionality through a programatic API. Most managers only need to allow requests to be routed to them via methods of the form `on`ManagerName`Request` and to hook into the configuration file parser. Instead, the Interest/Data management protocols provide rich access to NFD's internals.

Many management actions require the use of *control commands*; a form of signed Interests. These allow NFD to determine whether the issuer is authorized to perform the specified action. Management modules respond with *control responses* to inform the user of the commands success or failure. Control responses have status codes similar to HTTP and describe the action that was performed or any errors that occurred. Another type of actions is to query the current state of NFD instead of commanding for a change of state. Interests for this type of actions do not need to be signed. The returned data is not encrypted currently. In the future if data access control is desired, some data can be encrypted.

Most of the managers currently utilize dispatch tables for routing incoming requests to the correct processing method. All management protocol requests, whether commands or dataset 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. When the request is given to the manager's initial Interest handler method (as specified by `InternalFace::setInterestFilter`), the *verb* is used as the key to the dispatch table to locate the correct method for processing. These processing methods are referred to in the code as *verb processors*. Managers support a range of verbs, some of which need to be signed Interests (control commands) while others do not (datasets). Due to different processing requirements between these two types of requests, the managers maintain a separate dispatch tables for each: signed and unsigned verb tables and processors.

Dispatching to unsigned verb processors is typically done in the initial Interest handling method (`on`ManagerName`Request`). However, control commands must first be validated before dispatching to either methods of the form *on*ManagerName*ValidatedRequest* or `ManagerBase::onCommandValidationFailed` on validation and validation failure, respectively.

Managers use the `validate` method to invoke the `CommandValidator` through an indirection method inherited from `ManagerBase`; the common parent of all managers. `CommandValidator` is an NFD provided indirection around `libndn-cxx`'s

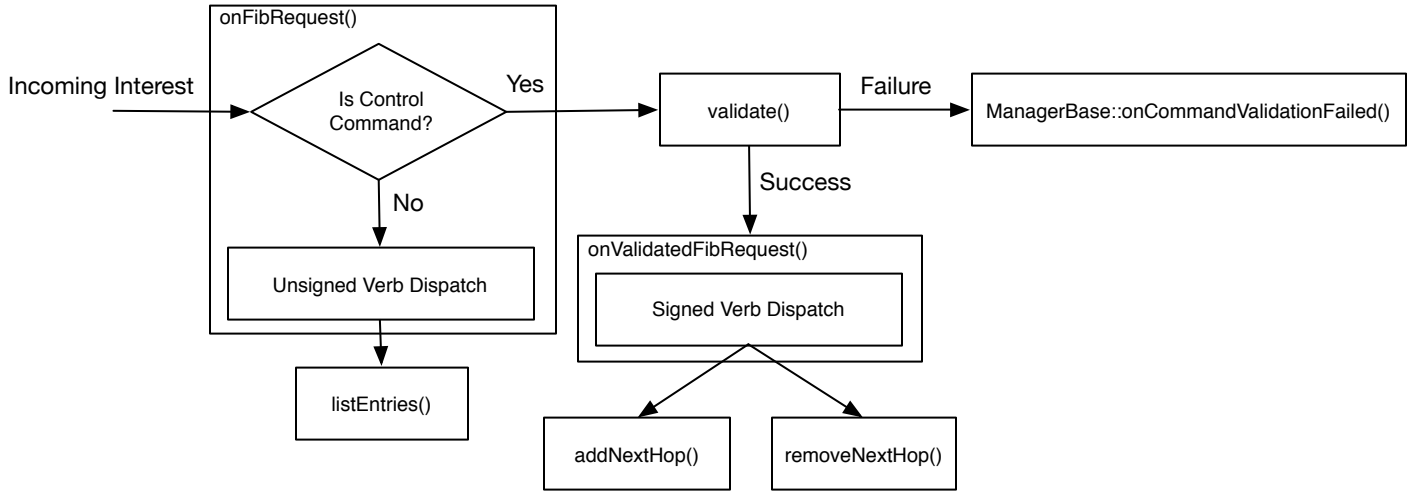

Figure 15: Overview of the FIB Manager's verb dispatch workflow

command authorization and validation functions. All NFD managers use `CommandValidator` (through `ManagerBase`) to determine whether or not a command is authorized (see Section 6.2.4).

The successful validation code path then performs command argument extraction, validation, and initializes an protocol specific implicit arguments before dispatching to the appropriate signed verb processor. Unknown and unsupported verbs typically fail at this point with a code 501 "Unsupported command" control response.
`ManagerBase::onCommandValidationFailed` will generate control responses with code 403 "Unauthorized command" to indicate the failure to the requester.

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

6.1 Managers

6.1.1 FIB Manager

The FIB Manager provides authorized users to modify NFD's FIB and publishes a dataset of all FIB entries and their next hops. At a high-level, authorized users can request the FIB Manager to:

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

The first two capabilities correspond to the `add-nexthop` verb while removing a next hop falls under `remove-nexthop`. The manager listens for Interests under the `/localhost/nfd/fib` and uses `onFibRequest` as the initial Interest handling namespace upon construction.

The FIB Manager supports uses of the following signed verb processors to handle control commands:

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

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

FIB Dataset On the unsigned request code path (i.e. `listEntries`), the FIB manager uses a `FibEnumerationPublisher` instance to publish FIB entries according to the FIB dataset specification. The `FibEnumerationPublisher` holds a reference to the FIB, and publishes FIB entries using the shared management internal face under the `/localhost/nfd/fib/list` prefix.

The FIB Manager's interaction with the dataset publisher is limited to calling `FibEnumerationPublisher::publish`, which is inherited from `SegmentPublisher`. On invocation, the `publish` will serialize the FIB in the form of a collection of `FibEntry` and nested `NextHopList` TLVs. Please refer to Section 6.2.2 for more details on the inner workings of the `SegmentPublisher`.

6.1.2 Face Manager

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

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

The Face Manager relies on the NFD configuration file's `face_system` section. This allows the manager to determine what protocol channels should be constructed for future Face creation. The `onConfig` method performs dispatching for `face_system` subsections (methods beginning with `processSection-`). All subsection processors are given the `ConfigSection` instance representing their subsection (a typedef around the boost property tree node) and a flag indicating whether or not a dry run is currently being performed. This allows NFD to test the sanity of the configuration file before performing any modifications.

Some subsection processors take a list of `NetworkInterfaceInfo` pointers. `onConfig` gets this list from the `listNetworkInterfaces` free function. The list describes all available network interfaces available on the machine. In particular, `processSectionUdp` and `processSectionEther` use the list for detecting multicast-capable interfaces for creating multicast faces.

The Face Manager maintains a protocol (`string`) to `shared_ptr<ProtocolFactory>` mapping (`m_factories`) to facilitate channel and Face creation. The mapping is initialized during configuration by the `processSection-` methods. Each subsection processor creates a factory of the appropriate type and stores it in the mapping. For example, the TCP processor creates a `shared_ptr<TcpFactory>` that can be referenced as "tcp". These factories are then used to create the protocol channels (e.g. `shared_ptr<TcpChannel>`) that will later be used to create Faces. Please refer to Section 2 for more details on the workings and interactions of the `ProtocolFactory`, `Channel`, and `Face` classes.

Command Processing On creation, the Face Manager listens for Interests on `/localhost/nfd/faces` and handles them in the `onFaceRequest` method. Validated control commands are dispatched by the `onValidatedFaceRequest` method to the appropriate verb processor:

- `createFace`: create unicast TCP/UDP Faces
- `destroyFace`: destroy Faces
- `enableLocalControl`: enable local control feature on requesting Face
- `disableLocalControl`: disable local control feature on requesting Face

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

`createFace` uses `FaceUri`s to parse the incoming URI in order to determine the type of Face to make. The URI's scheme (e.g. "tcp", "tcp4", "tcp6", etc.) is used to lookup the appropriate `ProtocolFactory` in `m_factories`. Failure to find a factory results in a code 501 "unsupported protocol" control response. Otherwise, `ProtocolFactory::createFace` is then invoked to attempt to create the Face. This method dispatches to `onCreated` and `onConnectFailed` on successful and failed Face creation, respectively. `onCreated` adds the new Face to the Face Table and generates a code 200 "success" control response while `onConnectFailed` results in a code 408 response with the reason for failure.

The Face Manager must protect NFD from errors that may result from `ProtocolFactory::createFace` call. However, the Face creation process is asynchronous (e.g. a DNS lookup may be required). As such, the provided try-catch block cannot guarantee that it will catch all generated exceptions. Instead, the Face Manager expects the `onConnectFailed` callback to be used to signal the failure.

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

Local control headers are enabled/disabled on a per-Face basis providing two capabilities called *local control features*:

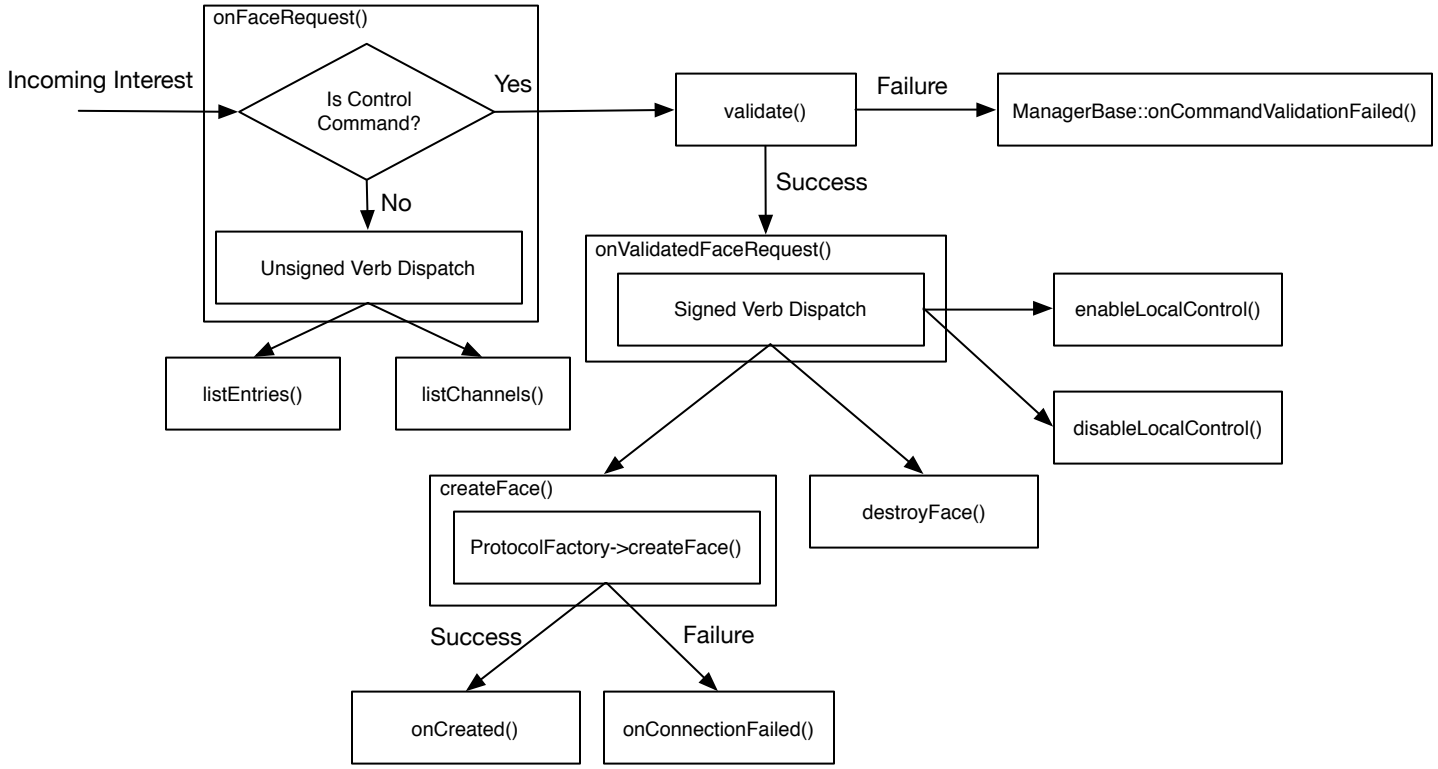

Figure 16: Overview of the Face Manager's verb dispatch workflow

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

As their names imply, the `(enable|disable)LocalControl` methods enable and disable the specified local control features on the Face sending the control command. Both methods utilize `extractLocalControlParameters` to perform common functionality: option validation, Face Table lookups, and ensure the requesting Face is local. Failure to find the requesting Face results in a code 410 "Face not found" response while non-local Faces trigger a code 412 "Face is non-local".

Datasets and Event Notification The Face Manager provides two datasets: Channel Status and Face Status. The Channel Status dataset lists all channels (in the form of their local URI) that this NFD has created and can be accessed under the `/localhost/nfd/faces/channels` namespace. Face Status, similarly, lists all created Faces, but provides much more detailed information such as flags and incoming/outgoing Interest/Data counts. The Face Status dataset can be retrieved from the `/localhost/nfd/faces/list` namespace. These datasets are each published by `SegmentPublisher` derivates: `FaceStatusPublisher` and `ChannelStatusPublisher` invoked by the `listFaces` and `listChannels` methods, respectively.

In addition to these datasets, the Face Manager also publishes notifications when Faces are created and destroyed. This is done using a `NotificationStream` instance triggered by the `onAddFace` and `onRemoveFace` methods. When the Face Manager is created, these two methods are set as subscribers to the Face Table's `onAdd` and `onRemove` events. Face notifications are published under the `/localhost/nfd/faces/events` namespace, which is ignored by the Face Manager. Instead, Face event notifications are expected to be served from the content store. See Section 6.2.3 for more information on `NotificationStream`.

6.1.3 Strategy Choice Manager

The Strategy Choice Manager is responsible for setting and unsetting forwarding strategies for routing prefixes and their children via the Strategy Choice table. However, NFD must have already installed a strategy in order for it to successfully set the active strategy (see Section 5). In other words, the manager allows users to choose a prefix's strategy from a well-known pool, but not add new or modify existing NFD strategies. Attempting to change to an unknown strategy will result in a code 504 "unsupported strategy" response. By default, there is at least the root prefix ("/") available for strategy changes, which defaults to the "best route" strategy. However, it is an error to attempt to unset the strategy for root (code 403).

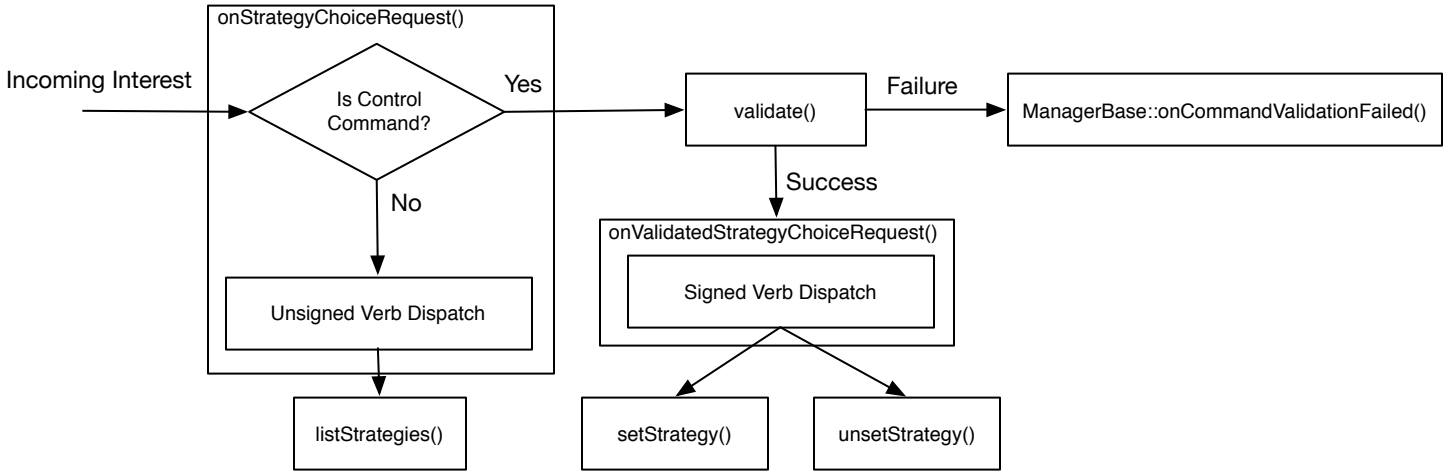

Figure 17: Overview of the Strategy Choice Manager's verb dispatch workflow

Internally, Strategy Choice Manager utilizes the same type of verb processing dispatch system as the FIB and Face Managers. It listens for incoming Interests on `/localhost/nfd/strategy-choice` and initially handles them in `onStrategyChoiceRequest`. Authorized control commands are dispatched to their verb-specific processing methods (`(un)setStrategy`) in `onValidatedStrategyChoiceRequest`.

The Strategy Choice Manager also provides a dataset of the active strategy for each prefix under the `/localhost/nfd/strategy-choice/list` namespace. Upon receiving a request, the manager uses a `StrategyChoicePublisher` instance to serialize the Strategy Choice table into `StrategyChoice` TLVs.

6.1.4 Manager Base

`ManagerBase` is the base class for all managers. This class holds the manager's shared `InternalFace` and provides a number of commonly used methods. In particular, `ManagerBase` provides indirection to the `CommandValidator` via the `validate` method and extracts/validates control parameters (control command arguments). `ManagerBase` also provides convenience methods for initializing and sending control responses (`setResponse` and `sendResponse`).

On construction, `ManagerBase` obtains a reference to a `CommandValidator` that will be used for control command authorization later. Derived manager classes provide the `ManagerBase` constructor with the name of their `privilege` (e.g. `faces`, `fib`, or `strategy-choice`). This privilege is used to specify the set of authorized capabilities for a given NDN identity certificate in the configuration file.

6.1.5 Forwarder Status

Forwarder Status (or `StatusServer`) provides information about the NFD and statistics. This includes the NFD's version, startup time, Interest/Data packet counts, and various table entry counts. `StatusServer` listens for Interests on `/localhost/nfd/status` and publishes Data packets with a 5 second freshness time to avoid excessive processing.

6.2 Utility Classes

6.2.1 Internal Face

Each manager is constructed around a shared `internal face` instance. Managers register their control command prefix with the internal face to receive commands and publish control responses. The internal face also holds the `CommandValidator` used to authorized NFD control commands. However, the validator is only accessed through `ManagerBase::validate`.

`InternalFace` is derived from `AppFace`, which defines `setInterestFilter`, `put`, and most importantly, `sign`. `AppFace` provides a `KeyChain` instance that is used by the managers to sign control responses and datasets.

6.2.2 Segment Publisher

`SegmentPublisher` provides a generalized method for segmenting and publishing datasets. `SegmentPublisher`s are constructed around an `AppFace` that is used for signing and posting Data as well as a name prefix under which the Data should

be published. Users of this class family need only call the `publish` method and the chosen publisher will handle all of the publications details.

`SegmentPublisher` itself is an abstract class. Dataset publishers extend the class to define a `generate` method that handles the serialization of the desired dataset. When `publish` is called, it will first invoke `generate` to fill an internal `EncodingBuffer` with protocol defined data.

After `generate` initializes the `EncodingBuffer`, `SegmentPublisher` will segment it into Data packets of up to 4,400 bytes with incrementing segment number name components. The last Data packet in the collection is marked by a final block ID referring to itself.

More concretely, consider the FIB dataset and its `FibEnumerationPublisher`. This publisher holds a reference to the FIB. When `generate` is called, the publisher will walk the FIB and serialize each entry into a series of `FibEntry` and nested `NextHopList` TLVs that represent the entirety of the FIB that are placed into the provided `EncodingBuffer`. This buffer and length of the serialized content are returned to the `publish` method so that it can chunk it into Data packets.

The `SegmentPublisher`'s `generate`-based approach to Data production allows it to be highly generalized and removes the need of re-writing boilerplate segmentation code. Adding a new dataset publisher is therefore very simple; the developer need only handle encoding their data source (e.g. the FIB in the case of the FIB dataset) without concern for how it should be packetized.

6.2.3 NotificationStream

NotificationStream serializes and publishes objects to an application Face (a class implementing the *AppFace* interface) under a specified namespace. The only constraint placed on publication by the *postNotification* method is that the object to be published has a no argument *wireEncode* method.

6.2.4 Command Validator

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

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

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

`CommandValidator` also supports the notion of a "wildcard" identity certificate for demonstration purposes to remove the "burden" of configuring certificates and privileges. Note, however, that this feature is security risk and should not be used in production.

6.2.5 General Configuration File Section Parser

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

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

6.2.6 Tables Configuration File Section Parser

`TablesConfigSection` provides parsing for the `tables` configuration file section. This class can then configuration the various NFD tables (CS, PIT, FIB, Strategy Choice, and Measurements) appropriately. Currently, the `tables` section on supports changing the default maximum number of Data packets that the content store can hold. 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`.

7 RIB Management

The RIB Manager that runs as a separate process, **NRD** (NFD RIB Daemon), manages the routing information base (RIB) and updates the FIB as needed. Logically, the RIB Manager is a part of NFD; however, it is implemented as a separate process to handle complex routing table manipulation while keeping packet forwarding logic lightweight and simple. Figure 18 shows the high-level interaction of the RIB Manager with NFD and other applications. A more detailed interaction is shown in 19 diagram.

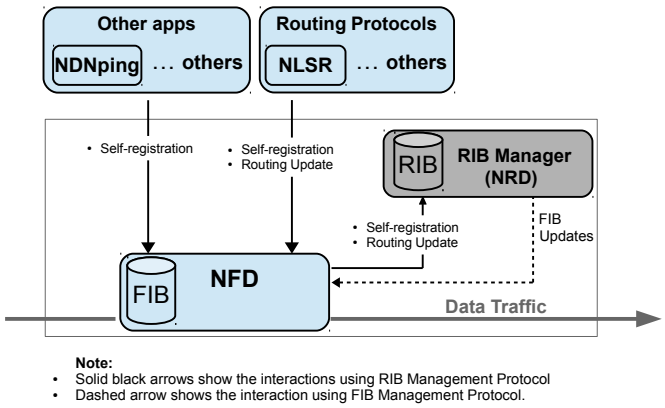

Figure 18: RIB Manager – system diagram

NFD provides various 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 flags off of the FIB Manager. It receives all registration requests, processes the included flags, and creates FIB updates as needed, which makes the forwarder leaner and faster. As the RIB can be updated by different parties in different ways, including various routing protocols, application's prefix registrations, and command-line manipulation by sysadmins, the RIB management module also provides a common abstraction to all these processes and generates a consistent forwarding table. Therefore applications should use the RIB management interface to manipulate the RIB, and only NRD should use the FIB management interface to directly manipulate NFD's FIB.

7.1 Initializing NRD

The Rib manager module is initialized by creating an instance of the **RibManager** class, called *Nrd*. This class provides abstractions for controlling and managing the RIB. When an instance of the RIB Manager is created, it does the following in the given order:

- looks for the *rib* block in the NFD configuration file and loads the localhost and localhop validation rules
- registers the control command prefixes */localhost/nfd/rib* and */localhop/nfd/rib*[4] with NFD. This allows NRD to receive the prefix registration requests.
- enables the local control header, which allows it to get the FaceId from where the prefix registration/unregistration Command Interest was received. This FaceId is used for self-registration of applications.
- subscribes to the FaceMonitor class to receive notifications whenever a Face is created or destroyed so that it can update the corresponding RIB entries.

7.2 Communicating with NRD

Applications, including routing protocols, may register or unregister routes through NRD by using a control commands sent to the RIB management module.

[4]If enabled in the NFD conf file.

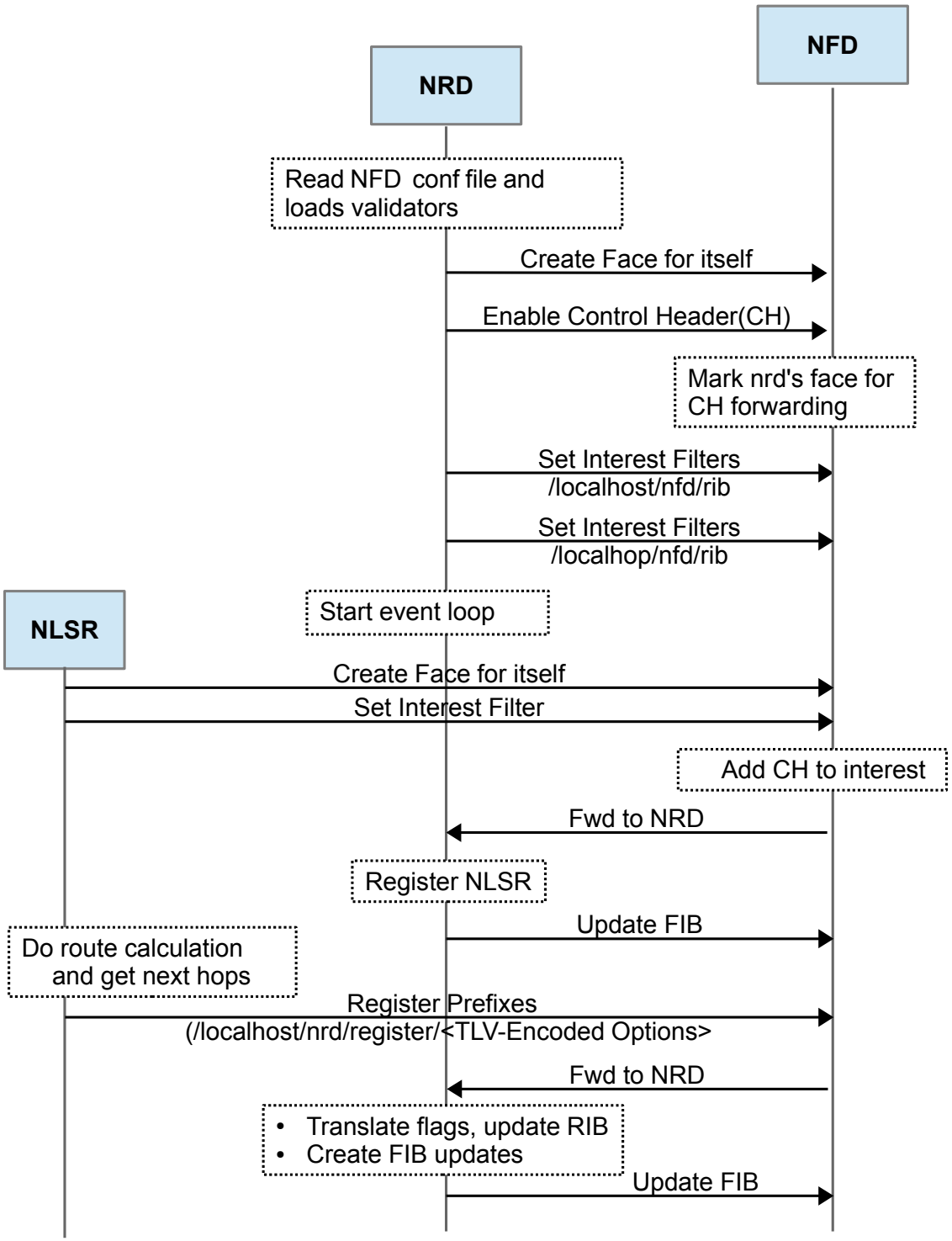

Figure 19: RIB Manager – timing diagram

7.2.1 Registering a route

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

- `Name`: Associated name prefix **(required)**
- `FaceId`: The face ID returned from the Face Management module after face creation.
  If the FaceId is set to zero or not set, the requesting face is used (self-registration).
- `Origin`: The producer of the command; defaults to 0 (Local producer application)
- `Cost`: Route preference; defaults to 0
- `Flags`: inclusive OR of route inheritance flags (Section 7.4)
- `ExpirationPeriod`: The duration for which the route is active (in milliseconds)

7.2.2 Unregistering a route

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

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

7.3 RIB Entry

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

- `name`: Associated name prefix
- `FaceId`: The nexthop face
- `origin`: The producer of the announcement. A prefix registration request can be sent by different parties. This field is used to differentiate between them. If one of the routing protocols or applications quits, this field helps to remove RIB entries added by that protocol or application.
  - `0`: Local producer application
  - `128`: NLSR
  - `255`: Static route
- `flags`: inclusive OR of route inheritance flags (Section 7.4)
- `cost`: Route preference
  - When there are multiple routes for the same Name prefix, a lower cost indicates a more preferred nexthop.
- `expires`: The duration for which the route is active (in milliseconds)

7.4 Prefix Registration Flags

The prefix registration flags allows fine grained control over prefix registration. The currently defined flags are discussed in the following:

- `CHILD_INHERIT`: Longer matching Name prefixes may use this route; true by default.
  Use `ndn::nfd::ROUTE_FLAG_CHILD_INHERIT` to set flag.
- `CAPTURE`: No shorter Name prefix route may be used for this prefix; overrides the `CHILD_INHERIT` flag. Use `ndn::nfd::ROUTE_FLAG_CAPTURE` to set flag.

Example RIB:

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

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

7.5 On Request

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

- Register Entry: The RIB Manager takes the passed parameters from the incoming request and searches for a RIB entry that matches the Name, FaceId and Origin of the incoming request. If the FaceId is 0 in the incoming request then it means that an application is trying to register itself with NFD (self-registration). For self-registration requests, the RIB Manager fetches the FaceId of the application from the Control Header and uses it for registration. If no match is found, the passed parameters are inserted as a new entry. Otherwise, the matching entry is updated. It also processes the flags of the received request and updates the relevant rib entries. Finally, the RIB manage adds nexthops to the FIB as needed.
- Unregister Entry: NRD takes the passed parameters and removes the corresponding nexthop from the FIB. If the removal is successful, the RIB entry with the same name, FaceId, and origin is removed from the RIB.

In both cases, the RIB Manager returns a control response with code 200 if the command is executed successfully.

7.6 Termination

The RIB Manager is an essential module of the NFD. Therefore, the NRD, which controls the RIB Manager, must be running simultaneously with the NFD to be functional. The NFD's startup script restarts the NRD if it crashes or quits unexpectedly. On the other hand, if the NFD quits or crashes, the NRD shuts down gracefully.

7.7 Extending RIB Manager

The RIB Manager currently supports only two commands – register and unregister. However, functionality of the RIB Manager can be extended by introducing more commands. For example, if a node needs to announce a prefix currently, it has to configure the routing protocol. A set of commands for advertising and withdrawing prefixes, with the help of any application, could provide a more unified way for the operators to publish name prefixes. However, this would also require support from the routing protocols, so that they can receive updates from the RIB Manager. Similarly, more registration flags can be introduced in the RIB Manager as needed.

8 Security

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

8.1 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[5] (both in the NFD configuration file, see Section 9.1). 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 9.1).

8.2 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 NRD.

8.2.1 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[6] 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.

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

[6] 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.

8.2.2 NFD Trust Model

With the exception of the RIB Manager (NRD), 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 9 and Section 6.

8.2.3 NRD Trust Model

NRD 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 NRD authenticates the `rib` command Interest, NRD will issue `fib` command Interests to NFD to set up FIB entries.

NRD's trust model 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 [12]. 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 [13].

NRD supports two modes of prefix registration: localhost and localhop. In localhop mode, NRD 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 NRD. Alternatively, the trust anchor could be the key of a testbed site or instituion, thus limiting NRD prefix registration to users at that site/institution.

In localhost mode, NRD expects to receive prefix registration requests from local applications. By default, NRD 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 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 8.

9.1 Configuration File

Many aspects of NFD are configurable through a configuration file, which adopts the Boost INFO format [14]. This format is very flexible and allows any combination of nested configuration structures.

9.1.1 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`
- **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 9.2.
- **face_system**: The face system section fully controls allowed face protocols, channels and channel creation parameters, and enabling multicast faces. Specific protocols may be disabled by commenting out or removing the corresponding nested block in its entirety. Empty sections will result in enabling the corresponding protocol with its default parameters.

  Version 0.2.0 of NFD contains the following face protocols:
  - **unix**: Unix protocol

    This section can contain the following parameters:
    - `listen`: controls whether the created Unix channel is in listening mode and creates Unix faces when an incoming connection is received (enabled by default)
    - `path`: sets the path for Unix socket (default is `/var/run/nfd.sock`)
  - **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 two subsections: `localhost_security` and `localhop_security`. The former controls authorizations for registering and unregistering prefixes in RIB from local users (through local faces: Unix socket or TCP tunnel to 127.0.0.1). `localhop_security` defines authorization rules for so called localhop prefix registrations: registration of prefixes on the next hop routers.

  Unlike the main `authorizations` section, the rib security section uses a more advanced validator configuration, thus allowing a greater level of flexibility in specifying authorizations. In particular, it is possible to specify not only specific

authorized certificates, but also indirectly authorized certificates. For more details about validator configuration and its capabilities, refer to Section 8 and Validator Configuration File Format specification [13].

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 [15]) 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.

9.1.2 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 [14] library and NFD implements an additional wrapper (ConfigFile) to simplify configuration file operations.

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

   As a general guideline, the callback should be able to process the same section multiple times in actual run mode without causing problems. This feature is necessary in order to provide functionality of reloading configuration file during run-time. In some cases, this requirement may result in cleaning up data structures created during the run. If it is hard or impossible to support configuration file reloading, the callback must detect the reloading event and stop processing it.
3. Update NFD initialization in `daemon/main.cpp` file. In particular, the new management module needs to be created somewhere around `initializeManagement` call, once created the second step callback needs to be added to ConfigFile class dispatch. Similar updates should be made to `reload` call in `main.cpp`.

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

9.2 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.

9.2.1 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` and `nrd` programs. 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).

9.2.2 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).

9.3 Hash Computation Routines

Common services also include several hash functions, based on city hash algorithm [7], 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.

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.

9.4 DNS resolver

DNS resolution is a common operation needed in various places to support face management (create IP-based channels and face). As such, NFD contains a helper class to perform DNS resolution in synchronous and in asynchronous manner. Synchronous operations are (and should be in any new code) limited to the initialization phase because they stall all other operations.

Therefore, all DNS resolution operations that needed to be performed in parallel with normal NFD packet forwarding (e.g., during UDP and TCP face creation based on management commands) must be performed in asynchronous manner. The only complication is that instead of receiving resolution result as a return value, the asynchronous version of resolver helper returns the result (or failure) via the specified callback.

To use resolution for protocol `X`:

- Include `core/resolver.hpp`
  - For synchronous resolution calls:
    `Resolver<X>::syncResolve(hostname, port)`
  - For asynchronous resolution calls:
    `Resolver<X>::asyncResolve(hostname, port, callbackWhenSucceed, callbackWhenFailed)`

There are also two convenience versions to simplify resolution operations for UDP and TCP protocols: `UdpResolver` and `TcpResolver`.

9.5 Event Emitter

The `EventEmitter` abstraction provides a light-way event subscription mechanism to get notifications (callback calls) when a specific event occurs. One of the primary uses of `EventEmitter` is the face system, where it is used to notify the subscribed entities of the arrival of new Interest and Data packet arrivals on the Face, or when the Face fails.

`EventEmitter` is very much like an ordinary callback mechanism, with the difference that it allows zero or multiple subscribers for the same event (i.e., zero or many callbacks per event).

When deemed suitable, the event emitter can be used as follows.

- **Event generator code:**
  - Include `core/event-emitter.hpp`.
  - Define a variable of templated class `EventEmitter`, specifying a list of parameters that will be passed to each event. That is, if the generated event needs to pass a string to the event subscribers (subscriber will have `void subscriber(const std::string& msg)` prototype), the variable should be defined as:

    ```
    EventEmitter<std::string> events;
    ```

  - Whenever an event occurs, call events as if it is a function, specifying the event parameters.

    ```
    events("test event");
    ```

- **Event subscriber code:**
  - Include your event generator header file.
  - Define function or method that will receive event notifications, e.g.,

    ```
    void onEvent(const std::string& msg) ...
    ```

  - When appropriate, subscribe the method or callback to the `EventEmitter`. To do so, you just need to add using `operator+=` your method or function to the event variable using bind construction:

    ```
    generator.events += bind(&MyClass::onEvent, this, _1);
    ```

  - When events are no longer required, use `generator.events.clear()` to remove all subscribers.

    Note that the EventEmitter implementation makes a simplifying assumption that event subscribers will only be removed together. In other words, after two or more subscribers are added go the `EventEmitter` instance, as shown below, it is impossible to remove an individual subscriber. The only option is to remove all subscribers at the same time. While this assumption is generally a drawback, it does not limit any functionality within NFD and allows a simpler and more optimized implementation of the event mechanism.

    If it is required to remove individual subscribers, you would need to modify the `EventEmitter` implementation or use some other event subscription implementation.

9.6 Face Status Monitoring Helper

As described in Section 6, NFD provides a way to notify interested applications about the creation of new faces and destruction of existing ones. This is required functionality for special applications that support NFD, in particular for RIB manager (to remove stale records) and `nfd-autoreg`, providing automatic prefix registration when a new face is created as a response to incoming TCP connection or UDP packet.

Generally, to obtain face status notifications, one needs to send out properly formatted Interests towards `/localhost/nfd/faces/e` and continue re-expressing these Interests as soon as they are satisfied or timeout. To simplify these operations, `FaceMonitor` class has been created. In order to receive face status notifications, an application needs to create an object of this class and register a callback. After this, whenever a new face is created or destroyed, the specified callback will be fired automatically with the argument that describes the event.

9.7 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`.

9.8 Global IO Service

The NFD packet forwarding implementation is based on Boost.Asio [4], 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.

References

[1] NDN Project Team, "NDN packet format specification (version 0.1)," http://named-data.net/doc/ndn-tlv/, 2014.

[2] ——, "NFD - Named Data Networking Forwarding Daemon (version 0.1.0)," Online: http://named-data.net/doc/NFD/0.1.0/, 2014.

[3] ——, "NFD management protocol," Online: http://redmine.named-data.net/projects/nfd/wiki/Management, 2014.

[4] C. Kohlhoff, "Boost.Asio," Online: http://www.boost.org/doc/libs/1_48_0/doc/html/boost_asio.html, 2003–2013.

[5] 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

[6] W. Pugh, *Skip lists: A probabilistic alternative to balanced trees*. Springer, 1989.

[7] Google, "The CityHash family of hash functions," Online: https://code.google.com/p/cityhash/, 2011.

[8] "Nfd v0.1.0 release notes," http://named-data.net/doc/NFD/0.1.0/RELEASE_NOTES.html.

[9] 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

[10] J. Shi, "ccnd 0.7.2 forwarding strategy," http://redmine.named-data.net/projects/nfd/wiki/CcndStrategy, University of Arizona, Tech. Rep., 2014.

[11] 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/

[12] Y. Yu, "NDN regular expression," http://redmine.named-data.net/projects/ndn-cxx/wiki/Regex, 2014.

[13] ——, "Validator configuration file format," http://redmine.named-data.net/projects/ndn-cxx/wiki/CommandValidatorConf, 2014.

[14] M. Kalicinski, "Boost.PropertyTree," Online: http://www.boost.org/doc/libs/1_42_0/doc/html/property_tree.html, 2008.

[15] NDN Project Team, "NDN-Cert," Online: https://github.com/named-data/ndncert, 2014.