

NLSR Developer's Guide

Vince Lehman, Muktadir Chowdhury, Nicholas Gordon, Ashlesh Gawande¹

¹University of Memphis

November 1, 2017

Abstract

The Named Data Link-State Routing Protocol (NLSR) is a Named Data Networking (NDN) routing protocol that populates NDN Forwarding Daemon's (NFD) Routing Information Base (RIB). The main design goal of NLSR is to provide a routing protocol to populate NFD's RIB and Forwarding Information Base (FIB). NLSR calculates the routing table using link-state or hyperbolic routing and produces multiple faces for each reachable name prefix in a single authoritative domain. NLSR will continue to evolve alongside the NDN protocol, NFD, and ndn-cxx. This document is meant to explain the design of NLSR including major module and data structures descriptions and the interactions between those components.

Contents

1	Introduction	2
1.1	NLSR Modules and Data Structures	2
1.2	Protocol Overview	2
1.2.1	Discovering Neighbors	2
1.2.2	Disseminating Routing Information	2
1.2.3	Calculating the Routing Table and Populating NFD's FIB	3
1.3	Dispatcher	3
2	Hello Protocol	5
2.1	Determining Neighbor's Status	5
2.2	Responding to Hello Interests	5
2.3	Failure and Recovery Detection	6
3	Sync Logic Handler	7
3.1	On Sync Update	7
3.2	Publish Routing Update	7
4	Link-State Advertisements	8
4.1	LSA Base Class	8
4.2	Adjacency LSAs	8
4.3	Coordinate LSAs	8
4.4	Name LSAs	8
5	Link-State Database	9
5.1	Retrieving an LSA	9
5.2	General Procedure	9
5.3	Scheduling LSA Builds	9
5.4	Building LSAs	9
5.5	Installing and Processing LSAs	9
5.6	LSA Expiration	10
5.7	LSA Refreshment	10

6	Routing Table	11
6.1	Routing Table Calculators	11
6.1.1	Link-State Routing Table Calculator	11
6.1.2	Hyperbolic Routing Table Calculator	11
6.2	Notifications for Newly Calculated Next Hops	11
7	Name Prefix Table	12
7.1	Adding an NPT Entry	12
7.2	Removing an NPT Entry	12
7.3	Updating an NPT Entry with New Routing Table Entries	13
7.4	Routing Table Entry Pool	13
8	FIB Interaction	14
8.1	Updating the FIB	14
9	Prefix Update Processor	15
9.1	Advertising and Withdrawing Routes	15
9.2	Security	15
10	NFD RIB Command Processor	16
10.1	Advertising and Withdrawing Routes	16
10.2	Security	16
11	LSDB Status Dataset	17
11.1	Requesting the dataset	17
12	Security	18
12.1	Creating Keys and Certificates	18
12.2	Certificate Publishing	19
13	Configuration File	20
13.1	Naming Conventions	20
13.2	General Section	20
13.3	Neighbors Section	20
13.4	Hyperbolic Section	21
13.5	FIB Section	21
13.6	Advertising Section	21
13.7	Security Section	22

1 Introduction

The Named-data Link State Routing protocol (NLSR) is an intra-domain routing protocol for Named Data Networking (NDN). It is an application level protocol similar to many IP routing protocols, but NLSR uses NDN's Interest/Data packets to disseminate routing updates. Although NLSR is designed in the context of a single domain, its design patterns may offer a useful reference for future development of inter-domain routing protocols.

NLSR supports name-based routing in NDN, computes routing ranks for all policy-compliant next-hops which provides a name-based multi-path routing table for NDN's forwarding strategy, and ensures that routers can originate only their own routing updates using a hierarchical trust model.

1.1 NLSR Modules and Data Structures

NLSR contains multiple modules that each contribute to the total realization of the protocol. Many of the modules interact with one another to trigger some behavior or to modify information in data structures. NLSR uses the following modules:

- **Hello Protocol** (Section 2) - determines the status of neighboring routers using periodic Hello Interests and notifies other modules when neighbors' statuses change.
- **ChronoSync** - provides network-wide synchronization of NLSR LSDBs. [?]
- **Sync Logic Handler** (Section 3) - handles sync update notifications from NSync by retrieving updated LSAs.
- **LSAs** (Section 4) - represent routing information published by the router.
- **LSDB** (Section 5) - stores the LSA information distributed by other routers in the network.
- **Routing Table** (Section 6) - calculates and maintains a list of next hops for each router in the network.
- **Name Prefix Table** (Section 7) - stores all advertised name prefixes and their next hops.
- **FIB** (Section 8) - maintains a shadow FIB which represents the intended state of NFD's FIB [?].
- **Prefix Update Processor** (Section 9) - listens for dynamic prefix announcements to advertise or withdraw name prefixes.
- **NFD RIB Command Processor** (Section 10) - listens for readvertise-to-NLSR commands to advertise or withdraw name prefixes that were inserted into NFD.

1.2 Protocol Overview

NLSR is designed to accomplish three main tasks: (1) discover adjacent neighbors; (2) disseminate and synchronize topology, name prefix, and hyperbolic routing information; and (3) calculate a routing table and populate NFD's FIB. The entire protocol is described in detail in the NLSR paper [?].

1.2.1 Discovering Neighbors

NLSR determines the adjacency status of neighboring routers using the Hello Protocol module (Section 2). When the Hello Protocol detects a status change for a neighbor, it will ask the LSDB module (Section 5) to update the router's advertised adjacency information.

Additionally, NLSR discovers the infrastructure for communicating with neighbors by querying NFD, which maintains a dataset of all information for its Faces.

1.2.2 Disseminating Routing Information

When a router's LSAs (Section 4) changes, the information of this change should be distributed to every other router in the network. The Sync Logic Handler module (Section 3) is used to notify the synchronization protocol of changes to the router's own LSAs as well as to learn of LSA changes from other routers in the network; the Sync Logic Handler module interfaces with ChronoSync to perform the two tasks.

When the Sync Logic Handler module learns of a new LSA, it will inform the LSDB module. The LSDB module will attempt to fetch the new LSA and will store it in the LSDB module's database if it can be retrieved. If the newly fetched LSA informs the router of previously unknown routing information, the LSDB module will inform other modules depending on the type of routing information:

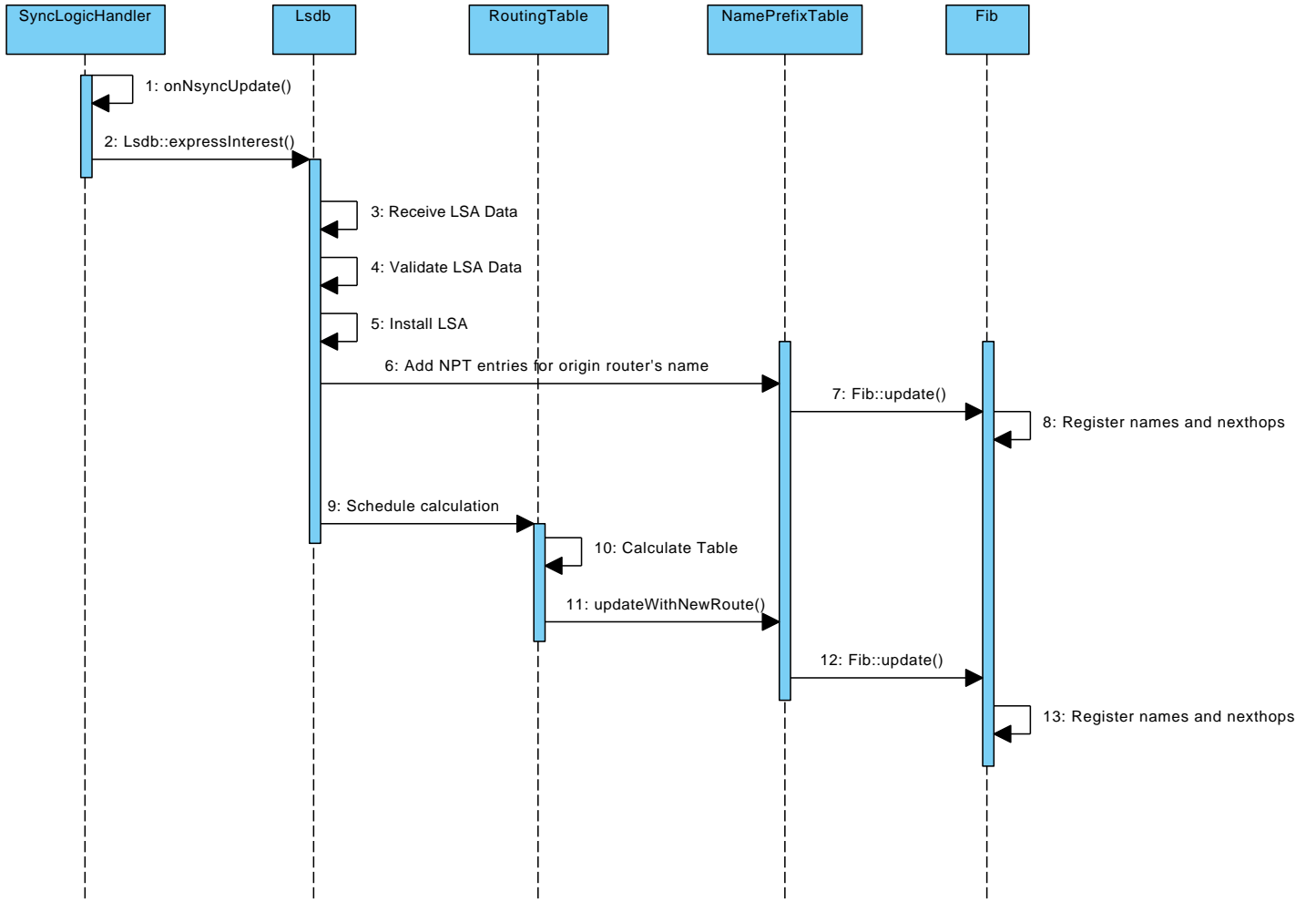


Figure 1: Simplified Diagram of the Actions of NLSR's Modules

- **Change in network topology** - the LSDB module will ask the Routing Table module (Section 6) to recalculate paths in the network
- **Change in name prefix advertisement** - the LSDB module will inform the Name Prefix Table module (Section 7), which will in turn notify the FIB module (Section 8) in order to add or remove the changed name prefixes.
- **Change in hyperbolic coordinates** - the LSDB module will inform the Routing Table module (Section 6), so the Routing Table module can calculate an up-to-date routing table.

1.2.3 Calculating the Routing Table and Populating NFD's FIB

When the routing table is calculated by the Routing Table module, the computed next hops are passed to the Name Prefix Table module. The Name Prefix Table module will then further pass the next hops to the FIB module to update NLSR's expected state of NFD's FIB. The FIB module will then perform the registrations or unregistrations with NFD's FIB.

A simplified diagram of NLSR's actions when receiving new routing information is shown in Figure 1. The remainder of this documentation will describe the purpose of and interaction between each module in more detail.

1.3 Dispatcher

NLSR takes advantage of a variety of ndn-cxx convenience mechanisms. Among these is the dispatcher. The dispatcher provides facilities for receiving and decoding ControlCommands, which simplifies the processing workflow. The dispatcher itself is [well-documented](#), so we will only give a brief overview here.

- Any prefixes that are registered, such as “prefix/register”, *must* be registered before top-level prefixes. All prefix registrations should go in `Nlsr::registerLocalhostPrefix`.
- The dispatcher can be used to accept incoming `ControlCommands` and respond to requests for datasets. Currently NLSR uses the dispatcher only for accepting incoming `ControlCommands`.
- Top-level prefixes cannot overlap. For example, you cannot register “/localhost/nlsr” and then “/localhost/nlsr/fib”. The second registration must be done as a sub-prefix of the first, i.e. the first prefix, and then “fib”.

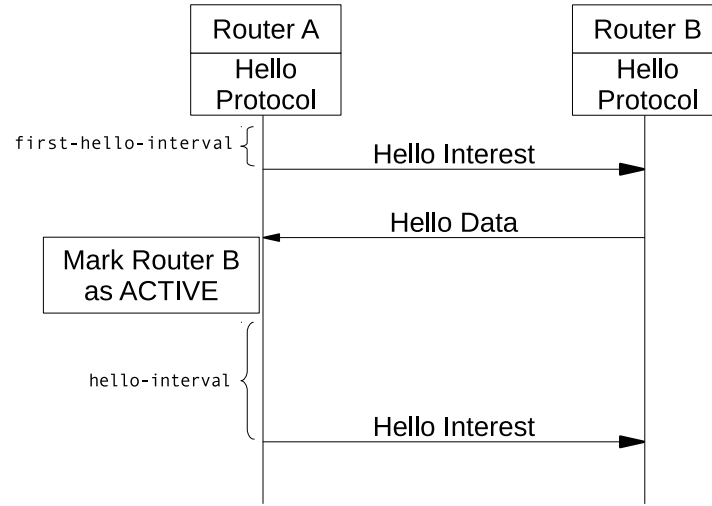


Figure 2: Router A determines the initial status of Router B

2 Hello Protocol

The Hello Protocol module periodically sends Hello Interests to learn the activity status of the router's neighbors. Hello Interests' names are constructed in the form: `/<neighbor's-router-prefix>/NLSR/INFO/<this-router's-prefix>`. If a neighbor responds to a Hello Interest, the neighbor is considered to be up and **ACTIVE**. A Hello Data's name is constructed using the following convention: `/<neighbor's-router-prefix>/NLSR/INFO/<this-router's-prefix>/<version>`. If a neighbor fails to respond to a configurable number of Hello Interests (**hello-retries**), the neighbors is considered to be down and **INACTIVE**. The Hello Protocol continues to send these periodic Hello Interests to each of its neighbors every **hello-interval** seconds. If the Hello Protocol detects a change in a neighbors status (i.e. a router that was previously **ACTIVE** is not responding to Hello Interests or a router that was previously **INACTIVE** responds to a Hello Interest), it will notify the LSDB (Section 5) to schedule a new Adjacency LSA build to include the updated neighbor information.

2.1 Determining Neighbor's Status

The Hello Protocol begins by scheduling Hello Interests to be sent to each neighbor of the router after **first-hello-interval** seconds. When the scheduled event is triggered, the Hello Protocol iterates through the list of neighbors first checking if there is already a Face to the neighbor. If there is a Face that has already been created, the Hello Protocol will construct and send a Hello Interest to the neighbor. If a Face has not been created for the neighbor, the Hello Protocol will attempt to create a Face to the neighbor and register the neighbor's router prefix. If the Face is created successfully, the Hello Protocol registers the Sync prefix, LSA prefix, and Key prefix using the Face ID returned by the Face creation command and sends out the Hello Interest. If the Face cannot be created, the Hello Protocol considers the failure as a Hello Interest timeout.

If the Hello Protocol receives Data in response to its Hello Interest, it will first ask the Validator module to verify that the Data is valid. Data is valid if the Data is legitimately signed (in the ordinary cryptographic way) and if the key name and Data name are of a certain format. If the Data is valid, the corresponding neighbor is set as **ACTIVE** and its timeout count is reset to zero. If the neighbor was previously **INACTIVE**, an Adjacency LSA build is scheduled to include the newly **ACTIVE** neighbor. If the Data is not valid, the packet is dropped.

If the Hello Interest sent to the neighbor times out, the corresponding neighbor's timed-out count is incremented. If the neighbor's timed-out count is less than **hello-retries** in the configuration file, the Hello Protocol will send another Hello Interest after **hello-timeout** seconds. If the neighbor's timed-out count equals the **hello-retries** value and the neighbor is currently marked as **ACTIVE**, the neighbor's status is set to **INACTIVE** and an Adjacency LSA build is scheduled.

2.2 Responding to Hello Interests

If the Hello Protocol receives a Hello Interest from another router, it will first verify that the Hello Interest came from one of its configured neighbors. If so, the Hello Protocol responds to the Interest with Hello Data. To optimize the time to respond to link recoveries, the Hello Protocol will then immediately send a Hello Interest to the neighbor if the neighbor is currently marked as **INACTIVE**.

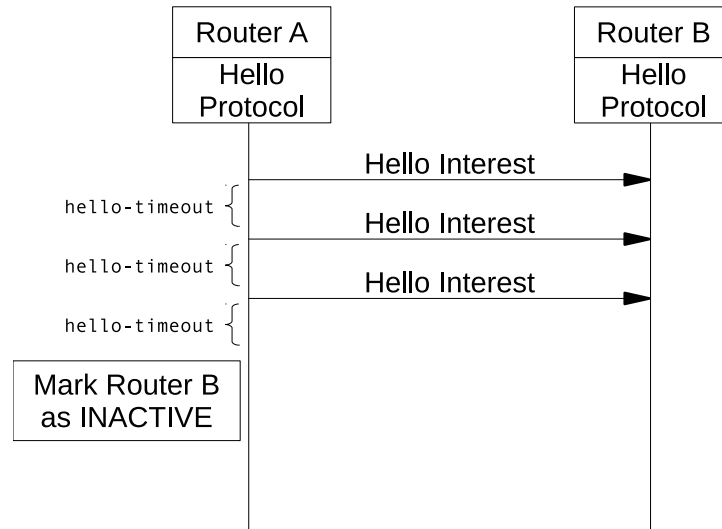


Figure 3: Router A determines that Router B has failed

2.3 Failure and Recovery Detection

The Hello Protocol will consider a neighbor as failed if the neighbor is currently **ACTIVE**, but Hello Interests sent to the neighbor have timed-out **hello-retries** number of times. A failure can also be detected if a `FaceEventNotification` is received with the information that a Face to the neighbor has been destroyed. The event is handled by the `Nlsr` class, but the triggered events simulate the actions of the Hello Protocol. If the neighbor was currently **ACTIVE**, the neighbor will be set to **INACTIVE**, the neighbor's timed-out count will be set to **hello-retries**, and an Adjacency LSA build will be scheduled.

The Hello Protocol will consider a neighbor as recovered if the neighbor is currently **INACTIVE**, but the Hello Protocol has received valid Data in response to a Hello Interest sent to the neighbor.

3 Sync Logic Handler

The Sync Logic Handler acts as the interface between the synchronization protocol and the NLSR application. The Sync Logic Handler receives notifications from the synchronization protocol when another router updates an LSA, where an update can be modification of the contents of the LSA, or just incrementing the sequence number to refresh it. The Sync Logic Handler then determines if the updated LSA should be retrieved. The Sync Logic Handler is also how NLSR notifies the sync protocol when its own LSAs are updated.

3.1 On Sync Update

When the sync protocol receives an update, the procedure roughly is this:

- For each name in the update:
- Check that the sequence number in the LSA is newer than the one stored in the LSDB.
- If so, tell the LSDB to fetch this new LSA. The LSDB will finish processing.

When the synchronization protocol receives a sync update, which may contain multiple distinct items, the names and sequence numbers of each item will be passed to `SyncLogicHandler::onNsyncUpdate()`. Since other syncs in the network blindly transmit what they think is new, we need to check that it's new *to us*, and we use the LSA sequence number to do that. The higher sequence number of the locally-stored LSA with the same name as in the update and the sequence number in the update is taken to be the newer one, noting that an absent LSA has a sequence number of 0. If the update is found to be newer, the Sync Logic Handler will call `Lsdb::expressInterest()`, which attempts to fetch the LSA represented by the update. Other LSDB methods will finish processing and installing the new LSA. (Section 5)

3.2 Publish Routing Update

When any of a router's LSAs are updated or refreshed by the LSDB, the LSDB will use the `SyncLogicHandler::publishRoutingUpdate()` method to notify the sync protocol that the sequence number for that LSA has changed. The Sync Logic Handler will also write the updated sequence number to file, so that a restarting router can continue publishing routing updates with sequence numbers larger than the sequence numbers it had published before. This is only an optimization. If a router were to reset its sequence number to 1, other routers would initially reject these LSAs as not being new. However, the LSAs in their LSDBs would eventually expire, since they are not being refreshed anymore. Once those LSAs expire, the LSAs that the restarted router is publishing would then be considered new. However, this process could take quite a while, so we optimize by resuming numbering where we left off.

4 Link-State Advertisements

Link-State Advertisements (LSAs) represent pieces of routing information distributed by routers.

4.1 LSA Base Class

All three LSA implementations inherit from an LSA Base class, `Lsa`, which maintains information that is included in each LSA. The `Lsa` class contains the following member variables:

- **Origin Router** - the router that advertised the LSA. Specifically, this is a name prefix that follows the NLSR convention of router naming.
- **Sequence Number** - a number used to indicate the LSA's version. Because sequence numbers are preserved between NLSR restarts, a higher sequence number also always indicates a *newer* LSA.
- **Expiration Time Point** - a time point indicating when the LSA is no longer valid. This currently is represented as a Unix timestamp (i.e. seconds since Jan 1, 1970).

4.2 Adjacency LSAs

Adjacency LSAs maintain an `AdjacencyList` which contains information about all the currently **ACTIVE** neighbors of the origin router. It also includes the number of active routers, not just the list itself. This aids in serialization.

4.3 Coordinate LSAs

Coordinate LSAs maintain the hyperbolic angle(s) and hyperbolic radius of the origin router.

4.4 Name LSAs

Name LSAs maintain a `NamePrefixList` which contains the name prefixes that are reachable at the origin router.

5 Link-State Database

The Link-State Database (LSDB) holds LSA information distributed by other routers in the network. The LSDB stores all types of LSAs and will trigger events when a new LSA is added, updated, or expires. The LSDB also handles LSA retrieval, performs LSA builds, and triggers routing table calculations.

5.1 Retrieving an LSA

The LSDB provides the `Lsdb::expressInterest()` method as a public interface to retrieve an LSA from the network. If LSA Data is returned, the LSDB will validate the Data using the Validator module. Then, it will perform the necessary LSDB modifications. If the LSA Interest times out, the LSDB will retry until it reaches a configurable maximum number of tries, or a configurable deadline passes.

The LSDB uses the SegmentFetcher system to retrieve LSAs. LSAs very often will exceed the maximum NDN packet size. In these cases, the LSA needs to be split into segments to be sent, so the LSDB uses the SegmentFetcher to send all LSAs. The SegmentFetcher can decide if the data actually needs to be split.

5.2 General Procedure

The LSDB is responsible for building, installing, and publishing NLSR's LSAs, as well as for installing and processing LSAs from other NLSRs. Generally, the functions of the LSDB are:

- Schedule the building of an LSA.
- Building the LSA.
- Installing the LSA into the LSDB.

5.3 Scheduling LSA Builds

LSAs need to be rebuilt whenever the routing information NLSR has changes. This includes events like neighbors becoming active, or when a prefix for advertisement is inserted by the Prefix Update Processor, which would cause an adjacency LSA or a name LSA rebuild, respectively. To improve performance, instead of directly building adjacency LSAs the first request schedules the build, and build requests that occur after the scheduling but before the actual event are aggregated (in other words, ignored), because they will be satisfied by the already-scheduled build. Some specifics are shown below.

- **Adjacency LSAs** – will only be scheduled if link-state routing is enabled. In particular, this means Adjacency LSA builds will *not* occur if hyperbolic routing is enabled. Note that adjacency LSAs will be built if dry-run hyperbolic routing is enabled, as the network is still using link-state routing to calculate paths.

5.4 Building LSAs

Building LSAs has a part common to all LSAs and a part specific to each LSA type. For example, all LSA types increment sequence number and have the same expiration length, and of course come from the same router. Additionally, all LSA builds cause a sync update publishing. However, each type of LSA includes different data, to represent different kinds of information that NLSR has. (Section 4) In particular:

- **Adjacency LSAs** – the number and a list of active neighbors is included.
- **Coordinate LSAs** – the hyperbolic radius and all hyperbolic angles are included.
- **Name LSAs** – the list of name prefixes that are accessible at this router is included.

5.5 Installing and Processing LSAs

LSA installation procedure is mostly the same across any type of LSA, but each type also has installation behavior specific to that type, too. For any LSA type, we need to schedule an expiration event, and we need to update several fields in the LSA. However, installing an adjacency or coordinate LSA causes a Routing Table calculation, but a name LSA does not, for example. Additionally, the name of the origin router is added as a “routable prefix” in the NPT. (Section 7).

- **Adjacency LSAs** – each adjacency in the LSA will be added as a “routable prefix” to the NPT. If the adjacencies have changed since the last version of this LSA, a Routing Table calculation needs to be scheduled, because the state of the network has changed. Of course, this is necessarily true if the LSA is new to us. Important to note is that we will also install and process *our own* adjacency LSA in this way.
- **Coordinate LSAs** – the router that the LSA is from will be added as a “routable prefix” to the NPT. If the radius and coordinates have changed since the last version of this LSA, a Routing Table calculation needs to be scheduled, because the state of the network has changed. As above, this is true for a new LSA. This is only done if the LSA is from a foreign router.
- **Name LSAs** – each name prefix in the LSA will be added to our NPT. This is only done if the LSA is from a foreign router.

5.6 LSA Expiration

LSAs expire so that the network can clean up when a router crashes. The amount of time an LSA lasts is configurable. When an LSA expires, we refresh it if it’s our LSA, and remove it from the LSDB if not. There is a “grace period” window that is appended to the end of the expiration period of all LSAs, to provide time for the originating router to refresh the LSA and for it to propagate back to us. In all expiration cases, the name of the origin router will be removed from the NPT. What happens when an LSA is removed from the LSDB differs based on the type of LSA:

- **Adjacency LSAs** – a Routing Table calculation needs to occur, since the state of the network has changed, at least from our perspective.
- **Coordinate LSAs** – a Routing Table calculation needs to occur, since the state of the network has changed, at least from our perspective.
- **Name LSAs** – the name prefixes in the LSA are removed from the NPT.

5.7 LSA Refreshment

NLSR will only refresh its own LSAs. Additionally, the procedure for refreshing an LSA is the same for all types:

- Increment the LSA sequence number
- Schedule another expiration event. The length of time to wait until refreshing is configurable, but it should probably be lower than the expiration time that was set when building the LSA initially. This prevents other routers from deleting our LSAs because the network is slow, for instance. The length of time is set by `lsa-refresh-time` in the configuration file.
- Publish an update to sync

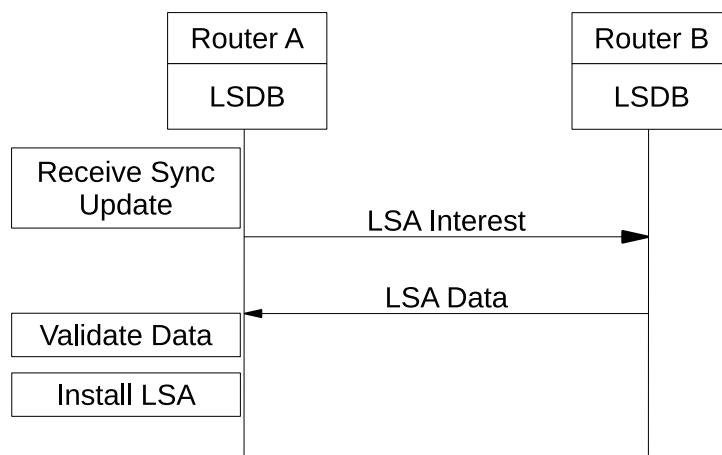


Figure 4: The general LSDB logic for each LSA type

6 Routing Table

The Routing Table module performs three main tasks: it performs the routing table calculations using a `RoutingTableCalculator` (Section 6.1), it stores the calculated routing table entries in a table, and notifies the Name Prefix Table module (Section 7) when there are changes to the calculated next hops.

6.1 Routing Table Calculators

The `RoutingTableCalculator` is a base class that provides functionality common to both link-state and hyperbolic routing. The Routing Table module uses the implementation class specific to the type of routing currently enabled.

6.1.1 Link-State Routing Table Calculator

The `LinkStateRoutingTableCalculator` class calculates the routing table uses Dijkstra's algorithm to calculate the shortest paths in the network. When `max-faces-per-prefix` is set to one, the `LinkStateRoutingTableCalculator` can simply run Dijkstra's algorithm. When `max-faces-per-prefix` is greater than one, indicating multi-path calculation, the `LinkStateRoutingTableCalculator` will iteratively perform Dijkstra's using only a single neighbor link as a next hop. The calculation will be performed using each neighbor in order to learn the path costs for each destination through each next hop.

6.1.2 Hyperbolic Routing Table Calculator

The `HyperbolicRoutingCalculator` class calculates the routing table using the Coordinate LSAs received from each router in the network to determine the cost from each of its neighbors to every other router in the network. The `HyperbolicRoutingCalculator` iterates through each of the router's neighbors calculating the hyperbolic distance from the neighbor to every other router in the network (excluding itself and the neighbor router). The `HyperbolicRoutingCalculator` then uses these calculated distances to add routing table entries to the destination with the neighbor as the next hop. The `HyperbolicRoutingCalculator` also adds a routing table entry to reach the neighbor itself; a routing table entry using the neighbor as a next hop to the neighbor with a cost of zero is added.

6.2 Notifications for Newly Calculated Next Hops

Once the Routing Table Module has finished calculating the routing table, it will update all of the Name Prefix Table's next hops. The Name Prefix Table Module will then update the next hops for each name prefix based on the newly calculated routing table. This process is described in more detail in Section 7.3.

7 Name Prefix Table

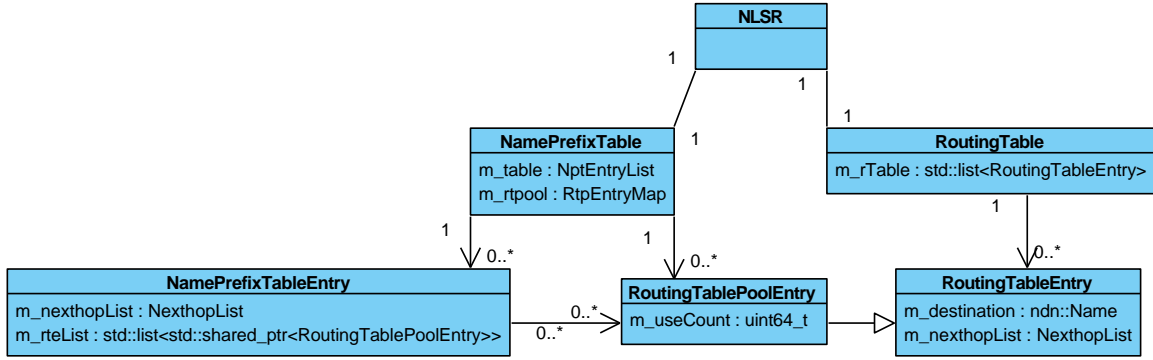


Figure 5: A

diagram of the NPT and Routing Table.

The “wire” arrows represent references (i.e. “x has y”), whereas the “solid” arrows represent inheritance (i.e. “y subclasses x”).

The quantification (e.g. 0..*) is standard UML.

The Name Prefix Table (NPT) is used by NLSR to maintain a list of all known name prefixes advertised by other routers, including router names. The NPT maintains a collection of NPT entries, where each entry represents a name prefix and all of its associated routing table entries. Additionally, to optimize the storage and association of the routing table entries, the NPT also maintains a collection of duplicated routing table entries, called routing table pool entries, which have an additional use count attribute. The NPT entries keep shared pointers to the appropriate routing table pool entries. If a name prefix is advertised by multiple routers, the name prefix will be represented by only one Name Prefix Table Entry, but will have multiple routing table pool entries which correspond with each origin router.

If a Name LSA exists with some advertised name prefix, then that prefix must have an entry in the NPT. So, if two routers advertise the same name prefix, i.e. their name LSAs contain a common name prefix, even if one router withdraws that common name prefix, the entry must remain in the NPT, because the other router still advertises it.

If any type of LSA for a remote router exists in the LSDB, the remote router’s name prefix must be in the NPT. An NPT Entry for a router name can only be removed when there are no more LSAs in the LSDB from the origin router. Note, even if some NPT entry has no next hops, it will *not* be removed from the NPT; it may later become possible to route to this prefix. These prefixes will be removed from the FIB, however.

7.1 Adding an NPT Entry

The `NamePrefixTable::addEntry()` method is the public interface for name prefixes to be added to the NPT. The name prefix as well as the router’s prefix which originates the name prefix are passed as parameters to the method.

If the name prefix is new, there will be no NPT entry for it, so one will be created. If the name prefix is not new, the existing entry will be updated, so the existing entry will also store this new origin router’s prefix, too. If after updating, the NPT entry has any next hops, which are associated to each of the origin router prefixes, the NPT will update the FIB to include that prefix and its next hops. The next hop list is sorted and truncated to be only as long as the `max-faces-per-prefix` variable. n

7.2 Removing an NPT Entry

The `NamePrefixTable::removeEntry()` method is the public interface for name prefixes to be removed from the NPT. The name prefix as well as the router’s prefix which originates the name prefix are passed as parameters to the interface.

This method removes an origin router prefix from some advertised name prefix. After this, there may be other origin routers that serve this name prefix, so this does not guarantee that the NPT entry will be deleted. If after updating the entry has any next hops, the NPT will update the FIB to reflect the change. Since the next hop list is sorted by ascending cost and its length truncated to `max-faces-per-prefix`, the contents of next hop list will not change if the removed origin router prefix was not already in the list passed to the FIB.

Note that even if the entry no longer has any next hops, it will be retained. All FIB entries for this prefix will be removed from the FIB, which will result in unregistrations from NFD, but the NPT entry will be retained. This is because it may become possible later to route to these origin routers again.

7.3 Updating an NPT Entry with New Routing Table Entries

When the Routing Table module has finished calculating, it will notify the NPT using the `NamePrefixTable::updateWithNewRoute()` method. The NPT will then make approximately $m \times n$ calls to `addEntry`, where m is the number of NPT entries and n is the number of origin routers for each m . That is, n will vary from one NPT entry to the next in most cases.

7.4 Routing Table Entry Pool

The Name Prefix Table has an internal data structure to help de-duplicate Routing Table information. Without this, each Routing Table entry has to be stored n times, if n is the number of prefixes that origin router advertises. Instead, each time a Routing Table entry would be fetched, the NPT first checks its internal data structure to see if that Routing Table entry is being used by another NPT entry. In that case those two NPT entries can share a pointer to that cached copy of the Routing Table entry.

This internal cache is smart, and will clean up from the cache unused entries when they are removed from the last NPT entry.

8 FIB Interaction

The FIB module interacts with NFD to perform registrations and unregistrations of routes. By registration, what is meant is the submission of a RIB route to the local NFD, which includes a name prefix, the Face ID of the nexthop, an expiration time, and the calculated cost from the Routing Table calculation. Additionally, NLSR sets a field to tell NFD that the route originates from NLSR, and sets a route inheritance flag.

The expiration time for a route is pegged at double the value of the LSA refresh time, which is defined by `lsa-refresh-time` in the configuration file. The route inheritance flag is set to capture, which forbids NFD from using a shorter prefix of the name prefix for forwarding.

More information about NFD's RIB can be found on the [Redmine wiki](#). An important thing to note is that NFD has a module called the FIB. Anywhere in this guide, the word "FIB" refers to the NLSR FIB, which models NLSR's expectation of how NFD would forward packets.

The FIB is directed by the Name Prefix Table, which registers and unregisters routes based on calculations by the Routing Table and advertisements from LSAs. The connection between the FIB and the NPT is through the `Fib::update()` method.

8.1 Updating the FIB

Generally, updating the FIB looks like this:

- Sort the list of next hops for the prefix, by cost.
- Take the cheapest `max-faces-per-prefix` hops. This can be set to have no limit, so all next hops are registered.
- Send a RIB route registration command for each next hop.
- Send a RIB route unregistration command for any next hops that have dropped out of the list. This includes next hops that became invalid since the last Routing Table calculation, as well as valid hops that are no longer in the top `max-faces-per-prefix` next hops.

If there are more passed next hops than the `max-faces-per-prefix`, the FIB module will only use the first `max-faces-per-prefix` number of next hops from the sorted list. If there are less passed next hops than `max-faces-per-prefix`, the FIB module will use all of the next hops. Specifically, when the NPT updates the FIB, the FIB creates entries so that it can compute the difference between the set of new next hops, and the set of old next hops that were registered at the last update. These entries are unique on the destination name prefix. The FIB will update an existing entry instead of creating a new one, which may involve unregistering old next hops, as mentioned above.

9 Prefix Update Processor

The Prefix Update processor allows manipulation of NLSR's advertised name prefixes with ordinary ControlCommands. Such commands may originate from something like `nlsrsrc`, the command line tool for manipulating NLSR.

9.1 Advertising and Withdrawing Routes

The processor accepts valid ControlCommands that are signed by the site operator's key. Additionally, the commands must be received on the `/localhost/nlsr/prefix-update/` prefix. The full condition list is specified in the validator rules in the configuration file.

The processor will send responses to commands.

9.2 Security

Prefix Update commands are similar to NFD RIB commands, but with one additional requirement, so they are more secure. In addition to being on the root-only `/localhost/` prefix, Prefix Update commands must be signed by the site operator's key. If the site operator's key were compromised, an attacker could create any number of NLSRs that impersonate the legitimate NLSR running at that site.

10 NFD RIB Command Processor

The NFD RIB Command Processor allows modification of NLSR's advertised name prefixes using NFD's RibMgmt commands. Such commands may originate from something like NFD's Readvertise module, which permits routes inserted in NFD to be propagated through to NLSR, so that NLSR can provide routing support for them.

10.1 Advertising and Withdrawing Routes

The processor accepts valid RibMgmt commands that have the name prefix to manipulate the origin of the route specified. No other validation is performed, as stated below.

The processor does not send any kind of response to commands.

10.2 Security

Any RibMgmt commands received on the `/localhost/nlsr/rib` prefix are considered secure, and are processed. This introduces a security hole because anyone who can send a RibMgmt command on this prefix can arbitrarily manipulate NLSR's advertised prefixes. However, because sending commands to this prefix requires root access, a would-be attacker will already have root access locally.

11 LSDB Status Dataset

NLSR makes available the entire contents of the LSDB upon request. A command can request LSAs of just a specific type, or request the collection of all LSAs. Unlike the Prefix Update and NFD RIB command processors, the LSDB Status dataset is available over *both* the `/localhost` and regular router prefixes. That is, you can request an arbitrary router's LSDB contents.

11.1 Requesting the dataset

- To request the local dataset, simply send an Interest of the form `/localhost/nlsr/lsdb/<dataset type>`.
- To request a remote dataset, send an Interest of the form `/<router name>/lsdb/<dataset type>`, where `<router name>` is whatever the router's name is. This is usually `general.network + general.site + general.router` from the configuration file.

Where `<dataset type>` is one of `names`, `adjacencies`, `coordinates` or `list`.

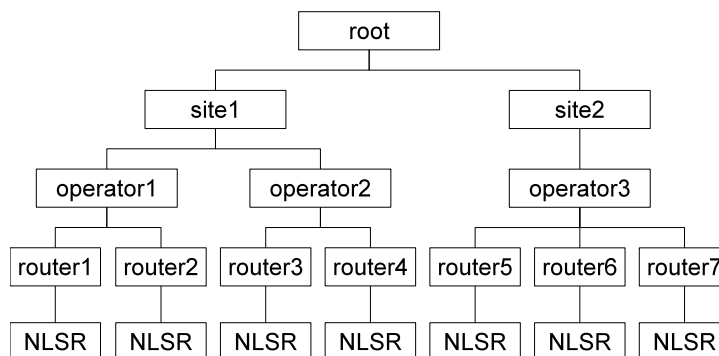


Figure 6: NLSR Trust Hierarchy

Table 1: Key Names

Key Owner	Key Name
Network	/<network>/KEY/<key>
Site	/<network>/<site>/KEY/<key>
Operator	/<network>/<site>/<operator>/KEY/<key>
Router	/<network>/<site>/<router>/KEY/<key>
NLSR	/<network>/<site>/<router>/NLSR/KEY/<key>

12 Security

The trust model of NLSR is semi-hierarchical. An example certificate signing hierarchy is shown in Figure 6. In this hierarchy, each entity's name and corresponding certificate name follow the convention described in Table 1.

12.1 Creating Keys and Certificates

The process to create keys and certificates for this hierarchy can be performed using the `ndnsec` [?] tools included with `ndn-cxx` [?]. The steps to create the keys and certificates is outlined below:

1. Create keys for Root:


```
ndnsec-keygen $ROOT_NAME > $ROOT_KEY_OUTPUT_FILE
```
2. Create certificate for Root:


```
ndnsec-certgen -N $ROOT_NAME -p $ROOT_NAME $ROOT_KEY_OUTPUT_FILE > $ROOT_CERT_OUTPUT_FILE
```
3. For each site, create keys and certificates:
 - (a) On the Site machine, generate keys for the Site:


```
ndnsec-keygen $SITE_NAME > $SITE_KEY_OUTPUT_FILE
```
 - (b) Copy `$SITE_KEY_OUTPUT_FILE` over to the machine where the Root certificate was created.
 - (c) Generate a certificate for the Site on the Root machine:


```
ndnsec-certgen -N $SITE_NAME -p $SITE_NAME -s $ROOT_NAME $SITE_KEY_OUTPUT_FILE > $SITE_CERT_OUTPUT_FILE
```
 - (d) Copy `$SITE_CERT_OUTPUT_FILE` over to the Site machine.
 - (e) Install the certificate on the Site machine:


```
ndnsec-cert-install -f $SITE_CERT_OUTPUT_FILE
```
 - (f) On the Site machine, create the Operator keys:


```
ndnsec-keygen $OP_NAME > $OP_KEY_OUTPUT_FILE
```
 - (g) On the Site machine, create the Operator certificate:


```
ndnsec-certgen -N $OP_NAME -p $OP_NAME -s $SITE_NAME $OP_KEY_OUTPUT_FILE > $OP_CERT_OUTPUT_FILE
```
 - (h) On the Site machine, create the Router keys:


```
ndnsec-keygen $ROUTER_NAME > $ROUTER_KEY_OUTPUT_FILE
```
 - (i) On the Site machine, create the Router certificate:


```
ndnsec-certgen -N $ROUTER_NAME -p $ROUTER_NAME -s $OP_NAME $ROUTER_KEY_OUTPUT_FILE > $ROUTER_CERT_OUTPUT_FILE
```
4. When NLSR starts, it will automatically create the NLSR keys and certificates for the router.

12.2 Certificate Publishing

In a network, every router must have the root certificate configured as a trust anchor for the **validator** in the configuration file. If two routers in a network do not share a common trust anchor, then when one seeks to validate the data of the other, they may be unable to establish trust in their signature. This is, of course, because of how the trust hierarchy is set up: you trust the person that signed some router's certificate, because it was signed by the site certificate, and the site certificate was signed by the region, etc., and the *nth* certificate was signed by the root certificate, which is your trust anchor, so you "just trust it". Moreover, if your trust anchor is *before* their trust anchor in the "chain", then they will be able to trust you, but you will not be able to trust them.

For each site, at least one router should publish the site certificate, and at least one router should publish the certificate of the site operator. Each router should publish its own certificate. This is a matter of performance; a network would work if all certificates for all nodes were kept centrally, but distributing the certificates in this way improves performance. All this information must be explicitly specified in the configuration file.

For example, the following configuration file indicates that NLSR should publish the site certificate and the router certificate:

```
...
security
{
    validator
    {
        ...
    }
    cert-to-publish "$SITE_CERT_OUTPUT_FILE" ; name of the site certificate file
    cert-to-publish "$SITE_CERT_OUTPUT_FILE" ; name of the router certificate file
    ...
}
```

13 Configuration File

NLSR's configuration file contains numerous parameters to control the behavior and performance of NLSR. The configuration file also includes the trust schema used by NLSR to verify LSA Data, Hello Data, and prefix update command Interests.

13.1 Naming Conventions

The NLSR naming convention is mostly arbitrary. For example, a router's name, maybe `%C1.Router/router1`, is composed of two parts. `%C1.Router` is the router "tag", and `router1` is the name or label of the router. Different entities are given different tags, depending on the context.

13.2 General Section

The **general** section in the configuration file includes parameters which deal with the general setup of the router, the behavior of the LSDB, and logging configuration.

There are three parameters used to configure the router prefix of the router. The router prefix is the name that other routers in the network know this router by.

- **network** - the name of the network to which the router belongs; e.g., `/ndn`.
- **site** - the name of the site to which the router belongs; e.g., `/edu/memphis`.
- **router** - the name to identify the router; e.g., `/%C1.Router/cs/pollux.t`

The router prefix is constructed by combining the three parameters following the format: `</network>/<site>/<router>`.

There are three parameters which affect the behavior of the LSDB.

- **lsa-refresh-time** - the time in seconds the router will wait before refreshing its LSAs (Default value: 1800; Valid values: 240 - 7200).
- **router-dead-interval** - the time in seconds after which an inactive router's LSAs are removed. The configured value for this parameter must be greater than **lsa-refresh-time**. (Default value: two times the value configured in **lsa-refresh-time**).
- **lsa-interest-lifetime** - the interest lifetime used for LSA Interests (Default value: 4; Valid values: 1 - 60).

The **log-level** parameter configures the verbosity of NLSR's logging. The possible **log-level** values are listed in increasing verbosity. That is, the value all the way to the left includes the values all the way to the right.

NONE < ERROR < WARN < INFO (Default) < DEBUG < TRACE < ALL

Note that a log level also enables all log levels to its left. That is, setting log level to **TRACE** causes **ERROR**, **WARN**, **INFO** and **DEBUG** messages. to also be logged.

The **general** configuration section also includes parameters to choose where the NLSR log file and the NLSR sequence number file are stored. The **log-dir** parameter is an absolute path to the directory where the NLSR log file should be written, and **seq-dir** is an absolute path to the directory where the NLSR sequence number should be written. The log directory must exist and be writable, or else NLSR will fail to start.

13.3 Neighbors Section

The **neighbors** section in the configuration file contains parameters that define the behavior of the Hello Protocol and the neighboring routers of the router.

- **hello-retries** - the number of times to retry a Hello Interest before deciding the neighbor is down (Default value: 3; Valid values: 1 - 10).
- **hello-timeout** - the interest lifetime for Hello Interests in seconds (Default value: 1, Valid values: 1 - 15).
- **hello-interval** - the time in seconds between sending each Hello Interest to a neighbor. (Default value: 60; Valid values: 30 - 90).
- **adj-lsa-build-interval** - when the Hello Protocol triggers an Adjacency LSA build, the LSDB will wait this amount of time in seconds before performing the Adjacency LSA build. This parameter is intended to allow for Adjacency LSA build requests to be aggregated and the build can then be performed once. (Default value: 5; Valid values: 0 - 5)

- **first-hello-interval** - the time to wait in seconds before sending the first Hello Interests (Default value: 10; Valid values: 0 - 10).
- **face-dataset-fetch-tries** - how many times to re-fetch the Face status dataset from NFD.
- **face-dataset-fetch-interval** - how often to fetch the Face status dataset from NFD. This dataset is needed to communicate with neighbors, but Face events are considered the main source of this information. This dataset is intended as a backup to that mechanism, so the interval is very long.

The **neighbors** section also includes multiple **neighbor** subsections, each of which configures a neighbor of the router. The **neighbor** subsection includes:

- **name** - the router prefix of the neighboring router
- **face-uri** - the face that should be used to connect to the neighboring router
- **link-cost** - the cost metric for the link connecting this router to the neighbor router.

NB: NLSR no longer creates and configures Faces in NFD! NLSR periodically fetches from NFD a dataset containing information about all of NFD's Faces. NLSR scans this information and internally configures neighbors with that information. Additionally, NLSR listens to activity from NFD concerning Faces. These two sources of information should cover all Faces; any neighbor not represented in these two sources of information will be assumed inactive or inaccessible, and will not be contacted.

13.4 Hyperbolic Section

Hyperbolic Routing is a greedy geometric routing technique available in NLSR. The best resource to understand how it works is its white paper. [?]

The **hyperbolic** section in the configuration file is used to enable/disable hyperbolic routing and to specify the hyperbolic coordinates of the router.

The **state** parameter indicates whether or not hyperbolic routing should be enabled. There are three possible values for this parameter: **on**, **off**, and **dry-run**. **on** enables hyperbolic routing; **off** disables hyperbolic routing (link-state routing is used); **dry-run** uses link-state routing to populate NFD's FIB, but will also perform the hyperbolic routing calculations and write them to the log file for debugging purposes.

The **radius** parameter defines the router's radius in the hyperbolic coordinate system and **angle** defines the router's angle(s) in the hyperbolic coordinate system. There can be (d-1) angular coordinates in d-dimensional hyperbolic routing. Currently the testbed uses 2-dimensional hyperbolic routing with one radial and one angular coordinate.

13.5 FIB Section

The **fib** section in the configuration file contains two parameters: one to limit the number of next hops registered for each name prefix, and the amount of time to wait before calculating the routing table after a request is made.

max-faces-per-prefix defines the maximum number of next hops that can be registered for a name prefix. This value is intended to reduce the FIB size for routers with a large number of neighbors. The default value for **max-faces-per-prefix** is 0 which indicates that all available next hops may be added to each name prefix. **max-faces-per-prefix** allows values between 0 and 60.

routing-calc-interval is the time to wait in seconds after a routing table calculation is requested before actually performing the routing table calculation. This parameter is intended to limit the number of routing table calculations, which may be performance intensive on some systems. The default value for **routing-calc-interval** is 15 seconds and can be configured to be in the range of 0 to 15 seconds.

13.6 Advertising Section

The **advertising** section includes a list of name prefixes that the router should advertise as reachable through itself. Each name prefix that should be advertised should be in the following format: **prefix /name/to/advertise**. This section allows for static configuration of the advertised prefixes, but prefixes can be dynamically advertised and withdrawn using the Prefix Update Processor.

13.7 Security Section

The **security** section of the configuration file includes the configuration for NLSR's validators and the locations of certificates that should be published by the router.

The **validator** subsection includes the configuration for the validator used by NLSR to verify the signatures of Hello Data and LSA Data.

The **prefix-update-validator** configures the validator used by the Prefix Update Processor to verify that prefix update command Interests are signed by the operator of the router.

The **security** section also allows configuration of which certificates should be published by the router using the **cert-to-publish** keyword. If the router should publish a certificate, the absolute path for the certificate file can be configured as **cert-to-publish** value.

References

- [1] Z. Zhu and A. Afanasyev, “Let’s ChronoSync: Decentralized dataset state synchronization in Named Data Networking,” in *Proceedings of IEEE ICNP*, 2013.
- [2] NDN Project Team, “NFD - NDN forwarding daemon,” <http://named-data.net/doc/nfd/>.
- [3] V. Lehman, A. M. Hoque, Y. Yu, L. Wang, B. Zhang, and L. Zhang, “A secure link state routing protocol for NDN.”
- [4] NDN Project Team, “ndnsec,” <https://github.com/named-data/ndn-cxx/tree/master/tools/ndnsec>.
- [5] —, “ndn-cxx,” <http://named-data.net/doc/ndn-cxx/>.
- [6] V. Lehman, A. Gawande, R. Aldecoa, D. Krioukov, L. Wang, B. Zhang, and L. Zhang, “An Experimental Investigation of Hyperbolic Routing with a Smart Forwarding Plane in NDN,” <https://named-data.net/wp-content/uploads/2016/07/ndn-0042-1-asf.pdf>, 2016.