NDN Memo: Automatic Prefix Propagation

Yanbiao Li[1], Alexander Afanasyev[2], Junxiao Shi[3], Beichuan Zhang[3], Lan Wang[4], and Lixia Zhang[2]

[1]Hunan University, lybmath_cs@hnu.edu.cn

[2]University of California, Los Angeles, [aa,lixia]@cs.ucla.edu

[3]The University of Arizona, [shijunxiao,bzhang]@email.arizona.edu

[4]The University of Memphis, lanwang@memphis.edu

Revision history

- **Revision 1 (September 23, 2015)**: Initial revision

ABSTRACT

When the producer application in Named Data Networking (NDN) architecture wants to make the data available for fetching, it registers data's prefix with the NDN forwarder on the same host machine as a producer application. Effectively, this registration provides a new option to the local forwarder where to forward interests for data, if they cannot be satisfied otherwise. To provide similar option to the remote NDN forwarders, they need to get configured manually, use dynamic routing protocols, or rely on opportunistic data discovery. This memo introduces a new protocol, *Automatic Prefix Propagation*, providing an optimal tradeoff between implementation complexity, deployment complexity, and overheads in environments where the host machine of the producer application is connected to the rest of NDN network via one or multiple gateway NDN forwarders. The automatic prefix propagation protocol uses the local security configuration (configuration of NDN certificates) and the inferred presence of gateway routers to automatically send (potentially aggregated) remote registration requests when prefix is registered locally and maintain this prefix while it is still present in the local routing information base (RIB).

1. INTRODUCTION

Producer applications in the Named Data Networking (NDN) architecture [14] express the intent to make data available for retrieval by registering data's prefix with the NDN forwarder(s)[10]. In the current management NDN API, this intent takes the form of a command interest [1] that is directed towards the routing information base (RIB) manager running in the instance of the NDN forwarder running on the same machine as the producer (*local forwarder*) [10]. If the request is authorized, the RIB manager creates necessary entries in the forwarding information base (FIB) in the local forwarder, allowing it to properly direct interests with the registered prefix towards the producer application, if interests cannot be satisfied by other means.

Directing interests from the remote NDN forwarders is a more complex issue, which is being addressed in a number of ways: *manual configuration*, *dynamic name announcement protocols*, and *opportunistic data discovery strategies*. Different methods have different tradeoff between the implementation complexity, usage complexity, and potential overhead in terms of unnecessary forwarded interests. Manually configuring remote forwarders (e.g., using the `nfdc` command) has trivial implementation cost and, when properly used, can result in minimal overhead. However, it is the most complex and tedious when comes to the usage complexity, unless a forwarding strategy relies on a pre-defined naming conventions to forward interests in a greedy fashion (e.g., using geo or hyperbolic coordinates embedded in the data names [12, 11]). The dynamic name announcements, e.g., using a routing protocol such as NLSR [11], minimizes the usage complexity and overhead, but has significant implementation and deployment costs. Opportunistic data discovery, e.g., using the AccessRouter strategy or the NCC strategy with automatically maintained FIB using the `nfd-autoreg` daemon program, is simple to use, but has relatively high implementation complexity and network overhead.

This memo introduces a new dynamic name announcement protocol, the *Automatic Prefix Propagation*, which minimizes the complexity and overheads in environments with one or more remote NDN forwarders acting as NDN gateways.[1] The protocol automatically triggers the remote prefix registration when an application registeres prefix locally, provided that (1) there is an active remote NDN gateway, and (2) the forwarder possesses a private key and the corresponding NDN certificate that matches the locally registered namespace. The

[1]The current version of the protocol assumes a single NDN gateway. In the future, the protocol will be extended to multi-gateway environments.

first condition is satisfied when "`/localhop/nfd`" prefix is present in the local RIB, e.g., configured by NDN auto-configuration [4]. The second requirement is fulfilled by the user requesting and installing NDN certificate that covers the desired namespace in the local forwarder. Coverage of the NDN certificate is determined using a basic trust schema [13], described in Section 3. Note that depending on to which namespace the user-installed NDN certificate corresponds, the remotely registered prefix can be shorter than the one registered by the application. This allows the forwarder to aggregate multiple local registrations into one remote action, reducing communication and bookkeeping overheads.

The rest of this memo is organized as follows. Section 2 demonstrates our motivation by an example. Section 3 reviews key design issues and proposes our solutions. The reference implementations are presented in Section 4, and Section 5 introduces a brief instruction of the proposed Automatic Prefix Propagation feature. Section 6 concludes the whole paper.

2. MOTIVATION EXAMPLES

Figure 1 shows two types of remote prefix registration and their corresponding forwarding processes. As shown in the left part, the tool *ndn-autoreg* is adopted in the router B, to enable an entry with a specified prefix (i.e., */ndn/ucla*) registered to its RIB when a connection from any end host is established. In this case, when an Interest toward the *app1* that runs in the host k arrives at the router B, all its connected hosts will then receive this Interest. In this process, the forwarding cost is much larger than really required. By contrast, the right part of this figure demonstrates a more ideal situation, where unnecessary forwarding costs are eliminated since the router B has learned enough knowledge from the host k, through the process called automatic prefix propagation.

3. DESIGN ISSUES AND OUR SOLUTIONS

In this section, we review a series of key design issues to facilitate automatic prefix propagation, and propose our solutions for each.

3.1 What Prefix to Propagate

The desirable propagation is to register prefixes carrying knowledges of local RIB entreis to the connected router, such that it will be aware of where to forward Interests under those prefixes. Here comes the first problem: **What prefix should be registered to the connected router by propagation?**

Given a local RIB entry, simply propagating its prefix will work. But it's unwise due to the risk that the size of remote RIB[2] will increase sharply, especially when there is a large number of hosts connected or there is a large number of applications run on some hosts. To this point, we seek for reasonable prefix aggregation to reduce not only the cost of propagation but also increments of the remote RIB.

But it's hard to determine a proper stable aggregation since you never know what's the next RIB entry. Fortunately, NDN's global hierarchical naming mechanism implicitly establishes connectivities between different parts of the whole architecture, such as security, forwarding and routing, etc. Actually, the namespace defined by a signing identity deos cover all prefixes (including sub-identities or application prefixes) it can sign, which enables us shift our focus from RIB entry prefixes to identities in local key-chain[3].

Thus, propagating an identity, that can represent both existing and potential RIB entries, will be a reasonable aggregation. Given a local RIB entry, we propose to propagate the identity in the local key-chain that can sign it, which is also used to sign registration commands for this propagation[4]. Further, if there are two or more satisfied identities, we select the shortest one to go deeper in aggregation. Figure 2 shows an example of such key-chain-involved propagation.

Finally, to complete registration commands for propagation, there are some other parameters should be carefully defined. Firstly, the route **Origin** should be specified as CLIENT and the **FaceId** should be set to 0 so that the incoming face will be selected when finalizing the registration on the remote RIB. Besides, there are two optional parameters, the route **Cost** and the command **Timeout**, that can be configured by users. Last but most important, the prefix registration **Flag** [9] must take the default setting (i.e., **CHILD_INHERIT** flag will take effect) no matter which flags are used in the corresponding local RIB entreis. Because the registered prefix, after aggregation, may be shorter than the actual one, while another flag (i.e., **CAPTURE**) blocks routes with shorter prefixes be used.

3.2 When to Propagate

In section 3.1, we have proposed the mechanism to determine a prefix to propagate for a given entry in the local RIB. The next question is **When should the propagation (i.e., register the selected prefix to the remote RIB) be performed?** Our general answer is the propagation should be performed automatically once the local RIB entry is inserted. Correspondingly, when a RIB entry is erased, it may be required to revoke the performed propagation (i.e., unregister the

[2]remote RIB representes the RIB on the connected router.

[3]local key-chain represents the key-chain owned by the local NFD RIB Daemon (NRD) [6].

[4]If the identity is scoped for NRD use (i.e., the last name component is 'nrd', its prefix with the last component eliminated will be registered while the whole identity is used to sign registration commands.)

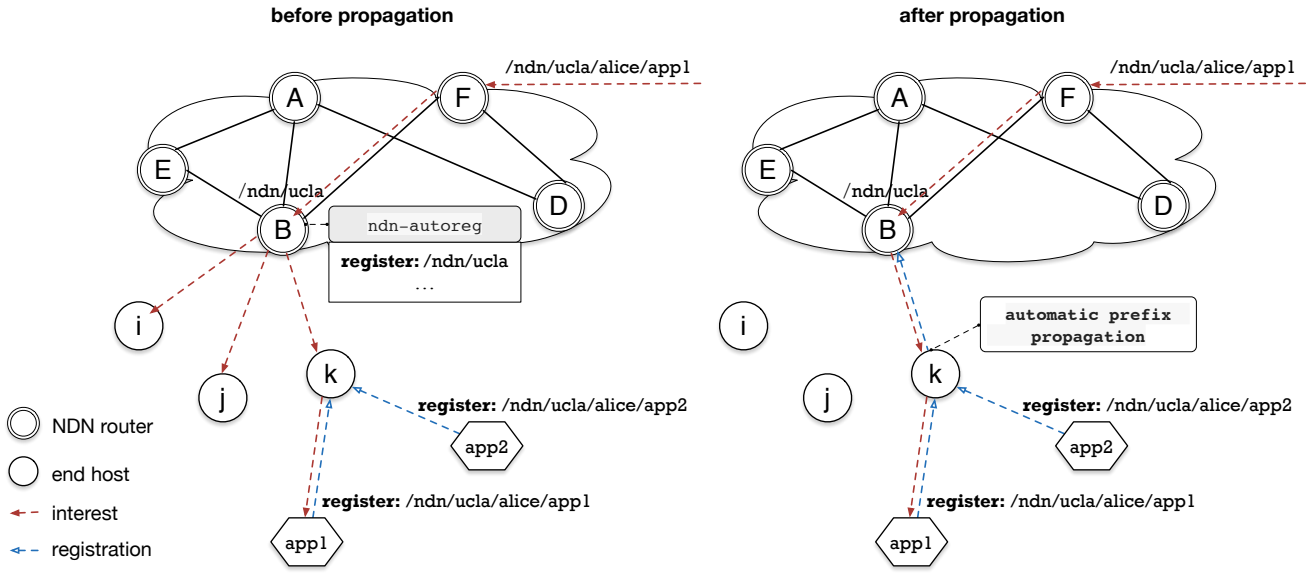

Figure 1: Motivation of prefix propagation

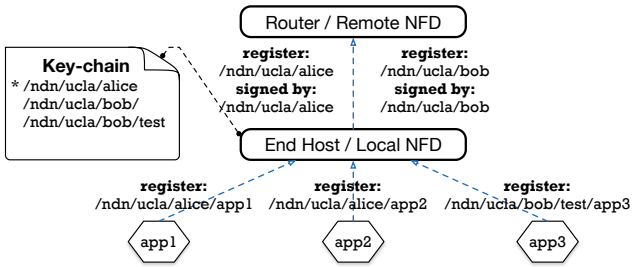

Figure 2: An example of prefix propagation.

registered prefix from the remote RIB).

But not all RIB entries will trigger propagations / revocations, e.g., entries [5] start with the NFD management prefix (i.e., */localhost/nfd*), and the entry represents the connectivity to the router (this will be discussed in detail later). Besides, due to prefix aggregation, the propagated entry may represent one or more local RIB entries. So, on one hand, the intended propagation should be abandoned if it has already been performed in response to a previous local RIB insertion. On the other hand, a propagated entry must be kept until all represented RIB entries are erased.

If a propagation / revocation is proved to be necessary, the corresponding propagated entry will be created / released. But there is some chances that no remote operations, i.e. register / unregister the propagated prefix to / from the remote RIB, will be set up immediately. For revocation, no remote operation will be triggered unless the revoking propagation has succeeded. Certainly, if there is no connectivity to the router, no remote operations should be set up immediately, but some of them will be pending (see section 3.7).

To distinguish whether there is a connectivity to the router, we define a **link local nfd prefix** as */localhop/nfd*. When a connectivity between local NFD and remote NFD is established[6], the **link local nfd prefix** will be registered to the local RIB. Correspondingly, such an entry will be erased if the connectivity is lost. Thereby, we can easily check the connectivity to the router by inspecting the **link local nfd prefix** in the local RIB. Besides, as mentioned above, the insertion / deletion of the **link local nfd prefix** will not lead to propagation / revocation, but can trigger other actions (see section 3.7 for detail).

3.3 Who are involved in Propagation

There are three logic entities involved in propagation, the applications run on end hosts, the local NFD and the remote NFD. Generally, the applications (i.e., producers) publish their Datas and listen to the corresponding Interests by registering the names of Datas to the local RIB via the local NFD. Then, the local NFD propagates an aggregated prefix from those Data names to the remote RIB via the channel between local NFD and remote NFD. So as the revocation.

3.4 How to Secure Propagations

To ensure propagations are reliable, a proper trust schema should be performed to secure propagations.

[5] these entires are registered for NFD mangement modules to receive NFD ControlCommands [3] and thus are restricted for local use only.

[6] How to establish such a connectivity is beyond this memos's scope and can be found otherwhere [4].

More specifically, the remote NFD will abandon the received registration commands for propagations, until the commands can pass the validation defined by the trust schema. The key issue is **how to enable the router verify a command generated and signed on a end host?** According to the NDN trust schema [13], the short answer would be having the command recursively signed by one of the trust anchor of the router.

More specifically, there should be some data base, on which a series of certificates are published. This data base may or may not reside in the end host according to the adopted management protocol[7] of certificates. To ensure a registration command for propagation be verified on the router, there are three basic requirements: 1) the certificate of the signing key of this command must be recursively signed by one of the router's trust anchor; 2) all certificates along such signing path have already published on the data base of certificates; 3) the router can access the data base to fetch required certificates. Besides, a series of formate rules can be configured on the router to help validate the command Interest as well as related certificate Datas.

Figure 3 shows an overview of such a trust schema. When *app3* register a prefix to the local NFD, the prefix */ndn/ucla/bob* is propagated to the remote NFD. While the registration command is signed by the identity */ndn/ucla/bob*, which is recursively signed by the identity */ndn* and the certificates of keys along the signing path have been published on the data base **DB**. Thus, the registration command can be verified on the router, as long as the end host can publish the certificate of */ndn/ucla/bob* on **DB**, */ndn* is set as one trust anchor on the router and it can access **DB** to fetch the certificates of */ndn/ucla/bob* and */ndn/ucla*. The corresponding validation process is simply demonstrated on figure 4.

3.5 How to Maintain Propagated Entries

This issue can be divided into two parts: **how to maintain propagated entries on the router?** and **how to maintain the propagated entries on the end host?**.

On the router, although the propagated entries have been registered to the RIB, they, unlike other RIB entries, are highly related to the status of the propagator (i.e., the end host). So, we propose to maintain the propagated entries as soft states, such that they will expire automatically after a some duration. Consequently, all propagated entries must be refreshed periodically to keep them work.

While on the end host, the situation is more complicated. As discussed in section 3.2, some propagations will not be performed if there is no connectivity to the router. However, there indeed are needs to propagate those prefixes, and they should be propagated immediately once the connectivity is activated. To this point, we just 'suspend' those propagations by maintaining the corresponding entries as well, and 'awake' them as long as there is a available connectivity to the router.

On the other hand, when a required revocation can not be performed due to connectivity problem, we will still release the corresponding entry. With the propagated entry released, the refresh process of this propagation will also stop. Consequently, the registration on the router's RIB will then expire after some duration.

Besides, maintenance of propagated entries is also useful to handler failures and deal with connectivity changes (see next sections for more detail). Actually, we design a state machine to describe different statuses of a propagated entry, and when and how to transit between them. Figure 5 shows the core parts[8] of the state machine in a simple and intuitive way. While the detail design and implementation of such a state machine will be introduced later.

3.6 How to Handle Failures

To this issue, we adopt an exponential back-off retransmission strategy. When the propagation fails, i.e., the corresponding registration gets a faillure from the router, it will be retried after a duration doubled each retry until reaches the maximum waiting period. Once a retry succeeds, the next retry waiting period will then be set back to the initial value. Both the initial and maximum waiting periods are configurable to users.

3.7 How to Deal With Connectivity Changes

No matter the connectivity (between the end host to a router) is lost, established or recovered, there are in total two atomic connectivity changes are involved: 1) connects to a router and 2) disconnects from a router. As mentioned in section 3.2, the **link local nfd prefix** will work as a 'alarm' of those connectivity changes.

On one hand, when the **link local nfd prefix** is inserted into the RIB, it means there is a connectivity to a router established or recovered. Then, we perform all 'suspend' propagations, represented by the maintained propagated entries, via the current connectivity.

On the other hand, the deletion of the **link local nfd prefix** from the RIB indicates current connectivity to the router was just lost. Then, until the lost connectivity is recovered or a new connectivity is established, it's unnecessary to perform any propagations nor revocations. Thus, any attempt for refresh or retry should be abandoned. Moreover, newly triggered necessary prop-

[7]Some existing tools can be used to manage the certificates in our case, such as **ndn-pib** [7], **ndns** [5] and **ndn-repo-ng** [8].

[8]Invalid transitions, valid transitions but without neither state switch nor action triggered, and the actions under state switches are not presented.

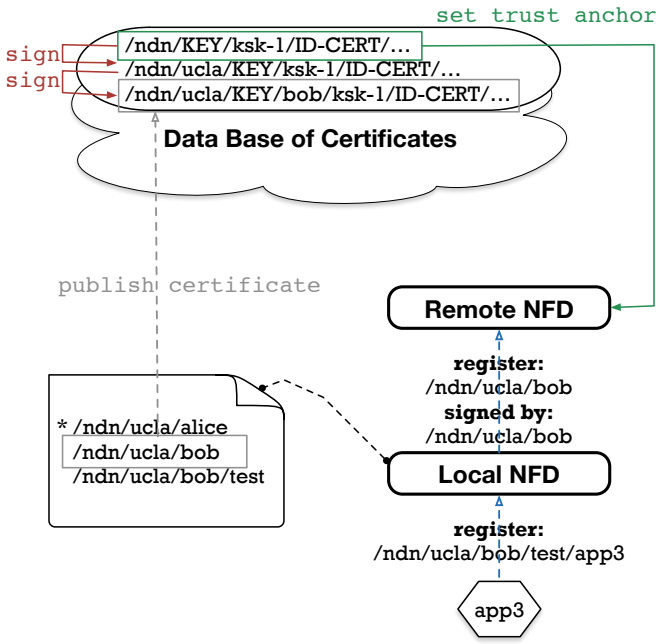

Figure 3: trust schema.

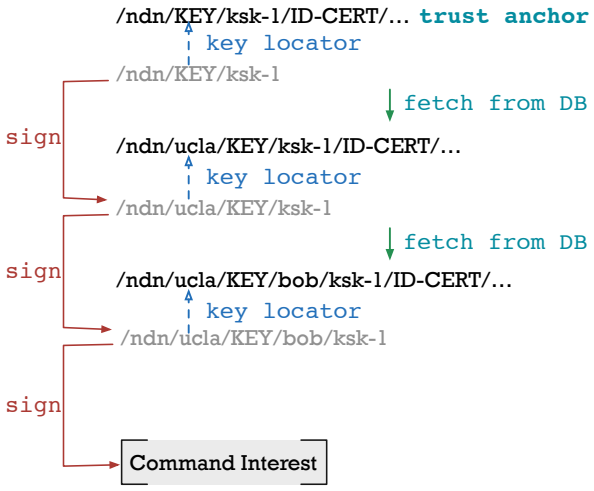

Figure 4: verification process.

agations will only lead to creations of propagated entries (but not any remote operations). So as newly triggered revocations, which will only result in some propagated entries' being released.

3.8 How to Response to KeyChain changes

As discussed in section 3.1, the propagated prefixes are highly related to the local key-chain owned by NRD. Once the key-chain changes, some propagations will be affected. Actually, the essential impact is previous selected prefixes for propagations may not be the best any more.

On one hand, for some propagated prefix, there may exist a better choice after a shorter identity is inserted to the key-chain. In this case, we will keep the propagated prefix the current best choice. On the other hand, the identity for some propagated prefix may be erased from the key-chain. Consequently, the corresponding propagation should be revoked, and its represented local RIB entries should be re-performed to set up some new propagations.

To prevent establishing a too close connection to the security module, we propose to response to those key-chain changes asynchronously. Namely, key-chain changes will not directly affect propagated entries immediately, and thus are not described in the propagated-entry state machine. Instead, whenever a propagation is about to be re-performed due to periodical refresh or retry, or as is "awaken" after connectivity establishment, the propagated prefix is checked against the key-chain to see whether it still works, i.e., is not only available but also the current best choice. If it is, re-perform the propagation as expected. Otherwise, response to the key-chain changes at this point as mentioned above.

4. REFERENCE IMPLEMENTATIONS

We implement the automatic prefix propagation feature as a separate module, **AutoPrefixPropagator**, of **NRD**. As depicted in figure 6, it works in cooperations with other two modules, **Rib** and **RibManager**, of **NRD**, and two NRDs, run on the end host and the router respectively, are involved to complete the propagation process. So as the revocation.

Then, we will review the implementation in detail with three logic parts: 1) public interfaces, 2) helpers and 3) propagated-entry state machine.

4.1 Public Interfaces

4.1.1 Constructor

```
AutoPrefixPropagator(ndn::nfd::Controller& controller,
                     ndn::KeyChain& keyChain,
                     Rib& rib)
```

When creating an instance of **AutoPrefixPropagator**, three arguments are required: 1) a reference to the NFD controller that used to send out commands for propagation / revocation, 2) a reference to the key-chain owned by the NRD that supplies identities, and 3) a reference to the managed RIB that maintains two important signals, *Rib::afterInsertEntry* and *Rib::afterEraseEntry*, will trigger propagations and revocations respectively.

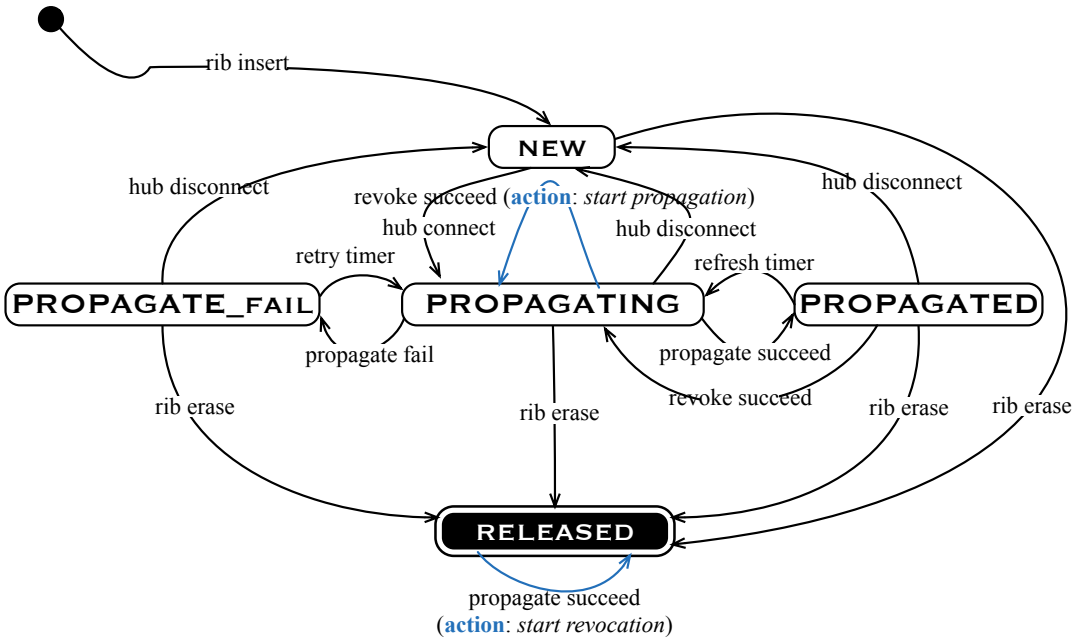

Figure 5: Overview of the propagated entry state machine

4.1.2 *loadConfig*

```
void
loadConfig(const ConfigSection& configSection)
```

This method is invoked when the **RibManager** is loading configurations from a specified NFD config file [2] at the *rib* section. While all propagation-related parameters are configured at the *rib.auto_prefix_propagate* subsection, which is loaded, parsed and then passed by to this method.

Table 1 presents the common parameters shared by all propagations, some of which are configurable and are actually set or modified in this method when loading configurations.

4.1.3 *enable*

```
void
enable()
```

This method is invoked by the **RibManager** to enable the automatic prefix propagation feature.

To achieve this, the method *afterInsertRibEntry*[9] is connected to the signal *Rib::afterInsertEntry*, which is emitted after an RIB entry was inserted. Correspondingly, another method *afterEraseRibEntry* is connected to *Rib::afterEraseEntry*, which is emitted after an existing RIB entry was erased.

4.1.4 *disable*

```
void
disable()
```

In contrast with *enable*, those two signal connections are released to disable the feature of automatic prefix propagation.

4.2 Propagation Helpers

4.2.1 *getPrefixPropagationParameters*

```
PrefixPropagationParameters
getPrefixPropagationParameters(const Name& localRibPrefix)
```

The return value, **PrefixPropagationParameters**, is a structure that could be used by the registration commands for prefix propagations / revocations. It consists an instance of **ControlParameters** (*parameters*) and an instance of **CommandOptions** (*options*), as well as a bool variable (*isValid*) indicates whether this set of parameters is valid (i.e., the signing identity exists).

This method is invoked to get the required parameters for prefix propagation. given a local RIB prefix *localRibPrefix*, find out, in local key-chain, a proper identity whose namespace covers the input prefix. If there is no desired identity, return a invalid **PrefixPropagationParameters**. Otherwise, set the selected identity as the signing identity of *options*. Meanwhile, set this identity (or its prefix with the last component eliminated if it ends with 'nrd') as the name of *parameters*.

4.2.2 *doesCurrentPropagatedPrefixWork*

[9] All methods of **AutoPrefixPropagator** are referred to without specifying the namespace.

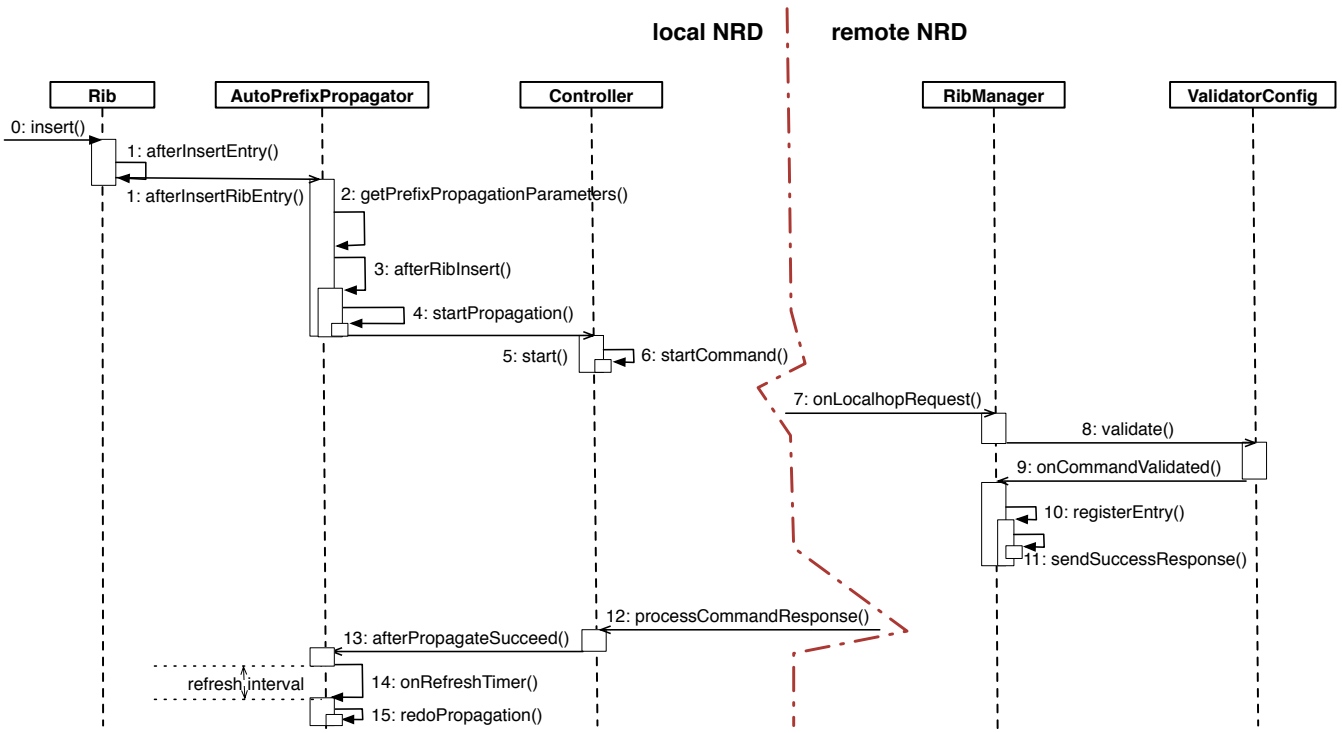

Figure 6: Simplified work flow of successful propagation

Table 1: Shared parameteres for prefix propagation.

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

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

```
bool
doesCurrentPropagatedPrefixWork(const Name& prefix)
```

This method is invoked before re-perform some propagation to check whether current propagated prefix still works. A propagated prefix still works if and only if the identity corresponding to this prefix still exists in the key-chain, and there is no shorter identity that covers this prefix. Otherwise, either current siging identity can not be found, or a better choice exists and should be adopted then.

4.2.3 *redoPropagation*

```
void
redoPropagation(PropagatedEntryIt entryIt,
                const ControlParameters& parameters,
                const CommandOptions& options,
                time::seconds retryWaitTime)
```

This method is invoked to refresh a successful propagation, to retry a failed propagation, or to awake a suspend propagation. No matter for which purpose, the propagation for the input propagated entry is performed with the input **ControlParameters**, **CommandOptions** and current setting of waiting period before retry,

as long as its propagated prefix still works (determined by *doesCurrentPropagatedPrefixWork*). Otherwise, the input propagated entry will be erased, while all local RIB entries represented by it will be re-handled to set up required propagations.

4.2.4 startPropagation

```
void
startPropagation(const ControlParameters& parameters,
                 const CommandOptions& options,
                 time::seconds retryWaitTime)
```

This method is invoked to send out the registration command for a propagation with input parameters. Two callbacks, *afterPropagateSucceed* and *afterPropagateFail*, are assigned for the case when the propagation succeeds, and the case when the propagation fails respectively.

Before sending out the command, two events, for refresh and retry respectively, are created (but not scheduled at this point) and passed by as arguments to those two callbacks, *afterPropagateSucceed* and *afterPropagateFail* respectively.

The retry event requires an argument to define the waiting period before next retry, which is calculated according to the back-off policy based on current waiting period (*retryWaitTime*) and *m_maxRetryWait*.

4.2.5 startRevocation

```
void
startRevocation(const ControlParameters& parameters,
                const CommandOptions& options,
                time::seconds retryWaitTime)
```

This method is invoked to send out the unregistration command for revoking a propagation with input parameters. Two callbacks, *afterRevokeSucceed* and *afterRevokeFail*, are assigned for the case when the revocation succeeds, and the case when the revocation fails respectively.

4.2.6 afterInsertRibEntry

```
void
afterInsertRibEntry(const Name& prefix)
```

This method is invoked once the signal *Rib::afterInsertEntry* is emitted. As discussed in section 3.2, not all rib entry insertions will lead to necessary propagations. Actually, the process is abandoned if the local RIB entry whose insertions trigger this invoking is scoped for local use only (i.e., the name of that RIB entry,*prefix*, starts with */localhost*). On the other hand, if *prefix* is just the **link local nfd prefix**, i.e., this invoking is triggered by establishing a connectivity to the router, *afterHubConnect* is invoked to awake all suspended propagations.

Even though *prefix* represents a RIB entry that can trigger propagation, the triggered propagation may be unnecessary, if there is no valid **PrefixPropagationParameters** found for this entry, or the propagation has already been processed.

Then, if there is a necessary propagation triggered, *afterRibInsert* is invoked to process this request further.

4.2.7 afterEraseRibEntry

```
void
afterEraseRibEntry(const Name& prefix)
```

Similar to *afterInsertRibEntry*, this method is invoked once the signal *Rib::afterEraseEntry* is emitted. If *prefix*, the name of the local RIB entry whose deletion triggers this invoking, is scoped for local use only, this invoking terminates immediately. While if this invoking if triggered by the deletion of the **link local nfd prefix**, i.e., the connectivity to the router was lost, *afterHubDisconnect* is invoked then to suspend all propagations by canceling their scheduled retry / refresh events. In all other cases, a revocation will be required.

However, if the there is no valid **PrefixPropagationParameters** found, the required revocation will not be performed. Besides, if the propagation to be revoked has not succeeded yet, or the corresponding propagated entry should be kept for other RIB entries, the required revocation will also be abandoned.

At last, if a revocation is really required to be performed, *afterRibErase* is invoked then.

4.3 Propagated-entry State Machine

Before go deep into the state machine, we introduce the propagated entry first. A propagated entry consists of a **PropagationStatus** indicates current state of this entry, as well as an event could be scheduled for either refresh or retry. Besides, it stores a copy of the signing identity for this entry. All propagated entries are maintained as a un-ordered map (*m_propagatedEntries*), where the propagated prefix is used as the key to retrive the corresponding entry.

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

- *NEW*, the initial state.
- *PROPAGATING*, the state when the corresponding propagation is being processed but the response is not back yet.
- *PROPAGATED*, the state when the corresponding propagation has succeeded.
- *PROPAGATE_FAIL*, the state when the corresponding propagation has failed.
- *RELEASED*, indicates this entry has been released. It's noteworthy that this state is not required to be explicitly implemented, because it can be easily determined by checking whether an existing entry

can still be accessed. Thus, any entry to be released is directly erased from the list of propagated entries.

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

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

A state machine is implemented to maintain and direct transitions according to the input events. Figure 7 lists all related events and the corresponding transitions.

Then, we will review all corresponding methods.

4.3.1 afterRibInsert

```
void
afterRibInsert(const ControlParameters& parameters,
               const CommandOptions& options)
```

Once this event happens, a new propagated entry will be created, i.e., from *RELEASED* to *NEW*. If there is a active connectivity to the router, *startPropagation* is invoked to perform this propagation, and the state of the created entry is further switched to *PROPAGATING*. Otherwise, this propagation will be suspended. The retry waiting period is set to *m_baseRetryWait*.

4.3.2 afterRibErase

```
void
afterRibErase(const ControlParameters& parameters,
              const CommandOptions& options)
```

Once this event happens, the propagated entry will be erased, i.e., switched to *RELEASED*. Besides, if there is a active connectivity and the corresponding propagated entry has succeeded, *startRevocation* is invoked to revoke this propagation. Since the **Cost** filed of *parameters* is default set, it should be unset to adapt to the unregistration command.

4.3.3 afterHubConnect

```
void
afterHubConnect()
```

Once this event happens, all suspended propagations are awaken by invoking *redoPropagation* for each of them.

4.3.4 afterHubDisconnect

```
void
afterHubDisconnect()
```

Once this event happens, all propagations are suspended by initializing their corresponding propagated entries.

4.3.5 afterPropagateSucceed

```
void
afterPropagateSucceed(const ControlParameters& parameters,
                      const CommandOptions& options,
                      const Scheduler::Event& refreshEvent)
```

When this event happens, beside the *PROPAGATING* sate, the propagated entry may also be in the *RELEASED* state, because this entry may be erased due to revocation before the result of this propagation gets back.

If the propagated entry does not exist (i.e., in the *RELEASED* state), an unregistration command is sent immediately to revoke this propagation. Actually, a copy of *parameters* is made and passed by to *startRevocation* with the **Cost** filed unset.

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

Figure 7: The transition table of propagated-entry state machine.

While if the state of the propagated entry is *PROPAGATING*, it should be switched to *PROPAGATED*, and the refresh event *refreshEvent* is scheduled to redo this propagation after a duration (*m_refreshInterval*).

4.3.6 *afterPropagateFail*

```
void
afterPropagateFail(uint32_t code,
                   const std::string& reason,
                   const ControlParameters& parameters,
                   const CommandOptions& options,
                   time::seconds retryWaitTime,
                   const Scheduler::Event& retryEvent)
```

When this event happens, beside the *RELEASED* state, the propagated entry may also be in the *PROPAGATING* state, because the same prefix may be propagated by other local RIB entry before the result of this revocation gets back.

If the propagated entry still exists, its state is switched to *PROPAGATE_FAIL*, and the retry event *retryEvent* is scheduled to redo this propagation after a duration defined by current waiting period for retry *retryWaitTime*.

4.3.7 *afterRevokeSucceed*

```
void
afterRevokeSucceed(const ControlParameters& parameters,
                   const CommandOptions& options,
                   time::seconds retryWaitTime,
                  )
```

When this event happens, beside the *RELEASED* state, the propagated entry may also be in any states expect the *NEW* state. Because the same prefix may be propagated by other local RIB entry before the result of this revocation gets back (*PROPAGATING*), and the result of that propagation may be back earlier (*PROPAGATED* or *PROPAGATE_FAIL*).

If the propagated entry is in the *PROPAGATING* state or the *PROPAGATED* state, *startPropagation* will be invoked, since the propagation on the router has been revoked but it should be kept in this case. As *param-*

eters is used for revocation (i.e., the **Cost** filed is not set), a copy of it is made and passed by to *startPropagation* with the **Cost** set as that of *m_controlParameters*.

4.3.8 afterRevokeFail

```
void
afterRevokeFail(uint32_t code,
                const std::string& reason,
                const ControlParameters& parameters,
                const CommandOptions& options)
```

When this event happens, the propagated entry stay in whatever state it is in, and nothing need to do in this case.

4.3.9 onRefreshTimer

```
void
onRefreshTimer(const ControlParameters& parameters,
               const CommandOptions& options)
```

When this event happens, the propagated entry must be in the *PROPAGATED* state. The *redoPropagation* is invoked to handle this refresh request.

4.3.10 onRetryTimer

```
void
onRetryTimer(const ControlParameters& parameters,
             const CommandOptions& options,
             time::seconds retryWaitTime,
            )
```

When this event happens, the propagated entry must be in the *PROPAGATE_FAIL* state. The *redoPropagation* is invoked to handle this retry request.

5. A BRIEF INSTRUCTION

In this section, we present a brief instruction with a simple example.

5.1 test environment

Two virtual machines are set up. One works as the end host whose IP address is *host_ip*, while the other works as the router whose IP address is *router_ip*. Besides, we run **repo-ng** as a data base of certificates on the host.

5.2 configure the end host

Firstly, start NFD with the config file *host.nfd.conf*.

host.nfd.conf

```
rib
{
  auto_prefix_propagate
  {
    cost 15
    timeout 10000
    refresh_interval 300
    base_retry_wait 50
    max_retry_wait 3600
  }
}
```

Then, start the **repo-ng** with the config file *repo.conf*.

repo.conf

```
repo
{
  data
  {
    prefix /ndn
  }
  command
  {
    prefix ""
  }
  storage
  {
    method sqlite
    path .
    max-packets 100
  }
  tcp_bulk_insert
  {
    host localhost
    port 9527
  }
  validator
  {
    trust-anchor
    {
      type any
    }
  }
}
```

At last, create the key-chain shown in figure 4, and publish their certificates onto the **repo-ng**. Then, export the certificate of */ndn* on the host, transfer it to the router.

5.3 configure the router

Start NFD on the router with the config file *router.nfd.conf*.

Set the router's trust anchor to */ndn*. i.e., copy the file of certificate of */ndn* (received from the end host) to whatever configured at *rib.localhop_security.trust-anchor.filename*.

5.4 establish connectivity

On the router, run the following command to start the **autoreg-server** with prefix */ndn* automatically registered when a new on-demand Face is created, such that the router can forward all prefixes start with */ndn* to the host once the connectivity from the host to the router is established.

```
nfd-autoreg --prefix=/ndn
```

On the host, run the following command to get connectivity to the router.

router.nfd.conf

```
rib
{
  localhop_security
  {
    rule
    {
      id "NRD Prefix Registration Command Rule"
      for interest
      filter
      {
        type name
        regex ^[<localhop><localhost>]<nfd><rib>[<register><unregister>]<>$
      }
      checker
      {
        type customized
        sig-type rsa-sha256
        key-locator
        {
          type name
          regex ^[^<KEY>]*<KEY><>*[<ksk-.*>]<ID-CERT>$
        }
      }
    }
    rule
    {
      id "NDN Testbed Hierarchy Rule"
      for data
      filter
      {
        type name
        regex ^[^<KEY>]*<KEY><>*[<ksk-.*>]<ID-CERT><>*$
      }
      checker
      {
        type customized
        sig-type rsa-sha256
        key-locator
        {
          type name
          regex ^[^<KEY>]*<KEY><>*[<ksk-.*>]<ID-CERT>$
        }
      }
    }
    trust-anchor
    {
      type file
      file-name anchor.cert
    }
  }
}
```

```
nfdc register ndn : /localhop/nfd udp4 : // < router_ip >: 6363
```

5.5 start test

On the router, inspect the RIB to ensure there is no entry with the name */ndn*.

On the host, run the following command to register a prefix /sample/ndn/ucla/bob/test to the NFD.

```
ndnpingserver /ndn/ucla/bob/test
```

Then, wait for a few seconds and inspect the RIB on the router. There should be a prefix */ndn* propagated from the end host.

6. CONCLUSION

In this memo, we propose the Automatic Prefix Propagation, that enables the end host spread some knowledge of local prefix registrations to the connected router automatically. The propagated prefix is selected with reasonable aggregation, while all propagated entries are maintained according to a specified state machine, handling propagation failures and connectivity changes efficiently.

7. REFERENCES

[1] Command Interests. http://redmine.named-data.net/projects/nfd/wiki/Command_Interests.

[2] Config file format. http://redmine.named-data.net/projects/nfd/wiki/ConfigFileFormat.

[3] Control Command. http://redmine.named-data.net/projects/nfd/wiki/ControlCommand.

[4] ndn-autoconfig. http://named-data.net/doc/NFD/current/manpages/ndn-autoconfig.html.

[5] ndns. https://github.com/named-data/ndns.

[6] NRD (NFD RIB Daemon). http://named-data.net/doc/NFD/current/manpages/nrd.html.

[7] Public key Info Base (PIB) Service. http://redmine.named-data.net/projects/ndn-cxx/wiki/PublicKey_Info_Base.

[8] Repo protocol and repo-ng. http://redmine.named-data.net/projects/repo-ng/wiki.

[9] RIB Management. http://redmine.named-data.net/projects/nfd/wiki/RibMgmt.

[10] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto, et al. Nfd developerâĂŹs guide. Technical report, Technical Report NDN-0021, NDN, 2014.

[11] A. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang. Nlsr: named-data link state routing protocol. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, pages 15–20. ACM, 2013.

[12] F. Papadopoulos, D. Krioukov, M. Bogua, and A. Vahdat. Greedy forwarding in dynamic scale-free networks embedded in hyperbolic metric spaces. In *2010 Proceedings IEEE INFOCOM*.

[13] Y. Yu, A. Afanasyev, D. Clark, V. Jacobson, and L. Zhang. Schematizing and automating trust in named data networking.

[14] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, et al. Named data networking (ndn) project. *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, 2010.