

# Traceroute for Named Data Networking

Susmit Shannigrahi, Dan Massey, Christos Papadopoulos

**Abstract**—This works talks about Traceroute for Named Data Networking. We discuss the issues and trade-offs for such an implementation.

## 1 Introduction

Traceroute in IP network is a useful tool for finding out network problems. Similarly for NDN, Such an application can be helpful for diagnosing network problem. However, while providing similar functionalities, traceroute for NDN would considerably differ from traceroute in IP network. In this article, we try to define what traceroute for NDN is, what should be the semantics for such a system and discuss about one implementation. Here are some of the naming convention that we are going to use here:

- Trace interest: Interest packet for the purpose of tracing a path.
- Normal interest: Interest packet for the purpose of fetching a content.
- Identifier: An unique string for identifying NDN entities.
- Consumer: An NDN entity which expresses normal or trace interest and gets back reply.
- Node: An NDN entity which implements buffering and forwarding.

## 2 What is traceroute in NDN?

Traceroute in the context of NDN is not easy to define. Like IP traceroute which returns a path between two given hosts, NDN traceroute would still return path(s) between NDN entities. Obviously, at one end of traceroute would be the consumer. However, consumers does not care where the content is coming from. Therefore, there can be several alternatives for the other end of traceroute for NDN.

When a consumer requests a content by expressing an interest, normally it goes to the nearest NDN entity that holds a copy of the content. This entity may be a node caching a copy of the content or a host actually publishing that content. Also, a certain content may be published by multiple publishers and content is cached at every intermediate NDN node it traverses. Therefore, doing a traceroute in a NDN network can take different meanings depending upon the context.

Assuming we start our trace for a content from the consumer end, we have several choices. These are as follows:

- Finding a path to the nearest host publishing that content. In absence of any strategy layer, this is the publisher nearest to the consumer.
- Finding all paths to nearest host publishing that content.
- Finding all paths to all reachable hosts publishing that piece of content.
- Finding all paths the nearest copy of that content. Depending on when and how we ask, this might lead to an intermediate node caching the content or to a publisher publishing the content.
- Finding all paths to all copies of the content. Here, we want to find paths to all copies of a given content, cached or otherwise.

Before deciding which of these approaches should be considered ideal for NDN traceroute, we are going to discuss each of these methods in details.

### 2.1 Finding one path to the nearest reachable publisher

This is the scenario similar to IP traceroute. The consumer sends one interest and gets back one path to the nearest publisher. When we talk about the publisher, we are interested in tracing path to the actual host publishing the content. To reach to the nearest publisher, we bypass intermediate caches. At the intermediate nodes, the interest gets forwarded over all possible outgoing faces for that name. When such an interest gets to the publisher, it replies and the first reply consumes the PIT entry. Note that the “nearest” publisher can change depending on network parameters. It simply means the publisher that replied first. In case there are multiple publishers reachable from a consumer, the nearest publisher may change.

Therefore, if we are trying to troubleshoot a network problem, this approach is not guaranteed to help us. The trace interest can be forwarded to a publisher while the normal interest can be forwarded to another. Also note that, like IP traceroute, the trace interest and actual interest can take different routes depending on the forwarding strategy in the network. If this happens, we can get a valid reply for our trace interest and still have problem of not receiving content.

### 2.2 Finding all paths to nearest reachable publisher

Finding all possible paths to the nearest publisher can be helpful to debug the problem mentioned above. If we

enumerate all possible paths to the nearest publisher, one of these will be used for forwarding actual interests. In case there is a problem in any of the possible routes, it can potentially lead to non-availability of content. Though this is certainly advantageous over the first method, this might be expensive in terms of time and number of interests required. Also, in a densely connected network such as the Internet, this method will enumerate a large number of possible routes. However, for the similar reasons described above, we can not be guaranteed to find path to the actual publisher responding to normal interests.

### 2.3 Finding all paths to all reachable publishers

Finding all paths to all publisher is helpful in smaller networks. This way we can map out the whole topology for a certain piece of content. This way, no matter which path the trace interest or the normal interest takes, we will find an exhaustive list of problems, if any, related to fetching that piece of data. For a large network, this is costly. However, assuming that the publishers would be strategically placed in such a network, there should not be too many publishers reachable from a given consumer. Also, we can limit our trace not exploring beyond a certain number of hops. This is the model we implemented, which is discussed in the later sections.

### 2.4 Finding nearest copy of the content

As each node in NDN caches the contents, we might want to trace to the path to the nearest content available. This can be a cache or in case it is not cached yet, an actual publisher. This matches perfectly with NDN paradigm where the consumer does not care about where the content is coming from. However, this does not help us troubleshooting a network problem. If the data is dynamically generated, it would mean that we will trace to the actual publisher. This would bring back problems we discuss in section 2.1.

### 2.5 Finding all copies of the content

This is pretty similar to the scenario above. If some content is unavailable, it means that the content is not cached and we are unable to reach the publisher. Finding all copies of the content is very expensive and does not provide much additional benefit over finding all publishers. If we choose to use this method, it is likely that it would enumerate all possible paths and nodes that the data has ever traversed.

Given the advantages and disadvantages of each of these methods, we think that the method discussed in section 2.3 would be ideal for our purpose.

## 3 Differentiating between normal and interest packets

For tracing a route without fetching actual content we need to differentiate between interests meant for tracing path to a certain content and interests for fetching that content. We will call them trace interests and normal interests, respectively. This distinction is necessary because when an interest packet arrives, NDN needs to know that we are just looking existing path to the content, not the actual content. NDN does not provide any special field in the interest packet for doing this. However, we can use a special name for achieving this. There are two ways we can create a special name. First one is to create a special namespace. We can add “/trace” at the beginning of each trace interest. For example, a trace interest for “/csu/index.html” would look like /trace/csu/index.html. This method has both advantages and disadvantages. As for advantage, we can have a special daemon/handler which sets an interest filter with NFD ( `NDN_set_interest_filter` ) for “/trace” namespace. Once an interest starting with “/trace” arrives, NFD forwards it to the daemon/handler to handle it appropriately. However, this method requires each of the nodes to run such a server/handler and a rule to forward matching interests to this server. Without such a server/handler and appropriate rule, NFD would not know what to do with the trace interest and discard it.

We also can add the “/trace” at the end. This way, a trace interest for “/csu/index.html” would look like “/csu/index.html/trace”. It would not take any special forwarding rule for forwarding the trace interests. In case some nodes decide not to run traceroute server/handler on them, it will be forwarded as the normal interest. However, for figuring out if an interest is a trace interest or a normal interest, we need to parse the name for each of the incoming packets and figure out what the last component of the interest is. This will be expensive even for a reasonable number of interests/second.

## 4 Caching of data vs caching of trace responses

Caching of data in a node is reasonable and does not interfere with our design of traceroute. However, caching of trace responses are useful. Every time we issue a trace response, we want to bypass the cache and get a fresh response. For example, imagine two nodes trying to trace to the same content via same intermediate node. One node is receiving the content without trouble but the other one’s packet is being dropped by the strategy layer. In this scenario, we don’t want a cached trace response saying that the path to content is fine. We want the trace packet to go up to the trace daemon/handler and find out that in spite of being connected to the content publisher, there is no route to the content because of the strategy.

we use a random number at the end of the trace interest name. So a trace interest for `"/csu/index.html"` would look like `"/trace/csu/index.html/12678664"`. We do this for two reasons. One of the reasons being what we discussed above, to avoid caching of trace responses. During interest forwarding, the cache is consulted first. Without having a way to bypass the cache, it is not possible for trace interest to reach the trace daemon/handler. In present architecture, an interest has 1:1 mapping to content. That means, one interest would fetch exactly one piece of content. In case we want to enumerate all possible paths to a publisher or content, the intermediate nodes have to send multiple distinct interests if multiple paths are available. Also, the trace daemon needs to keep track of interests it already processed, otherwise it might forward the same interests again and create a loop.

## 5 Iterative vs non-iterative traceroute

We need to decide whether we want to make our trace iterative or non-iterative. In the iterative mode, the consumer sends one interest packet to the immediate neighbors. The neighbors reply with forwarding information, if any. On receiving this information, the consumer again sends another interest, with instruction to exclude the neighbor from the path. The neighbor forwards this trace interest and gets back reply from another node. This goes on until the whole path is explored.

In the non-iterative mode, the consumer sends one interest. The intermediate nodes forward the interest and wait for reply or replies. Once the replies arrive, it consolidates them and send back a new reply upstream.

In iterative mode, consumers control the whole process. It sends an interest, gets a reply back with possible forwarding paths. It then can choose which path to explore. Also, as the client gets intermediate responses, it has much finer control over the whole process. For example, it can set the timers about how long it should wait for getting a reply. However, depending on whether we are trying to explore all possible paths for a given content or not, this might be costly in terms of number of interest packets exchanged. Also, another problem with iterative trace route is lack of expressiveness of interest packets. Once we get back a reply from an intermediate node, the consumer need to indicate that this node should not reply further. For this, an option might be the exclude filter. However, in this case, there is no direct way to exclude the already visited nodes. As a work around, we can append the node identifier to the interest name. When an intermediate node finds out that it's identifier is in the interest name, it forwards the packet. However, exclude filter is meant to match qualifying content objects against an interest. We can hack it for our purpose, but it will be just a temporary solution.

In the non-iterative mode, number of interests are less than iterative mode. However, returning the reply messages gets complicated. We want to keep the content objects unchanged for verification at the client. Suppose we have  $n$  paths for a content from an intermediate node. In case we want to enumerate all paths, we will embed  $n$  content messages in our reply. If the previous node has  $m$  paths, we now have a content object which have  $m$  embedded messages, each of these messages having several messages inside them. Also, using this method complicates setting the timeout value at consumer. How long does the consumer wait before timing out? The delay between sending and receiving reply would depend on length of the path and also on several other factors such as processing speed at intermediate nodes.

## 6 Best layer for traceroute

Before we discuss the methods of implementing traceroute for NDN, we need to figure out which layer it should reside in. We have two choices for placing this in the NDN stack. One obviously is the application layer, the other is strategy layer.

If we place it in application layer, it would work like this: The consumer sends an trace interest packet. A server is placed on each node which listens for this special interest. Once NFD forwards this interest to the application server, the server handles the packet appropriately and tells the NFD daemon about the action it should take. Depending on the position of this node, this action might be forwarding the interest packet or sending a reply back. This method, however, has a limitation. In the current implementation, the NFD daemon and the application layer is connected though a socket. There can be hypothetical scenarios where all the trace interests are forwarded as per the FIB entries but the actual interests are dropped. In this case, we will find available routes to a certain content/publisher but fetching the actual content would be impossible.

We therefore argue that while implementing traceroute is possible at the application layer, best possible choice would be to integrate it with the strategy layer where it will know about the forwarding strategy decisions.

## 7 Architecture & Design

In our implementation, we choose to enumerate all paths to all reachable publishers of a certain content. In a smaller than Internet size testbed, we can find out all possible routes to a certain content and effectively finding the topology map for a certain content. This will help us finding possible faults in the network. Also, we implemented the non-iterative version as we discussed, where the consumer sends one interest and waits for reply.

## 7.1 Interest packets

The trace interest packets are very similar to the normal interests. We append a “/trace” at the beginning of the interest name for identifying it as trace interest packet. We also append a random number at the end the interest name for identifying duplicate interests at the trace server. As already discussed earlier, an interest for “/csu/index.html” would look like, e.g., “/trace/csu/index.html/123456”.

## 7.2 Data Packets

The data packet is a normal content object containing the path information.

## 7.3 Parsing FIB table

As we discussed earlier, our application is at application layer which does not have direct access to NFD data structures and values. However, NFD publishes the FIB table as a html page. We parsed this page to figure out the FIB. However, this is just the forwarding table, we don’t have access to the strategy layer. We assume for our implementation that there are no strategies in place and the interest packets are forwarded as per the routes found in the FIB.

## 7.4 Identifiers

For finding out the actual path that the trace interests are going to take, we need to use a unique identifier for each nodes. This unique identifier can be anything, for our purpose we will use IP addresses at each node. Note that we don’t use IP addresses for any other purpose but identification of nodes. Right now, IP address allows us to identify the faulty nodes. We expect that in actual NDN network, there would be some kind of unique identifier for this function.

## 7.5 Timeout Values

NDN uses a 4 second default timeout value. For our purpose, when forwarding the trace interests, we use this default value for timeout. However, the amount of timeout required at the client depends on several factors such as length of the path and delay at intermediate nodes. Therefore, this is flexible and dependent on the user. By default, after a timeout, the client re-expresses the interest two more times before giving up.

## 7.6 Remote vs Local Content

For figuring out if an incoming trace interest is for something that is local, we parse the FIB for the given content name. All content that is being published will have an entry in the FIB. Moreover, remote entries will have a “remote” flag in the FIB entry. If such a flag exists, we know that the content is remote. Otherwise, the content is local.

## 7.7 Handling Duplicate Trace Packets

NFD does loop avoidance by default. However, as we handle the trace packets at the application layer and forward these by adding and deleting multiple routes, we might create loops. For avoiding this, we keep track of all the trace interests that pass thorough each node and drop the duplicates. The random number at the end of trace interests is particularly helpful for this purpose. As this is unique, we just keep track of this number for discarding duplicates.

## 7.8 Forwarding

The trace server is the actual engine of the whole process. Upon start up, it registers a interest filter with NFD for incoming trace packets. As a result of this, NFD forwards all trace packets to the server. The server does a duplicate checking and discards if the incoming packet is a duplicate. It then extracts the actual content name by removing the first and the last component of the name. It then look in the FIB for finding out if there is any entry for the interest name. If there is no such entry, it sends back a message saying that there is no path for the content. If there is such a entry, it looks if a “remote” flag is associated with the entry. If not, this is a local content. The server sends back a message saying that it is local content. In case, there is a remote entry in the FIB, the server looks up where this entry points to. This results in a list of one or more outgoing faces. The server duplicates these entries by adding new routes, but for the trace interest. For example, if an intermediate node has remote entries for “/csu/index.html” via face 34 pointing to 10.0.0.1 and face 35 pointing to 10.0.0.2, the server creates two interest packets, say, “/trace/csu/index.html/123456” and “/trace/csu/index.html/567892”. It then adds two routes to NFD using

```
nfdc add-nexthop /trace/csu/index.html/123456 tcp
↳10.0.0.1
```

and

```
nfdc add-nexthop /trace/csu/index.html/567892 tcp
↳10.0.0.2
```

It then expresses interest for the two interest packets. Once it receives the replies, it parses them, add its own identifier to the path and sends the replies. If any of the paths times

out, it indicates it on the reply message. Finally, it cleans up the routes it added to NFD.

```
reply = identifier + "NO ROUTE"
note name in duplicate_entries
replies = replies + reply
send replies
```

## 8 Implementation

In our implementation, we have two programs, one is a trace client another is a trace server.

### 8.1 Trace Client

The trace client is a simple program which asks for a name and a timeout value. Here is the pseudo-code for the client. This is the pseudo code for the client:

```
read (name) (timeout)
prepend "/trace" to (name)
append <random number> to (name)
construct interest packet from name
express interest using the interest packet
listen for reply until timeout
while(timeout count) < 3:
    reexpress interest
    timeout_count++
```

### 8.2 Trace Server

```
Note node identifier (IP address)
use NDN_set_interest_filter with name /trace
while(TRUE):
listen for incoming trace interests
if incoming trace interest is not duplicate_
->entries:
    content_interest_name = interest name
    without first & last component

    if content_interest_name in FIB:
        path_exists = Yes;
        if path is local:
            reply = identifier + "LOCAL"
            replies = replies + reply
            note name in duplicate_entries
        else if path is remote:
            find how many remote paths are in FIB:
            for each path:
                prepend /trace and append <random number>
->to name
                find remote IP for this path
                add a route to FIB for this name
                express interest and wait for reply
                if received reply:
                    paths = parse reply
                    for each path in paths:
                        reply = identifier + "FWD" + path
                        replies = replies + reply

                else if timed out:
                    reply = identifier + "NO REPLY"
                    replies = replies + reply
                    delete added route from FIB
    else:
        path_exists = No;
```