NDN-Trace: A Path Tracing Utility for Named Data Networking

Siham Khoussi
National Institute of Standards and Technology (NIST)
siham.khoussi@nist.gov

Davide Pesavento[*]
National Institute of Standards and Technology (NIST)
davide.pesavento@nist.gov

Lotfi Benmohamed
National Institute of Standards and Technology (NIST)
lotfi.benmohamed@nist.gov

Abdella Battou
National Institute of Standards and Technology (NIST)
abdella.battou@nist.gov

ABSTRACT

In this paper we propose NDN-Trace, a path tracing utility to determine the characteristics of the available paths to reach a given name prefix in NDN-based networks. While the traceroute tool in IP networks is based on an iterative process, with each iteration incrementally traversing more hops along the path to the target, we adopt a non-iterative approach, with the tracing process done at the application layer. Our design supports multi-path tracing that can be used to trace paths to NDN forwarding nodes, applications, or content store caches, while providing path information (node identifiers and round-trip times), as well as optional metrics such as those related to content stores. NDN-Trace leverages NDN's native Interest/Data exchange and does not require changes to NDN forwarding. We present a C++ implementation of our design, and show experimental results that demonstrate its capabilities. We also discuss open issues and future work, including an approach to implement path tracing within the NDN forwarder itself.

CCS CONCEPTS

• **Networks → Network protocol design**; *Network measurement*; *Network monitoring*;

KEYWORDS

Path tracing, Traceroute, Named data networking, Information centric networking

1 INTRODUCTION

Because of fundamental differences between IP and NDN (Named Data Networking [6]), path tracing in NDN requires a novel approach [7]. Since routing and forwarding in IP networks are based on IP addresses, the IP traceroute tool [5] was developed to determine network reachability and latency information for an endpoint identified by a given IP address. NDN is based exclusively on named objects, with routing and forwarding based on name prefixes that do not refer to unique endpoints. An Interest is forwarded by the network nodes based on its name, and retrieves content either from some producer application (or content publisher), or from the Content Store (CS) of one of the on-path nodes. Consecutive transmissions of Interest packets with the same prefix may be forwarded along different paths, and may reach different data sources (producers or caches). Therefore we need to be able to discover all paths to a given name prefix, which makes the implementation of path tracing in NDN more challenging. Thus, multi-path forwarding introduces new requirements in the design of path tracing for NDN. Note that leveraging and improving the multi-path forwarding capability is an active area of research [4, 12, 17].

A path tracing design for NDN should support a number of use cases, including: discovering one (such as the nearest) or more cached copies of a given content, finding one or all paths to an application or content publisher (bypassing any on-path caches), tracing one or more paths towards an NDN router and identify all hops for diagnostic purposes, measure the round-trip time (RTT) to retrieve some content or reach a forwarder. Consequently, the NDN name being traced can be: a name prefix belonging to an application's namespace, the name of a content that can be cached somewhere in the network, or the name (assigned by the network operator) of any node running an NDN forwarder. Finally, in addition to providing basic path information (node identifiers and round-trip times), the goals of tracing can be extended to include returning other metrics such as those related to content stores [11, 16].

With these goals in mind, we designed **NDN-Trace**, a path tracing utility that can be used to determine the characteristics of available paths to a given name prefix in NDN-based networks. To reduce the deployment barrier of this utility, we decided to start with a design that does not require changes to the forwarder. The option of having path tracing implemented inside the forwarder itself is discussed as part of our future work. NDN-Trace uses NDN's native Interest/Data exchange and exhaustively explores multiple paths using a lightweight forwarding strategy tailored to this purpose.

This paper is organized as follows. Section 2 explores related work in terms of previous ICN traceroute proposals. The design of NDN-Trace is described in section 3, followed by a detailed explanation in section 4 of how we estimate round-trip times. In section 5 we introduce our C++ implementation and present preliminary evaluation results. Open issues and future work are discussed in section 6. Section 7 concludes the paper.

2 RELATED WORK

Despite the critical importance that a path-tracing tool assumes in Information-Centric Networks, the matter has received relatively little attention from the scientific community so far.

[*] Also with Sorbonne Universités, UPMC Univ Paris 06, Laboratoire d'Informatique de Paris 6 (LIP6).

ICN '17, September 26–28, 2017, Berlin, Germany
2017. ACM ISBN 978-1-4503-5122-5/17/09…$15.00
https://doi.org/10.1145/3125719.3125738

Asaeda et al. [3] propose Contrace, a tool for measuring the round-trip time between routers in Content-Centric Networks (CCN). The Contrace client creates and sends trace requests encoded as special CCNx packets. Each on-path router attaches its own Report block, containing arrival timestamp and node identifier, to the trace request and then forwards it to its upstream neighbor router. This process is repeated until the request reaches either the content producer or a cache with a copy of the content. At this point, a Reply message is generated and propagated back to the Contrace client following the PIT state. In order to support multi-path tracing, Contrace requires PIT entries created by trace requests to be kept alive until a configured timeout expires. This represents a significant departure from the "flow balance" principle established by NDN. Moreover, Contrace does not provide RTT measurements between the client node and each intermediate router on the path to the content; instead, it only provides the RTT between the client and the node where the content is found, similar to the Ping tool.

In their ICN Traceroute protocol specification, Mastorakis et al. [8] take a different approach, which resembles the mechanism used by the IP traceroute tool. The ICN Traceroute client iteratively discovers the path to a named content by issuing Interests with a progressively increasing HopLimit counter. This counter is decremented by one at each hop traversed by the Interest, and when it reaches zero, the forwarding of the trace request stops, while a reply containing the forwarder's name is sent back to the client following the reverse path. A new PathSteering header is introduced, which is constructed hop-by-hop while the reply travels back to the client, and is then included in subsequent trace requests that need to be forwarded along the same path. This design is unable to reliably support multi-path tracing, as there is no guaranteed mechanism for the client to choose a new path that was not explored before. Thus, the client cannot tell whether it discovered all available paths.

Shannigrahi et al. [13] tackle several design questions and tradeoffs that we also faced, and that will be discussed in the remainder of the paper. The solution they offer, however, does not distinguish between single and multi-path tracing, does not consider RTT measurements, and does not support tracing of cached content.

3 THE NDN-TRACE PROTOCOL

The architecture of NDN-Trace consists of three interacting components:

- A *client* application, run by the user, which sends the initial trace request, and displays the discovered path(s) once the tracing session is completed.
- A *daemon* that processes trace requests and replies, and performs round-trip time measurements. It registers the `/Trace` name prefix with the local NDN forwarder.
- A *forwarding strategy*, installed by the daemon under the `/Trace` prefix, that handles trace request forwarding and performs Content Store lookups on behalf of the daemon.

The strategy and the daemon must be running on every NDN node in the network. However, only the node that initiates a trace request is required to run the client application. See section 6.3 for an alternative design that requires neither a forwarding strategy nor a separate daemon.

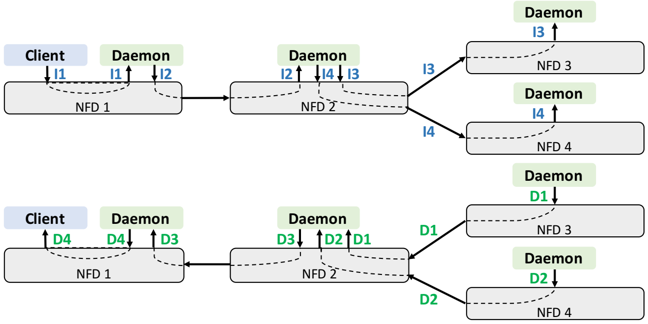


Figure 1: Forwarding of NDN-Trace requests and replies. On each node, NFD performs the standard CS/PIT/FIB lookups. The second node is a forking point for the trace request: two new Interests I3 and I4 are created, and each one is sent to a different next hop. When the two Data D1 and D2 come back, the daemon combines them into a single reply D3.

NDN-Trace operations leverage the conventional NDN Interest/Data exchange: a *trace request* is an Interest packet, and a *trace reply* is a Data packet[1].

3.1 Trace request

Upon user invocation, the NDN-Trace client initiates a tracing session by expressing a trace Interest, whose name includes the following components:

- a fixed `/Trace` prefix, this *must* be the first component;
- a parameter P_1 to denote whether single-path or multi-path tracing is requested: this is currently a binary setting, but can easily be extended to an integer representing the maximum number of paths to explore at each hop;
- a parameter P_2 that indicates the type of tracing: "application" (matching FIB entry toward a producer application), "cache" (matching Content Store entry), or "any", meaning that tracing will stop at either a producer or a cache, whichever is found first;
- the name to trace;
- a random nonce that is regenerated at every hop: this is needed to prevent the forwarder from aggregating Interests from different tracing sessions in the same PIT entry;
- the identifier of the chosen outgoing face for the Interest, used to steer the request toward a particular next hop.

For example:

`/Trace/<`P_1`>/<`P_2`>/<NameToTrace>/<TraceNonce>/<FaceId>`

Note that P_1, P_2, TraceNonce, and FaceId can be combined into a single name component using a suitable reversible encoding. For clarity of exposition, we will keep them separate here.

When a trace Interest is received by NFD (the NDN Forwarding Daemon [2]), it is forwarded according to the FIB to the trace daemon of that machine. For the first hop, the daemon is effectively running on the same node as the trace client application.

[1] Throughout the paper, we use the terms "trace request" and "trace Interest", and "trace reply" and "trace Data" interchangeably.

Each daemon maintains a Pending Trace Table (PTT) data structure. When a daemon receives a trace Interest, it searches its PTT for an entry with the same name-to-trace and the same network nonce as the incoming Interest (note that this is the nonce specified by the NDN Interest packet format, not the application-level nonce that appears in the trace request name). If a match is found, this arrival is considered a loop, and a `PATH_LOOP` reply is immediately sent back to the previous hop. Otherwise, a new entry is inserted into the table, together with the arrival time of the request.

Then, the daemon employs NFD's management protocol to interrogate the FIB for any entries that match the traced name prefix, thus obtaining the list of next hops for that prefix and their respective cost. If the list is empty, it means that this node does not know how to reach the target name prefix, therefore the trace daemon replies with a `NO_ROUTE` error. Else, the list is traversed in increasing order of routing cost. If the trace request parameter P_1 indicates that this is a single-path trace, only the first (lowest cost) next hop is considered, otherwise all entries are processed. For each of them, and only if P_2 is set to "application" or "any", the daemon checks if the corresponding face points to a local application. If that is the case, a partial reply is recorded in the PTT for this next hop, and processing continues with the following entry in the list of next hops. Otherwise, a new trace Interest is expressed. This Interest is identical to the incoming Interest, except for two things: (1) the trace nonce contained in the name is refreshed; (2) the outgoing face identifier, also in the name, is replaced with this next hop's FaceId. In particular, the NDN-layer nonce is kept unchanged from the incoming Interest, to ensure that the request loop detection is effective[2]. The daemon keeps track of all these new trace Interests in the PTT, along with the time at which they were expressed. Eventually, when all Interests for the same tracing session are either satisfied, expired, or nacked, the daemon generates a trace reply as explained in 3.2.

A daemon may also choose to reply with a `PROHIBITED` error, in case the target name belongs to a set of prefixes for which the network operator has blocked trace requests.

Inside NFD, trace Interests are handed over to our custom trace strategy, that as a first step examines the parameter P_2 in the name. If P_2 indicates that the client wishes to trace a cached object, the strategy performs a Content Store lookup on the traced name only, as opposed to the full Interest name. If a matching Data packet is found, the strategy responds to the Interest with a Nack containing a custom reason, in order to let the daemon know that the target has been reached. On the other hand, if P_2 has a value other than "cache", or if the CS lookup fails, the strategy simply sends the trace request on the face specified in the Interest name.

This whole process is repeated at every node encountered on the path(s), until all trace requests either reach the target prefix, or hit an error condition, or time out.

3.2 Trace reply

A trace reply is an NDN Data packet sent by an NDN-Trace daemon in response to a trace request. Replies are forwarded back downstream following the PIT entries left behind by the corresponding trace Interests, as per the regular NDN forwarding semantics. The Data packet payload contains the following set of values for each node traversed during the tracing session:

- The node identifier: this can be anything that allows to identify an instance of a forwarder, globally or locally within a network. Conceptually, this should be the node's name, or a combination of the node's name and the ID of the face on which the trace Interest was received. Globally unique names can be hierarchically assigned to a router by the network operator in a manner similar to how domain names are employed in today's Internet. In our implementation we have chosen to use the NFD ID, which is the NDN name of the public key used by the forwarder.
- The measured round-trip time from this node to the target name prefix.
- The time spent by the trace daemon on this node to prepare and send all trace requests and to process all the corresponding replies. See section 4 for how this number is used in the final RTT computation. The two most expensive operations that are taken into account here are the trace Data signing operations and the queries executed via NFD's management protocol, which also involve cryptographic operations.

Upon receiving a trace Data packet, the daemon records its arrival time, which is used to compute the round-trip time between this node and the upstream node that sent this reply. As soon as all PTT entries for a tracing session are either satisfied, expired, or nacked, the daemon creates a new Data packet containing the identifier and the timing information for this node, as previously explained, plus the values returned by all upstream nodes, up to the point where the tracing stopped (for whatever reason). In case of multi-path tracing, the daemon extracts the information returned from each path, and concatenates them together to form a single[3] Data packet, which is then sent downstream as usual.

Eventually, the NDN-Trace client will receive a reply containing all the information generated by each hop along the path (or paths) to the traced name prefix. As a last step, the client computes the RTT values as described in section 4, and displays the results accordingly.

4 MEASURING ROUND-TRIP TIMES

Based on the above description of Interest and Data processing at the trace daemon, it is clear that Interest/Data packets for path tracing incur higher processing time as they traverse NDN forwarding nodes than regular Interest/Data packets from user application traffic. Consequently, we need to compensate for this difference when computing round-trip times by properly accounting for the additional processing delay, as we will see in this section.

When node i sends trace Interests to n upstream neighbors ($n = 1$ if single-path tracing), it measures the response time to each one of them when the corresponding trace Data is received. All these

[2] Since trace Interests are terminated at every hop by the trace daemon, and new Interests have a different name, NFD can no longer detect loops, thus the responsibility of loop detection is now assumed by the daemon.

[3] The reply Data packet may need to be segmented by the daemon if its total size exceeds the maximum allowed by NDN (8800 octets). In that case, the standard NDN naming conventions for sequence-based segmentation are used [18].

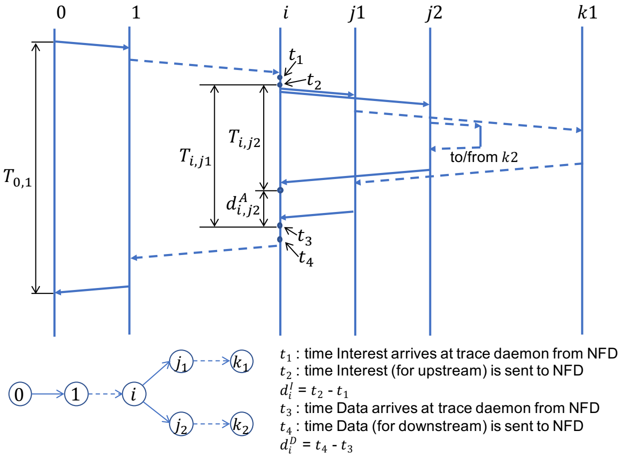

0
1
i
$j1$
$j2$
$k1$
t_1
t_2
$T_{i,j2}$
$T_{i,j1}$
$T_{0,1}$
to/from $k2$
$d^A_{i,j2}$
t_3
t_4
0
1
i
j_1
k_1
j_2
k_2
t_1 : time Interest arrives at trace daemon from NFD
t_2 : time Interest (for upstream) is sent to NFD
$d^I_i = t_2 - t_1$
t_3 : time Data arrives at trace daemon from NFD
t_4 : time Data (for downstream) is sent to NFD
$d^D_i = t_4 - t_3$

Figure 2: RTT measurement: a time-sequence diagram illustrating the different overhead and delay components.

measurements are then aggregated in one Data packet and sent to node i's downstream. Two quantities are measured by node i for each upstream node j:

- $T_{i,j}$: the time between sending an Interest to j and receiving the corresponding Data back.
- $D_{i,j}$: the delay at the trace daemon on node i due to processing the request/response to/from node j. Note that if k is the node hosting the traced content, and thus will locally handle and not forward the trace request, then $D_{k,k}$ denotes the time it takes to process the request at the local forwarder's trace daemon.

While $T_{i,j}$ is computed in a straightforward manner (as the difference between request and response times for a trace Interest sent by i toward j as shown in Fig. 2), $D_{i,j}$ is made up of three components. Mathematically:

$$D_{i,j} = d^I_i + d^D_i + d^A_{ij}$$

where:

- d^I_i is the *Interest processing overhead* at node i's daemon.
- d^D_i is the *Data processing overhead* at node i's daemon, and includes signature verification of the incoming Data packet and signing of the outgoing one[4].
- d^A_{ij} is the *aggregation delay* at node i incurred by the trace Data received from node j. This is due to the fact that, in the multi-path case, each reply is held at node i until a response is received from all upstreams, so that one single Data packet aggregating information from all paths through i can be created and sent downstream (note that $d^A_{ij} = 0$ for single-path tracing).

The information that node i sends to its downstream node includes the $T_{i,j}$ and $D_{i,j}$ for each upstream neighbor j used to forward a trace request. Node i will also receive similar information from these upstream nodes regarding their own upstream nodes, and this information will be concatenated with node i's own information. Obviously, as a trace Data packet is propagating downstream toward the client, its payload increases in size due to the nesting of information from each hop along the path. Once at the client, the packet will contain all the information needed for the client to be able to identify all paths that have been successfully traced.

For each discovered k-hop path, spanning nodes $0, 1, 2, \ldots, k$ (where 0 refers to the node where the trace client is running), the RTT for the whole path is computed as follows:

$$RTT_k = T_{0,1} - \sum_{i=1}^{k-1} D_{i,i+1} - D_{k,k}$$

From the response time measured at node 0 ($T_{0,1}$) we subtract all the overhead due to the trace daemon at all intermediate nodes as well as at the end node ($D_{k,k}$). The result approximates the RTT experienced when retrieving an already signed Data packet from node k. However, for dynamic Data that cannot be signed ahead of time (such as for the `ndnpingserver` application when signing a ping reply message), the RTT could be higher than estimated by this equation.

When needed, the client can also compute the RTT to each intermediate node m on this path to k, as follows:

$$RTT_m = T_{0,1} - \sum_{i=1}^{m} D_{i,i+1} - T_{m,m+1}$$

where node $m + 1$ is the upstream neighbor of m on the path to k (which has RTT_k as its round-trip time).

5 IMPLEMENTATION AND EVALUATION

We implemented a prototype[5] of NDN-Trace in C++. The trace client and daemon are based on the ndn-cxx library [14], while the trace strategy has been implemented for NFD [2].

5.1 Command-line tools

The NDN-Trace client program is called `ndntrace`, and it can be run in the following way:

```
ndntrace -n <NAME> [-s|-m] [-p|-c|-a] [-C]
         [-t TIMEOUT] [-r COUNT]
```

where:

- `-n` specifies the traced name prefix. This is the only mandatory parameter.
- `-s` and `-m` choose between single-path and multi-path tracing, respectively. It is the equivalent of parameter P_1 described in section 3.1. If this option is not specified, the program defaults to single-path tracing.
- `-p|-c|-a` is the equivalent of parameter P_2, i.e. it indicates the type of tracing session: `-p` to trace a producer application, `-c` to find a cached copy, `-a` (the default) for either of them.
- `-C` activates the retrieval of extended Content Store statistics, such as hit and miss ratio for the target name prefix, number of matching Data packets stored in the cache and their size, and so on. This is disabled by default.

[4] The reply signing time can be estimated, for example by maintaining a moving average from previously processed trace replies.

[5] The complete source code is available at `https://github.com/usnistgov/ndntrace`.

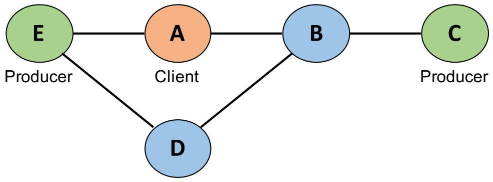


Figure 3: NDN topology used for the evaluation. Path 1 is $A \rightarrow E$; path 2 is $A \rightarrow B \rightarrow C$; path 3 is $A \rightarrow B \rightarrow D \rightarrow E$.

- `-t` indicates the timeout after which a trace request expires and should be aborted. The argument is used to set the InterestLifetime of the requests.
- `-r` specifies the number of times that a trace request is retransmitted in case a timeout happens. By default, no retry is attempted.

The NDN-Trace daemon is called `ndntraced`, and currently accepts no options. In the future, we plan to make the daemon’s behavior more configurable via a configuration file, which can be provided by the operator through a command-line option.

5.2 Sample NDN-Trace execution

To show NDN-Trace in action, we deployed a small network of 5 nodes on the Emulab testbed [1]. The network topology is represented in Fig. 3. The nodes were connected to each other via wired Ethernet, each link with a capacity of 1 Gbps and a delay of 10 ms. All 5 machines were running the Linux operating system (UBUNTU14-64-STD image), and the latest released version of NFD and ndn-cxx (0.5.1 at the time of writing). We then installed the NDN-Trace stack: the forwarding strategy and the daemon on every node, while the client application was installed only on the node issuing the trace commands (node A in Fig. 3). Finally, we spawned a trivial producer application, serving the prefix `/example`, on nodes C and E.

In the case of single path tracing, we expect NDN-Trace requests to follow the path that has the lowest cost among all available next hops (see section 3.1). In the experiment topology this path is $A \rightarrow E$, based on static routes that we set on each node beforehand. If we run the command:

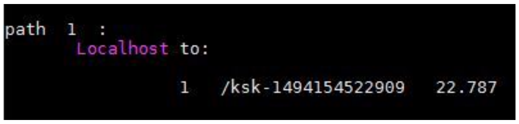
```
ndntrace -n /example -s -p
```

on node A, we indeed obtain the expected output, as we can see from the screenshot in Fig. 4. Note that “localhost” in the tool output refers to the node on which the client is running.

In order to run a multi-path trace, we simply replace the `-s` option with `-m`, i.e.:

```
ndntrace -n /example -m -p
```

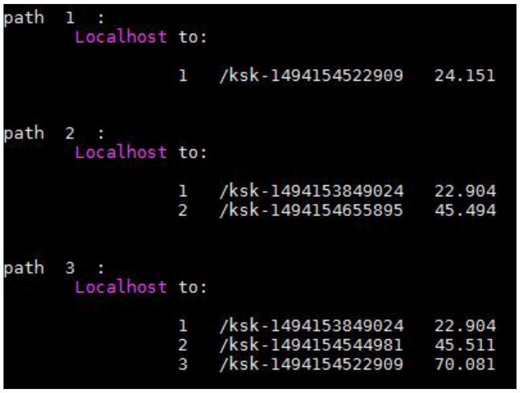
This command discovers all paths to any producer application that can publish content under the `/example` name prefix. In our case, three paths are displayed by the trace client, as shown in Fig. 5: path 1 is $A \rightarrow E$, path 2 is $A \rightarrow B \rightarrow C$, and path 3 is $A \rightarrow B \rightarrow D \rightarrow E$. For instance, the three rows for path 3 listed at the bottom of the screenshot correspond to the three non-localhost nodes on that path. Each row shows the abbreviated NFD ID of the node and the corresponding RTT, as experienced by A and estimated according to the RTT_k expression in section 4.



```
path  1  :
      Localhost to:

                 1   /ksk-1494154522909   22.787
```

Figure 4: Screenshot of a single-path tracing session.



```
path  1  :
      Localhost to:

                 1   /ksk-1494154522909   24.151

path  2  :
      Localhost to:

                 1   /ksk-1494153849024   22.904
                 2   /ksk-1494154655895   45.494

path  3  :
      Localhost to:

                 1   /ksk-1494153849024   22.904
                 2   /ksk-1494154544981   45.511
                 3   /ksk-1494154522909   70.081
```

Figure 5: Screenshot of a multi-path tracing session.

Table 1: Average and standard deviation of the RTT reported by the two tools on the emulated topology. “Baseline” is the total two-way link delay.

	Path 1	Path 2	Path 3
Baseline	20 ms	40 ms	60 ms
`ndnping`	24.26 ± 0.4 ms	50.61 ± 0.7 ms	70.33 ± 1.4 ms
`ndntrace`	21.91 ± 0.5 ms	47.96 ± 1.0 ms	69.12 ± 0.8 ms

5.3 Comparison with NDN ping

In order to demonstrate the accuracy of the results obtained by NDN-Trace, we use NDN ping on the topology of Fig. 3, and compare the measured round-trip times as reported by the two tools. We test both tools on all 3 paths listed in the previous section. Table 1 shows the average RTT values obtained from running each tool 100 times on each path.

We observe from these results that the RTT values measured by NDN ping are on average slightly higher than those reported by NDN-Trace. This is easily explained if we consider how NDN-Trace calculates the round-trip times. As described in section 4, NDN-Trace subtracts the request/reply processing delay experienced by the trace daemons at each hop along the path, including the final node, where the producer application is running. Conversely, NDN ping does not perform such accounting, and the reported RTT is simply what is measured by the ping client, which includes the overhead incurred by the ping server when signing the ping reply packet. We independently measured the entity of the delay caused by one Data signing operation, and obtained an average value of about 3 ms on the hardware used for the experiments, which is consistent with the difference between the two tools.

6 DISCUSSION AND FUTURE WORK

In this section we outline some limitations of the current NDN-Trace design, and discuss potential solutions and future work.

6.1 Carrying signatures in trace replies

When an NDN-Trace daemon receives a reply from its next hop, the path information is extracted, appended to the values measured at the current hop, and finally everything is serialized into a new Data packet. This means that the cryptographic signature from the upstream trace daemon is discarded after verification, and is replaced by this node's signature. When the reply eventually reaches the client application, the packet carries only the signature of the trace daemon running on the same machine as the client.

This has the benefit of making replies smaller, because only the payload (i.e., the path information reported by all upstream nodes) is propagated from each hop to its downstream, while all other Data packet fields, including their TLV (Type-Length-Value) encoding overhead, are discarded. On the other hand, the duty of verifying Data signatures falls on every daemon on the traced path(s). This may not be desirable in some scenarios. For example, the intermediate nodes may not have the resources to perform signature verification, which may involve fetching the entire certificate chain, on behalf of the trace client.

An alternative method for constructing the reply packet consists in concatenating the full Data packets received from the upstream hops, including the Name, MetaInfo, and Signature TLV fields, and inserting the resulting blob into the new Data packet, along with the path information generated by the current node. Thus, the client that originated the trace request can inspect the signatures of all intermediate nodes, and therefore the information returned by each hop can be verified and trusted independently. We leave a more detailed comparative analysis of the two methods as future work.

6.2 Eliminating the custom forwarding strategy

Our NDN-Trace implementation requires the use of a specific forwarding strategy for the `/Trace` name prefix, as detailed in section 3. This requirement is dictated by two reasons: (1) a strategy can perform a Content Store lookup for the traced name prefix, if requested by the client; (2) a strategy can steer outgoing trace Interests toward the correct face, as specified by the daemon.

The first requirement can be lifted by augmenting NFD's management protocol with a Content Store API. That would enable a privileged local process to enumerate the Content Store or query existing entries. As for the second requirement, we can leverage an NDNLPv2 header field called "NextHopFaceId" [15]. This field allows local applications to tell the forwarder which face should be preferred to forward an outgoing Interest. With these two changes, NDN-Trace does not need a custom forwarding strategy anymore.

6.3 In-forwarder implementation

The design described in section 3 has the important advantage of not requiring modifications to the NDN forwarding engine, and as such it can be deployed independently of the forwarder, and it is not tied to any particular forwarder variant. It also greatly simplifies the implementation of the trace daemon itself. These were the main reasons why we chose this approach for our proof of concept.

Other designs are certainly possible, in addition to those already proposed in the literature (section 2). In particular, we sketched a design in which the tracing functionality is known to the forwarder. In this case, all NDN-Trace daemon operations are performed by the forwarder itself, while the client application maintains the role of triggering a trace request and displaying the results to the user.

This approach could provide substantial performance advantages, as it would remove all intra-node communications with the trace daemon. The presence of a `/Trace` prefix on an incoming Interest facilitates splitting trace traffic into a "slow path", to avoid affecting regular data plane traffic. Moreover, there would be no need to go through the management protocol to query the forwarder's FIB, thus saving costly serialization and signing operations. We intend to further explore all the ramifications of an in-forwarder implementation and study its performance impact as future work.

6.4 Discovering all cached copies

While the current NDN-Trace design can be used to discover cached content, it will only find the first encountered copy on the explored path (or at most one copy per path in the multi-path case). In order to discover all cached copies in the network, we plan to augment NDN-Trace to report other cached copies back to the client. This can be accomplished by either proceeding further along the path when a cached copy is found, or iteratively resubmitting new trace requests from the client with an indication to ignore the nodes whose cached copies are already known.

This brings up the related issue of path search termination, which also applies to the other path discoveries discussed above: if a trace request does not encounter the target name prefix (content producer or cache) after a certain number of hops have been traversed, do we need to stop the process, and if yes, how? Some ICN flavors, such as CCNx, already contain a HopLimit field in their packet format. NDN does not provide this functionality in its TLV format [9], but it would be straightforward to implement an equivalent feature at the application layer, by including a time-to-live counter in the name of the trace Interest. Another approach is to just rely on an eventual looping of the trace Interest to stop any further propagation.

7 CONCLUSION

The new networking paradigm introduced by NDN, in particular its stateful multi-path forwarding plane with caching and named-based routing, requires a new solution to network path tracing. We propose NDN-Trace as one such solution that can be used to discover forwarding path information for a given name prefix. For ease of deployment, we chose an application-layer implementation for the first version of our path tracing utility. While we initially evaluated NDN-Trace through emulation, we intend to deploy and test it on the NDN testbed [10] soon. NDN-Trace is still a work in progress: future versions will support more of the features discussed here and will also provide an in-forwarder implementation.

DISCLAIMER

Any mention of commercial products or reference to commercial organizations is for information only; it does not imply recommendation or endorsement by NIST, nor does it imply that the products mentioned are necessarily the best available for the purpose.

REFERENCES

[1] 2017. Emulab – Network Emulation Testbed. (2017). http://www.emulab.net/

[2] Alexander Afanasyev et al. 2016. *NFD Developer’s Guide*. Technical Report. NDN, Technical Report NDN-0021, Revision 7. https://named-data.net/publications/techreports/ndn-0021-7-nfd-developer-guide/

[3] Hitoshi Asaeda, Xun Shao, and Thierry Turletti. 2017. *Contrace: Traceroute Facility for Content-Centric Network*. Internet-Draft draft-asaeda-icnrg-contrace-02. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-asaeda-icnrg-contrace-02 Work in Progress.

[4] Giovanna Carofiglio, Massimo Gallo, and Luca Muscariello. 2016. Optimal multipath congestion control and request forwarding in information-centric networks: Protocol design and experimentation. *Computer Networks* 110 (2016), 104–117.

[5] Van Jacobson and S Deering. 1989. Traceroute tool. (1989).

[6] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. 2009. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 1–12.

[7] Dirk Kutscher, Suyong Eum, Kostas Pentikousis, Ioannis Psaras, Daniel Corujo, Damien Saucez, T Schmidt, and Matthias Waehlisch. 2016. *Information-centric networking (ICN) research challenges*. Technical Report.

[8] Spyridon Mastorakis, Jim Gibson, Ilya Moiseenko, Ralph Droms, and David Oran. 2017. *ICN Traceroute Protocol Specification*. Internet-Draft draft-mastorakis-icnrg-icntraceroute-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-mastorakis-icnrg-icntraceroute-01 Work in Progress.

[9] Named Data Networking Project. 2017. NDN Packet Format Specification. (2017). https://named-data.net/doc/NDN-TLV/current/

[10] Named Data Networking Project. 2017. NDN Testbed. (2017). https://named-data.net/ndn-testbed/

[11] Dario Rossi and Giuseppe Rossini. 2012. On sizing CCN content stores by exploiting topological information. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*. IEEE, 280–285.

[12] Klaus Schneider and Beichuan Zhang. 2017. *How to Establish Loop-Free Multipath Routes in Named Data Networking*. Technical Report. NDN, Technical Report NDN-0044. https://named-data.net/publications/techreports/ndn-0044-1-loopfree-routing/

[13] Susmit Shannigrahi, Dan Massey, and Christos Papadopoulos. 2017. *Traceroute for Named Data Networking*. Technical Report. NDN, Technical Report NDN-0055, Revision 2. https://named-data.net/publications/techreports/ndn-0055-2-trace/

[14] The NFD Team. 2017. ndn-cxx: NDN C++ library with eXperimental eXtensions. (2017). https://named-data.net/doc/ndn-cxx/current/

[15] The NFD Team. 2017. NDNLPv2: Consumer-controlled forwarding. (2017). https://redmine.named-data.net/projects/nfd/wiki/NDNLPv2#Consumer-Controlled-Forwarding

[16] Edmund Yeh, Tracey Ho, Ying Cui, Michael Burd, Ran Liu, and Derek Leong. 2014. VIP: A framework for joint dynamic forwarding and caching in named data networks. In *Proceedings of the 1st international conference on Information-centric networking*. ACM, 117–126.

[17] Cheng Yi, Alexander Afanasyev, Lan Wang, Beichuan Zhang, and Lixia Zhang. 2012. Adaptive forwarding in named data networking. *ACM SIGCOMM computer communication review* 42, 3 (2012), 62–67.

[18] Yingdi Yu et al. 2014. *NDN Technical Memo: Naming Conventions*. Technical Report. NDN, Technical Report NDN-0022. https://named-data.net/publications/techreports/ndn-tr-22-ndn-memo-naming-conventions/