NDNREST: RESTful applications over Named Data networking

Harshavardhan Kadiyala, Rain Gu ,
Susmit Japhalekar , Ravneet Kaur

***Abstract*—Named Data Networking (NDN) is an information-centric networking architecture that can route data based on the content names instead of the source and destination address. NDN uses interest packets to request data from content providers and data packets to receive data. At first look, NDNs pure pull-based communication model seems to match the request-reply mechanics of REST interactions. However, modern RESTful communication involves passing client-side information and application state in requests. In this project, we implemented changes to the existing packet format of NDN to facilitate client-side information in interest packets; we will explore potential threats to the privacy of REST clients due to shared in-network cache and propose changes to the forwarding strategy which will help in effectively running modern RESTful applications over the NDN communication architecture. Our primary goal is to show NDN can support modern RESTful communication patterns effectively with minimal changes to existing NDN architecture.**

I. Introduction

In today's Internet, Web is a universal platform for various kinds of services, from familiar content browsing and media streaming to purpose-built applications hosted in browsers and in stand-alone agents like node applications. Many web applications are deployed over HTTP protocol [1] [2], which are based on a request/response model running on top of a TCP connection to the server. A client sends a request in the form of an HTTP request containing a URI [3], request meta-information, and possible body content. The server responds with an HTTP reply containing entity meta-information, and possible entity-body content [4].

Majority of these Web applications use a transactional paradigm known as Representational State Transfer (REST) [5]. REST improves scalability by distributing application state from servers to clients. A RESTful request is self-contained and carries all the information necessary for a service to process the request. Without the client-side context, a RESTful service may be inefficient or impaired, or may not be able to function at all. The familiar HTTP cookie is a simple form of distributed context, where a service uses the HTTP protocol to convey tokens, often opaque, to its clients. The tokens are typically unique to each client; this allows the service to associate multiple requests from a given client together. The cookie may carry client-side state directly or may be used as a reference to state held at the server.

The Named Data Networking (NDN) [6] project aims to develop a new Internet architecture that can capitalize on strengths and address weaknesses of the Internets current host-based, point-to-point communication architecture in order to naturally accommodate emerging patterns of communication.

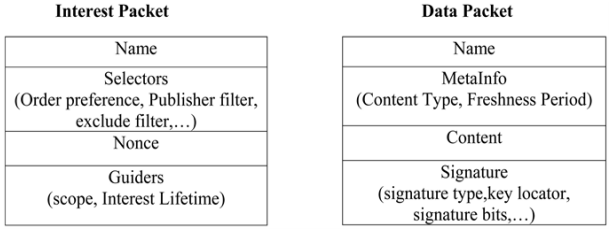


Fig. 1: NDN packets [7]

By naming data instead of their locations, NDN transforms data into a first-class entity. It also enables several radically scalable communication mechanisms such as automatic caching to optimize bandwidth.

In this report, we will present a new framework for running RESTful applications over NDN called NDNREST. We do not believe it is either desirable or efficient to simply reproduce HTTP bit-for-bit within an NDN protocol envelope. Rather, we explore the approach of enabling NDN-based Web clients to pass the necessary meta-information and application state to Web server applications that use NDN. We also address a potential problem like flow imbalance and potential privacy concerns that can be caused due to client data being added into interest packets.

In section III we will discuss the related work. Section IV, we will discuss details about the implementation of NDN over REST and the change in the protocol to support REST style communications, along with the potential drawbacks of our approach. We also address the privacy concerns of the data cache in NDN, which is implemented as the Content Store(CS), and proposed a user-initiated solution. Section V specifies our evaluation strategy and results.

II. Background

A. Named Routing

Communication in NDN [7] is driven by receivers i.e., data consumers, through the exchange of two types of packets: Interest and Data. Both types of packets carry a name that identifies a piece of data that can be transmitted in one Data packet (see Fig.1).

To carry out the Interest and Data packet forwarding functions, each NDN forwarder [7] maintains three data structures: a Pending Interest Table (PIT), a Forwarding Information Base (FIB), and a Content Store (CS) (see Fig.2), as well as a Forwarding Strategy module that determines whether, when and where to forward each Interest packet. The PIT stores all

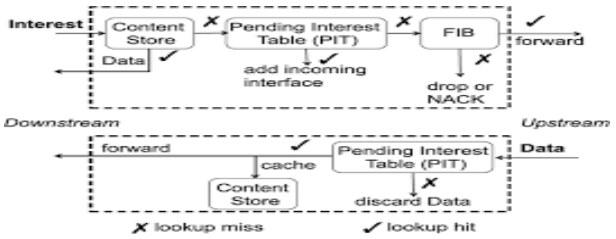


Fig. 2: Forwarding Process at an NDN Node [7]

the Interests that a router has forwarded but not satisfied yet. The Content Store is a temporary cache of Data packets the router has received. The forwarding information base (FIB) table use the names to route a request to next hop routers that are closer to the data provider by performing longest-prefix matching. The hierarchy also allows name resolution and data routing information to be gathered across similar data source names, which is the key point to enhance scalability and flexibility of the network architecture.

NLSR [8] is a routing protocol in NDN that populates NDNs routing Information Base. The main design goal of NLSR is to provide a routing protocol to populate NDNs FIB. NLSR calculates the routing table using link-state or hyperbolic routing.

III. Motivation and Related work

In this section, we will explore problems with implementing RESTful applications on current NDN network and existing solutions.

As we view the current state of RESTful communication. we see that RESTful clients have data to send in their requests, in the form of HTTP header meta-data and other application-specific RESTful states. But, in the NDN interest packets, all of the client-side context and meta-data associated with a request must be encoded in the Interest name field: no other field is present in the base NDN architecture [4]. Encoding client data inside name can cause adverse effects in other parts of NDN network. NDN forwarders use name to forward interest packet; if names contain a large amount of client data each forwarder has to process long names before passing interest packet to next hop; this can cause long delays in forwarders and can decrease overall network throughput.

Moreover, Many client requests are intimately bound to the client context data associated with them; the context and the request are carried together in the HTTP communication protocol messages. The client-specific data tends to make each request unique even when clients are accessing common resources or services. For example names such as https://www.example.com/¡data¿ are possible in HTTP, ¡data¿ field depend on client specific processing at runtime. On the other hand, NDN Content objects are immutable, and the object names are bound to fixed data. Services are not able to return different results based on client specific processing unless the clients use unique names in their requests.

In NDNREST, we transfer client data as a part of extended interest packet. This approach is simple as NDN network uses TYPE-LENGTH-DATA format [6] which can be dynamically changed at runtime to inject new information without affecting other parts of NDN network. Our approach will not produce any extra delays during propagation and supports dynamic names. However, it breaks one of the major assumptions of NDN network that interest packet has fixed size and size is less than the data packet size.

These assumptions in the NDN can cause flow imbalance in the network when a large amount of data is sent through interest packet. A RESTful application that required a larger client payload is quite a common scenario in general REST applications [4]. The data being transferred through interest packet can be large enough to make bandwidth accounting for Interests more important. We addressed this problem in NDNREST by adding interest data based flow balance to default forwarding strategy of NDN forwarder.

One of the key features of NDN is the router-side content caching provided by Content Store, it significantly increases the overall performance of communication in NDN by taking advantage of data packets that are independently named and verifiable at all ends. However, this in-network feature is also a potential leaking point of the privacy of data consumers. The previous works [9] have shown that there is a possibility for “simple and difficult-to-detect” timing attacks [10] through which an adversary can uncover the recent history of content retrieved by its neighboring users, which takes the Content Store in their shared NDN router as “oracles” using a designated sequence of probing.

There are several works [10], [9], [11] which have proposed potential solutions that are designed for mitigating RTT Timing Attack on the privacy of Content Store in NDN and other similar caching stores in Content Oriented Network. The solution can be classified into three groups: (1) naive solution, (2) probabilistic caching, (3) delay-based caching. There are also works [12], [13], [14] that demonstrates how to collaborate with other NDN routers to form Collaborative Caching, but in NDNREST, we only address the solutions within the scope of local NDN network.

A naive solution to this problem is not to cache any data packets in content stores. This solution provides perfect privacy for all data consumers and still fit into NDN architecture, however, the benefit from one of the main advantages of NDN that the data packets are perfect candidates for in-network caching is completely disappeared and degrades the overall performance. There are other solutions proposed that essentially results in same performance as no cache situation. In *Wait Before Reply* [10], all requests are delayed for an interval of time equivalent to the round trip time that they took when they are originally fetched from producers even if requested content is available in the Content Store.

The other group of solutions propose to introduce randomness into the caching policy based on the internal states of a router which is unknown to any users. This will provide probabilistic privacy for users hence the assumptions that the adversary can make is also probabilistic. Psaras et al. [11] proposed that the router could generate randomness based on its position in the forwarding path as well as the available space in the cache. Once a request is received, the decision whether the data packet should be cached is replied on the random

number generated. By using this solution, the adversary can not make any solid conclusion to the request history of its neighbours since only a randomly chosen subset of requests were cached. However, once the target name is cached by chance, the adversary can still exploit its neighbour's activities. Therefore, the trade-off between privacy and performance lies on the size of this cached subsets.

The last group of solutions to this problem advocates delaying the requests several times before they can be normally retrieved from Content Store. Acs et al.[10] proposed *Delay the First k*, In this approach, each request will be intentionally delayed k times, where k is a random number. The main advantage of this approach is that the popular names would unlikely to suffer from a long delay, and will not be randomly dropped from the cache. However, the algorithm that selects k should be carefully determined and possibly require the consideration of the popularity of names. The disadvantage of this approach is that the unpopular names suffer from longer latency.

In the second and third group of solutions, the authors intentionally weak the guarantee of achieving perfect cache privacy in order to improve the overall performance. The randomness of the fact that a certain name is cached is also applied to the conclusion an adversary can make, which will lower the chance of making a solid conclusion by the adversary. In our solution, we adopt the delay-based solution in which users can explicitly mark the name as private, and the subsequent public requests from other clients to the same name will result in a cache miss. This will only delay the request at most once since the public request to the name will make data associated with the name being cached in the content Store, this would prevent the adversary from distinguishing how this name is cached.

IV. Methodology

In this section, we will explore changes made to existing NDN infrastructure for running transactional and interactive REST- or Web-like applications over NDN. We focus on the key issue as we see it: how can servers obtain the client meta-data and context information that is associated with client requests?. We introduce new packet format, major advantages that come with adding a new field to interest packet, we will also discuss major disadvantages of running REST applications on NDN with new packet format and solutions to these potential problems.

A. Changes to existing Interest packet format

Existing interest packet is a fixed size packet which can serve NDN style producer-consumer communication pattern effectively. Current interest packet offers an efficient way to request a data object from producer using a named prefix. It has options to bypass the cache and specify interest validity. It contains a consumer's public key which can be used by the producer to encrypt requested data before transferring. All these features mentioned above can be used by a REST-based application and get benefits of enhanced security and named

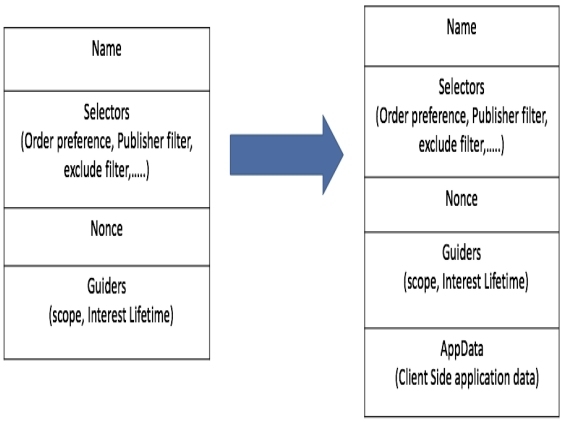


Fig. 3: Changed packet format

data transfer. However, to invoke requests which are updating (put) or creating a new resource (post) in a REST application requires the network to support the transfer of client-side data to a server. To facilitate this with existing NDN protocol, an application requires multiple handshakes and a requirement for a special name to be registered by the client, which helps server to contact back. These hand-shakes can be costly and result in performance degradation of applications.

In NDNREST, we introduced changes to Interest packet structure which will enable the efficient transfer of client-side data to a server. A new field APPDATA (see Fig 3) is added to interest to carry client information. APPDATA organized as a collection of bytes and can be accessed using NDNREST API calls for put and post. We made this changes in existing client-side library PyNDN2 [15] which is used by NDNREST IV-D for low-level NDN functionality.

The Interests with the APPDATA field has the following benefits:

1) The APPDATA field is opaque to routers. No special name components are present, and no special name processing takes place at routers. This new packet can be transferred using existing NDN network.
2) If a client Interest packets name a cacheable object, intermediate routers can perform normal CS processing and return the cached data.
3) The application context information travels directly with the Interests; the client context, name, and returning data remain bound together.
4) The application data can be transferred just once and no additional round-trips are needed.

Along with many above advantages mentioned above, we have seen some potential problems with adding unsolicited data to interest packets.

1) Adding client data in Interest packets increases Interest packet size, possibly substantially. The NDN property of flow balance assumes that Interest packets will generally be small compared to the corresponding Data packets. Pushing unsolicited data might compromise that property and cause flow imbalance in NDN network.

2) Different REST clients share common content store which can be a privacy issue.

In the remaining part of this section, we will discuss solutions to above problems.

B. Interest data based flow balancing in NFD

NDN's hop by hop flow balance is one of the 6 design principles of NDN [16]. It enables each node to control load over its links by deciding which link should be used to forward interest packet, router commits bandwidth for the returned data packet. By limiting the number of interest packets sent, each router and client node in the network control how much data it will receive through that link. We believe this fundamental flow balancing approach in NDN has an inherent flaw as they assume data returned for each interest packet has the same size which may not be the case for real applications.

By adding client-side data to interest packets, we are breaking initial assumptions that interest packets are of fixed size doesn't require flow balancing. Interest packets coming from NDNREST can contain large client-side data and are required to be considered when flow balance is done in NDN forwarder. In this project, we implemented a simple interest packet flow balance into NFD.

Algorithm 1 Pseudo code for Flow balance of Interest packets

```
1: function FINDELIGIBLENEXTHOP(Interest, outLink)
2:     NextHopsList = getNextHopsList(Interest.prefix)
3:     ForEach Hop in NextHopsList do
4:       FaceData[Hop.LinkId()] ≤ minimumdatasize
5:       minimum_data = FaceData[Hop.LinkId()]
6:       nextHop = Hop
7:     outlink = nextHop
8: end function
9: function AFTERRECEIVEINTEREST(inLink, Interest)
10:    findEligibleNextHop(Interest, outLink)
11:    FaceData[outLink.Id()] += Interest.size()
12: end function
13: function BEFORESATISFYINTEREST(InLink, Interest)
14:    FaceData[InLink.Id()] -= Interest.size()
15: end function
```

The algorithm 1 is designed to keep track of data carried on every active request sent through each link and identify a link with lowest active interest data. In NDN every data packet follows the same path as interest packet. So we can keep track of the total size of interest data transferred through a link for which data producer not yet sent the data packet back. "afterReceiveInterest" method gets called every time a new interest packet arrives at the NDN forwarder, In this callback, we will find next hop with lowest interest data transferred for which data packet not yet arrived using "findEligibleNextHop" method. Once we found the link to be used to transfer Interest packet, it will be forwarded in that link and FaceData dictionary is used to keep track of link id and amount of data transferred. When the data packet arrives at the hop "beforeSatisfyInterest" will be called and the interest data will be removed from the link in which data packet arrived. Our implementation is evaluated and is performing better than existing flow control at balancing interest packet load between links.

C. Cache Privacy Concern

To provide privacy for clients using content Store in NDN router that is shared by multiple users locally, we propose the solution that delays at most once when there is potential private leak regarding the name requested. The protection scheme does not modify any caching policy or mechanism in current NDN router's design. Compared to the other previous work, our solution requires user as the initiator to initiate its own privacy concern by explicitly marking the request as a private one.

1) Threat Model: The interesting point regarding this privacy concern in local NDN network with shared NDN router is that the adversary is relatively powerless but still able to make accurate assumptions about its neighbours' recent request history. To be specific about the term " neighbour", it refers to all the users that share a particular NDN router as their first hop in NDN network. An adversary in this model has no additional access to any part of the network, such as the hosts of honest users and NDN router, and can send Interest packet with any desire names. In other words, the adversary is not able to monitor, alter, or block the traffic in any part of the local network. The next section shows how exactly an adversary is able to learn about useful information only by measuring several RTTs for different requests.

2) RTT Timing Attack: The core of this Timing Attack is based on two facts that (1) all data packets received by an NDN router will be cached in Content Store and (2) the latency of fetching a cached content in a router is significantly longer compared to fetching the ones that are not cached. Therefore, only by measuring and comparing RTTs of fetching some content, an adversary can determine if a particular content has been recently fetched by the other users that share this NDN router as the first hop with the adversary. Let *target_name* being the name that the adversary wants to confirm that has been recently fetched by any of its neighbors, and *cached_name* being any known cached names in the router. An adversary A can perform the attack as follow:

1) sends Interest packet with *target_name*, and records RTT_{target_name}.
2) sends Interest packet with *cached_name*, and records RTT_{cached_name}.
3) sends Interest packet with *non_cached_name*, and records $RTT_{non\_cached\_name}$.

Now IF $(RTT_{target\_name} - RTT_{cached\_name}) < \epsilon$ for negligible ϵ AND $(RTT_{non\_cached\_name} > RTT_{cached\_name})$, the previous work has shown that A can conclude target_name has been recently fetched by any of its neighbors with high probability. There is also works show that the cache replacement policy has only a small impact on the success

rate of this Timing Attacks.

3) User-Initiated Delay Once Protocol: In our solution User-Initiated Delay-Once**(UIDO)**, we add two data structures **PTable** and **PubList** to maintain the privacy information and modify the Interest Packet dispatching procedure of the names for detecting private request. Any user now can choose to mark the name as a private request with a random nonce. The nonce associated with this private name should only be known by the user-generated.

To maintain the private request information, an extra data structure called **PTable** is used to stores records in the format as (name, nonce, delayed) in each entry. For example, when a user issues an Interest Packet with name "/a$private+1", the possible entry in **PTable** would be ("/a/", "1", false). Multiple entries with the same name but different nonce are allowed to stay in the meantime. An entry in **PTable** will be removed only when a public request with the same name("/a/" in this case) is received by the router. The boolean *delayed* is used to check if this private request has already been delayed for once when there are multiple entries with same name existing in **PTable**.

Another data structure named **PubList** is used to keep track of the names that are recently being queried through public requests, with the format of (name, timeToLive). Each entry in **PubList** is associated with an attribute(*timeToLive*) to indicate the liveness of this name as being public one. In other words, for every name in a public request, the name will be marked as public for a time interval with random length, and timeToLive will reset to a new value each time the name is requested if the name was in **PubList**. The router also needs to periodically check if any of the entries times out in the **PubList** and removes them from **PubList**.

Putting them all together, the protocol for a router to dispatch and process an Interest packet with name **n** now becomes as follows:

If n is not marked as private, the router:

- Checks if n is in **PubList**:
- YES: updates timeToLive of n in **PubList** and proceeds.
- NO: checks if there are entries in **PTable** with name n:
  - YES: removes all entries with name n in **PTable**. Adds n into **PubList**. Delay once.
  - NO: adds n into **PubList** and proceeds.

If n is marked as private with nonce a, the router:

- Checks if n is in **PubList**:
- YES: considers this request as public one and proceeds.
- NO: checks if **PTable** have any entries with name n:
  - YES: checks if any of these entries have nonce a.
    - YES: proceeds.
    - NO: insert (n, a, true) into **PTable**. Delay once.
  - NO: insert (n, a, true) into **PTable**, and proceeds.

Let's re-exam the RTT Timing Attack with this new protocol. The user that wants to request the *target_name* can mark it as a private request. The adversary's request with this

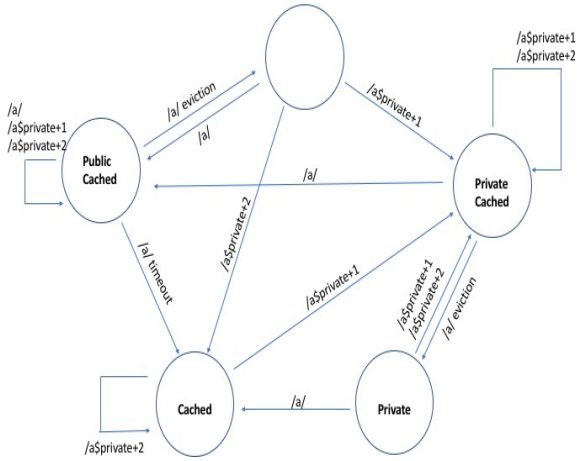


Fig. 4: State Machine of A private request "/a/private+1"

target_name will not result in a cache hit since this public request would have to delay once. Instead, the adversary can also send a private request to *target_name*, which will also be delayed since the nonce that adversary provides should not match the nonce stored in **PTable**. In either case, the following requests to this *target_name* may result in cache hit if it is not evicted yet, but the adversary cannot conclude that *target_name* is requested by the other users since the adversary also had requested for this name and it could be a reason for the cache hit.

However, there is still a way for the adversary to learn about the activities of the other users. In NDN router, there are no explicit limitations on the number of requests that each user can send, so any user is able to flood requests with a different name until all old entries in Content Store are evicted and replaced with new ones. Consider the following attack. Before user sends the request for *target_name*, the adversary can send a public request to *target_name*, which will add *target_name* to **PubList**. Then the adversary flood requests with other names to make sure *target_name* is no longer in the Content Store. Now the user sends the private request to *target_name*, and since *target_name* may still be in **PubList**, this request is marked as public, and cached in Content Store. Now if adversary requests again for the *target_name* and get cache hit, it can conclude that the user has just requested for *target_name*.

To address this problem, the protocol now adapts as follow: when a name in Content Store is being evicted, the router removes the entry with this name in **PubList**. Now the full protocol can be described as the FSM in Fig 4, which demonstrates the state change with respect to a private request "/a$private+1". The states are Public, Private, and Cached, in which Public and Private are mutually exclusive. The empty states can be either the name is never requested yet or just recently evicted from Content Store and removed from **PubList** as well. Public state means the "/a/" is in **PubList**, Private state means "/a$private+1" is in **PTable**, and Cached means "/a/" is in Content Store. The name "/a$private+2" is only one instance of a private request that has name "/a/" but with a different nonce.

4) UIDO Limitations: The privacy concerns of Content Store is especially high when the population that shares a certain NDN router is minimal (only one user and one adversary). Our protocol is able to protect user's privacy in such case and with the experimental results shown that the overall performance is not significantly degraded with this privacy protection. As the number of users increases when push this privacy concerns to global scope, the larger size of anonymity set naturally provide another class of the protection of privacy. Even though we only target at protecting the cache privacy in a local scope, the protocol should be also working in the global scope in every NDN router since the local case with a minimal number of users is the worst case for privacy protection.

The other limitation besides the performance degrade is the extra information that the router needs to maintain for protecting privacy such as the **PTable** and **PubList**. Even though the size of each entry is not very large but there is no limitation or restriction to prevent the flooding of private requests that may significantly occupy the space resource in NDN router which is quite limited. In the future work, it's possible to add time-based or popularity-based eviction policy in **PTable** as well to save space in the router. Another related concern is the privacy marker "/private+nonce", which will always increase the Interest packet size by some bytes. The naming issue related to NDN architecture is getting worse in this design. Some may argue that we could modify the internal layout of Interest Packet by adding a private bit to indicate whether it is a private request, however, this approach would save few bytes in name but also require the adjustment in NDN router when dispatching the packets received.

As for User-Initiated part, this design can also be viewed as a limitation compared to the other previous solutions that the NDN router makes all decision about privacy protection by considering all requests are potential privacy required requests. However, in our opinion, the privacy concerns vary from user to user, thus not all request needs to be considered as private one and this decision would reduce the amount of work on processing the private requests. In our solution, even though the user now needs to make a decision about its own privacy, the amount of extra work is minimal compared to make a normal public request in the original design.

D. Client Interface for NDNREST

The interface design for NDNREST is similar to that of Flask [17], a micro-framework for Python based REST applications. In this section, we will give an introduction to interfaces NDNREST provides.

In REST every resource is uniquely addressable using a uniform and minimal set of commands that is GET, POST, PUT DELETE.

1) GET method is used to read or retrieve a representation of a resource.
2) POST is most-often utilized to create new resources.
3) PUT is most-often utilized for update capabilities, PUTing to a known resource URI with the request body containing the newly-updated representation of the original resource.
4) DELETE is used to delete a resource identified by a URI

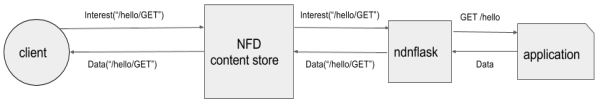


Fig. 5: Mechanism of Rest over NDN

NDNREST uses PyNDN2 [15] library to access low-level functionality required to talk to NDN local hub. Original PyNDN2 is updated with new packet format for NDN Interest and used for NDNREST's put and post functionality.

```
from ndnflask import Flask
app = Flask(NFD)

@app.route("/hello", methods=['GET'])
def hello():
    return "Hello World!"

app.run()
```

NDNREST client library creates a persistent server face interface with NDN forwarding daemon and registers itself with the local hub when an application comes up (refer Fig 5). It creates an identity and generates RSAKeyPair. It exchanges public key required to sign packets with local hub. NDNREST receives REST commands from the application and registers prefixes using function "route". With client-side data support, NDNREST can support POST and PUT methods using the same interface. In these methods, request body contains a newly-created or newly-updated representation of the resource. NDNREST client library provides various a cache control features for a client to control in-network cache. It helps the client to specify cache specific parameters like freshness which is used by the content store to determine the validity and it also helps clients to maintain private caches in the content store.

We tested compatibility of NDNREST with existing Flask applications in GitHub and able to run them on NDN network with no changes to existing application code.

1) Cinema microservice [18]
2) Blockchain application [19]
3) Scikit learn models [20]
4) ChatterBot application [21]

V. Evaluation

A. Evaluation goals

1) The Performance impact of NDNREST.
2) The Effectiveness of Interest packet based flow balance and decreasing load on links.
3) The Performance impact of private cache on performance.

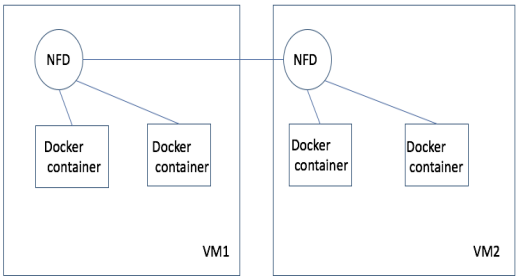


Fig. 6: Test setup

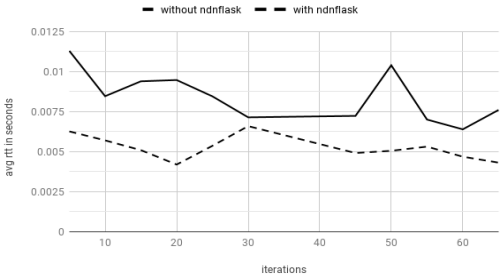
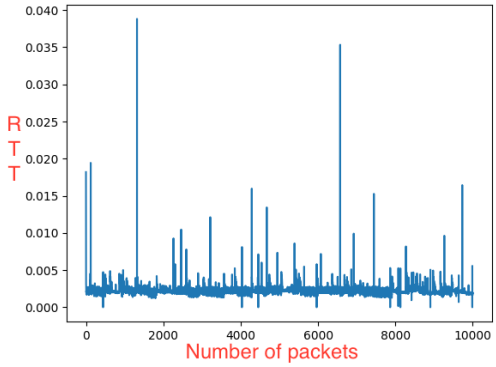
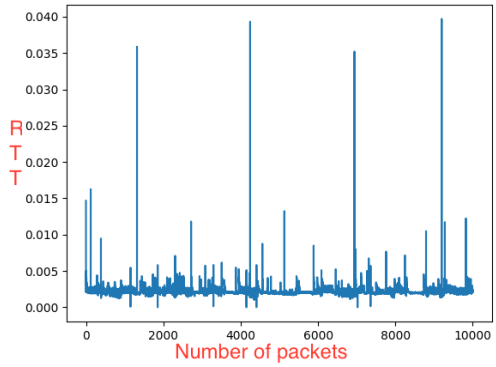


Fig. 7: Average round-trip time



(a) With New Interest packet flow balance.



(b) Without Interest packet flow balance.

Fig. 8: Performance of Flow balance.

B. Test Setup

We have two VMs deployed on the same physical machine. In each VM, NDNREST applications are deployed as a docker container and each VM has an NDN router deployed as docker container to connect to NDN network.

C. Performance impact of NDNREST

We also evaluated NDNREST round-trip times and compared it with round-trip times of same data transfer without using NDNREST. The application that doesn't use NDNREST uses names to transfer client-side data. We compared round-trip time(RTT) of Interest packet with NDNFlask and without NDNFlask. We measured the average round-trip time for iterations 5 to 65. Fig 7 shows the graph of average round-trip time for Interest packet with NDNFlask and without NDNFlask. Average round-trip time taken by normal Interest packet to get data was 9ms while with NDNFlask it took 5ms, reduction of delay by 44%. We were able to achieve this reduction because of a decrease in additional processing delays of long names in each forwarder.

D. Evaluation of new Flow balance algorithm

NDN router deployed in each VM is replaced with our NDN router with new interest-based flow balance. Several instances of server application are created on VM2, and each instance registers itself with deployed NDN router creating multiple links for the same name. We used NDNREST client application deployed on VM1 to send 10000 put requests each containing 360 BYTES to the server creating congestion in the network.

We computed round trip time for each request-reply and plotted a graph with original NDN flow balance and with new flow balance. The figure (ref 8b) has more spikes than figure (ref 8a) indicating that new NDN interest flow algorithm helps in spreading the flow evenly and decreasing link overload.

E. Evaluation of private cache

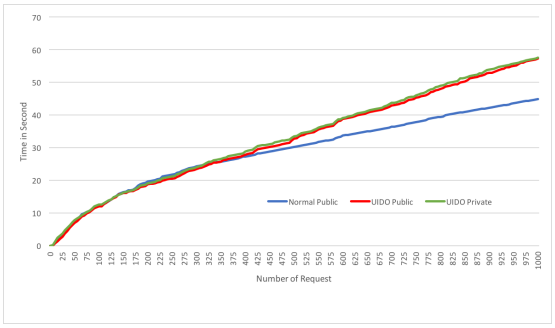
Due to the extra data structures and protocol procedures added to the NDN routers, the performance is expected to be decreased compared to the original NDN routers. The purpose of this experiment is not to prove the correctness of our UIDO privacy protocol, but to present what is the performance impact that comes with the new privacy protection. To test the performance of our UIDO privacy protocol, we design the experiment to measure the round-trip times by making 1000 requests with different patterns of names belonging to three categories: (1) Normal Public:Original NDN router, (2)UIDO Public: UIDO NDN router with all requests as public requests, and (3)UIDO Private: UIDO NDN router with all request as private ones. The reason we are not measuring the cache hit rate is that with UIDO protocol, the cache hit does not change in any situation. The intentional delay in some cases does not change the status of entries in private content Store, but the only request has to wait for a short interval of time. Therefore, measuring the overall RTTs is more reasonable to demonstrate the performance impact of the new router.

Since our privacy protocol is targeted at resolving the privacy issue in the local scope, only one local NDN router

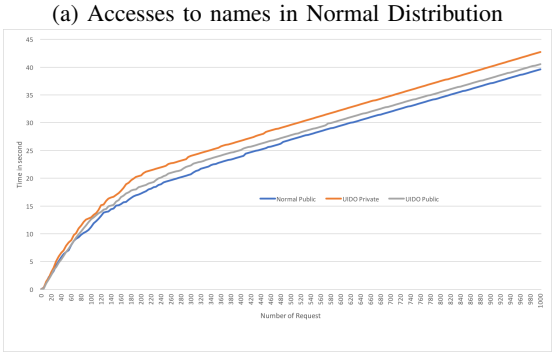
is required and no further hops to other NDN router on the way to the echo server. To simulate the other class of latency such as propagation delay, and queuing delay outside the local scope, we add the randomly varying latency on server side range from 0.0 to 0.25 sec. The server also has registered all possible names and connected to the local NDN router. In the local NDN network, we also use only one NDNREST client as the user instead of creating multiple instances, because, in the aspect of the new NDN router, the potential latency is only associated with names requested which are independent of sources. As for the private requests, all of them will choose nonce randomly to represent multiple users.

We measure the RTTs with two access patterns of names: (1) multiple names follow Normal Distribution on their popularity, (2) Access to multiple names in Uniform Distribution, and (3) all accesses to one single name. The first and second pattern represents the average case that popular names tend to be requested much more frequently to simulate the common access patterns in the real world. The third pattern is the best case for the original NDN router where nearly every request will result in a cache hit, but it is also the worst case for UIDO NDN router sending all private requests with the different nonce and all of them will be delayed due to protection. This scenario is designed to find the upper bound of the performance of our UIDO router.

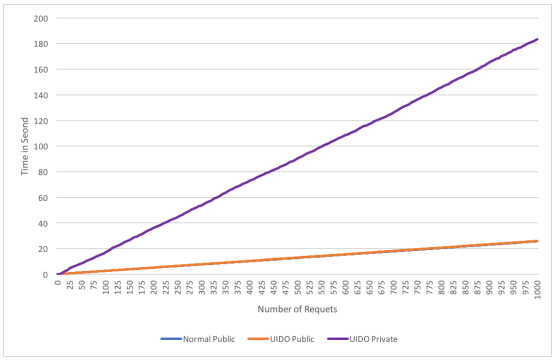
The results are shown in Fig 9(a) demonstrates that on the accesses patterns in Normal Distribution, the performance with UIDO regardless of public or private degrades compared to the original NDN router especially when the number of requests getting bigger. The results Fig 9(b) has also shown with UIDO the performance is not as good as the original one but not asymptotically significant. For both patterns, the performance difference between UIDO Public and Private is also about the same. In Fig 9(c), which is the worst case for our UIDO Private, the performance of UIDO Private is much worse compared to the other two candidates which are nearly overlapped. These results met our expectation that with UIDO, the overall performance on average cases is decreased but not significantly. The performance on a worst case scenario can be considered same as not using any cache at all and the penalty is proportional to the latency retrieving the Data packet from the server since each request would trigger the delay.



(a) Accesses to names in Normal Distribution



(b) Accesses to names in Uniform Distribution



(c) Accesses to A Single Name

Fig. 9: Performance impact with UIDO

VI. Conclusion

In this report, we proposed a new NDNREST client library with changed interest packet format and also improvements to existing NDN forwarder which will help current HTTP based RESTful applications to run efficiently and securely on NDN network. We also proposed a privacy protection protocol that minimizes the length of latency while keeping relatively same performance in average case as using the original NDN router. We believe that NDN's fixable architecture helped these changes to be compatible with existing infrastructure and helped NDNREST applications to co-exist with other pure NDN applications. We envision that in future NDN network, every class of applications will have its own extensions to default protocol and forwarding strategy yet co-exist in the same core NDN network. This will enable a low-cost transition to NDN network as it will be easy to make interfaces backwards compatible with existing HTTP or TCP based infrastructure, ultimately contributing to NDN's success as a mainstream content delivery network.

References

[1] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext transfer protocol–http/1.0," Tech. Rep., 1996.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," Tech. Rep., 1999.

[3] L. Masinter, T. Berners-Lee, and R. T. Fielding, "Uniform resource identifier (uri): Generic syntax," 2005.

[4] I. Moiseenko, M. Stapp, and D. Oran, "Communication patterns for web interaction in named data networking," in *Proceedings of the 1st international conference on Information-centric networking*. ACM, 2014, pp. 87–96.

[5] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[6] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[7] "NSF Named Data Networking project online," http://www.named-data.net/, accessed: 2017-10-08.

[8] 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*. ACM, 2013, pp. 15–20.

[9] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*, ser. CCS '00. New York, NY, USA: ACM, 2000, pp. 25–32. [Online]. Available: http://doi.acm.org/10.1145/352600.352606

[10] G. Acs, M. Conti, P. Gasti, C. Ghali, and G. Tsudik, "Cache privacy in named-data networking," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. IEEE, 2013, pp. 41–51.

[11] I. Psaras, W. K. Chai, and G. Pavlou, "Probabilistic in-network caching for information-centric networks," in *Proceedings of the second edition of the ICN workshop on Information-centric networking*. ACM, 2012, pp. 55–60.

[12] N. Laoutaris, S. Syntila, and I. Stavrakakis, "Meta algorithms for hierarchical web caches," in *Performance, Computing, and Communications, 2004 IEEE International Conference on*. IEEE, 2004, pp. 445–452.

[13] D. Wessels, "Configuring hierarchical squid caches," *National Laboratory for Advanced Network Research*, 1997.

[14] K. Claffy and D. Wessels, "Internet caching protocol (icp), version2," *IETF RFC2186*, 1997.

[15] "NDN client library with TLV wire format support in native Python, https://github.com/named-data/pyndn2, accessed: 2017-11-15."

[16] "ndn design principles, http://named-data.net/project/ndn-design-principles, accessed: 2017-11-15."

[17] "a micro-framework for REST based Python applications, http://flask.pocoo.org, accessed: 2017-11-15."

[18] "Cinema microservice, https://github.com/umermansoor/microservices, accessed: 2017-11-15."

[19] "Blockchain application, https://github.com/dvf/blockchain, accessed: 2017-11-15."

[20] "Scikit learn models, https://github.com/amirziai/sklearnflask, accessed: 2017-11-15."

[21] "ChatterBot, https://github.com/chamkank/flask-chatterbot, accessed: 2017-11-15."