Implementation and Evaluation of a Controller-based Forwarding Scheme for NDN

Elian Aubry*†, Thomas Silverston‡ and Isabelle Chrisment*†
* Université de Lorraine, LORIA CNRS UMR7503, 54506 Vandoeuvre-lès-Nancy, France
† Inria Nancy – Grand Est, 54600 Villers-lès-Nancy, France
‡ NICT, National Institute of Information and Communications Technology, Tokyo, Japan
Email: elian.aubry@loria.fr, thomas@nict.go.jp, isabelle.chrisment@loria.fr

Abstract*—Named-Data Networking (NDN) is a novel clean-slate architecture for Future Internet. It has been designed to take into account a new use of the Internet and especially accessing content for a large number of users, and it integrates several features such as in-network caching, security or multipath. As NDN relies on content names instead of host address, it cannot rely on traditional Internet routing, and it is therefore essential to propose a routing scheme adapted for NDN. To this end, in this paper, we present SRSC, our SDN-based Routing Scheme for CCN/NDN and its implementation. SRSC relies on the SDN paradigm.A controller is responsible to forward decisions and to set up rules into NDN nodes. We implement SRSC into NDNx and we also deploy an NDN testbed within a virtual environment and real ISP topology in order to evaluate the performances of our proposal with real-world experiments. We demonstrate the feasibility of SRSC and its ability to forward *Interest* messages in a fully deployed NDN environment, while keeping low overhead and computation time and high caching performances.

I. Introduction

Named Data Networking (NDN) [1] is a new networking architecture proposed to address today's Internet requirements. Indeed, the Internet is now mostly used to access content [2], and the current Internet architecture has not been designed for this purpose. The NDN clean-slate architecture introduces a host-to-content communication paradigm and relies on new features such as in-network caching, data encryption, mobility support or multipath, and has attracted a huge research community, providing novel mechanisms for caching [3], congestion control [4], or deployment [5].

The NDN architecture is a cache network and nodes can store content for future requests. Content is therefore spread into the network closer to users. The NDN protocol relies on two messages: *Interest* and *Data*, which are the request and reply respectively. When a user requests a content, it sends an *Interest* into the network, which is forwarded up to a node having the content. However, the NDN architecture does not have any scheme to forward *Interest* messages in the network. NDN first relies on flooding to forward *Interest*, wasting network resources and preventing its deployment at the Internet scale. Routing schemes such as NLSR [6] have therefore been proposed but it does not fully benefit from the potential of caching features.

In this paper, we present our implementation of SRSC, a SDN-based Routing Scheme for CCN/NDN. SRSC is a forwarding scheme for NDN, whose originality is to rely on the SDN paradigm and operate in a native NDN environment (i.e., without TCP/IP). SDN allows decoupling the control and data plane, and NDN nodes become forwarding devices only and the controller is responsible to compute the routing decision.

We deploy SRSC into a virtual environment through a new Docker [7] testbed. A virtual testbed is used to evaluate our routing scheme at large scale within a real ISP topology and demonstrate the efficiency of our proposal though real-world experiments. The contributions of this paper are threefold:

- We implement our routing scheme SRSC into NDNx;
- We design a new virtual NDN testbed easily deployable on all platforms;
- We evaluate SRSC through extensive experiments in areal testbed environment.

Our controlled-based routing scheme succeeds in forwarding *Interests* to closest Content Stores and is a potential candidate for routing in NDN. The experiments show that the SRSC controller exhibits a low computation time, while at the same time traffic overhead of SRSC is limited to 18% and caching performances remain high-level with 47% of Cache Hits in our NDN testbed.

The rest of the paper is organized as follows. Section II provides a general overview of the NDN architecture and Section III describes the implementation of SRSC into NDNx. Section IV presents our virtual testbed with Docker and the experimental deployment. The SRSC implementation is evaluated through real-world experiments in Section V. Section VI discusses the implementation, and Section VII surveys the related work. Section VIII concludes the paper and presents some future works.

II. Named-Data Networking Overview

The Internet host-to-host communication paradigm based on TCP/IP is no longer adapted to massive content distribution at the Internet scale. Users are interested by content and not its location and the NDN architecture has been proposed to overcome these limitations. The NDN architecture relies on host-to-content paradigm where a packet addresses a content names and not an host location. A content name can for instance be described with a URL such as /netflix/tv/hd/show.avi. NDN is also a caching network and each node can store content for further requests. Thus, content is spread into the network,

Messages	Prefixes	Steps	Phases
InterestBoot	/allNodes/controller/<IdController>	Controller Announcement	Bootstrapping
InterestDiscover	/neighbours/node/<IdNode>	Neighbour Discovery	Bootstrapping
InterestNeighbours	/controller/<IdController>/node/<IdNode>/<Neighbours>	Topology Discovery	Bootstrapping
InterestNewContent	/controller/<IdController>/incache/node/<IdNode>/data/<Data>	Content Announcement	Forwarding/Bootstrapping
InterestPath	/controller/<IdController>/path/node/<IdNode>/data/<Data>	Rule Request	Forwarding
InterestCreatePath	/node/<IdNextNode1>/install/<IdNextNode2>/.../<IdNode2>/data/<Data>	Path Establishment	Forwarding
InterestDeleteContent	/controller/<IdController>/deletecache/node/<IdNode>/data/<Data>	Content Removal	Forwarding

TABLE I: SRSC Control *Interest* defined for communication between nodes and controllers. Control *Data* share the same format and carry requested information.

closer from users, reducing de-facto the delay to users and enhancing the users' Quality of Experience.

For a brief history about the NDN architecture, the Content-Centric Networking (CCN) architecture has been proposed by PARC in 2009. Then, it has forked into two different projects due to copyright issues [8]: the NSF-funded NDN project [1] and the CCN project (PARC) [9]. The two architectures shared the same common baseline up to 2013 (CCNx 0.8) and then evolved independently. The open source NDNx implementation has now attracted a huge part of the research community. The NDN/CCN harmonization [10] is however an ongoing topic at ICN Research group (IRTF).

The NDN protocol relies on two messages: *Interest* and *Data*; *Interest* is the message to request content, and *Data* is the reply message transporting the data. A NDN node's architecture is composed of three components: (i) the Content Store (CS), a memory for storing content; (ii) the Forward Information Base (FIB), a table with entries (name prefixes and network interfaces) to forward *Interest* (e.g., routing table), and (iii) the Pending Interest Table (PIT), a table keeping track on which interfaces to forward the data back to requester.

When a user requests a content, it sends an *Interest* into the network, which is forwarded up to a node having the content. This node can be the origin content publisher as well as any other node that owns the content. When a node receives an *Interest*, it checks if the content is in its cache (CS), and if so, it sends the *Data* message through the same interface. *Data* follows the reverse path up to the requester, and each node along the delivery path can decide to store the content according to caching strategy for further requests. Otherwise, it looks up for a matching entry in its FIB to forward the *Interest* into the network.

The NDN architecture does not provide any routing mechanisms to forward *Interest* and it floods the network, wasting resources and preventing its deployment at the Internet scale. We have therefore propose SRSC [11], a SDN-based Routing Scheme for CCN/NDN, as novel mechanism to populate FIB in order to forward efficiently *Interest* into NDN network.

III. SDN-based Routing Scheme for CCN/NDN

The SRSC protocol relies on the SDN paradigm and uses a controller, which is in charge of the forwarding decisions. Basically, upon receiving an *Interest*, a node requests the forwarding rules to the controller, which sends back the path toward the content. The node will therefore update its FIB and forward the *Interest* through the corresponding interface. The controller is responsible for a slice of network – such as an Autonomous System in today's Internet – and has to discover the topology, nodes and content in the network.

In the following section, we describe (i) the SRSC Control Messages exchanged between the controller and nodes, and (ii) the algorithms we have implemented in NDNx at the node side and controller side. The NDN Forwarding Daemon process (NFD), which is responsible to forward messages, was modified to use SRSC. All other NDNx components are unchanged (PIT, FIB, CS).

SRSC relies on two distinct phases to achieve its goal: Bootstrapping and Forwarding. Bootstrapping is for binding the nodes to their controller; the controller therefore learns the network topology. Forwarding is the essential step to forward *Interest*, i.e., requesting rules to controller, adding rules into nodes' FIB and forwarding *Interest* to the nearest Content Stores.

A. SRSC Control Messages

In SRSC, nodes contact a controller to obtain forwarding information for content. The controller has to acquire the entire knowledge of the topology, associated nodes and content, while nodes have to discover their controller and neighborhood, and announce the available content in their Content Store. In SRSC, controller and nodes communicate using NDN messages. To this end, we define specific control messages (i.e., signaling) to allow nodes to communicate with controller and exchange information with each other. These control messages are specific NDN *Interest* and *Data* messages with reserved prefix names, which are presented in Table I. Some *Control Data* messages reply to *Control Interest* as in NDN communication (e.g., *InterestDiscover* and *InterestPath*).

The prefixes of the control messages use fields carrying valuable information such as <IdController> and <IdNode>, i.e., identity of controller and nodes: (e.g., serial number). <Neighbours> is a list of couples <IdNode>/<IdController> to identify the slice of network (i.e., a single domain managed by the same controller). <Data> refers to the content name (i.e., prefix) requested by users. In our control messages, *Control Data* does not transport effective data, but transports instead forwarding information (e.g., forwarding rules) or information about neighbours' identities.

B. SRSC Algorithms

The SRSC protocol has been implemented into two distinct phases: Bootstrapping and Forwarding.

Bootstrapping: This is the necessary phase for both controller and nodes. As presented in Algorithm 1, in order to learn the network topology, the controller broadcasts an *InterestBoot* message (l 1). Then, all nodes associated with it respond with an *InterestNeighbours*, containing a list of all their neighbours and controllers. The controller can then update information about the topology (l 3), and it can detect border nodes, which are nodes from its neighbours' list that are bound to another controller (l 5). Finally, for each received *InterestNewContent*, the controller updates its content's location table (l 9).

On the other side, nodes perform Algorithm 2. Upon receiving the *InterestBoot*, nodes associate themselves with the controller and broadcast the message if they are not already bound to any other controller (l 5), else they drop it. They also broadcast an *InterestDiscover* to discover their neighborhood (l 7), and reply to each other with a *Control Data* carrying their identity and their associated controller (l 15). Each node sends their neighborhood information to the controller with *InterestNeighbours* (l 12). During Bootstrapping, nodes also announce their content to controller (*InterestNewContent*) (l 18).

Algorithm 1 SRSC Bootstrapping for Controller

```
//Start Bootstrapping
1. Controller.broadcast(InterestBoot)
2. For All InterestNeighbours received from Node Ni do:
3.   Controller.Update(Topology)
4.   If Ni.GetControllerID() != Controller do:
5.     Controller.AddBorderNode(Ni)
6.   End If
7. End For

//Content Tracking
For All InterestNewContent received from Node Ni do:
9.   Controller.AddNewContent(NewContent, Ni)
10. End For
```

Forwarding: This is the crucial phase of SRSC, as nodes will learn the forwarding rules toward the content. From a node's point of view, the forwarding phase is described in Algorithm 3. While receiving an *Interest*, a node checks if the content is already in its Content Store (l 2), if there is a PIT entry for the same content (l 4) and if a FIB entry already exists (l 6). In other cases, the node will request the forwarding rule from the controller (l 10). While receiving the forwarding rule from the controller (*DataPath*), a node gets the entire path to the content. It first extracts the next hop to forward the *Interest* (l 16), updates its FIB and sends *InterestCreatePath* (l 19) to the next hop in order to install forwarding rules in the following nodes on the path. *InterestCreathPath* will be forwarded hop by hop so that each node on the path does not have to request the controller again and can immediately update its FIB, reducing the number of control messages (l 22 to 28). At the end of the process, the first node has therefore established a route to the destination and can forward the original *Interest* message along the path to the Content Store.

Algorithm 2 SRSC Bootstrapping for Node

```
//Topology discovery
1. Node receives InterestBoot from Controller
2. If Node.AlreadyBindToController() do:
3.   Node.drop(InterestBoot)
4. Else
5.   Node.BindTo(Controller)
6.   Node.broadcast(InterestBoot)
7.   Node.Broadcast(InterestDiscover)
8. While Node.NbrDataDiscoverReceived()!=Node.NbrFaces()
do:
9.     Node receives DataDiscover from Neighbour
10.    Node.AddInNeighboursList(Neighbour)
11.  End While
12.  Node.Send(Controller,InterestNeighbours(NeighboursList))
13. End If
14. For All InterestDiscover received from Node Ni do:
15.   Node.Send(Ni, DataDiscover(Node,Controller))
16. End For

//Content Location
17. For All NewContent available do:
18.   Node.Send(Controller,InterestNewContent(Node,NewContent))
19. End For
```

Algorithm 4 details the forwarding process on the controller side. From the bootstrapping phase, controller already knows the topology and available content. Controller extracts prefix names from requests (*InterestPath*) and selects the more appropriate nodes possessing the content (l 3, l 4). Indeed, the same content can be replicated in several Content Stores and in our current implementation, the controller chooses the shortest path to the destination (SRSC chooses the less costly path, but as we attribute the same weight on all links, the less costly is also the shortest). If the content is not available in the network, the controller selects a border node to forward the *Interest* and reach another network. Another controller will therefore be responsible to get the content using the same process. Note that one could have a controller sending multiple paths for a same content but we keep this case for future improvement.

IV. Docker Experimental Testbed

In order to evaluate our implementation of SRSC, we deploy a virtual experiment testbed. Virtualization allows testing the real implementation of a protocol and its behavior in real-world scenarios. It is different from simulation experiments where simulations evaluate a simplified prototype without taking into account numerous effects arising in real-world

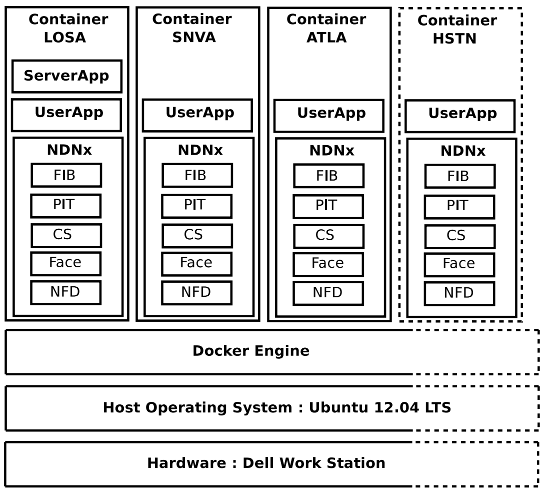

Fig. 1: Architecture of the Docker virtual testbed

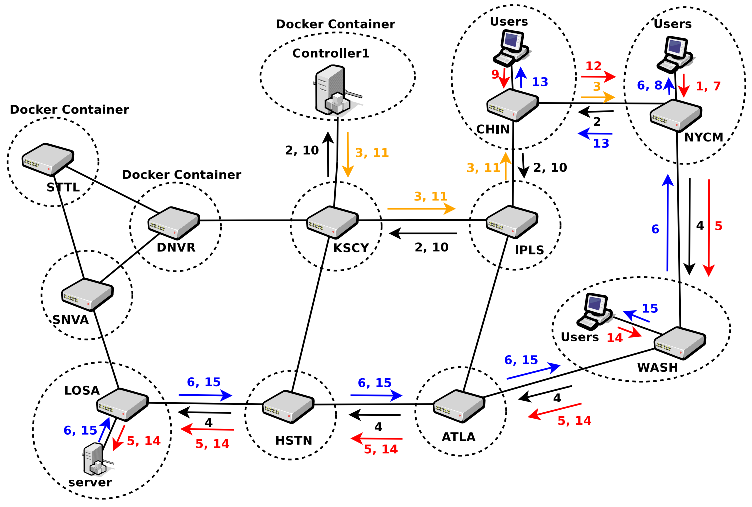

Fig. 2: Docker Experiment testbed: SRSC is implemented into NDNx and deployed on Abilene Topology. Four routing scenarios. **Full Routing Scenario (FRS):** messages 1 to 6; **Cache Scenario (CS):** messages 6 and 7; **Close Routing Scenario (CRS):** messages 9 to 13; **Direct Forwarding Scenario (DFS):** messages 14 and 15.

Algorithm 3 SRSC Forwarding for Node

```
//Send Request to Controller
1. For All Interest(Content) received do:
2.  If Node.IsInCS(Content) then
3.    Node.Send(Content)
4.  Else If Node.AlreadyInPIT(Content) do:
5.      Node.UpdateEntry(PIT,Content)
6.     Else If Node.AlreadyInFIB(Content) do:
7.         Node.Send(FIB,Content)
8.         Node.AddEntry(PIT,Content)
9.       Else
10.        Node.Send(Controller,InterestPath(Node,Content))
11.       End If
12.    End If
13.  End If
14. End For

//Receive Path from Controller
15. For All DataPath received from Controller do:
16.  NextNode ← Node.ExtractNextNode(DataPath)
17.  Content ← NodeExtractContent(DataPath)
18.  Node.AddEntry(FIB,Content,NextNode)
19.  Node.Send(NextNode,InterestCreatePath)
20.  Node.Send(NextNode,Content)
21. End For
22. For All InterestCreatePath received from Ni do:
23. NextNode ← Node.ExtractNextNode(InterestCreatePath)
24.  Content ← NodeExtractContent(InterestCreatePath)
25.  Node.AddEntry(FIB,Content, NextNode)
26.  If not Node.IsLastNode(InterestCreatePath) do:
27.    Node.Send(NextNode,InterestCreatePath)
28.  End If
29. End For
```

Algorithm 4 SRSC Forwarding for Controller

```
1. For InterestPath received from Node Ni do:
2.  RequestedContent ← ExtractPrefixe(InterestPath)
3.  If ContentList.Has(RequestedContent) do:
4.   Destination ← GetNode(RequestedContent)
5.  Else
6.   Destination ← GetBorderNode()
7.  End If
8.   Path ← Controller.GetPath(Ni, Destination)
9.   Controller.Send(Ni, Path)
10. End For
```

experiments. In addition, the use of virtualization allows deploying large number of nodes and performing large-scale experiments while still experimenting in laboratory facilities. Scalability is indeed an important concern while testing large-scale networking architecture.

In this context, we rely on Docker [7], a software containerization platform. Containers and virtual machines have similar resource isolation and allocation benefits, but containers are more portable and efficient. The virtual testbed architecture is shown on Figure 1, where each node is represented by a container encapsulating NDNx and our implementation of SRSC into the NDN Forwarding Daemon. The containers have also user space, from which user applications generate requests. The Docker containers are not tied to any specific infrastructure: they can be run on any computers or hardware, and in any clouds. Thus, our virtual testbed is portable and can be deployed on all kind of infrastructures. Then, we can create any topology where each node is deployed into a container and has as many interfaces as links in the network. The containers are built with Ubuntu 12.04 LTS system and our entire testbed

Parameters	Values
Topology	Abilene
Catalog Size	10,000
Number of Users	11
Requests per user	10,000
Popularity Model	Zipf(α=1.1)
Cache Size	1,000
Caching Strategy	LCD
Replacement Policy	LFU
Metrics	Cache Hit, #Messages

TABLE II: Large-scale Experiment Parameters

runs on a single workstation with 32GB of memory and Intel Xeon(R) CPU E5-1620 v2 with 8 cores at 3.70GHz.

For our experiments, we use the Abilene topology as it is an Internet-like topology commonly used for evaluating large-scale networking architecture (Figure 2). On this topology, we can place a server and its catalogue of content and users can be located at any nodes of the topology. The names of a node is the US city in which it is located such as Los Angeles (LOSA) or Atlanta (ATLA). We also deploy probes at each node to collect all the traffic.

V. SRSC EVALUATION

A. Scenarios and Experiment parameters

In order to evaluate our SRSC implementation within our Docker virtual testbed, we have performed two different kinds of experiments. For the first experiment, we have distinguished all the actual scenarios for a node to forward an *Interest* with SRSC. We therefore evaluate these use cases and thus the ability of our SRSC proposal to perform routing within the NDN environment. The second experiment evaluates the global performances of SRSC in a large-scale scenario, with a higher number of users, requests and traffic load. We therefore evaluate the traffic overhead of SRSC and the overall caching performances with our routing scheme.

With SRSC, we can distinguish four distinct scenarios for a node to forward *Interests*. Indeed, on one hand a node can have to perform the full process and request a rule to the controller; on the other hand, a node can already have the forwarding rule in its FIB and can forward an *Interest* without any request to the controller. In these following scenarios, a single content is requested to show the ability of SRSC controller to achieve its goal and to allow forwarding *Interest* toward the content. We therefore measure the total delay for a request and its reply (i.e., the delay between the sending of an *Interest* and the reception of the corresponding *Data* message).

Use-case scenarios: We distinguish four distinct scenarios for a node to forward *Interest*: (i) Full Routing Scenario; (ii) Cache Scenario; (iii) Close Routing Scenario, and (iv) Direct Forwarding Scenario. In order to explain these four scenarios, we consider the case where the server stores a catalogue of files and is located on node LOSA (Figure 2). Three users are located on nodes CHIN, NYCM and WASH and the SRSC controller is connected to node KSCY.

Full Routing Scenario (FRS) is the general case when a NDN node does not have any rules in its FIB to forward an *Interest*, and has to request rules from the controller. Thus, the user connected to node NYCM requests a content, which is only stored at the server (step 1 in the figure). Node NYCM has no information about content in its FIB or its Content Store and it sends a specific control message *InterestPath* to the controller to get rules to reach the content (2); The controller computes the path between node NYCM and the server connected to node LOSA and sends its reply (3); Node NYCM sets up the rules in FIB along the path to the server (node LOSA) by using a *InterestCreatePath* (4), and it forwards the original *Interest* toward the content (5). Finally, the server sends the *Data* back to the user (6).

Cache Scenario (CS) is the case where a requested content is already present in the node's Content Store. A user sends an *Interest* for a content (7) that has been cached into node NYCM; the node can therefore satisfy the request and reply with *Data* (8). In this case there is no need to contact the controller and it is the best case that can happen for NDN where content has been spread close to users.

Close Routing Scenario (CRS) is the case where an *Interest* is forwarded up to the nearest Content Stores but not necessarily the origin server. Indeed, NDN is a caching network and several replicas of the content can be stored in different nodes' Content Stores. For instance, the user at node CHIN requests a content available in node NYCM (9). As node CHIN has no rules to reach node NYCM, it contacts the controller by sending an *InterestPath* (10). The controller computes available paths up to the content and sends the rules to node CHIN (11). Note that two paths are available (node LOSA and NYCM) and the SRSC controller selects the shortest path to the content. Upon receiving rules from the controller, node CHIN sets up the rule in its FIB and forwards the *Interest* to NYCM (12), which replies with *Data*. The CRS scenario is similar to FRS scenario but content can be found on closer nodes instead of origin server due to previous requests in the network.

Direct Forwarding Scenario (DFS) is the case where rules are already in a node's FIB, but the content is not available in Content Stores. It can be due to some replacement strategies to manage node memory efficiently. A user in node WASH sends *Interest* to get content, which is forwarded up to the server as this rule is set in its FIB (14). The server replies with *Data* back to the user (15).

Performances evaluation scenario: Besides the first experiment to evaluate the routing ability of SRSC in all use-case scenarios, we evaluate the performances of NDN architecture with our SRSC routing scheme. We measure the traffic overhead of SRSC and the overall caching performances.

For this experiment, we use a catalogue with 10,000 contents (i.e., files) located on the server on node LOSA; each file has a size randomly chosen between 1 to 2000 bytes. Each of the 11 nodes of the Abilene topology has a user, and each user performs 10,000 requests. The content popularity follows a

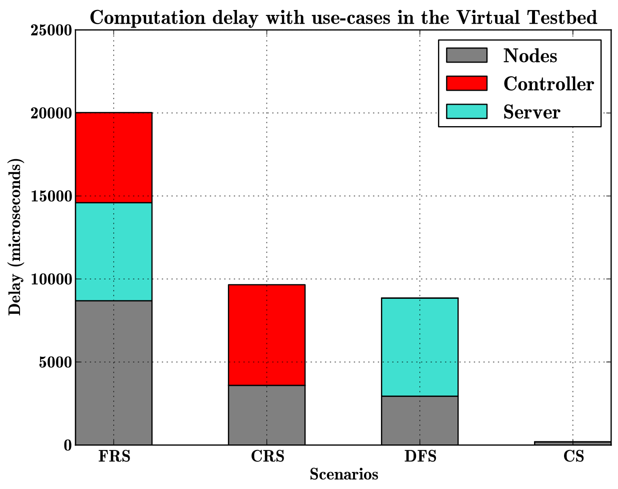

Fig. 3: Computation time with SRSC within Docker NDN testbed for the four use-case scenarios.

ZipF distribution (α=1.1) commonly used in the literature [3]. Thus, there will be a total of 110,000 requests toward contents according to their popularity. The NDN nodes have a cache size of 1,000 contents and use Leave Copy Down (LCD) as the caching strategy and Least Frequently Used (LFU) as the replacement policy. Table II summarizes the parameters of this experiment.

B. Results

The results of the experiments are described in this part. We first present our results for the routing use-cases in SRSC. Then we show the overall evaluation of NDN/SRSC in lager-scale scenario.

Use-case scenarios: For all these scenarios, we evaluate the processing time to get content at the server, controller and nodes side. The processing time of the controller corresponds to the time to look up the content and compute the path. For nodes, the processing time is to look up the tables (CS, PIT, FIB), update entries (FIB, PIT) and forward messages. For the server it corresponds to the time to sign content and respond to request. In our experiments, we performed 10 runs of each scenario and show the average value on Figure 3. Each histogram represents a scenario with the distinct processing time. We do not take into account link latency as this depends on the topology, and choose to focus on the performances of the SRSC implementation.

From all the scenarios, the delay to access content is larger with **FRS** (20 ms). Indeed, in this case the full process is performed and a node contacts the controller, which computes path up to the server and sets up rules in nodes. More precisely, the processing time for nodes represents the larger part of the processing time, as several nodes will be involved in the process to forward an *Interest* to the destination. This is the general case of SRSC, which obviously generates much more requests (i.e., processing time and traffic overhead).

CRS and **DFS** show a similar overall processing time (9-10 ms). However, this processing time is not composed of the same part. With **CRS**, an *Interest* is forwarded up to a Content Store: there is no processing time at the server and the controller counts for the larger part of the processing time. The controller's processing time in **CRS** is slightly larger than the one in **FRS** as the controller can have to compute more than a path toward several Content Stores. Nodes processing time in **CRS** is also slightly lower than with **FRS**, as there are fewer nodes involved in the routing process. Indeed, a path to a closer Content Store is shorter than path to the origin server (LOSA). Regarding **DFS**, as rules are already in nodes' FIB, there is no processing time at the controller level and *Interest* is forwarded up to the server. The larger part of the delay is therefore from the server side. The nodes' processing time is smaller than with **FRS** as rules are already in FIB and there is no look up in PIT or CS or any update. Fewer nodes are also involved in this scenario. Note that the **DFS** scenario could also forward *Interest* to a closer Content Store instead to the origin server; this scenario can potentially have a smaller processing time.

For the **CS** scenario, the processing time is negligible (<1 ms) as the closest node already has the content in its Content Store; neither controller nor server are involved. **CS** illustrates the typical case where NDN shows its efficiency: Content has been spread close to users and requests stay local, reducing de-facto the overall delay to transport content and enhancing quality of experience.

This experiment is a proof of concept of the implementation of our routing scheme SRSC and its deployment into Docker virtual environment. We have tested all representative scenarios of SRSC and demonstrate the feasibility of our proposal. With SRSC, the controller and the server represent the largest part of the processing time (FRS, CRS, DFS). These cases are necessary steps at the beginning before content is spread into the network. At steady state, NDN will fulfill its objective to cache content in the network. Thus, **CS** will therefore become the most frequent scenario; closest nodes will therefore satisfy requests without any messages to the controller, reducing the SRSC processing time.

Performances evaluation scenario: We evaluate the overall performance of SRSC in a larger-scale experiment and traffic load. In Table III, we present the number of messages in the network, as well as the volume of data transported in the network. The Cache Hit has been computed and is also presented.

First, all the requests have been satisfied (110,000 requests), and there has been no packet loss in the experimental testbed. The number of *Interests* (113,874) is larger than the Requests, as some *Interests* are duplicated in the NDN network to reach content. There are also fewer *Data* messages (75,333), which are the replies to *Interests*. Indeed, when nodes have the requested content in their Content Store (Cache Hit), *Data* messages are not spread into the network.

SRSC has also generated 304,575 *Control Interests* and 44,764 *Control Data* that are specific messages for the communication between the nodes and the controller. One can observe that SRSC generates a lot of messages (349,339) compared with NDN messages (209,207). However, these control messages count for only 27.3MB of the traffic, and the NDN messages have generated 152.2MB of traffic (NDN *Interest* and *Data*). Thus, the SRSC overhead represents 18.0% of the overall traffic, and this is satisfactory for real-world experiments. However, reducing the overhead is a typical concern for protocol engineering and we keep this topic for future work. For instance, nodes communicate periodically with the controller, and the number of update messages can be tuned to reduce drastically the number of control messages in the network.

We also compute the Cache Hit in the network, which is the probability to find content in cache. This is the most important metric to evaluate the efficiency of the NDN architecture. In our experiment, the Cache Hit reaches 47%, which means that almost half of the requests have found content in nodes' Content Stores instead of the original server. In other words, the load at the server has been reduced by a factor two, which is a very important performance improvement. Reducing the load at the origin server is one of the NDN objectives and comes directly from the in-network caching capabilities of NDN, distributing the load within the network. In addition, our routing scheme SRSC has also the ability to forward request up to the closest node and not necessarily on the path up to the origin server, alleviating the load at server and enhancing the Cache Hit performance.

To sum up our findings, the originality of our work is to implement and evaluate our routing scheme SRSC into real-world experiments. We have implemented SRSC into NDNx and performed experiments to evaluate its performances. We show that SRSC succeeds in forwarding *Interest* toward content; SRSC is therefore a potential candidate for a routing scheme in a full NDN environment. In SRSC, the processing time is shared between network entities (controller, server, nodes) and depends on the use-case to forward *Interest*. In addition, SRSC introduces a low traffic overhead, while it maintains a high-level of caching performances. We have also deployed an NDN testbed that can be used for further evaluations of new protocols or applications.

	Number of messages	**MBytes**
NDN Interest	133,874	7.8
NDN Data	75,333	144.4
SRSC Control Interest (overhead)	304,575	22.8
SRSC Control Data (overhead)	44,764	4.5
Total NDN	209,207	152.2
Total SRSC (overhead)	349,339	27.3
Traffic Overhead (Ratio)	-	18%
Cache Hit Ratio	47%	-
#Hit	86,278	-
#Miss	96,683	-

TABLE III: Performance evaluation of SRSC in large-scale experiment scenarios

VI. Discussion

The SRSC protocol relies on controllers and a single controller cannot store all the information about content and location. The controller can therefore be distributed into several entities [12] as for P2P network. For the P2P analogy, a controller would be similar to BitTorrent websites that keep track of the torrents' information on the Internet but do not own any content. The scalability of such architecture has been largely demonstrated with the success of this application on the Internet.

A controller is also responsible for a domain and if an *Interest* cannot be resolved, the controller forwards it to other domains through border nodes, where other controllers will be in turn responsible to find rules to the content. This is similar to the "hot-potato routing" with BGP in today's internet [13]. Popular content will also be cached locally and inter-domain communications will remain limited.

From our implementation experience, we have not focused on reducing the SRSC traffic overhead and keep this topic for future updates. As this is a common topic in protocol engineering, the traffic overhead can be largely reduced with simple improvements. The number of periodic messages can be tuned, and control messages can be aggregated to reduce the traffic overhead. For example, instead of sending an *InterestNewContent* for each available content in CS, a message can aggregate several content announcements.

The size of the FIB should also stay limited as the NDN nodes have constrained memory resources. Thus, FIB entries will be deleted if they have not been used for a while upon expiration of a timer. Entries for popular content will remain in the FIB and memory will be reallocated for newer requests.

VII. Related Work

Information-Centric Networking and especially Named-Data Networking has attracted a huge research community, and there has been a lot of work to improve its architecture (e.g., caching strategies, congestion control, multipath). More recently, routing has become an important topic, as it is essential for future deployment at Internet-scale. At the same time, several research projects aim at experimenting NDN into testbed facilities in order to favor future deployment.

Routing in NDN: Hoque et al. propose NLSR [6], one of the first routing schemes for NDN. NLSR extends OSPFN routing protocol [14] and is now included into the NDNx distribution. NSLR does not rely on the IP protocol and uses also *Interest* and *Data* messages to exchanges control information between nodes. Our proposal SRSC differs from NLSR for several reasons: (i) SRSC forwards *Interest* to the closest nodes, while NLSR forwards *Interest* up to the origin server. SRSC can therefore reduce the length of the path to access content; (ii) NLSR nodes exchange routing information among them, while SRSC relies on the SDN paradigm and a controller is responsible to manage all this information.

Differently, Ascigil et al. [15] and Zhang et al. [16] propose another approach, which consists in forwarding *Interest* up to the server or on the paths already taken by *Data* messages carrying the same content. In this proposal, each node must keep track of all *Data*, increasing the use of memory and CPU at each node.

Saino et al. [17] use a hash routing scheme in Information Centric Networks. This solution enables forwarding *Interest* using prefixes' hash and caching content in a single node defined by this hash. However, this solution does not select the shortest path to the destination: once a path to content is set, messages always follow the same path and this can increase the delay within large networks.

Finally, Rosensweig et al. [18] add new tables in nodes enabling to monitor incoming traffic and forward *Interest* according to these statistics. But tracking all the incoming traffic in each node without coordination between them can produce a huge amount of redundancy, and seriously impact the network's performances with all the memory access needed.

In our previous work [11], we present the theoretical foundation of our routing scheme SRSC. Our scheme relies on the SDN paradigm and a controller is responsible to compute the shortest path to content or one of its replicas in the network, and to set up forwarding rules into nodes' FIB. Besides evaluating a prototype through simulation experiments with NS-3 and ndnSim, we detail in this paper the full implementation of our proposal SRSC into NDNx and real-world experiments performed into a large-scale virtual testbed.

NDN experimental Testbed: There have been several research projects to deploy NDN into testbed for experiments and evaluation. For instance, the Doctor project [19] uses virtual tesbted based on Docker and a gateway between the IP and NDN networks to collect real traffic from users. Virtualization allows experimenting with real implementation on generic hardware and deploying large-scale testbed at reduced infrastructure cost.

The Offelia testbed [20] has been built to evaluate Information Centric Networking architecture with SDN. However, it is not a full native NDN environment as Openflow is the interface between NDN messages and the IP network.

Other testbed initiative also relies on Banana Pi routers [21]. It is therefore more costly to perform any modification of the experimental platform compared with a virtual environment.

VIII. Conclusion

In this paper, we have presented the implementation of our NDN routing scheme SRSC, which has been implemented into NDNx and deployed in our Docker virtual testbed. We have first demonstrated the feasibility of our controlled-based routing scheme to forward *Interest* to closest Content Stores. The computation time is limited for the controller, which can scale to a large number of nodes and users. We have then shown that our routing scheme has low overhead (18%) and still exhibits high caching performances (47% of Cache Hit). SRSC is therefore a suited solution for a routing scheme in a full NDN environment.

Future work will extend the evaluation of our routing scheme with real user traffic such as video streaming, and compare with other routing protocols such as NLSR. The calibration of SRSC parameters can also help reducing the number of control messages in the network and multipath feature can also be experimented.

Acknowledgments

This research work has been supported by the French ANR DOCTOR project (ANR-14-CE28-0001).

References

[1] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Named data networking. In *ACM Conext 2009*.

[2] Cisco White Paper. Cisco Visual Networking Index: Forecast and Methodology, 2015–2020. 2015.

[3] Cesar Bernardini, Thomas Silverston, and Olivier Festor. A comparison of caching strategies for content centric networking. In *IEEE Globecom 2015*.

[4] Lorenzo Saino, Cosmin Cocora, and George Pavlou. Cctcp: A scalable receiver-driven congestion control protocol for content centric networking. In *IEEE ICC 2013*.

[5] Tin Yu Wu, Yu Wei Wu, and Kai Lin Cheng. An efficient ndn-based load adjustment scheme for reduction of energy consumption. In *IEEE CIT 2014*.

[6] A K M Hoque, Syed Obaid Amin, Adam Alyyan, Beichuan Zhang, Lixia Zhang, and Lan Wang. Nlsr: Named-data link state routing protocol. In *ACM ICN 2013*.

[7] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[8] FAQ Named-Data Project:. https://named-data.net/project/faq/#How_does_NDN_differ_from_Content-Centric_Networking_CCN.

[9] Priti Goel, Eric Holmberg, Mark Konezny, Ramesh Ayyagari, and Dick Sillman. Ccnx packet processing on parc router platform. In *Proceedings of the 2Nd ACM Conference on Information-Centric Networking*, ACM-ICN '15, pages 211–212, New York, NY, USA, 2015. ACM.

[10] Alex Afanasyev and Lixia Zhang. Ndn/ccn harmonization: Identifying ndn/ccnx1.x commonalties and differences. a high-level discussion summary. IRTF ICNRG, September 2016.

[11] Elian Aubry, Thomas Silverston, and Isabelle Chrisment. SRSC: SDN-based Routing Scheme for CCN. In *IEEE NetSoft 2015*.

[12] Kévin Phemius, Mathieu Bouet, and Jérémie Leguay. Disco : Distributed sdn controllers in a multi-domain environment. In *IEEE NOMS 2014*.

[13] Renata Teixeira, Aman Shaikh, Timothy G. Griffin, and Jennifer Rexford. Impact of hot-potato routing changes in ip networks. In *IEEE/ACM Transactions on Networking, Vol. 16, Iss. 6 PP. 1295-1307, December 2008*.

[14] Lan Wang, AKM Mahmudul Hoque, Cheng Yi, Adam Alyyan, and Beichuan Zhang. OSPFN: An OSPF Based Routing Protocol for Named Data Networking. Technical report, Named Data Networking, 2012.

[15] Onur Ascigil, Vasilis Sourlas, Ioannis Psaras, and George Pavlou. Opportunistic off-path content discovery in information-centric networks. In *IEEE LANMAN 2016*.

[16] Yan Zhang, Tao Huang, Jiang Liu, Jian ya Chen, and Yun jie Liu. Reverse-trace routing scheme in content centric networking. *The Journal of China Universities of Posts and Telecommunications*, 20(5):22 – 29, 2013.

[17] Lorenzo Saino, Ioannis Psaras, and George Pavlou. Hash-routing schemes for information centric networking. In *ACM ICN 2013*.

[18] E.J. Rosensweig and J. Kurose. Breadcrumbs: Efficient, best-effort content location in cache networks. In *IEEE INFOCOM 2009*, pages 2631–2635, 2009.

[19] ANR DOCTOR project<ANR-14-CE28-000>. http://www.doctor-project.org/.

[20] Stefano Salsano, Nicola Blefari-Melazzi, Andrea Detti, Giacomo Morabito, and Luca Veltri. Information centric networking over sdn and openflow: Architectural aspects and experiments on the ofelia testbed. *Comput. Netw.*, 57(16):3207–3221, November 2013.

[21] Benjamin Rainer, Daniel Posch, Andreas Leibetseder, Sebastian Theuermann, and Hermann Hellwagner. A low-cost ndn testbed on banana pi routers. *Communications Magazine, IEEE*, 54(9):6, oct 2016.