POLITECNICO DI TORINO
Repository ISTITUZIONALE

Efficient Caching through Stateful SDN in Named Data Networking

Original
Efficient Caching through Stateful SDN in Named Data Networking / Mahmood, A.; Casetti, C.; Chiasserini, C.; Giaccone, P.; Harri, J.. - In: TRANSACTIONS ON EMERGING TELECOMMUNICATIONS TECHNOLOGIES. - ISSN 2161-3915. - STAMPA. - 29:1(2018). [10.1002/ett.3271]

Availability:
This version is available at: 11583/2691328 since: 2018-02-27T14:15:38Z

Publisher:
John Wiley & Sons

Published
DOI:10.1002/ett.3271

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
Wiley postprint/Author's Accepted Manuscript

This is the peer reviewed version of the above quoted article, which has been published in final form at http://dx.doi.org/10.1002/ett.3271.This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions.

(Article begins on next page)

18 April 2024

Efficient Caching through Stateful SDN in Named Data Networking

A. Mahmood†, C. Casetti†, C.F. Chiasserini†, P. Giaccone†, J. Härri‡
† Dept. Electronics and Telecommunications, Politecnico di Torino, Italy
‡EURECOM, 450 route des Chappes, 06410 Sophia Antipolis, France
E-mails:{ahsan.mahmood,casetti,carla.chiasserini,paolo.giaccone}@polito.it, jerome.haerri@eurecom.fr

***Abstract*—Named Data Networking (NDN) is an innovative paradigm to provide content based services in future networks. As compared to legacy networks, naming of network packets and in-network caching of content make NDN more feasible for content dissemination. However, the implementation of NDN requires drastic changes to the existing network infrastructure. One feasible approach is to use Software Defined Networking (SDN), according to which the control of the network is delegated to a centralized controller, which configures the forwarding data plane. This approach leads to large signaling overhead as well as large end-to-end (e2e) delays. In order to overcome these issues, we propose to enable NDN using a stateful data plane in the SDN network. In particular, we realize the functionality of an NDN node using a stateful SDN switch attached with a local cache for content storage, and use OpenState to implement such an approach. In our solution, no involvement of the controller is required once the OpenState switch has been configured. We benchmark the performance of our solution against the traditional SDN approach considering several relevant metrics. Experimental results highlight the benefits of a stateful approach and of our implementation, which avoids signaling overhead and significantly reduces e2e delays.**

***Index Terms*—Stateful SDN, Named Data Networking, OpenState.**

I. Introduction

An increasing usage of content-based applications such as video sharing, social media networking, and e-commerce, has led to a dominant use of the Internet as a *content distribution network* (CDN). However, implementing content distribution in legacy IP networks is challenging. Indeed, the communication model in legacy IP networks is based on the packet exchange between pairs of hosts, which requires mapping content to location endpoints as well as deploying CDNs or P2P networks [1]. In this scenario, Named Data Networking (NDN) emerges as a promising paradigm in which hosts address the content in network packets rather than contacting the host containing the content, hence greatly simplifying content distribution. Moreover, each network node in NDN is equipped with a content store, which caches a copy of already delivered contents. In this way, future content requests can be satisfied from the network nodes instead of the content server.

Content caching is therefore a key component of NDN, and significantly benefits both users and network operators. From the user's perspective, the ability to retrieve content from intermediate nodes in the network, reduces delays and enhances the quality of experience. From the operator's point of view, the network overhead is greatly reduced, especially if multiple users request the same content (e.g., popular videos and live sport streams). In light of such advantages, future 5G systems will integrate caching capabilities in the network, especially to provide media and entertainment services [2].

Implementing NDN requires, however, a drastic change in the legacy network infrastructure. In this regard, Software Defined Networking (SDN), which also represents one of the core technologies in the network evolution towards 5G [3], can be leveraged to realize the NDN concept. Indeed, in SDN control plane and data plane are separated, and a logically centralized controller programs the data forwarding plane by keeping a global view of the network. As a consequence, SDN enables a programmable network, lowers operational costs and allows for flexible services. This allows us to easily install the functionality of an NDN node on top of the SDN controller that is responsible to process and react to NDN packets received from all the switches in the data plane.

An existing implementation of the SDN paradigm is OpenFlow (OF) [4], which is a standard protocol enabling the communication between the SDN controller and the network devices. In OF, the controller installs match-action forwarding rules on the network switches. Since the switches are stateless and the rules are static, the switch must interact with the controller for any "new" action or change of action to be performed in the data plane, hence leading to a large communication delay and a large amount of control traffic. In many time-critical network applications, where contacting the controller may be unfeasible, some control logic can be delegated to the switches, which allows modification of the forwarding rules based on local network events. Such an approach, enabling swift local decisions at the switch without involving the controller, requires to support states within the switches and is called *stateful* data plane, in contrast to the vanilla stateless OF data plane. OpenState [5] is a recent extension of OF that enables a stateful SDN approach, leveraging a standard OF switching architecture.

In this work, we combine the stateful SDN approach with the NDN technology, and realize the functionality of the NDN node by attaching a local cache to a stateful SDN switch, implemented through OpenState. The switch autonomously provides the required content provisioning functionality without interacting with the SDN controller. The main advantages

C. Casetti, C.F. Chiasserini and P. Giaccone are also with CNIT, Parma, Italy.

of our approach are the simplicity, since the NDN control logic is embedded directly within the switch, and the smaller latency with respect to vanilla SDN, since the NDN control logic does not need any interaction with the controller to operate. In particular, our main contributions are as follows:

- *Architecture:* we propose an NDN network architecture that enables direct support of caching within SDN switches;
- *S/N-DN node:* we design a stateful SDN/NDN (S/N-DN) node, i.e., a stateful SDN switch capable to perform the functionality of an NDN node;
- *Implementation:* we provide a stateful data plane implementation of the S/N-DN node using OpenState, thus avoiding any interaction with the controller at runtime;
- *Performance evaluation:* we evaluate the performance of our solution in an emulated environment considering several relevant metrics. Moreover, we benchmark our solution with a traditional, stateless OF data plane implementation of the S/N-DN node.

The remainder of the paper is organized as follows. Sec. II gives an overview of stateful SDN and of NDN, which are the two pillars of our work. Sec. III introduces our solution, and Sec. IV presents the data plane implementation of our approach. Sec. V describes the methodology we use to evaluate our solution, while Sec. VI presents experimental results. Sec. VII discusses related works and highlights the novelty of our solution. Finally, conclusions are drawn in Sec. VIII.

II. Preliminaries

In this section, we give an overview of stateful SDN and NDN technologies as these are the two pillars on which our solution is based. First, we describe stateful SDN as well as the platforms that help to implement a stateful data plane. Then we explain the NDN architecture along with its main components.

A. Stateful SDN

Existing SDN technologies such as OpenFlow [4] forces separation of control and data plane of a network. As a consequence, all the network intelligence is contained in the logically centralized controller that is responsible to govern the “dumb” switches. This means that the controller becomes responsible for all the network-wide and local decisions, hence resulting in large signaling overhead and latency.

In stateful SDN, part of the network intelligence is moved from the controller to the switches; specifically, the controller is involved in making network-wide decisions since it has a global view of the network, while switches make decisions that only rely on switch-local states.

Stateful switches maintain states for all the incoming traffic flows, with each flow being identified by a flow key. Based on the current state, the switches apply match-action rules on the packets belonging to the flows, hence reducing the need to rely on the controller to make local decisions. In this regard, OpenState [5] extends OpenFlow to configure stateful data plane. In order to enable the stateful functionality, OpenState switches run programmable eXtensible Finite State Machines (XFSMs), where an XFSM implements two tables: a *state table* and a *flow table*. The state table stores the current state for each active flow, while the flow table stores the match-action rules. Differently from a vanilla OF flow table, in OpenState, one of the fields used for matching a flow is the current state and the action allows to change the state. Interestingly, this approach enables the implementation of the XFSMs using almost the same hardware architecture as vanilla OF switches, exploiting the efficient lookup in a Ternary Content-Addressable Memory (TCAM). Indeed, for any incoming packet, first its state is looked up from the state table, then the action and the next state are chosen based on both the current state and the packet header, according to the flow table. When accessing the state table to lookup and update the state of a flow, OpenState defines two different keys: the *lookup_scope* defines the packet header fields used for lookup, and the *update_scope* defines the ones used for update. Their distinct definitions allow a more flexible packet processing, as discussed in [5]. The complete OpenState protocol specification is available at [6], while the feasibility of implementing hardware-based OpenState switches is addressed in [7].

P4 [8] is an alternative approach to OpenState to implement stateful SDN. Specifically, P4 is a high level programming language used to describe the packet processing within a switch with a much higher flexibility than OpenFlow, in which the set of available matching fields is fixed. Indeed, P4 allows an arbitrary definition of the packet parsing and processing, and thus enables programmable, protocol independent switches. A P4 program is developed obliviously from the underlying switching architecture and a P4 compiler has the responsibility to tailor the required packet processing to the underlying hardware architecture, designed according to a specific processing model [9], [10].

Unlike OpenState, where the states are embedded within the available TCAMs, in P4 the states are available through external objects, thus leaving stateful data plane implementation as optional. For this reason, we have implemented the NDN node through OpenState, which natively supports a stateful data plane in OF-compatible switches.

B. Named Data Networking (NDN)

Named Data Networking (NDN) [11] is an architecture to support the Information Centric Networking (ICN) approach that aims to change the traditional IP network into a content-oriented one. In legacy IP networks, packets in the network are destined to specific IP addresses; in contrast, NDN packets are destined to a content, thus the routing is based on content identification (i.e., data names). Moreover, each network node in NDN is able to cache contents, thus content requests can be satisfied by multiple nodes in the network. The routing is anycast and based on two types of packets: Interest and Data, and both carry a content name. This name is hierarchically structured, similar to URLs, and each name is divided into components, e.g., `/polito/tng/courses/A.pdf`. The Interest packet is generated by the *data consumer* and sent to the NDN network. The network nodes forward the Interest packets towards the *data producer* (i.e., the server hosting the content) based on the content name. The Data packet gets back to the consumer by following the reverse path of the corresponding Interest packet. The nodes typically cache the contents received back from the data producer. Thus, in the case a node receives an Interest packet for a locally stored content, the node will answer directly with the Data packet, avoiding the forwarding of the packets to/from the data producer.

In order to manage the forwarding of Interest and Data packets, each NDN node features the following data structures:

1) Content Store (CS), which caches a copy of already requested contents;
2) Cache Lookup Table (CLT), which allows to understand if a requested content is already stored in the CS;
3) Pending Interest Table (PIT), which stores forwarded Interests along with the interface from which the Interests were received, when they cannot be satisfied from the CS;
4) Forwarding Information Base (FIB), which contains forwarding rules based on the content names; it is the equivalent of IP routing tables in the context of legacy IP networks.

When an Interest packet arrives at an NDN node, the latter first extracts the content name and checks the availability of the requested content in its CS through the CLT; if the content is available, then the Data packet is sent back to the consumer. Otherwise, the content name is looked up in the PIT to find any matching entry; if it exists then the incoming interface of this Interest is inserted in the PIT. In case there is no matching entry in the PIT, then the Interest is forwarded towards the producer based on the FIB. At this point, a new entry is created in the PIT, which records the pending request in terms of content name and incoming interface. Inside the FIB, the node performs longest prefix matching on the content name to forward the Interest packet; if no matching entry exists in the FIB, then the Interest is discarded.

When a Data packet arrives, the content name in the packet is looked up in the PIT and the content is forwarded to all the interfaces listed in the PIT entry. Once the content is transmitted towards the consumers, the PIT entry is deleted. In addition, a copy of the content is cached in the CS and the CLT is updated, so that future Interest packets can be locally satisfied. In the case the CS is full, one content is evicted to make space for the new content to store, and the CLT is updated accordingly.

III. The Stateful S/N-DN Approach

Here we propose our stateful S/N-DN solution, which combines the SDN and NDN technologies, i.e., we equip a stateful SDN switch with the components that allow us to implement the NDN approach therein. As a result, we can make NDN-related decisions within the switch without interacting (at runtime) with the SDN controller.

As discussed in detail in Sec. VII, previous works have combined SDN and NDN, but failed to fully integrate the two technologies. Indeed, stateless SDN switches cannot make local decisions on how to process NDN Interest/Data packets, thus a dedicated agent, external to the switch, is typically envisioned for the actual packet processing. Our stateful SDN approach, instead, integrates the NDN data structures (CLT, PIT) directly within the switch and avoids the interaction with an external NDN agent.

We begin this section by explaining our proposed stateful S/N-DN architecture. Then, we highlight the challenges we face, along with the solutions we envision to enable NDN in an OF network. Finally, we present a toy example to better clarify the proposed solution.

A. The proposed architecture

Owing to the benefits of incorporating caching capabilities in an SDN network, we propose that each OF switch is attached with a local cache. We call this node comprising the OF switch and the cache, *cache-equipped switch*. We envision that the cache-equipped switch, upon receiving content requests from a user, should primarily try to satisfy the request by exploiting its own cache. If the content is not available therein, the request should be remotely satisfied either from the data producer or from a cache-equipped switch along the path towards the data producer. Although the above mentioned functionality can be provided by either following the stateful or the stateless approach, we envision that the cache-equipped switch functions in the stateful manner, enabled by OpenState technology. In this way, the switch can make decisions on its own, by maintaining the states of pending requests, thereby eliminating the need to interact with the controller at runtime. It follows that this approach requires no control traffic (at least to operate the NDN control logic) after bootstrapping the switches, which reduces the end-to-end user latency.

Our network architecture, as shown in Fig. 1, is based, for simplicity, on a single SDN domain. The SDN domain consists of cache-equipped switches, an SDN controller and the hosts, which are typically the terminals handled by users. The cache-equipped switches are connected with a tree topology to the server, which acts as the data producer and owns the complete catalog of the requested contents. As mentioned earlier, the cache-equipped switch firstly tries to satisfy the content requests using its own cache, otherwise the request is forwarded towards the server. In response, the server sends the requested content to the host, and a copy of each content is stored in the switch cache for future requests. It is fair to assume that the delay for content retrieval is negligible when the content is stored in the cache within the switch.

Importantly, the cache-equipped switches must forward requests based on the content name, which is not possible in legacy IP networks. Each of the hosts in Fig. 1 acts as an NDN consumer that generates Interest packets. The server acts as an NDN producer, which responds to the hosts with Data packets. Finally, the functionality of an NDN node is implemented inside the cache-equipped switch. Hence, we name the cache-equipped switch with NDN functionality as *S/N-DN Node* (i.e., an integrated SDN-NDN node) and use this term hereinafter. As explained in Sec. II-B, the main components of an NDN node are: PIT, FIB, CLT and CS. Thanks to the availability of local states, the OpenState switch can support the behavior of the PIT and of the CLT, thus the switch can forward

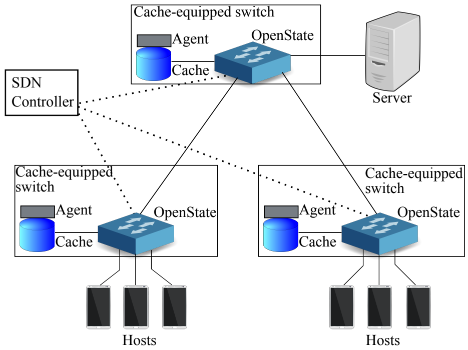


Fig. 1. The Stateful S/N-DN architecture.

autonomously the content requests either to the cache or to the server.

The cache combines a *cache agent* and an actual storage, implemented with either volatile or non-volatile memory. The cache agent receives the messages from the switch and interacts with the local storage. The main functionalities are the following:

- whenever an Interest packet is received, the requested content is read from the storage and then a Data packet is sent to the requesting host; notably, thanks to the CLT implemented within the switch, an Interest packet is received only when the content is available in the local storage;
- whenever a Data packet is received, the content is written on the cache. In case of an eviction, a control message is sent to the switch with the fingerprint (as described in the following section) of the evicted content, to properly update the CLT implemented within the switch.

To support such interface functionalities between the switch and the cache, a simple embedded computer (e.g., Raspberry Pi) could be adopted.

B. *Challenges and solutions*

It is not straightforward to enable NDN over an OF network. Notably, TCP cannot be adopted as transport protocol; indeed, it is not suitable for NDN because its end-to-end congestion and flow control are not suitable to interact with caching schemes and to support one-to-many communications, as required in an NDN architecture [12]. Thus, we use UDP/IP protocol, i.e., we consider that NDN Interest/Data packets are included in the payload of UDP packets. Second, an OF-switch cannot process NDN packets, as content identification is not possible. The OF protocol allows the switches to parse only a limited set of fields from an incoming packet, in particular it is not possible to parse the data field at the UDP layer. Following the approach introduced in [13]–[16], we piggyback the NDN semantics in the UDP/IP header fields. Specifically, we compute a fingerprint of the content name and store it in the UDP source and destination ports. Hence, the switch can identify the content referred by Interest packets using the fingerprint. Last, the available space in the packet header is limited; only 32 (16+16) bits are available in the UDP port fields. For variable name components, an option is to partition the available bits into groups, one for each name component. However, if the number of name components grows large, the number of bits per component decreases, hence the probability of fingerprint collision increases. The solution we recommend in such case is to store the fingerprint in other OF-matchable fields such as IPv6 source and destination addresses.

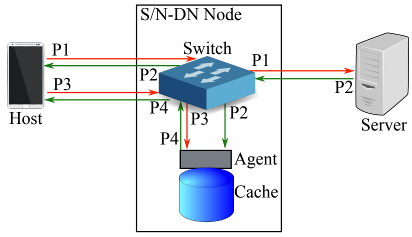
C. *A toy example*

In order to better explain our solution, we consider the toy scenario depicted in Fig. 2 (top), and use it to explain the main interactions between the network entities and the corresponding packet format, as shown in Fig. 2 (bottom). The scenario includes a host requesting content A, before and after the content is stored in the cache. For simplicity, we assume a simple Ethernet network connecting a host, a server and a cache to the OpenState switch. In the following, we report the sequence of exchanged packets. Notably, no interaction occurs between the switch and the SDN controller.

1) The host generates an Interest packet (P1) for content A and sends it to the switch inside the S/N-DN node. The hash of the content name, denoted by "hash(A)", is coded using 32 bits of the UDP source and destination ports, while the NDN Interest packet is inserted in the UDP payload. Since the content is not stored in the cache, the switch forwards the request (P1) to the server.
2) The server generates a response packet (P2) in which the Data is inserted into the UDP payload field. The response packet is sent to the switch inside the S/N-DN node, which does not only forward it to the host, but also places a copy of the packet in the local cache. In the response packet, the server uses multicast IP and MAC destination addresses since, in general, the data could be directed to multiple hosts, whose requests were pending in the switch.
3) The host generates another Interest packet (P3) for the content A and sends it to the switch. Since the content A is already stored in the cache, the switch forwards the request to the cache instead of the server.
4) The cache agent sends back the content to the host through a Data packet (P4). This packet is destined directly to the requesting host, since content retrieval from the cache is instantaneous and it is not possible to have pending requests for a content in the cache.

IV. Stateful Caching Implementation

We now describe the stateful data plane implementation of an S/N-DN node. To this end, as a first step, in Sec. IV-A we consider an OpenState switch and the associated controller. On top of the controller, we develop an application to properly preconfigure the OpenState switch so that packet forwarding can be performed based on content names. This simple application lays the foundation for developing a more complete application (detailed in Sec. IV-B) that allows the controller to configure the stateful switch to work as an S/N-DN node. Note that this enhanced application enables the switch to provide a



	Ethernet	IP	UDP ports	UDP payload
P1	Eth(Host)-->Eth(Server)	IP(Host)-->IP(Server)	hash(A)	Interest(A)
P2	Eth(Server)-->Eth(Multicast)	IP(Server)--> IP(Multicast)	hash(A)	Data(A)
P3	Eth(Host)-->Eth(Server)	IP(Host)-->IP(Server)	hash(A)	Interest(A)
P4	Eth(Cache)-->Eth(Host)	IP(Server)--> IP(Host)	hash(A)	Data(A)

Fig. 2. Exchange of messages in the toy example when content A is requested.

prototypical support for the NDN data structures (e.g., PIT and CLT), thus integrating the main NDN control functionalities within the switch.

A. NDN Packet Forwarding using OpenState

We describe a basic network application that controls the switch destination port of NDN Interest and Data packets based on the content name, instead of the usual forwarding based on layer-2/3 headers. We assume just one host acting as a data consumer and one server acting as a data producer, connected as in Fig. 2.

Each content is identified by a fingerprint applied on its name. The state is associated to the content. Depending on the current state, the request is forwarded to a distinct port, i.e., either the server or the cache. This behavior is described by the state machine diagram in Fig. 3. Specifically, there are three possible states: default, pending and stored. Initially, the state of any content is set to "default". When a request for a content in default state is received, the request is sent to the server and the content becomes "pending". When the content is pending, all new requests for it will be dropped, since redundant. On the contrary, in the case of a data packet from the server, the content will be sent to the cache and to the host and the content will become "stored". When the content is stored, any request for the content will be forwarded to the cache, and the content data from the cache will be forwarded to the host. If the content is evicted from the cache, the cache agent notifies the switch using a dedicated packet (e.g., adopting a special MAC address), which triggers the state of the content again in the "default" state. Table I shows the actual flow table that is installed on the switch by the controller, just during the bootstrapping of the S/N-DN node, and that implements specifically the XFSM described in Fig. 3. Note that the identification of Data packets or Interest packets is performed assuming a single layer-2 network connecting server and hosts, but it can be easily generalized to layer-3 networks. To lookup and update the state, we use the content fingerprint as key for the state table. Since the content fingerprint is stored in the

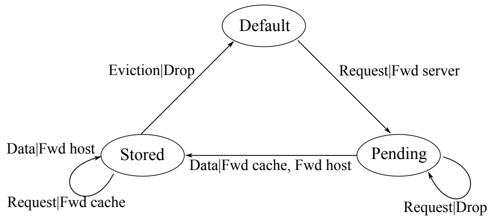


Fig. 3. State machine diagram for NDN packet forwarding. The notation on the arrow is event|action.

TABLE I
FLOW TABLE FOR NDN PACKET FORWARDING

Match fields		Actions	
State	**Event**	**Action**	**Next state**
Default	MACsrc=* MACdst=*	Fwd to server	Pending
Pending	MACsrc=server MACdst=*	Fwd to cache Fwd to host	Stored
Pending	MACsrc=* MACdst=*	Drop	Pending
Stored	MACsrc=cache MACdst=EVICTED	Drop	Default
Stored	MACsrc=cache MACdst=*	Fwd to host	Stored
Stored	MACsrc=* MACdst=*	Fwd to cache	Stored

TABLE II
EXAMPLE OF STATE TABLE IN AN S/N-DN NODE

Flow key	**State**
*	Default
UDP dst port = 888 UDP src port = 777	Pending
UDP dst port = ... UDP src port = ...	...

UDP ports of the packet, we set the lookup_scope and the update_scope equal to the UDP source and destination ports.

As an example, Table II shows the state table corresponding to a pending content, whose fingerprint corresponds to UDP ports 888 and 777. Notably, while Table I is fixed and pre-configured during bootstrapping, Table II stores the active flows (i.e., stored and pending contents) and its occupancy varies with the time.

B. S/N-DN Node using OpenState

We extend the basic NDN packet forwarding in OpenState for one host, described in the previous section, to implement the full functionality of the S/N-DN node with multiple hosts. As shown in Fig. 4, we have an OpenState switch connected with the cache, through the cache agent, at port P_C. The server is connected to the switch through port P_S, and the switch contains U user ports from P_1 to P_U. The stateful switch combines two state machines: XFSM 1 is responsible to forward the traffic to the cache or the server, mimicking the behavior of PIT and FIB of a standard NDN node; XFSM 2 is used for learning the MAC addresses of the hosts connected at the U user ports and guarantees layer-2 connectivity.

Each of XFSM 1 and XFSM 2 contains a state table and a flow table; thus, in our application we have four tables: state table 1, flow table 1, state table 2 and flow table 2. The flow table of XFSM 1 extends the flow table described earlier in Table I. For simplicity, in the following we discuss in details the case with two user ports P_1 and P_2 (i.e., $U = 2$). Later, we will comment on how to generalize it to an arbitrary U. The flow entries of the two tables are shown in Tables III and IV, in decreasing order of priority. The "pending" states, referred to a specific content, are coded as:

- "Pending10": the request received from P_1 is pending;
- "Pending01": the request received from P_2 is pending;
- "Pending11": the requests received from P_1 and P_2 are pending.

Upon arrival of an Interest packet at XFSM 1, the content state is looked up from state table 1. Assuming that the content has been requested for the first time, the content is in the "default" state and the Interest packet is forwarded to the server (i.e., entries 1 and 2 in Table III). If the arrival port is P_1 (or P_2), then the next state of the flow is "Pending10" (or "Pending01").

Moreover, the Interest packet is sent to XFSM 2 where the switch operates the standard MAC learning and forwarding procedure, following the approach proposed in [5]. The corresponding flow table is shown in Table IV. The lookup_scope of XFSM 2 is the packet destination MAC address, which is used to look up the state of the flow from state table 2. The next state depends on the ingress port of the incoming packet, i.e., if the in-port is P_1, then the next state is "portP1". The update_scope of XFSM 2 is the source MAC address of the packet. Thus, state table 2 stores the correspondence between the MAC address and the corresponding switch port.

Coming back to XFSM 1, entries 3–8 of Table III correspond to a content in "pending" state, i.e., the content has been forwarded to the server and the switch is waiting for the server response. Meanwhile, if another request arrives for the same content, the "pending" state will be updated to reflect the actual set of ports from which the requests arrived. Entries 9–11 of Table III manage the response from the server carrying the Data packet with the requested content. The switch forwards the content to the ports coded in the "pending" state and to the cache, and then changes the state of the content into "stored". As a result, any future request for this content will be forwarded to the cache instead of the server, coherently with entry 14. Then, the cache agent will answer with the requested content, using the destination MAC address corresponding to the requesting host. When such Data packet enters the switch, entry 13 will allow the correct forwarding to the destination port through XFSM 2. Finally, when the content is removed from the cache, the cache agent informs the switch to change the state of the corresponding flow from "stored" to "default" (entry 12).

C. S/N-DN node for $U > 2$

We discuss here how to generalize XFSM 1 for a generic number of user ports U. The "pending" state is now coded as "Pending$X$" where X is a binary string with 1 in position i whenever the Interest for a specific content is pending from port i. The total number of pending states is $2^U - 1$. For example, "Pending1010" means that the request is pending from user ports 1 and 3. Now, entries like 1–2 in Table III must be equal to U, i.e., one entry for each user port. Entries like 3–8 must be one for each pair of user port and "pending" state, thus summing to a total of $U(2^U - 1)$ entries. Entries like 9–11 in Table III must be one for each "pending" state, thus summing to $(2^U - 1)$. Finally, entries 12–14 in Table III will remain the same.

Table V summarizes the number of entries, which grows as

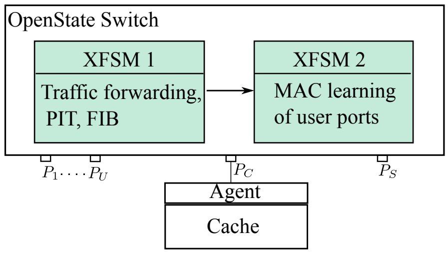


Fig. 4. Architecture of a stateful S/N-DN node.

TABLE III
FLOW TABLE 1 OF S/N-DN NODE, U=2

No.	Match fields: state	Match fields: event	Actions: action	Actions: next state
1	Default	MACsrc=*, MACdst=*, in-port=P_1	Fwd to server, Goto XFSM2	Pending10
2	Default	MACsrc=*, MACdst=*, in-port=P_2	Fwd to server, Goto XFSM2	Pending01
3	Pending10	MACsrc=*, MACdst=*, in-port=P_1	Goto XFSM2	Pending10
4	Pending10	MACsrc=*, MACdst=*, in-port=P_2	Goto XFSM2	Pending11
5	Pending01	MACsrc=*, MACdst=*, in-port=P_1	Goto XFSM2	Pending11
6	Pending01	MACsrc=*, MACdst=*, in-port=P_2	Goto XFSM2	Pending01
7	Pending11	MACsrc=*, MACdst=*, in-port=P_1	Goto XFSM2	Pending11
8	Pending11	MACsrc=*, MACdst=*, in-port=P_2	Goto XFSM2	Pending11
9	Pending10	MACsrc=server, MACdst=*	Fwd to cache, Fwd on port P_1	Stored
10	Pending01	MACsrc=server, MACdst=*	Fwd to cache, Fwd on port P_2	Stored
11	Pending11	MACsrc=server, MACdst=*	Fwd to cache, Fwd on ports P_1,P_2	Stored
12	Stored	MACsrc=cache, MACdst=EVICTED	Drop	Default
13	Stored	MACsrc=cache, MACdst=*	Goto XFSM2	Stored
14	Stored	MACsrc=*, MACdst=*, in-port=*	Fwd to cache, Goto XFSM2	Stored

TABLE IV
FLOW TABLE 2 OF S/N-DN NODE, U=2

Match fields: state	Match fields: event	Actions: action	Actions: next state
Default	in-port=P_1	Drop	portP1
Default	in-port=P_2	Drop	portP2
portP1	-	Fwd on port P_1	-
portP2	-	Fwd on port P_2	-

TABLE V
NUMBER OF ENTRIES IN XFSM 1 AND XFSM 2

Table	State transition or mapping	Num. entries
Flow table 1	Default → Pending	U
	Pending → Pending	$U(2^U-1)$
	Pending → Stored	2^U-1
	Stored → Stored	2
	Stored → Default	1
Flow table 2	Default → PortP	U
	PortP → PortP	U
State table 1	Content → Stored	$C+1$
State table 2	MAC → PortP	$U+3$

$O(U2^U)$. The number of entries in state table 1 corresponds to the number of contents in "stored" or "pending" state. Thus, it is bounded by the maximum number of contents in the cache (equal to C) plus the number of contents that are "pending". Notably, one additional entry is present in the state table for the "default" state. Instead, the number of entries in state table 2 is equal to the number of MAC addresses learned by the switch, which is upper bounded by $U+3$ assuming one MAC address learned for each user port, one entry for the cache, one entry for the server and one default entry.

D. Enhanced flow table definition in XFSM 1

Table V highlights the limited scalability of the flow table definition in XFSM 1 when U is large. This is mainly due to the self transition from/to "pending" state, whenever the state of a pending content must be updated to include the port from which a new request has been received. In other words, if the current state for a content is "pending$X$", with X being the adopted bit string representation of the arrival ports, and a new Interest packet for the same content arrives from port i, the new state "pending$Y$" is obtained by setting the ith bit of Y to one, i.e., using the logical OR operation:

$$Y = X \vee 2^{U-i+1} \qquad i = 1, \dots, U. \tag{1}$$

Thus, to support this simple state transition in flow table 1, the controller has to pre-configure all the possible transitions from a given "pending$X$" state and a given i arrival port to the new state "pending$Y$", producing $U(2^U-1)$ entries as discussed before. This approach is what we described in Sec. IV-C and it is actually what we have experimented since it is the only option available when adopting the OpenState switch emulator provided in [6]. Nevertheless, the original paper on OpenState [5] envisions the possibility of computing simple operations on the current state in order to compute the new state. In this case, (1) could be coded into a single entry in flow table 1, substituting all the $U(2^U-1)$ entries from "pending" → "pending". Similarly, with a single entry, it would be possible to configure all the "default" → "pending" transitions.

Furthermore, if OpenState was able to operate a programmable action based on the current state, it would be possible to avoid also the 2^U-1 entries "pending" → "stored" states. Indeed, given "pending$X$" state, when a Data packet arrives from the server, the set of user ports where to forward the packet (in addition to the cache) could be defined as the set of destination ports where the ith bit of X is 1. This would allow to code the transitions "pending" → "stored" into just 1 single entry.

TABLE VI
ENHANCED VERSION OF XFSM 1

Table	State transition	Num. entries
Flow table 1	Default → Pending	1
	Pending → Pending	1
	Pending → Stored	1
	Stored → Stored	2
	Stored → Default	1

In summary, Table VI shows the final number of entries if the programmability of OpenState would be allowed in terms of simple logical operations to compute both the new state and the actions to apply. The number of entries in the flow table would be $O(1)$ allowing the approach to scale for a large number of user ports.

V. PERFORMANCE EVALUATION METHODOLOGY

Here we introduce the methodology we use to evaluate the performance of our system. We start by presenting the performance metrics we adopt, then we describe the stateless approach for implementing the S/N-DN node, against which we benchmark the performance of our stateful approach. Finally, we describe our testbed and evaluation settings.

A. Performance metrics

In order to evaluate the system performance, we look at the following metrics:

- *End-to-end (e2e) delay*, measured in ms: time difference between the generation of an Interest packet and the arrival of the Data packet with the requested content at the host;
- *Cache download probability*, $P_{cache} \in [0,1]$: computed as fraction of content requests that are satisfied by directly downloading the content from the local cache;
- *Control traffic*, measured in bit/s: amount of traffic exchanged between the OF switch and the SDN controller over time, considering only the messages related to our NDN application;
- *Memory occupancy*, measured in kbytes: memory needed to store the state entries and flow entries in the switch.

B. Stateless caching in SDN

As a benchmark for our approach, we consider an S/N-DN node implemented through a standard, stateless OF switch. In the following, we use the terms *stateful approach* and *stateless approach* to represent, respectively, stateful and stateless data plane implementations of the S/N-DN node.

The architecture of a stateless S/N-DN node is shown in Fig. 5. An OF switch consists of U user ports, one port P_C connected to the cache through the cache agent, and one port P_S to connect to the server. The switch is controlled by the SDN controller, which stores the NDN-specific data structures, i.e., PIT, FIB and CLT. In addition, the controller is also

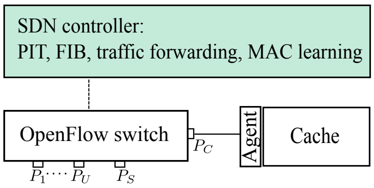


Fig. 5. Architecture of a stateless S/N-DN node.

responsible to forward the traffic towards the cache or the server and to run a reactive MAC learning application. To learn the MAC address of a host connected to a port, upon receiving the first packet from the host, the controller installs a flow entry in the OF switch, which is used to forward the subsequent packets to the host. Note that, in a network of stateless S/N-DN nodes, each of the OF switches is attached to a local cache but only one SDN controller stores the PIT, CLT and FIB tables for all the S/N-DN nodes.

The sequence of exchanged packets is shown in Fig. 6. The scenario and the packet formats are the same as in Fig. 2, i.e., the host requests content A twice, before and after the content is stored in the cache. In more detail, first, the host sends the Interest (P1) for content A to the switch, which then sends an OF packet_in message to the controller carrying a copy of P1. The controller creates an entry in the PIT, indicating that content A is needed by the host port. Then, it sends a packet_out message to the switch to forward the Interest (P1) to the server port. Hence, P1 is forwarded to the server that sends back the Data packet (P2) carrying content A. Upon receiving the content, the switch again contacts the controller by sending out a packet_in message carrying P2. The controller now checks its PIT table and then instructs the switch, by sending a packet_out message, to send the content to the host port and to the cache port. After the previous step, content A has reached the host and it is stored in the cache. Now, if the host requests again the same content (P3), then a copy of the Interest packet is again forwarded to the controller in a packet_in message to check in the CLT if the content is available in the cache. Since now the content is in the cache, the controller instructs the switch, through a packet_out message, to forward the Interest (P3) to the cache port. The cache agent responds with the Data packet (P4) directed to the host MAC address. This packet is received by the switch and then forwarded to the host based on the local MAC matching rules. In addition, when a content is evicted from the cache, the cache agent sends an eviction message to the controller, such that the controller updates the corresponding CLT.

In the above scenario, the whole packet exchange, excluding the packets between the switch and the controller, are the same as those depicted in Fig. 2 for our proposed stateful approach. Thus, the two approaches are equivalent from a functional point of view. The stateless approach requires that the switch interacts with the controller for each Interest packet, since the main NDN data structures (PIT and CLT) are managed by the controller and not directly by the switch as in our stateful

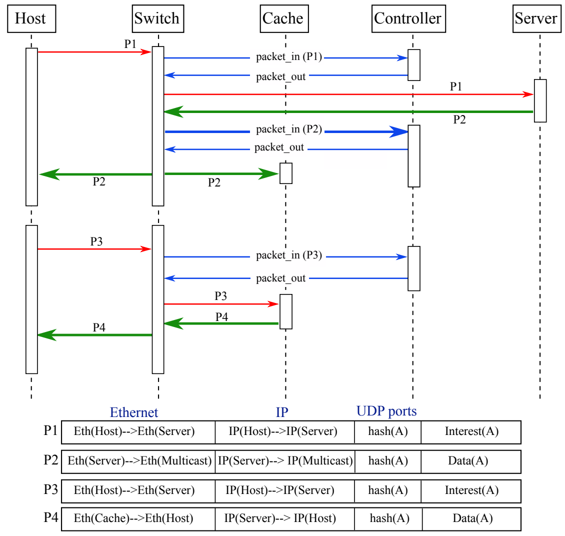


Fig. 6. Packet interaction in stateless implementation of S/N-DN node when content A is requested.

S/N-DN node.

C. Testbed implementation

We create our testbed on a server installed with Ubuntu 16.04. Mininet[1] is used to emulate our network scenario. We use OpenState virtual switch provided in [6] for stateful S/N-DN node and a standard OpenFlow v1.3 virtual switch for stateless S/N-DN node. The switch is managed by Ryu controller. The applications running on Ryu to support the stateful and stateless S/N-DN nodes have been developed in Python. Furthermore, the server and the cache agent (including the cache) are implemented as Mininet hosts running their respective software written in Python using Scapy[2] tool. The server can receive content requests and respond with actual content data. The cache agent is able to: (i) save content received from the server in a local Least Recently Used (LRU) cache, (ii) respond to content requests received from the host by sending the content data, and (iii) inform the switch about eviction of any content from the cache. We use a reference NDN traffic generator[3] to generate a trace of NDN Interest packets. This trace of content requests is replayed by the host at a specified rate using the `tcp_replay` Linux utility.

D. Evaluation of performance metrics

In order to evaluate the memory occupancy of the flow tables, we use the standard OpenFlow "flow_stats" request and reply messages. The SDN controller sends the flow_stats request message and receives back the memory occupied by each of the installed flow entries. In the stateful approach,

[1] http://www.mininet.org
[2] http://www.secdev.org/projects/scapy/
[3] https://github.com/named-data/ndn-traffic-generator

TABLE VII
SCENARIO PARAMETERS

Parameter	Description
d_{ss}	One-way delay between switch and server [ms]
d_{sc}	One-way delay between switch and controller [ms]
$\hat{C}$	Normalized cache size

since the flow tables do not change after the initial configuration, the flow_stats request and reply messages are exchanged only once in the beginning.

For the evaluation of the control traffic, we use Wireshark[4] to count the number of control messages (exchanged between the OF switch and the SDN controller) as well as to compute the size of each message. Furthermore, we only consider the control messages required for content retrieval; we do not include the messages required for initial configuration or the ones for retrieving flow statistics.

E. Evaluation scenario

We setup the scenario described in Fig. 1, comprising of an OpenState/OpenFlow switch, a cache, a server and a host. The switch is configured by the application running on top of the Ryu controller. The server and the cache run their respective software, while the host replays the traffic generated by the NDN Traffic Generator. A total of 1000 content requests are periodically generated at the rate of 1 packet/s. This guarantees that the time interval between two subsequent requests is always greater than the e2e delay of the first request, regardless of the stateful/stateless approach. The content catalog contains 100 data items, each of fixed size. The content for each Interest packet is chosen at random across the catalog according to a given distribution of content popularity. The system parameters are summarized in Table VII. Therein, d_{ss} represents the one-way propagation delay between the OF switch and the server, while d_{sc} represents the one between the OF switch and the SDN controller. Furthermore, $\hat{C} \in [0, 1]$ represents the normalized cache size, i.e., the size C of the cache normalized by the catalog size, both measured in terms of content items.

VI. Experimental Results

Given the aforementioned evaluation settings, we perform experiments on our testbed to validate the proposed approach and assess its performance. We now assume uniform distribution of content popularity and data size[5] equal to 1200 bytes. Fig. 7 shows the end-to-end (e2e) delay of the content requested by the host, for $\hat{C} = 1$ and $d_{ss} = 100$ ms. In the case of our stateful approach, d_{sc} does not affect the e2e delay experienced by the host, since, once the switch has been configured, it never interacts with the controller. The figure clearly shows that there are two phases in the system: a transient phase during which the cache is not full and the switch stores a copy of each of the distinct content items in the cache, and a stationary phase which starts as soon as the cache is fully utilized. During the transient phase, a requested content not stored in the cache is fetched from the server and the incurred e2e delay is around 218 ms, which is the round-trip time between the switch and the server plus the processing time. Conversely, during the stationary phase, all of the 100 contents are available in the cache and the e2e delay is always around 18 ms, which is coherent with the processing time evaluated in the transient phase.

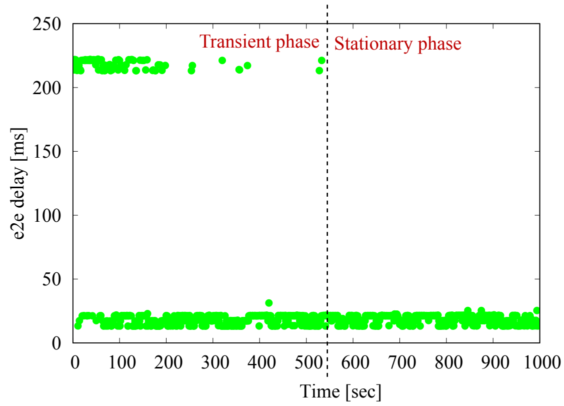


Fig. 7. The e2e delays during transient and stationary phases for the stateful approach, when $\hat{C} = 1$ and d_{ss}=100 ms.

In the following we will present our results during the stationary phase.

A. End-to-end delays

Fig. 8 depicts the e2e delays of the content requested by the host for different values of d_{ss}. The results refer to the stateful approach with $\hat{C} = 0.5$. At $d_{ss} = 100$ ms, for each of the requested content, the e2e delay is centered around either 18 ms or 218 ms, depending on the availability of the content in the cache. Indeed, even if the cache is always fully utilized in the stationary phase, it cannot store all of the 100 contents due to its limited size. Thus, some contents are evicted to make room for newly requested items, according to the LRU eviction policy. Similarly, at $d_{ss} = 50$ ms, the e2e delays are centered around either 18 ms or 118 ms. Finally, at $d_{ss} = 0$ ms, even the content that is not in the cache is retrieved instantly from the server, thus the e2e delays is centered around 18 ms.

Fig. 9 instead compares the stateful and the stateless approaches in terms of the distribution of the e2e delay. In particular, it shows stacked bar plots, for different normalized cache sizes, representing the probability that the e2e delay lies within the specified intervals. In the case of stateful approach, the e2e delay is distributed in the intervals 0–99 ms and 200–299 ms, depending on the availability of the content in the cache. For the stateless approach, instead, much larger e2e delays are observed: they now range between 200 and 299 ms, and between 600 and 699 ms. This is because the stateless approach relies on the communication with the Ryu controller to implement the functionality of the S/N-DN node, which depends on the parameter d_{sc}. Moreover, the impact of the normalized cache size is also evident for both approaches.

[4]https://www.wireshark.org/

[5]Data size comprises also the standard NDN header for content identification.

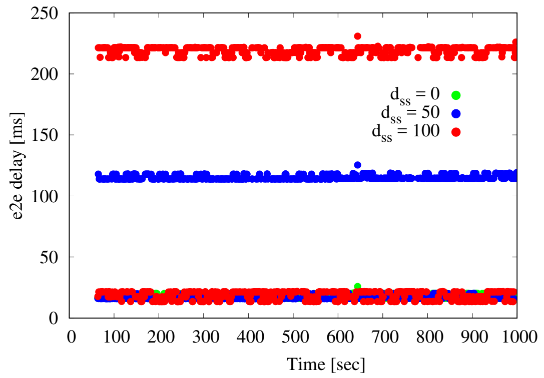


Fig. 8. e2e delays during the stationary phase, for the stateful approach and $\hat{C} = 0.5$.

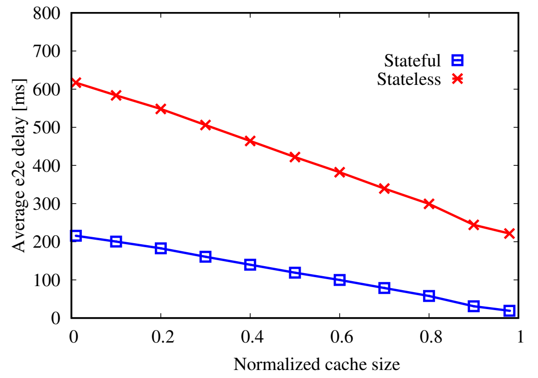


Fig. 10. Average e2e delay for d_{ss}=100 ms and d_{sc}=100 ms.

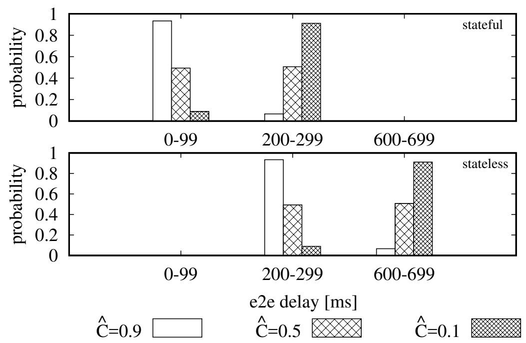


Fig. 9. Probability density function of the e2e delay for stateful and stateless approaches, for d_{ss}=100 ms and d_{sc}=100 ms.

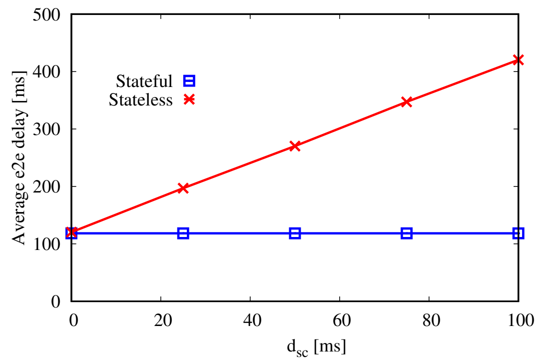


Fig. 11. Average e2e delay for d_{ss}=100 and $\hat{C} = 0.5$.

At $\hat{C} = 0.9$, approximately 93% of the content requests are satisfied by the cache, hence incurring lower e2e delays, while for the remaining 7% requests the contents are fetched from the server resulting in much larger e2e delays. At $\hat{C} = 0.5$, approximately 50% of the content requests are satisfied by the cache, while at $\hat{C} = 0.1$, approximately 9% and 91% of the content requests are satisfied, respectively, from the cache and the server.

Fig. 10 compares the average e2e delay, computed over all content requests, for the stateful and stateless approaches, as a function of $\hat{C}$. The delay of the stateful approach is approximately three times less than that of the stateless approach for small $\hat{C}$, while the difference becomes two times for large $\hat{C}$. We plot the average e2e delay as a function of d_{sc} in Fig. 11. The figure clearly shows that the average delay increases linearly with d_{sc} for the stateless approach, while the performance is independent of parameter d_{sc} in the stateful case. Indeed, recall that, unlike the latter, the stateless solution heavily relies on the communication with the SDN controller in order to implement the NDN functionalities, hence its performance significantly depends on the communication latency between the controller and the switch.

B. *Cache download probability*

We compute the cache download probability as $\hat{C}$ varies as well as for content requests that are generated according to uniform and Zipf (with $\alpha = 0.75$) distributions of content popularity. Moreover, the behavior of the cache download probability does not depend on the adopted (stateful/stateless) approach, rather on the generation process of the content requests. This is because, as previously discussed in our setup, the time to retrieve a requested content is much lower than the time interval between two requests. As shown in Fig. 12, the cache download probability exhibits a linear behavior for the uniform popularity, and non-linear for Zipf popularity; both behaviors are well known and in accordance with Che's approximation for LRU caches [17].

C. *Control traffic for stateless approach*

The OF control traffic is essential in the stateless data plane implementation of the S/N-DN node, as explained in Sec. V-B. In particular, 4 OF messages (see Fig. 6) are exchanged between the switch and controller when the requested content is not available in the cache. In addition, while operating in the stationary phase, the cache is full and each unavailable content, after being fetched from the server, is stored in the cache in place of the least recently used content. This

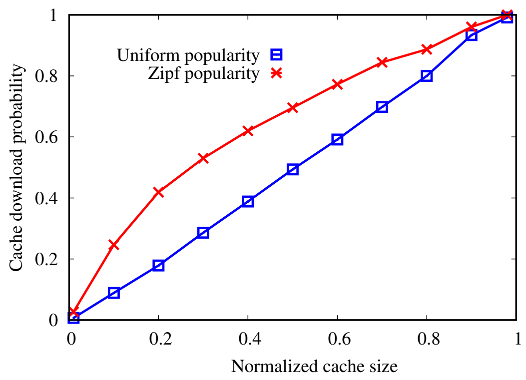


Fig. 12. Cache download probability for d_{ss}=100 ms and d_{sc}=100 ms.

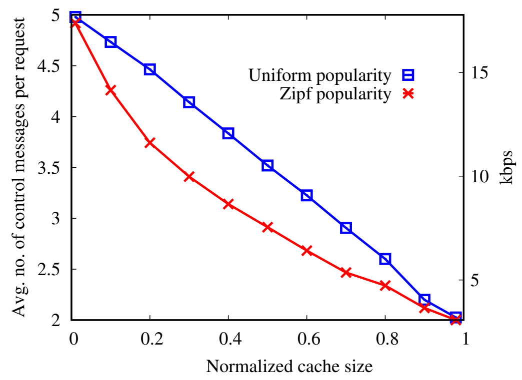


Fig. 13. Control traffic for the stateless approach, for d_{ss}=100 ms, d_{sc}=100 ms and one request per second.

leads to an additional control message sent to the controller informing about the evicted content, thus a total of 5 OF messages are exchanged. On the other hand, only 2 OF messages are exchanged when the content is available in the cache. Furthermore, note that the installation of the flow entry for the destination MAC address of the host takes place during the transient phase, hence it is not included in this computation. In summary, in our scenario, the average number of control messages for the stateless approach can be computed as follows:

$$5(1 - P_{cache}) + 2P_{cache}. \quad (2)$$

Fig. 13 shows the average number of control messages per content request, for different values of $\hat{C}$, under uniform and Zipf distributions of content popularity. At small $\hat{C}$, around 5 OF messages are exchanged, since the corresponding P_{cache} is close to 0. As $\hat{C}$ increases, P_{cache} increases and thus the number of messages decreases. If the cache stores all content items (i.e., $\hat{C} = 1$), then $P_{cache} = 1$ and only 2 OF messages are exchanged.

In addition to evaluate the number of control messages, we also measure the corresponding bandwidth. Table VIII shows the empirical size of the Ethernet frames of the OF messages that are exchanged when the requested content is available in the cache and when it is not. We consider two possible cases: either the carried data is small (equal to 100 bytes) or large (equal to 1200 bytes, almost the maximum allowed to avoid IP fragmentation in the OF traffic).

Table IX shows the network overhead for the two approaches. The network overhead is defined as the OF control traffic exchanged with the controller in stationary conditions, i.e., when the cache is full, normalized by the data traffic received from the server. In the stateless approach, the network overhead depends on the availability of the content in the cache as well as the size of the data packets, since a copy of the data packet may be sent to the controller carried by an OF packet_in message. When the data is small and is not locally cached, the overhead can reach almost 800%, since more than 1.1 kbytes (refer to the size of the corresponding 5 OF messages in Table VIII) are exchanged for just 100 bytes of data. The overhead is instead much smaller when the data

TABLE VIII
ETHERNET FRAME SIZE OF OF MESSAGES FOR DATA BETWEEN 100 AND 1200 BYTES

Scenario	OF message	Size
Content not available	packet_in(Interest)	276 bytes
	packet_out(Interest)	106 bytes
	packet_in(Data)	354-1454 bytes
	packet_out(Data)	122 bytes
	packet_in(eviction)	276 bytes
Content available	packet_in(Interest)	276 bytes
	packet_out	106 bytes

TABLE IX
CONTROL TRAFFIC OVERHEAD FOR THE STATEFUL AND STATELESS APPROACHES

Approach	Scenario	Data packet	Normalized OF traffic
Stateful	any	any	0%
Stateless	Content not available	100 bytes	799%
		1200 bytes	180%
	Content available	100 bytes	269%
		1200 bytes	31%

is large, since the OF traffic is better amortized. Nevertheless, in the best case, i.e., the large data locally available at the cache, the overhead is still approximately 30%.

We also show, in Fig. 13, the average bandwidth required for the stateless approach for each content request, and the required bandwidth for the large data packets, assuming 1 request per second. The required bandwidth varies between 3 kbps and 18 kbps, which is small due to the limited request rate considered in the experiment. Notably, this depends on the request generation rate and the cache size.

In our proposed stateful approach, on the other hand, after the initial pre-configuration, the switch acts autonomously without any interaction with the SDN controller (excluding the legacy monitoring messages), thus the number of OF messages as well as the network overhead is null.

D. Memory occupancy

Table X shows the empirical memory required for each entry in the flow tables of XFSM 1 and XFSM 2. Instead, state tables

TABLE X
EMPIRICAL MEMORY OCCUPANCY FOR XFSM 1 AND XFSM 2

Table	State transition	Memory (per-entry)
Flow table 1	Default → Pending	168 bytes
	Pending → Pending	152 bytes
	Pending → Stored	176 or 192 bytes
	Stored → Stored	152 or 160 bytes
	Stored → Default (eviction)	152 bytes
Flow table 2	Default → PortP (MAC learning)	128 bytes
	PortP → PortP (MAC forwarding)	88 bytes

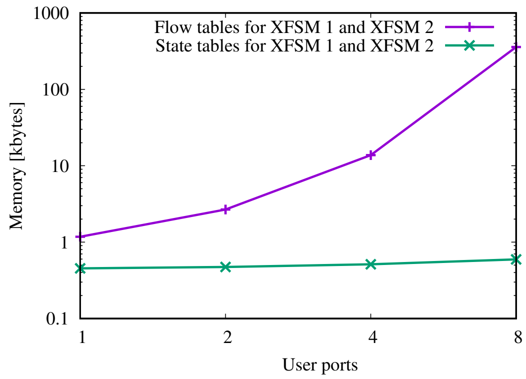


Fig. 14. Empirical memory occupancy in the OpenState switch for a stateful S/N-DN node.

are not present in standard OF architectures, hence no message is available to query their memory occupancy. In the following results we estimate their occupancy by using as reference: (i) $32+U+1$ bits for each entry of state table 1, obtained as 32 bits (i.e., the content fingerprint) plus U bits to code the set of destination ports plus 1 additional bit for the stored state, and (ii) $48+U$ bits for each entry of state table 2 (i.e., 48 bits of MAC address that serves as flow key in XFSM 2 plus U bits to code the state).

Fig. 14 shows the total memory occupancy for both the flow tables and the state tables, respectively, assuming $C = 100$ contents. Since the number of flow entries grows as $O(U2^U)$ as discussed in Sec. IV-C, the memory occupancy of the flow tables increases rapidly by increasing the number of user ports U. However, it can be observed that the memory occupied by the flow tables inside the switch for $U = 8$ is approximately 350 kbytes, which is feasible for hardware implementation based on TCAM memories [18]. Regarding the state entries, they depend on the user ports as well as the cache size (see Table V) but their occupancy remain below 0.6 kbyte, thus they are negligible.

With regard to the stateless approach, the whole functionality of the NDN node is implemented in the SDN controller while the switch plays a minor role; as a result, the number of flow entries installed in the switch is limited. Specifically, only U entries are installed, one for each MAC address associated with each user port. Note, however, that the small memory occupancy required by the stateless approach comes at the cost of large e2e delays.

VII. Related Works

NDN is an ICN architecture whose baseline implementation was carried out under the project called Content Centric Networking (CCN) [1]. The significance of deploying SDN and ICN capabilities in future networks has led to a considerable amount of research that aims at combining both the technologies.

A first family of works implements ICN over SDN, without an integration among the two layers. The works in [13], [19] implement an OF-based CCN node, in order to provide CCN functionalities such as caching and name forwarding in an SDN-based data plane. The CCN node consists of an OF switch, a wrapper and a CCNx daemon (i.e., a reference CCN implementation). The wrapper pairs each port of the switch with a CCNx interface, thus making it responsible for the communication between the switch and the daemon. A similar approach is taken in NDNFlow [20] where an ICN node is implemented by installing CCNx daemon on a legacy OF switch. The authors consider a network scenario involving both ICN-enabled as well as legacy OF switches. They implement an ICN module in the SDN controller, which uses an ad-hoc protocol on the south-bound interface of the controller, in parallel to OpenFlow, to control the ICN-enabled OF switch. In order to implement a CCNx node, computing resources must be integrated with the switch to run the CCNx daemon. Instead, in our stateful S/N-DN node the processing of NDN packets runs directly on the switch (thanks to the supported state machines) and thus allows to achieve full line-rate performance.

In [14] and [21], switches are not ICN-aware, making the SDN controller responsible for managing ICN request and response packets. This solution is similar to the stateless approach we compare with, and incurs large delays due to the required interaction between the switch and the controller, differently from our stateful approach which completely avoids such interaction. Very recently, other works have investigated how to integrate ICN in a 5G architecture using a stateless SDN approach. Indeed, [22] provides a 5G-ICN architecture implemented with the network slicing paradigm, while [23] combines NDN and SDN in the context of vehicular networks. Due to the centralized nature of approach, both works suffer limited reactivity as perceived by the users due to the continuous interaction between the controller and the switch to forward properly data and interest packets. The work in [24] proposes to combine ICN, C-RAN and SDN in heterogeneous networks. The adopted approach is based on logically centralized controllers responsible for all the ICN related operations (e.g., content addressing and matching), thus incurring large processing overload on the controllers and limiting large-scale deployment of heterogeneous networks. The paper suggests to install matching rules directly on the SDN switches to reduce the interaction with the controllers. This enhanced version is similar to the stateless approach considered in our work.

The work [25] is one of the initial works that combine SDN with ICN. Differently from our work, the considered ICN architecture has three main components: rendezvous, topology management and forwarding. The network function of topology management and rendezvous are realized in two centralized nodes, called topology manager and rendezvous server respectively. The topology manager builds forwarding identifiers, used for source routing, using bloom filter-based encoding scheme [26]. The forwarding network function is delegated to the SDN controller. This centralized approach for packet forwarding incurs significant scalability issues. In contrast, we focus on a decentralized approach according to which NDN Interest/Data packets are routed without involving the controller.

Similar to our idea of implementing an NDN node using a stateful SDN data plane, NDN.p4 [27] provides a preliminary implementation of NDN node using P4 abstraction [8] to program the data plane. In particular, [27] enables a P4 switch to process NDN Interest and Data packets. The switch implements PIT and FIB tables but cache lookup is not implemented. In our work, we adopt OpenState instead of P4, and we allow the switch to implement the PIT and CLT, thus it operates autonomously the forwarding of NDN traffic among cache, server and hosts. In addition, we evaluate the performance of the whole S/N-DN node considering relevant metrics such as latency, memory occupancy in the switch and control traffic.

The work most similar to ours is [28], which studies the feasibility of "breadcrumb" forwarding in OpenState switches. Such forwarding is a reverse path forwarding scheme, which maintains states for the opposite direction of a flow. The paper concentrates on a hardware proof-of-concept implementation based on FPGA. Moreover, the authors discuss few applications (e.g., CCN node, MPLS switching) which can be implemented by "breadcrumb" forwarding. In particular, the work describes the implementation of the PIT. The XFSM discussed in [28] for the PIT is very similar to our XFSM 1, proposed in Sec. IV-C, and thus suffers the same scalability limitations since the flow entries in the flow table grows as $O(U2^U)$. However, it does not support explicitly the content storage, and thus it does not provide the additional functionalities (as CLT, content eviction, automatic forwarding to cache or to the server, integration with the MAC learning/forwarding) that we support to fully enable the S/N-DN node.

Finally, in our work we do not investigate the effect of the specific caching policy adopted in NDN networks. We just consider a basic user-driven approach in which the content requests from the users dictate the contents stored in the cache and the eviction is managed with a traditional LRU policy. More advanced policies can be devised to optimize the efficiency of the caching scheme. In this context, [29] discusses the challenges for content caching and routing to provide video streaming service in ICN mobile wireless networks. The proposed video streaming solution takes into account content popularity to devise an efficient content caching strategy. Moreover, it provides a mechanism to improve content delivery by considering mobility of the nodes and selecting optimal content providers. In our work, we do not investigate the possible implementation of such schemes through a stateful approach, even if we expect that some of them could be easily implemented.

VIII. Conclusions

We proposed a novel solution to implement NDN leveraging the programmability of stateful SDN switches. Our architecture comprises stateful SDN switches, each of which is attached to a local cache. This combination of stateful switch and cache can successfully replicate the behavior of an NDN node, and we therefore referred to it as stateful S/N-DN node. We implemented the stateful S/N-DN node using OpenState, and a system testbed using Mininet, OpenState and Ryu SDN controller so as to evaluate the performance of our solution. We also benchmarked our stateful approach with a stateless data plane implementation of the S/N-DN node. We highlighted that, in a traditional stateless approach, the OpenFlow switch must rely on the communication with the controller in order to implement the functionality of the NDN node, thus resulting in large overhead and large end-to-end delays. On the contrary, our stateful approach does not need to involve the controller after the initial configuration of the OpenState switch, hence it yields zero control traffic and short latency.

IX. Acknowledgements

This work is partially supported by the EC through the H2020 5G-TRANSFORMER project (Project ID 761536), and the H2020 HIGHTS project (grant no. 636537) . EURECOM acknowledges the support of its industrial members, namely, BMW Group, IABG, Monaco Telecom, Orange, SAP, ST Microelectronics, and Symantec. In addition, the authors would also like to thank Marco Bonola, Salvatore Pontarelli and Angelo Tulumello for their comments and the discussion on their OpenState implementation.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *ACM CoNEXT*, December 2009, pp. 1–12.

[2] "5G and media & entertainment," 5G-ppp White paper, January 2016. [Online]. Available: https://5g-ppp.eu/wp-content/uploads/2016/02/5G-PPP-White-Paper-on-Media-Entertainment-Vertical-Sector.pdf

[3] "Vision on software networks and 5G," 5G-ppp White paper, Software networks WG, January 2017. [Online]. Available: https://5g-ppp.eu/wp-content/uploads/2014/02/5G-PPP_SoftNets_WG_whitepaper_v20.pdf

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[5] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming platform-independent stateful OpenFlow applications inside the switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014.

[6] "OpenState SDN project home page." [Online]. Available: http://www.openstate-sdn.org/

[7] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone, "Stateful OpenFlow: Hardware proof of concept," in *IEEE HPSR*, July 2015, pp. 1–8.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[9] "Protocol independent switch architecture." [Online]. Available: http://schd.ws/hosted_files/p4workshop2015/c9/NickM-P4-Workshop-June-04-2015.pdf

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, Aug. 2013.

[11] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014.

[12] Y. Ren, J. Li, S. Shi, L. Li, G. Wang, and B. Zhang, "Congestion control in named data networking – a survey," *Computer Communications*, vol. 86, no. Supplement C, pp. 1 – 11, 2016.

[13] X. N. Nguyen, D. Saucez, and T. Turletti, "Providing CCN functionalities over OpenFlow switches," Research Report, Aug. 2013. [Online]. Available: https://hal.inria.fr/hal-00920554

[14] S. Salsano, N. Blefari-Melazzi, A. Detti, G. Morabito, and L. Veltri, "Information centric networking over SDN and OpenFlow: Architectural aspects and experiments on the OFELIA testbed," *Computer Networks*, vol. 57, no. 16, pp. 3207 – 3221, 2013, information Centric Networking.

[15] A. Ooka, S. Ata, T. Koide, H. Shimonishi, and M. Murata, "OpenFlow-based content-centric networking architecture and router implementation," in *Future Network Mobile Summit*, July 2013, pp. 1–10.

[16] S. Eum, M. Jibiki, M. Murata, H. Asaeda, and N. Nishinaga, "A design of an ICN architecture within the framework of SDN," in *IEEE ICUFN*, July 2015, pp. 141–146.

[17] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: modeling, design and experimental results," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, Sep 2002.

[18] "SDN system performance." [Online]. Available: http://www.pica8.com/20pica8-deep-dive/sdn-system-performance/

[19] X. N. Nguyen, D. Saucez, and T. Turletti, "Efficient caching in content-centric networks using OpenFlow," in *IEEE INFOCOM*, April 2013, pp. 67–68.

[20] N. L. M. van Adrichem and F. A. Kuipers, "NDNFlow: Software-defined named data networking," in *IEEE NetSoft*, April 2015, pp. 1–5.

[21] M. Vahlenkamp, F. Schneider, D. Kutscher, and J. Seedorf, "Enabling ICN in IP networks using SDN," in *IEEE ICNP*, Oct 2013, pp. 1–2.

[22] R. Ravindran, A. Chakraborti, S. O. Amin, A. Azgin, and G. Wang, "5G-ICN: Delivering ICN services over 5G using network slicing," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 101–107, May 2017.

[23] S. H. Ahmed, S. H. Bouk, D. Kim, D. B. Rawat, and H. Song, "Named data networking for software defined vehicular networks," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 60–66, 2017.

[24] C. Yang, Z. Chen, B. Xia, and J. Wang, "When ICN meets C-RAN for HetNets: an SDN approach," *IEEE Communications Magazine*, vol. 53, no. 11, pp. 118–125, November 2015.

[25] D. Syrivelis, G. Parisis, D. Trossen, P. Flegkas, V. Sourlas, T. Korakis, and L. Tassiulas, "Pursuing a software defined information-centric network," in *2012 European Workshop on Software Defined Networking*, Oct 2012, pp. 103–108.

[26] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "LIPSIN: Line speed publish/subscribe inter-networking," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 195–206, Aug. 2009.

[27] S. Signorello, R. State, J. Franois, and O. Festor, "NDN.p4: Programming information-centric data-planes," in *IEEE NetSoft*, June 2016, pp. 384–389.

[28] G. Bianchi, M. Bonola, and S. Pontarelli, "On the feasibility of "breadcrumb" trails within OpenFlow switches," in *IEEE EuCNC*, June 2016, pp. 122–127.

[29] C. Xu, P. Zhang, S. Jia, M. Wang, and G. M. Muntean, "Video streaming in content-centric mobile networks: Challenges and solutions," *IEEE Wireless Communications*, vol. 24, no. 5, pp. 157–165, October 2017.