μNDN: an Orchestrated Microservice Architecture for Named Data Networking

Xavier Marchal, Thibault Cholez, Olivier Festor

μNDN: an Orchestrated Microservice Architecture for Named Data Networking

Xavier Marchal, Thibault Cholez, Olivier Festor
Universite de Lorraine, CNRS, Inria, LORIA
Nancy, France
{xavier.marchal,thibault.cholez,olivier.festor}@loria.fr

ABSTRACT

As an extension of Network Function Virtualization, microservice architectures are a promising way to design future network services. At the same time, Information-Centric Networking architectures like NDN would benefit from this paradigm to offer more design choices for the network architect while facilitating the deployment and the operation of the network. We propose μNDN, an orchestrated suite of microservices as an alternative way to implement NDN forwarding and support functions. We describe seven essential micro-services we developed, explain the design choices behind our solution and how it is orchestrated. We evaluate each service alone and the whole microservice architecture through two realistic scenarios to show its ability to react and mitigate some performance and security issues thanks to the orchestration. Our results show that μNDN can replace a monolithic NDN forwarder while being more powerful and scalable.

CCS CONCEPTS

• **Networks → Network architectures**;

KEYWORDS

Information-Centric Networking, Network Functions Virtualization, Software-Defined Networks, Microservice Architecture, Network Management

ACM Reference Format:
Xavier Marchal, Thibault Cholez, Olivier Festor. 2018. μNDN: an Orchestrated Microservice Architecture for Named Data Networking. In *5th ACM Conference on Information-Centric Networking (ICN '18), September 21–23, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3267955.3267961

ICN '18, September 21–23, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-5959-7/18/09…$15.00
https://doi.org/10.1145/3267955.3267961

1 INTRODUCTION

In recent years, there has been a trend toward the softwarisation of networks, and in particular the Network Functions Virtualization (NFV) [7] paradigm is seen as a great enabler to foster more innovation in networking by replacing specialized and expensive dedicated network equipment by commodity servers running Virtualized Network Functions. NFV appears to be a great opportunity for disruptive protocols like Named Data Networking (NDN) [32] because NFV infrastructure constitutes a substrate where NDN could be deployed incrementally and at low cost on commodity hardware alongside common protocols. This could help ISPs to progressively deploy ICN when and where needed.

NFV also introduces new design patterns inherited from the software world to networking and, among them, microservice architectures emerge as a promising way to improve the flexibility and scalability of network functions by giving more opportunities to deploy them in an efficient manner. For example, it is well acknowledged that in ICN, *Content Stores* are not useful in every node but rather in particular points of the network, as demonstrated in [5, 27]. So, we expect several benefits from running NDN as a virtualized microservice architecture compared to a bare metal monolithic implementation, in particular:

- The incremental deployment of NDN over an existing NFV environment;
- More efficient NDN topologies;
- Better horizontal scalability to deliver better performance when needed;
- Ability to deploy dynamically on-path security functions;
- Easier development and improvement of each module separately (less coupling and complexity).

However, microservice architectures also face new challenges. Indeed, if each service is simpler, this also exacerbates the need for a proper chaining and orchestration of microservices to build consistent network topologies which tend to be more complex. The full potential of a microservice architecture, which heavily relies on the dynamic reconfiguration of the network according to the operational needs, can only be

achieved if the architecture is properly orchestrated. Therefore, to reveal the desirable properties of the microservices, we also developed a manager entity.

In this paper, we propose the first realization of NDN forwarding functions using microservice architecture, which encompasses several contributions:

- The design of seven microservices to fulfil NDN requirements and ensure the good performance and security;
- Examples of possible combinations to make efficient NDN network topologies;
- A central manager component to monitor and orchestrate the microservices according to predefined rules;
- The evaluation of the whole architecture under two scenarios challenging the performance and the security of the μNDN network;
- The open source implementation of the proposed architecture.

The rest of the paper is organized as follows. Section 2 presents the related work regarding the microservice architectures for networking and the initiatives to use network softwarization technologies to deploy ICN. Section 3 describes the seven microservices composing our architecture and how they can be assembled. Then, we explain how they are managed and orchestrated in Section 4. Section 5 presents our implementation and the evaluation of our architecture under different scenarios. Finally, Section 6 discusses the lessons to be learned and Section 7 concludes the paper.

2 STATE OF THE ART

2.1 NFV and SDN initiatives for ICNs

Only a few recent initiatives leverage the Network Function Virtualization architecture and the related technologies to support the deployment of ICN. The DOCTOR project proposes a virtualized and orchestrated architecture to deploy NDN [3] with a particular emphasis on the security aspects. More precisely, they show that the proper monitoring of the Virtualized Network Functions and the orchestration of countermeasures can mitigate some security issues of ICN, like the Interest Flooding Attack or the Content Poisoning Attack. vICN [24] has the particularity to propose a generic object-oriented programming framework using a precise representation of the different resources used to build virtualized ICN topologies. Both approaches have similarities like the fact to use containers and virtual switches for the Virtual Network Infrastructure and the fact to follow the ETSI guidelines concerning the Management and Orchestration of NFV [8]. MiniCCNx [2] also uses similar technologies but to offer an emulation environment for experimentation rather than to build a production network, while the authors of [9] propose their Future Internet Testbed with Security architecture to perform ICN experiments.

Close to the NFV concept but in a 5G perspective, network slicing, where several network protocols cohabit in the same infrastructure thanks to the virtualization, is envisioned by the IRTF ICNRG [20] as a possible scenario for future ICN deployment. The authors of 5G-ICN [21] propose such an architecture which enables mobility as a service, but they mainly focus on the data plane and we miss details on the control plane. To improve the performance of NFV-based networks, the authors of [19] call for the support of hardware acceleration and propose a programmable data plane (PDP) that can enable on-the-fly customization of L2-L7 stacks to integrate hardware accelerators. Their example instantiates an accelerated ICN router which is decomposed in six modules but they are not designed to be autonomous and are not managed.

Other initiatives try to enhance network processing with on-path information processing in ICN. The authors of [25] consider Named Function Networking over CCN and propose to distribute processing functions across the network by adding a resolver in CCN nodes for λ-expressions embedded in names and that can be identified with a specific prefix. To the same extent, the NFaaS [12] framework (Named Function as a Service) uses cloud-based techniques to deploy lightweight VM based on Unikernels [14] for on-demand execution of services on network nodes. The VM are stored in a *Kernel Store* component that can also manage the VM execution so that a service is able to migrate between nodes over time. These two works aim to execute information processing functions over the network nodes while our microservices are fully dedicated to networking tasks. However, since they operate at a different level, it should be possible and fruitful to use both approaches at the same time.

Focusing on a single concept of network softwarisation, more studies have coupled ICN with Software Defined Networking (SDN) for various purposes. SRSC [1] is a controller-based forwarding scheme for NDN to perform the route management as an alternative to NLSR, [31], [30], [23] and [18] use SDN, and more precisely OpenFlow to transport ICN over IP while also optimizing the caching on the path, but [30] and [18] must twist the use of IP address and port fields to contain hashes of content names in order to allow OpenFlow switches to process Interests, while [23] rather chose to use IP options and payload, and [31] chose to extend OpenFlow for a cleaner approach. While fundamentally different in the internal operation, our architecture shares with the ones cited above the centralized nature of route management.

2.2 Microservice architectures for networking

The microservice architecture was first described by the software architects J. Lewis and M. Fowler [6] as a way to split a monolithic application as a set of small ones that handle a well-defined single task (the service or function) and are able to communicate with each other through a common protocol (mostly over HTTP RESTful API). The microservice architecture is an extension of Service Oriented Architectures [11] but with additional concepts and it is mostly used in cloud web applications and in IoT.

Much of the literature, like [22] or [4], give as advantages of microservice architectures easier development and maintenance tasks compared to monolithic applications that grow and become more complex over time with the addition of new functionalities, and also a better scalability and reactivity to emerging technologies. On the opposite, using a microservice architecture also comes with a few drawbacks which are mainly additional deployment complexity and the need to design an external communication layer. They also tend to use more resources, like shown in [29] where T. Ueda et al. benchmarked a web application using both a monolithic and a microservice implementation. They show that switching from a monolithic to a microservice architecture increases the performance cost up to 79.2% due to context switching, cache misses, etc. Moreover the virtual network infrastructure, in their case made of Docker virtual bridge, reduces the throughput up to 33.8%.

Microservices can be used for any kind of application and some research works like [13] try to create automatic tools to help people to switch from monolithic software applications to microservices. More precisely, Levcovitz et al. propose a methodology, based on a dependency graph, to identify microservice candidates in a monolithic application (the loosely coupled parts), and validate it on a real application. Advanced management architectures dedicated to microservices are also proposed, like the self-managing microservices architecture in [28]. With the rising of network softwarization initiatives like ClickOS [16] or Unikernels [14] and the emergence of new standards like NFV, network functions tends to follow the same path toward the creation of scalable and robust network architectures from small orchestrated functions with the promise of reducing network operators operational and capital expenditures while fostering innovation.

In summary, the microservice architecture is promising but has not yet been fully applied to networking problematic. At the same time, there is a clear trend toward the use of network softwarization (SDN and NFV) to facilitate the deployment of ICN but all the previous works use off-the-shelf monolithic ICN routers, thus limiting the flexibility and the options of the virtualized ICN network. In this paper, we have the challenging objective to propose a new microservice architecture specifically designed for ICN to get the full benefit from NFV.

3 MICROSERVICES FOR NDN

In this section, we describe the different microservices we developed as Virtualized Network Functions (VNF). We divide them into two groups: core routing functions that can replace the key routing functionalities provided by the NDN Forwarding Daemon (NFD), and support functions that can perform additional on-path processing of packets. We will indifferently use the term microservice or module in the rest of the paper.

3.1 Common definitions and design constraints

Before presenting the different microservices, we define here their shared foundations. The design of the services follows three key guidelines. The first is to be fully compatible with the NDN protocol to be able to communicate with any other implementation. The second is to force the decomposition into distinct services as much as possible, to exacerbate the good and bad aspects of such architecture. The third is to have the links between our modules to follow a pipeline logic to chain network functions. More precisely, a module does not know which one is above or under it and can be linked to any other one, even if all combinations do not make sense. The central manager described in Section 4 is the only one to have a global view. Other management architectures could have been envisioned to orchestrate microservices (distributed or self-management [28]) but are beyond the scope of this paper.

Yet, many challenges still arise:

- How to decompose a monolithic NDN router?
- How to link the microservices?
- How to process distinct *Interest* and *Data* flows?

In the following description of the different modules, the ingress and egress *Faces* refer as which module initiate the connection to create the link. An ingress *Face* listens to incoming traffic/connection. Also, each microservice may be oriented or not. An oriented module only processes *Interest* or *Data* on its *Faces*, while a non-oriented can process both.

We will also refer to the effective cardinality of a module which is the number of different sources or destinations it can distinguish when forwarding, but it has no relation to the number of endpoints a module can handle concurrently. The cardinality can take two values:

- An effective cardinality of "1" on one side means that a module should be connected to a single other. If it has to handle more than one neighbour, it is still possible to forward packets to all its source or destination *Faces*

in a multicast fashion. This can happen because, for a source, it does not have any clue to which one to send back the response, or for a destination, it does not store information to route the packet correctly.
- An effective cardinality of "N", means that a module is able to forward packets to the right source(s) or destination(s), which implies to maintain some routing information.

All the microservices described below are able to report their status (at least *Faces* statistics, but also service-specific values) for monitoring, like described in Section 4.

3.2 Description of the microservices

3.2.1 Core routing functions. Our first key decision was to separate the PIT and FIB functions. This is challenging because they are known to be tightly coupled to implement NDN forwarding. However, the idea to split ICN core network functions as distributed services has already been introduced in a CCNx report[1]. We chose to split PIT and FIB because we think that it can be useful to have a distinct PIT in some specific scenarios. For example, to scale up the PIT alone when it is a bottleneck. Also, like NFD, we need a third function to select the right *Interest* or *Data* pipeline. So, the three core routing functions have the following purpose:

- **Name Router (NR)**: this function handles the prefix registration requests sent by the producers and performs the routing based on packets' name. It is similar to the *Forward Information Base* (FIB) in NFD. This module is the only one that is "mandatory" in a μNDN network because it is the only one that can listen to incoming route advertisements from producers. This module is oriented and has two types of ingress interfaces: *Faces* that handle the consumers have a cardinality of 1 and *Faces* that handle the routing information have a cardinality of N. The egress *Faces* have a cardinality of N.
- **Backward Router (BR)**: this function keeps track of the *Interest* packets that pass through it in order to forward later the *Data* packets back to the right consumers. Since it stores user requests, it can also handle a retransmission mechanism. This function is similar to the *Pending Interest Table* (PIT) in NFD. It is oriented so *Interest* packets can only come from ingress *Faces* and *Data* packets from egress *Faces* and the cardinality of ingress and egress sides are respectively N and 1.
- **Packet Dispatcher (PD)**: this function aims to split a mixed *Interest* and *Data* traffic that comes from the same *Face* and redirect each traffic type to different outgoing *Faces*. It is used to avoid packet dropping when chained with "oriented" modules that can only process one type of ingress traffic. NFD does something similar when it receives packets from its *Faces* in order to select the right processing pipeline (*Interest* or *Data* processing). We think that this module is more suited at the edge of the network to handle external traffic that is not aware of the specific traffic management performed by μNDN but should not be useful inside the network. The cardinality of the module is N for both ingress and egress sides.

3.2.2 Support functions. These microservices can be considered as on-path modules because they are not used to route network packets. They are not oriented and have a cardinality of 1 for both ingress and egress sides. They can improve the performance, the reliability or the security of the network. We can extract two of these functions from the monolithic NFD:

- **Content Store (CS)**: this function aims to store recent *Data* packets that pass through it in order to reuse them later. Since it is optional in NFD (CS size can be set to zero), it is perfectly logical to make it a standalone function.
- **Strategy Forwarder (SF)**: this function is used to forward packets to one or more selected destination(s) based on a strategy. In NFD, it occurs after a FIB matching and the strategy can be defined for any prefix registered in the FIB and is hierarchical. Extracting this function enables more general forwarding rules because it can be used between any module (no longer dependent on the *Faces* selected after a FIB matching). The strategies can be simple ones used at a low level like load-balancing, fail-over, etc., or other more specific to NDN. The strategy rules are only applied for egress *Faces* and the cardinality depends on the actual strategy in use.

We can also take advantage of the NFV paradigm to propose more specific functions. For instance, some security functions that would be too demanding or too specific can be implemented as microservice:

- **Signature Verifier (SV)**: this function checks the signature of *Data* packets. The verification of the packets' signature is not implemented in NFD due to the lack of universal trust model and the heavy cost of the verification. But in μNDN, this function can be configured with a trust model to follow and placed in a different process or even in a different server which reduces its impact on the other network components. To store the keys, this function can use both a local storage as the public keys repository or an NDN Public Information Base if it exists. This function then reports the name of

[1] https://github.com/PARC/CCNxReports/raw/master/2014/5.1CCNx1.0ImplicationsforRouterDesign.pdf

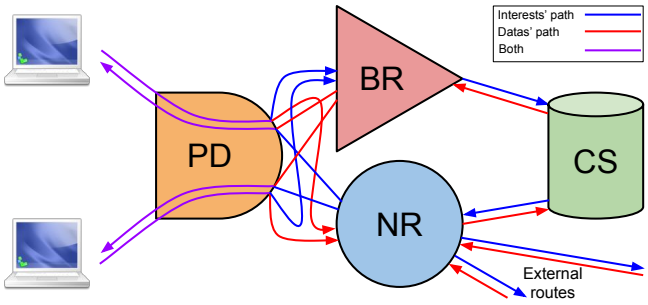

Figure 1: Example of service chaining that is equivalent to NFD (with multicast policy)

every failed packet to an upper entity that can deploy countermeasures if needed. If packet signature cannot be checked at line speed, the module can perform statistical verification.

- **Name Filter (NF)**: this function simply drops NDN packets based on their name (strict or partial matching). It is used to avoid specific packets in the network by filtering traffic at the edge, or to limit the scope to a given path for a service in case of multiple available paths (for example, to limit the diffusion of videos for a specific service to a single path).

These two functions may also communicate with each other in order to create something similar to a firewall by making the *Signature Verifier* directly append rules in a *Name Filter* with the aim to reduce fake packet verification. These functions could be easily used between regular NFD nodes because of their on-path nature. The main properties of each microservice are summarized in Table 1.

3.3 Network examples of possible combinations

We illustrate in Figure 1 an instance of μNDN that mimics the network functionalities of a monolithic NFD instance. For this service, the three core modules (*Content Store*, *Backward Router* and *Name Router*) are needed to simulate the three inner-tables of NFD (CS, PIT and FIB). The *Packet Dispatcher* module handles the external connections and the pipeline selection based on the packet type so the clients (consumers and producers) only need to connect to a single entry point of the μNDN network like they would do with a monolithic forwarder. In our case, the NR module is not able to use name-based strategies other than multicast forwarding, but this can be solved by adding a *Strategy Forwarder*. By default, the only logic used by the NR is to forward *Interest* packets to all *Faces* that can handle prefixes of these packets, which is equivalent to the multicast strategy in NFD.

Figure 2 shows an example of a more complex topology involving five different microservices. There are three points

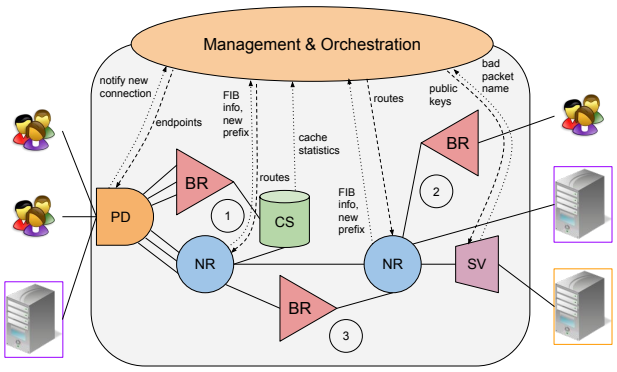

Figure 2: Example of a small managed μNDN network

of interest that are highlighted in the Figure. The first is the practical use of the previous topology that mimics NFD. We believe that this kind of chaining is more prone to be placed at the edge of the network to propose bidirectional *Faces* to clients. The second point of interest shows that it is possible not to follow the "default" NDN pipeline and only use what is needed. Here, a client cannot use the same *Face* to send consumer and producer traffic because of the oriented chaining so they must specialize their *Faces* for one kind of traffic (but they can still do both by using two or more *Faces*: at least one per traffic type). The third point of interest is when we decide to link two *Name Routers* together. This can create a routing loop like the one in our example, when the same prefix is available on both sides. To avoid packets to loop in the network, a *Backward Router* (or any efficient loop avoidance function) must be placed between them to avoid a broadcast storm (this can be automatically performed by the manager).

4 MANAGEMENT AND ORCHESTRATION OF MICROSERVICES

4.1 The manager

To enable the full potential of a microservice architecture, which heavily relies on the dynamic reconfiguration of the network according to the operational needs, a powerful manager is needed (or VNF Manager in NFV nomenclature). Even if microservices can be managed one at a time for simple tasks like editing their configuration, when the network grows, the need to have (at least) a third-party program to manage these microservices altogether increases. The manager is in charge of a part or of the whole autonomous system (AS) running μNDN. In particular the manager is in charge of these key operations to guarantee the performance and the security of the network:

- Deploy and dynamically link the microservices where and when needed;

Table 1: Summary of the main properties of each microservice

Name	Function	Oriented	Ingress/Egress cardinality
Name Router	Route *Interest* packets	Yes	1/N
Backward Router	Route back *Data* packets	No	N/1
Packet Dispatcher	Split *Interest*/*Data* traffic	Yes	N/N
Content Store	Cache *Data* packets	No	1/1
Strategy Forwarder	Forward *Interest* packets	No	1/1 or N
Signature Verifier	Verify packets' signature	No	1/1
Name Filter	Filter on packets' name	No	1/1

- Update their running configuration;
- Scale up the bottleneck modules accordingly;
- Deploy countermeasures in case of attack.

Each module has a management interface connected to the manager through the control plane. This interface can be implemented in different ways, for example by using the name of *Interest* packets as commands like NFD, or by using more common solutions like an HTTP RESTful API. If we want to follow the NFV paradigm, it may be better to use the second option since the first is specific to ICN. To react to network events, the manager needs to store and process information reported by the microservices. Table 2 lists some metrics that are useful to monitor. These values are used by the manager and/or the operator to improve the QoS of the network. For example, *Unsolicited Data* and cache statistics are used to detect some ICN attacks like the Content Poisoning Attack (CPA), while faces and routes statistics are used to adapt the topology over time.

Inside the managed network, a route propagation protocol like NLSR [10] is not mandatory because the manager can be in charge of route propagation routines and update the routes in a centralized SDN fashion each time a route (un)registration is notified by a NR module or when a NR joins or leaves the network. In fact, the manager just has to check the status of the known routes (static routes) and to change the configuration of NR modules that are concerned by these changes (dynamic routes). More precisely, any NR module that can reach the one that triggers the routine(s) will be concerned. For inter-AS route management, the deployment of specific microservices, for example supporting NLSR, and placed at the edge of the managed network is a good solution to have a protocol agnostic communication of routes between the manager and the microservices.

In μNDN, we use a centralized management architecture which is sufficient for our needs. Of course, more advanced management architectures are possible like [28] to avoid a single point of failure but are left for future works.

Table 2: Microservices' suggested values to monitor

Name	Values
Name Router	Route statistics
Backward Router	Unsolicited *Data* packets Retransmitted *Interest* packets
Packet Dispatcher	User traffic statistics
Content Store	Hit/Miss count
Signature Verifier	Name of failed packets
Name Filter	Drop count

4.2 Dynamic network changes

One of the most interesting properties of a managed microservices network is its ability to dynamically evolve, in particular by scaling up slow modules when congestion occurs, or by adding additional functions on the fly based on the operational needs. Concerning the scaling, two modules are typically expected to be slower than the others: SV and BR (what is confirmed by our evaluation in Section 5). The first because signature verification at line speed is known to be computation intensive, and the second because its stateful nature makes it the slowest part of NFD. Moreover, those two cases are representative on the way to scale modules with different ingress cardinalities, respectively 1 for SV and N for BR.

Figure 3 illustrates the way we expect the scaling to occur for a 1/1 function (SV for this example). The scaled function must be first surrounded by SR with a load balancing strategy to distribute packets among the modules and BR to aggregate the traffic from the multiple instances of the scaled function. In this way, the whole thing can be seen as a box with the same 1/1 cardinality and that contains the desired function. By definition [6], microservices can also directly exchange control messages with each other, which can be interesting when scaling some specific functions like CS: the scaled CS can collaborate with its counterparts to avoid duplicate cache entries in the local cluster they form.

Figure 4 illustrates the method to perform the scaling of the BR function. In the case of the BR function, the previous

method is not applicable because it would just move the bottleneck to the additional BR module used to ensure the connectivity consistency. It is still possible to perform the scaling for this function but it forces the next module to duplicate the return traffic toward the BR instances, what can impact its performances. On the ingress side, SF modules are deployed, one per ingress *Interest* traffic source, to maintain the routing property of the scaled function. With this method, it is possible to get duplicate traffic when a retransmission occurs, but this can be solved by adding a lightweight filter on SF modules to drop duplicate packets if required.

For functions that store dynamic information like CS and BR, a synchronization between the scaled modules may be needed, especially when a scale down occurs. In this case, the manager orders SF modules not to forward *Interest* packets to the BR module that will be destroyed and ask it to pass its table on the other module before destroying it, so that to minimize information loss that would result in forwarding errors and retransmissions. Another possibility is for the cluster of scaled modules to use a shared memory, like etcd.

Three different strategies (sequences of actions) are possible to insert a new module in an existing chain. To explain them, we consider the case when two modules M1 and M2 are already linked together and a third module M3 is inserted between them. In the first two strategies, the link between M1 and M2 is removed first so the choice must be made between dropping the incoming packets or buffering them, depending on the strategy. Finally, new links are set between M3 and M2 and between M1 and M3 to get the new chain. The third strategy consists of postponing the M1-M2 link deletion to directly create the links M3-M2 and M1-M3, and finally to remove the link M1-M2. This sequence may duplicate traffic during the last step, but the chain is never broken during the insertion process.

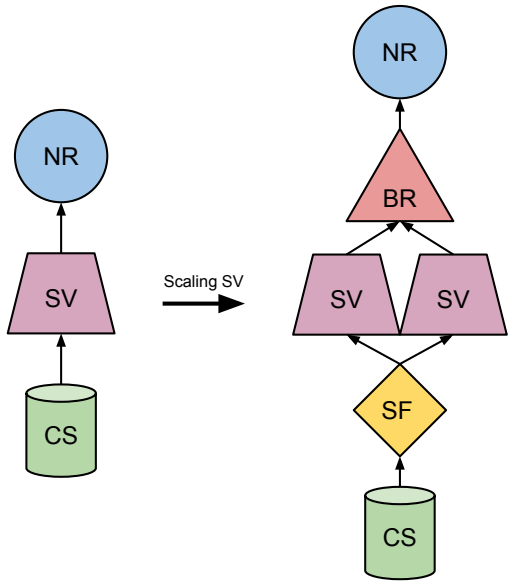

Figure 3: Scaling procedure of a *Signature Verifier* when strictly following connectivity constraints

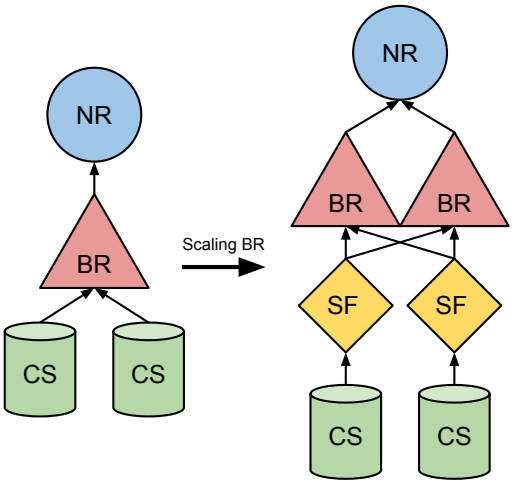

Figure 4: Possible scaling procedure of a *Backward Router*

5 EVALUATION

5.1 Experimental environment

The microservices are written in C++ while the manager is written in Python[2] and both are released in open source[3]. Most of the functionalities described in the paper is already implemented. The following paragraphs highlight some implementation points.

5.1.1 Implementation of the microservices. The microservices currently work as an overlay of IP because it allows faster prototyping than a lower layer development over Ethernet. Our modules can currently communicate indifferently over TCP or UDP to forward NDN packets, but we will support direct communications over Ethernet in the near future.

All the microservices are currently single-threaded but some of them could be easily multithreaded, even if horizontal scaling already allows to dedicate more resources to a function. In practice, the modules are free to connect to any number of endpoints so they do not strictly follow the cardinality defined in Section 3, which is only enforced by the manager. This may lead to other interesting function chaining patterns in the future, even if we did not have explored the full extent of it.

[2]with the help of Networkx, Twisted and Klein libraries
[3]https://github.com/Kanemochi/NDN-microservices

The longest prefix match used by BR, NR and NF functions and the shortest suffix match used by CS are implemented thanks to an indexed N-ary tree data structure using NDN name as key. The index is checked first to help saving time when a strict matching occurs. The other modules use flat data structures like arrays or hashmaps.

The behavior of the CS function is currently a bit simplified compared to NFD for an easier implementation. We made the following minor changes that only prevent consumers to retrieve outdated packets:

- Concerning *Interest* packets, we read the *MustBeFresh* field to decide if the CS must be used: if it is set to "true", the cache is bypassed;
- Concerning *Data* packets, we read the *freshness* field before taking the caching decision: if it is set to zero, the packet is never cached.

5.1.2 Implementation of the manager. The manager is divided in five parts: (1) the network view as a graph, (2) the orchestration engine, (3) the module management interface, (4) the key management, and (5) the REST API.

(1) The representation of the network topology takes the form of a directed graph that stores all the required management information, for instance: routes for *Name Router*, CPU usage of every module, etc. It has associated procedures to perform management tasks that are based on the graph like route propagation.
(2) The orchestration engine ensures that the microservices images and the virtual network are well deployed. It relies on the Virual Network Infrastructure (VNI), here a Docker engine, for the creation and the deletion of the containers. It is also in charge of retrieving on a regular basis containers statistics that are used to take management decisions, for example to trigger the scaling process.
(3) The module management interface is used to vehicle information between the manager and the modules. It identifies the containers and send asynchronous management messages (in JSON format) via an UDP socket. A proper RESTful management API is planned for future work.
(4) The key management engine acts as a PKI being in charge of the synchronization of the public keys with the different instances of the *Signature Verifier* function. It also handles the signature verification when a new route is asked by a producer.
(5) The REST API allows external software to get information from the network graph and to interact with the manager. Based on this API, we implemented a web-based GUI for the manager. The visualization of the dynamic topology of the managed network is implemented with D3.js and the different management functionalities can be used thanks to AJAX forms.

Table 3: Average throughput of each microservice

Module	Throughput (Mbps)	
	Bare-Metal	Container
Name Router	1,820	1,595
Backward Router	1,304	1,090
Packet Dispatcher	1,761	1,635
Content Store (freshness = 0)	1,760	1,538
Content Store (freshness > 0)	1,031	979
Content Store (from cache)	2,447	2,061
Strategy Forwarder	1,756	1,540
Signature Verifier (RSA2048)	515	401
Signature Verifier (ECDSA256)	122	101
Name Filter	1,804	1,593

5.2 Unit testing of microservices

For the following experiments, we use a DELL PowerEdge R730 server that features two Intel 2.4 GHz octo-core Xeon processors (E5-2630 v3) with Hyper-Threading and Turbo Boost enabled, 128GB of RAM and two 400GB SAS SSD in RAID0 for the operating system (Ubuntu 18.04 server) and Docker containers. The server uses Docker CE 18.03 and ndn-cxx library v 0.6.1. Packets are transported over TCP/IP and the containers are attached to a docker bridge virtual network. The size of *Data* packets is set to 8192 octet.

To know how many resources must be allocated by the orchestrator for each module and what are the possible bottlenecks, we have to evaluate the throughput of each one individually. In this experiment, we use a simple topology considered as the best-case scenario where the tested module is placed between a single consumer and a single producer running NDNperf instances [15]. Table 3 gives the achieved throughput for every module in two contexts: if they are executed on the host or inside a container. Some modules are given under different configurations when it is relevant.

Based on these results, we can identify that the *Signature verifier*, the *Backward Router* and the *Content Store* (in a specific case) functions are the slowest. As expected, the SV module is slow due to its computation intensive nature, but this is not critical since the verification is more prone to be done on demand and at the edge of the network when a suspicious traffic must be investigated. Concerning the *Content Store* function, because its inner operations have different costs, its tendency to become a bottleneck really depends on the cache hit ratio and the quantity of received *Data* packets that are candidates for caching (i.e. with a freshness value > 0). When all *Data* packets are candidates,

Table 4: Performance comparison between NFD and its microservice equivalent

	Microservices				NFD
	PD	CS	BR	NR	
%CPU core usage	100	59	89	64	100
Throughput (in Mbps)	776				527
Latency (in ms)	2,63				3,88

a *Content Store* becomes slower than the *Backward Router* when the cache hit is lower than 20% due to the increased time spent to store *Data* packets in its table. The results in Table 3 also show an average performance degradation of 13% when the microservices are run in containers, what was expected and seems to be reasonable.

Then, in Table 4 is given a performance comparison between an NFD instance the minimal "corresponding" μNDN instance presented in Figure 1. It is not really fair to compare the two directly because their advanced functionalities differ, but it gives a first idea of μNDN capacity. We can see that the equivalent μNDN chain is faster than the NFD instance (around 50% better throughput) and has a lower processing latency, but at the price of a higher CPU consumption (3 times more CPU cycles). In this case, the *Packet Dispatcher* is clearly the bottleneck. This is a limitation of this specific network topology (Figure 1) where the PD module has to handle twice the traffic of both the consumer and the producer (because they are on the same side of the network), compared to the other modules that handle each packet only once. However, like said before, the PD function is not complex and could be easily multithreaded. When PD is not the bottleneck (for example, when the producer is directly accessible through NR), the reported throughput becomes 968 Mbps with associated CPU usages of 67, 100 and 71 respectively for CS, BR (the new bottleneck) and NR.

5.3 Performance management

To experiment the scaling properties of μNDN, we built a network with 3 modules in the following order: a *Content Store*, a *Backward Router* and a *Name Router* (same as Figure 4 but with only one CS). In this experiment, the *Content Store* is used as a pure traffic forwarder (no caching possible). We also limited the CPU usage of the BR modules to 67% of a single core to artificially accentuate the bottleneck and better see the effect of the scaling process. Each time a scalable function meets the following conditions, it can be scaled up: (1) it must not be at the edge of the network (to avoid breaking TCP connections outside the managed network), (2) it must be defined as scalable (automatically deployed functions are not defined as scalable by default), and (3) it must have an associated CPU usage above 80% of its capacity over the last reporting period. If all the conditions are met, an orchestration routine scales up the function. In the same way, when a scaled function has a CPU usage below 20%, an orchestration routine scales it down. The scaling routine is executed every 20 seconds.

Figure 5 shows the throughput reported by the consumer and the CPU usage of all microservices that constitute the network. At the beginning, a consumer is started with a window of 32 *Interests* packets to overload the network, resulting in an average throughput of 635 Mbps limited by the *Backward Router* (BR1, which has reached its CPU capacity of 67%). Then, the scale up process is triggered and follows the procedure described in Section 4: the manager first creates a *Strategy Router* and inserts it between the *Content Store* and the actual *Backward Router*. Then it deploys up to 2 additional BR modules (BR1.1 and BR1.2) and links them to the newly deployed SF and the NR modules. During the scaling process, we can notice that the throughput increases with around 960 Mbps for a scaling factor of 2 (when BR1.1 is created), but the throughput then decreases to 715 Mbps when the third BR module is created. This can be explained because the NR module uses already 100% of its allocated CPU core and has no more left capacity to forward the additional traffic. However our scaling procedure could be improved by taking into consideration the topology and the dynamic load of the other modules. In this way, it would not have added the third useless BR module knowing that NR was already at its limit. Finally, we slow down the consumer by setting an *Interest* packet window of 1 to trigger the scale down process. This is illustrated in the last third of Figure 5: the clones are removed when their reported CPU usage reaches 0% (and their respective curves do not continue past this point). At the end of the experiment, the network is in the same state as it was at the beginning.

5.4 Security management

The goal of this experiment is to test the capability of μNDN to dynamically deploy a countermeasure against a Content Poisoning Attack (CPA). We built a network that leaves some space for a CPA similar to the one that can affect NFD [17], as depicted in Figure 6. It is made of a chain with two *Content Stores* and one *Name Router* modules, the CS modules are placed at the edges of the network and the *Name Router* is placed in the middle. The flaw of this network is that a CS is directly connected to the producer side of NR. More precisely, once a good provider connects to CS and registers its prefix, the NR module routes packets to the good provider through the CS that will forward the *Interest* packets that do not match any cache entry to all known endpoints (because a CS does not know the concept of a route). Consequently, if a bad provider connects to the same CS, it can take advantage of the

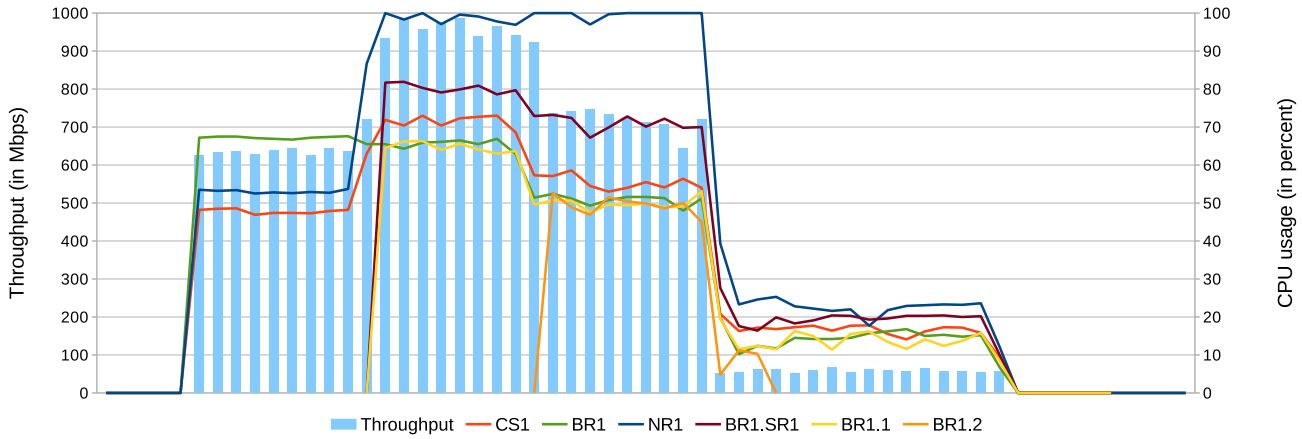

Figure 5: Throughput and CPU usage of the network during a scaling event

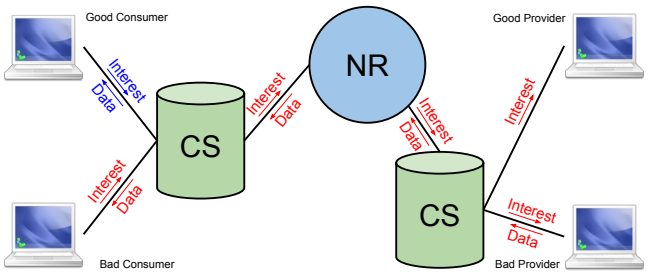

Figure 6: Content poisoning scenario

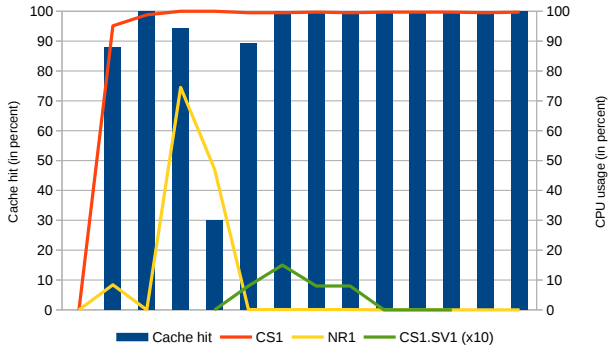

Figure 7: Cache hit over time under Content Poisoning Attack

already created route of the good provider (preventively, the NR module drops any *Data* packet received from a *Face* that has not previously registered the concerning prefix properly). Like in [17], the attacker collaborates with a bad consumer to perform the CPA: the bad consumer asks for bad contents to be inserted in the cache. The good consumer is configured to ask a limited range of content to set the cache hit to 100%.

Figure 7 shows a histogram that represents the cache hit of the consumer-side CS and different curves that represent the CPU usage of the modules (we do not plot the producer-side CS for the readability). Once all the contents in the range are cached, the cache hit reaches 100%. When the producer-side CS fully serves the consumers, no more traffic is forwarded to NR, which explains the 0% CPU usage. Then, the bad consumer starts the cache pollution which results in a decrease of the cache hit and an increase of the CPU usage of the other modules in the network. The first report of the cache hit ratio to the manager after the attack does not show enough difference to appear suspicious (third bar), but the next one shows a drop of the cache-hit from 94% to 30%, triggering the verification process. At this moment, the manager asks the orchestrator to deploy a SV module between the consumer-side CS and NR (the location where the problem was detected). SV is configured with the public keys of good producers and will drop any packet that fails the verification process. Its execution makes the cache hit quickly come back to its original value, mitigating the attack. At the end, the bad consumer should stop after repeated timeouts. Finally, the manager restores the network back to its original state by removing SV when it does not receive reports of fake *Data* packets for 10 periods (the report period is set to 2 seconds). For an efficient usage of SV modules in the network, a manager should be able to move them dynamically over time (and/or deploy new ones) toward the source of the attack, thus limiting the verification only to the smallest concerned part of the network.

6 DISCUSSION

With hindsight, our most questionable design choice is the separation of the PIT and FIB functions in two distinct modules because it resulted in some significant drawbacks. We had to define specialized *Faces* for a specific incoming traffic (*Interest* or *Data*) which makes these modules "oriented" (see Table 1) if we want to avoid traffic broadcast (*Interest* for the PIT and *Data* for the FIB). Moreover, the asymmetric cardinality of NR (1/N) and BR (N/1) makes them harder to scale up separately. Indeed, scaling routing functions like BR (NR was not shown but it is like a fusion of the two scaling methods described with the additional constraint to have consistent routes between scaled instances) increases the complexity of the network management and does not appear to be very efficient according to our experiments, even if our implementation still needs to be optimized.

A unified module regrouping PIT and FIB functions appears to be the most reasonable choice for the moment. This forwarding module could perform both actions on all its *Faces*, thus reducing the network complexity. An intermediate solution to the development of a whole new PIT+FIB component is to regroup our three core routing functions (NR, BR and PD) in a single VM or container, as it is possible in NFV to build a Virtualized Network Function with several VNF Components inside. The standalone BR can be kept for very specific scenarios like in [26] where the PIT is enhanced with off-path functionalities to improve content delivery. By opposition, all the on-path support functions (including the CS) can easily take advantage of the microservice paradigm.

7 CONCLUSION

In this paper, we presented μNDN, an alternative way to implement NDN forwarding and support functions using microservices, in order to make NDN fully benefit from NFV features. Our first work was to split the three key functions of the NDN forwarder into dedicated Virtualized Network Functions. To have a complete architecture, we also developed a manager as well as other microservices to cover important needs of a network operator related to performance and security. The microservices make it possible to design more efficient NDN networks by only deploying the right function where and when needed but at the price of a greater management complexity, i.e. more chaining and orchestration tasks, to build and operate the network.

Even if a direct comparison with NFD is biased by the specific features enabled on both sides, our evaluation still highlighted the benefit of the microservice architecture. Indeed, splitting the functions resulted in a greater throughput than the current single-threaded implementation of NFD without much optimization effort. We also showed that our orchestrator can scale-up a bottleneck component and add security modules on the fly to mitigate performance or security issues. Finally, the source code of the different pieces of software composing μNDN is released in open-source.

Nearly all the functionalities presented in this paper are implemented but a few missing are still under development. In particular, in our future work, we will make our microservices directly communicate over Ethernet to avoid any TCP/IP overlay and keep optimizing our architecture to improve chaining and packet delivery. We will also make our architecture compatible with NFV standards like the TOSCA description language. Finally, we are interested in exploring further the possibilities offered by μNDN by adding new functionalities to NDN as microservices.

Acknowledgement

This work is partially funded by the French National Research Agency (ANR), DOCTOR project, under grant <ANR-14-CE28-0001>.

REFERENCES

[1] Elian Aubry, Thomas Silverston, and Isabelle Chrisment. 2017. Implementation and Evaluation of a Controller-Based Forwarding Scheme for NDN. In *AINA 2017 - IEEE 31st International Conference on Advanced Information Networking and Applications*. IEEE, Taipei, Taiwan, 144 – 151. https://doi.org/10.1109/AINA.2017.83

[2] Carlos M.S. Cabral, Christian Esteve Rothenberg, and Maurício Ferreira Magalhães. 2013. Mini-CCNx: Fast Prototyping for Named Data Networking. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-centric Networking (ICN '13)*. ACM, New York, NY, USA, 33–34. https://doi.org/10.1145/2491224.2491236

[3] Théo Combe, Wissam Mallouli, Thibault Cholez, Guillaume Doyen, Bertrand Mathieu, and Edgardo Montes De Oca. 2017. A SDN and NFV use-case: NDN implementation and security monitoring. In *Guide to Security in SDN and NFV*. Springer. https://hal.inria.fr/hal-01652639

[4] Namiot Dmitry and Sneps-Sneppe Manfred. 2014. On micro-services architecture. *International Journal of Open Information Technologies* 2, 9 (2014).

[5] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, K.C. Ng, Vyas Sekar, and Scott Shenker. 2013. Less Pain, Most of the Gain: Incrementally Deployable ICN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/2486001.2486023

[6] Martin Fowler. 2014. Microservices a definition of this new architectural term. (2014). https://martinfowler.com/articles/microservices.html

[7] Network Functions Virtualisation Industry Specification Group. 2013. *Network Functions Virtualisation (NFV)*. White Paper. ETSI. https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV_White_Paper2.pdf

[8] Network Functions Virtualisation Industry Specification Group. 2014. *Network Functions Virtualisation (NFV); Management and Orchestration*. Group Specification GS NFV-MAN 001. ETSI. http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf

[9] P. H. V. Guimaraes, L. H. G. Ferraz, J. V. Torres, D. M. F. Mattos, M. P. Andres F., M. E. Andreoni L., I. D. Alvarenga, C. S. C. Rodrigues,

and O. C. M. B. Duarte. 2013. Experimenting Content-Centric Networks in the future internet testbed environment. In *2013 IEEE International Conference on Communications Workshops (ICC)*. 1383–1387. https://doi.org/10.1109/ICCW.2013.6649453

[10] AKM Hoque, Syed Obaid Amin, Adam Alyyan, Beichuan Zhang, Lixia Zhang, and Lan Wang. 2013. NLSR: named-data link state routing protocol. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*. ACM, 15–20.

[11] Kevin Khanda, Dilshat Salikhov, Kamill Gusmanov, Manuel Mazzara, and Nikolaos Mavridis. 2017. Microservice-based iot for smart buildings. In *Advanced Information Networking and Applications Workshops (WAINA), 2017 31st International Conference on*. IEEE, 302–308.

[12] Michał Król and Ioannis Psaras. 2017. NFaaS: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ACM, 134–144.

[13] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. 2016. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175* (2016).

[14] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGPLAN Not.* 48, 4 (March 2013), 461–472. https://doi.org/10.1145/2499368.2451167

[15] Xavier Marchal, Thibault Cholez, and Olivier Festor. 2016. Server-side performance evaluation of NDN. In *3rd ACM Conference on Information-Centric Networking (ACM-ICN'16)*. ACM SIGCOMM, ACM, Kyoto, Japan, 148 – 153. https://doi.org/10.1145/2984356.2984364

[16] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 459–473.

[17] Tan Nguyen, Xavier Marchal, Guillaume Doyen, Thibault Cholez, and Rémi Cogranne. 2017. Content Poisoning in Named Data Networking: Comprehensive Characterization of real Deployment. In *15th IFIP/IEEE International Symposium on Integrated Network Management (IM2017)*. Lisbon, Portugal, 72–80. https://doi.org/10.23919/INM.2017.7987266

[18] Xuan Nam Nguyen, D. Saucez, and T. Turletti. 2013. Efficient caching in Content-Centric Networks using OpenFlow. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 67–68. https://doi.org/10.1109/INFCOMW.2013.6562846

[19] D. Perino, M. Gallo, R. Laufer, Z. B. Houidi, and F. Pianese. 2016. A programmable data plane for heterogeneous NFV platforms. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 77–82. https://doi.org/10.1109/INFCOMW.2016.7562049

[20] A. Rahman, D. Trossen, D. Kutscher, and R. Ravindran. 2018. *Deployment Considerations for Information-Centric Networking*. Internet-Draft draft-irtf-icnrg-deployment-guidelines-00. Internet Research Task Force. https://tools.ietf.org/html/draft-irtf-icnrg-deployment-guidelines-00 Work in Progress.

[21] R. Ravindran, A. Chakraborti, S. O. Amin, A. Azgin, and G. Wang. 2017. 5G-ICN: Delivering ICN Services over 5G Using Network Slicing. *IEEE Communications Magazine* 55, 5 (May 2017), 101–107. https://doi.org/10.1109/MCOM.2017.1600938

[22] Chris Richardson. 2017. Microservice Architecture. (2017). http://microservices.io/patterns/monolithic.html

[23] S. Salsano, N. Blefari-Melazzi, A. Detti, G. Morabito, and L. Veltri. 2013. Information Centric Networking over SDN and OpenFlow: Architectural Aspects and Experiments on the OFELIA Testbed. *Comput. Netw.* 57, 16 (Nov. 2013), 3207–3221. https://doi.org/10.1016/j.comnet.2013.07.031

[24] Mauro Sardara, Luca Muscariello, Jordan Augé, Marcel Enguehard, Alberto Compagno, and Giovanna Carofiglio. 2017. Virtualized ICN (vICN): Towards a Unified Network Virtualization Framework for ICN Experimentation. In *Proceedings of the 4th ACM Conference on Information-Centric Networking (ICN '17)*. ACM, New York, NY, USA, 109–115. https://doi.org/10.1145/3125719.3125726

[25] Manolis Sifalakis, Basil Kohler, Christopher Scherb, and Christian Tschudin. 2014. An information centric network for computing the distribution of computations. In *Proceedings of the 1st ACM Conference on Information-Centric Networking*. ACM, 137–146.

[26] Vasilis Sourlas, Leandros Tassiulas, Ioannis Psaras, and George Pavlou. 2015. Information resilience through user-assisted caching in disruptive content-centric networks. In *IFIP Networking Conference (IFIP Networking), 2015*. IEEE, 1–9.

[27] Yi Sun, Seyed Kaveh Fayaz, Yang Guo, Vyas Sekar, Yun Jin, Mohamed Ali Kaafar, and Steve Uhlig. 2014. Trace-Driven Analysis of ICN Caching Algorithms on Video-on-Demand Workloads. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. ACM, New York, NY, USA, 363–376. https://doi.org/10.1145/2674005.2675003

[28] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. 2015. An Architecture for Self-managing Microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud (AIMC '15)*. ACM, New York, NY, USA, 19–24. https://doi.org/10.1145/2747470.2747474

[29] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload characterization for microservices. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 1–10.

[30] M. Vahlenkamp, F. Schneider, D. Kutscher, and J. Seedorf. 2013. Enabling ICN in IP networks using SDN. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*. 1–2. https://doi.org/10.1109/ICNP.2013.6733634

[31] N. L. M. van Adrichem and F. A. Kuipers. 2015. NDNFlow: Software-defined Named Data Networking. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. 1–5. https://doi.org/10.1109/NETSOFT.2015.7116131

[32] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 66–73. https://doi.org/10.1145/2656877.2656887