





# Micro-Chain: A Cluster Architecture for Managing NDN Microservices

Otávio A. R. da Cruz   [ Federal University of Rio Grande do Sul | [otavio.augusto@ufrgs.br](mailto:otavio.augusto@ufrgs.br) ]


Antonio A. S. da Silva  [ Federal University of Rio Grande do Sul | [aassilva@inf.ufrgs.br](mailto:aassilva@inf.ufrgs.br) ]

Paulo Mendes  [ Airbus | [paulo.mendes@airbus.com](mailto:paulo.mendes@airbus.com) ]


Denis L. do Rosário  [ Federal University of Pará | [denis@ufpa.br](mailto:denis@ufpa.br) ]

Eduardo C. Cerqueira  [ Federal University of Pará | [cerqueira@ufpa.br](mailto:cerqueira@ufpa.br) ]

Julio C. S. dos Anjos  [ Federal University of Ceara Campi Itapaje | PPGETI | [jcsanjos@ufc.br](mailto:jcsanjos@ufc.br) ]

Carlos E. Pereira  [ Federal University of Rio Grande do Sul | [cpereira@ece.ufrgs.br](mailto:cpereira@ece.ufrgs.br) ]

Edison P. de Freitas  [ Federal University of Rio Grande do Sul | [edison.pignaton@ufrgs.br](mailto:edison.pignaton@ufrgs.br) ]

 Programa de Pós-graduação em Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Av. Osvaldo Aranha, 103, Centro Histórico, Porto Alegre, RS, 90035190, Brasil.

**Received:** 16 December 2023 • **Accepted:** 26 July 2024 • **Published:** 03 October 2024

## Abstract

Network Functions Virtualization (NFV) and Information-Centric Networking (ICN) are promising networking paradigms for the future of the Internet. Concurrently, microservice architecture offers an attractive alternative to monolithic architecture for software development. This work addresses a scenario composed of these concepts, where an ICN network must be deployed and managed using ICN microservices. In this scenario, ICN microservices must be created, connected, configured, and monitored at runtime, which is not trivial. To address these challenges, this work proposes Micro-Chain, an architecture for deploying, scaling, and linking ICN microservices. The architecture consists of four modules, relationships between them, and core operations. A Micro-Chain implementation is presented as proof of concept, which has a threshold-based scaling process and a placement method to minimize the number of hops for an ICN microservice chain. The evaluation assesses a scale-on-demand scenario in a cluster with three nodes. The results demonstrate that 1) the developed solution can scale on demand, 2) the communication overhead is 0.632%, and 3) the placement of microservices affects network performance.

**Keywords:** Information-Centric Networking, Network Functions Virtualization, Microservice Architecture, Node Cluster, Network Management

## 1 Introduction

The traditional location-based communication model employed by TCP/IP is not fully compatible with the Internet's operation, particularly with the Web protocols that use name-based content requests. Information-Centric Networking (ICN) appears as an alternative to solve these issues. Specifically, ICN adopts a request-response communication model that uses content data names directly at the network layer, providing a more appropriate architecture for content acquisition on the Internet. As a promising candidate for the future of Internet communication, ICN has the potential to transform the delivery of network services and accommodate the evolving digital landscape [Singh and Ujjwal, 2020].

ICN offers substantial benefits in network architecture compared to the TCP/IP paradigm. For instance, ICN devices store content locally, leading to faster and more reliable content retrieval, reducing latency, and improving delivery performance, especially for frequently accessed content. The distributed caching approach improves network resilience, guaranteeing continuous access to content even during node or link failures. The capability of ICN to separate content from specific locations enables user access to content at different network points, which can be leveraged in scenarios involving mobile users and devices [Singh and Ujjwal, 2020].

The widespread adoption and deployment of ICN face several challenges and obstacles. One of the primary challenges is the fact that applications are typically IP-based. Adapting these applications to be compatible with ICN may require extensive adaptations. Therefore, there must be robust support to stimulate companies' adoption of ICN. Another challenge to consider is the required replacement of network hardware, including routers, switches, and other network equipment, which requires both time and monetary resources. Within this context, the implementation of ICN can be leveraged by using Network Functions Virtualization (NFV) [Marchal *et al.*, 2018]. Specifically, generic hardware can implement ICN network functions and other existing protocols, facilitating the gradual transition to this new paradigm while minimizing long-term expenses.

Traditionally, software is implemented using a monolithic architecture characterized by high interdependencies among their components and functionalities [Alencar *et al.*, 2022]. The tightly coupled nature of the monolithic architecture harms application maintenance and updates, leading to potential inefficiencies in system management. Moreover, applications over monolithic resemble singular software entities, requiring complete application replication for horizontal scaling. However, these applications comprise multiple software components with distinct resource demands, resulting

in the inability to scale up these components independently and inefficiencies in resource utilization. Addressing these limitations is crucial, which has led to exploring alternative software design approaches to improve scalability, resource usage efficiency, and overall application robustness [Cerny *et al.*, 2022].

Microservices architecture emerged primarily to address the scalability and interdependence issues associated with the monolithic approach [Cerny *et al.*, 2022]. This architecture divides an application (or service) into small, loosely coupled modules that can be independently developed and deployed. Compared to the monolithic architecture, the microservices architecture can provide advantages in terms of scalability, resource usage optimization, modularity, isolation, and flexibility. When using microservices, each system module (microservice) can be scaled independently according to its requirements. For example, highly-trafficked microservices can be scaled without impacting others. Modern network approaches are exploring the concept of microservices to enhance network resource utilization and meet the demanding needs of diverse applications [Marchal *et al.*, 2018]. In addition, the small size of the microservices contributes to understandability. It enables teams to focus on specific system parts, speeding up the development process and improving maintainability and code reuse.

In this context, Virtualized Network Functions (VNF)s can be implemented from a microservices architecture for enhancing the use of ICN [Marchal *et al.*, 2018], where the advantages of both concepts can be leveraged. However, managing and orchestrating microservices poses considerable challenges. Microservices must be deployed, connected, and monitored in runtime. Additionally, current solutions focus on managing and orchestrating cloud applications. Since VNFs microservices handle network operations, modifications in the data link and network layers are required. To support these requirements, it is necessary to define an architecture that determines an organization of entities and their relationships to perform these operations.

This paper proposes Micro-Chain, an architecture to orchestrate and manage ICN microservices. More specifically, it deals with the ICN implementation called Named Data Networking (NDN) [Zhang *et al.*, 2014]. Micro-Chain defines a group of modules and their relationships for managing and monitoring NDN microservices across a cluster of nodes. For this, concepts and technologies used in cloud computing, such as Kubernetes and Prometheus, are extended. Micro-Chain operations are tested in a scaling scenario with multiple nodes, which employs several of the operations suggested for the architecture. In this scenario, a microservices placement problem that degrades communication performance is identified.

The main contribution of this work is the development of an architecture for deploying and managing an NDN local network composed of multi-nodes and based on microservices. The secondary contributions are:

- An architecture with an online implementation accessible for validating other solutions based on NDN. The implementation leverages tools like Kubernetes and Prometheus, benefiting from robust community support

and regular updates.

- The development of a method for placing microservices.
- A discussion on how to deal with NDN microservices and identify challenges and future work.

The remainder of this paper is organized as follows. Section 2 explains the main concepts necessary to understand the proposed solution. Section 3 presents the related work. Section 4 describes the proposed architectural solution and the implementation details of this architecture. Then, in Section 5, the addressed application scenario is defined, the proposed architecture is evaluated, and the results are presented. Finally, Section 6 discusses the outlines, and Section 7 concludes the paper.

## 2 Background

This section presents the main concepts about NDN and its basic functioning. Subsequently, a set of NDN microservices and a solution to manage them on demand are described.

### 2.1 Named Data Networking

NDN is one of the most used implementations of ICN, where the communication in an NDN is based on two types of packets: Interest and Data. First, a client requests content by sending an interest packet containing the content name. This Interest travels through the NDN, looking for the requested content. Then, the device that receives the Interest and has the content stored responds by sending the Data packet with the requested content. NDN devices execute three data structures and forwarding policies:

- **Content Store (CS):** A temporary cache of received Data packets to satisfy future Interests. CS employs a strategy to define whether content should be stored locally;
- **Forwarding Information Base (FIB):** It contains information about how to route Interest to get closer to the data source based on their names. A name prefix-based routing protocol populates the FIB and can have multiple outgoing interfaces and next-hop for each prefix;
- **Pending Interest Table (PIT):** Stores all Interests forwarded but not yet satisfied. Each PIT entry records the name of the data transported in the Interest, together with its input and output interface(s) to assist in routing the Data packets along the reverse path of the Interest.

Fig. 1 shows the operation of an NDN device according to the incoming packets received. When an Interest packet (red arrow) arrives at the device, it is first checked whether the content is stored locally in the CS. The content is sent to the requesting party using a Data packet (green arrow) if it exists. If the content does not exist locally, the device continues to check whether there is any entry in the PIT for the name of the requested content. If existent, the input face of Interest is added to PIT. If there is no entry in the PIT, the device checks the FIB for the next hop and forwards the packet. The packet

can also be responded to with a NACK or discarded if there is no adequate match in the FIB.

In addition, when a Data packet arrives at the device, it is checked whether the name is registered in the PIT. If it exists, the data packet is forwarded to all registered faces. In parallel, the CS can store data according to the defined strategy. The data is discarded if no record in the PIT exists.

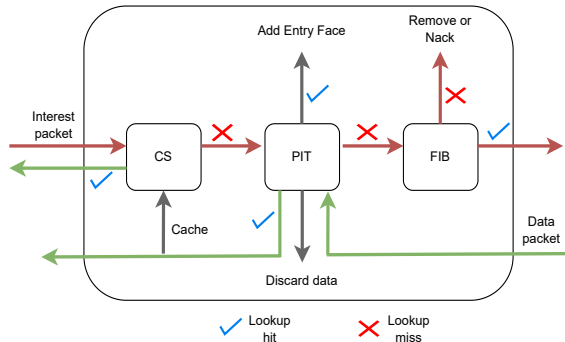


Figure 1. Packet processing and forwarding by an NDN device.

In more dynamic networking scenarios, as is the case of satellite networks, as shown by Dyerowicz and Mendes [2017], such NDN basic operation may need to be enhanced with other functions, such as push-communication mechanisms, aiming to reduce the communication overhead. Therefore, having the NDN engine developed based on microservices may bring advantages.

## 2.2 NDN microservices

The Microservices architecture is frequently used to develop services that execute information processing functions distributed across network nodes. As this architectural approach gained prominence, its applicability extended into the network domain. In this context, protocols developed through the microservices paradigm are employed to execute network tasks and operations. These protocols utilize Microservices architecture's inherent modular and scalable characteristics to improve system adaptability on demand.

Marchal *et al.* [2018] developed a set of seven NDN microservices from the monolithic Named Data Networking Forwarding Daemon (NFD) implementation. The microservices are:

- **Name Router (NR)**: It forwards an interest packet to the next device towards the server. It is similar to the FIB of NDN, and it only requires microservice because it is the only one that can listen to route announcements from the server.
- **Backward Router (BR)**: Used to forward a Data packet towards the requesting client. This function is similar to the PIT of NDN.
- **Content Store (CS)**: This function aims to store Data packets that pass through it to reuse them later.
- **Strategy Forwarder (SF)**: This function is used to forward packets to one or more selected destination(s) based on a given strategy. The strategies can be load-balancing, failover, or more specific to NDN.
- **Packet Dispatcher (PD)**: Responsible for forwarding each type of traffic to different outputs. This module

selects the correct processing pipeline and is best suited to handle external traffic at the network's edge.

- **Signature Verifier (SV)**: NDN data packets have a signature field to prevent cyber attacks, and this function verifies this signature.
- **Name Filter (NF)**: This function is a filter used to avoid specific packets on the network based on name.

The authors also developed a runtime Manager responsible for the NDN control plane and on-demand operations. More specifically, the operations performed by the Manager are i) **Horizontal scaling**: It increases or decreases the number of microservice instances based on CPU usage; ii) **Create and destroy microservices**: New microservices can be deployed or destroyed as needed, for example, during the scaling up process, when new instances of microservices are created; iii) **Create and destroy links**: Microservices are connected in sequences to enable a set of desired functionalities; iv) **Monitor metrics**: Each microservice can provide informative metrics to the Manager for decision-making.

Additionally, to compare performance between monolithic NFD and microservices NDN, Marchal *et al.* [2018] performed experiments evaluating CPU usage, throughput, and latency of both. The results indicate that microservices have higher throughput and lower latency at the cost of higher CPU consumption. Therefore, a trade-off must be considered when evaluating which architecture to use for a given scenario.

The work developed by Marchal *et al.* [2018] presents significant contributions to the usage and management of NDN microservices. However, the evaluation was conducted on a single machine, which does not align with real applications like the Internet. In this context, NDN microservices require deployment and management across multiple devices, which is more challenging.

To address the limitations of the work by Marchal *et al.* [2018], da Cruz *et al.* [2024b] proposed Micro-Chain, an architecture designed to support the deployment and scaling of NDN microservices in a cluster. However, the work lacks sufficient architectural detail, making it challenging to comprehend and, consequently, to utilize and reproduce.

In another work, da Cruz *et al.* [2024a] utilized NDN microservices to deploy and manage an NDN network in a military network setting, where a device is required to be replaced due to its low battery level. However, this work does not address the automatic scale of NDN microservices.

## 3 Related Works

According to the motivation and goals of this work, this section discusses works in the contexts of NFV, Software-Defined Networking (SDN), ICN, and microservice architecture; management and orchestration in NFV context; and placement problems.

### 3.1 NFV, SDN, ICN, and Microservice Architecture

Aldaoud *et al.* [2023] investigated the potential of integrating ICN, NFV, and SDN technologies. The work concludes that

combining these concepts can enhance networks' reliability, scalability, mobility, flexibility, and robustness. In addition, the authors discuss the utilization of ICN, NFV, and SDN in the context of 5G, 6G, and satellite networking, where these technologies have the potential to improve performance.

In ICN, the use of NDN has been explored in several contexts. The authors Wang *et al.* [2021] and Dulal *et al.* [2022] focus on NDN data plane or change part of this plane to enable communication between devices. Additionally, Qi and Wang [2023] and Fang and Wolf [2023] deal with the executing functions based on the NDN content naming and forwarding process. Conversely, other studies approach the NDN control plane. One of the strategies used in this regard is a SDN controller modified to support NDN operations.

Khalid *et al.* [2023] and Kalafatidis *et al.* [2022] extend the functionalities of an SDN controller for NDN, where the controller is mainly responsible for configuring routes between network devices. However, the authors do not address the challenge of scaling on demand, *i.e.*, dynamically adjusting network resources in response to system requirements.

Microservices is an architecture traditionally used to develop applications in cloud environments. However, Chowdhury *et al.* [2019], Nekovee *et al.* [2020] and Marchal *et al.* [2018] addressed the use of microservice to develop VNFs. The authors emphasize that this approach leverages microservice characteristics, such as independence, modularity, and continuous delivery, for NFV.

This work addresses the integration of ICN and microservices architecture concepts. The interest of previous works in addressing these concepts, whether jointly or separately, indicates their importance for current and future networks. Furthermore, the works presented in this section argue that microservices architecture is a viable option for developing VNFs. This approach can potentially improve VNFs flexibility, scalability, maintenance, development, and redundancy.

### 3.2 Management and Orchestration in NFV Context

In their review of management and orchestration in NFV architecture, Kaur *et al.* [2022] highlighted that container technology has been gaining attention for developing VNFs, mainly due to the gains in scalability, resource utilization, and agile software development presented by containers. These characteristics make containers more suitable than Virtual Machines (VMs) when: 1) Applications were developed using microservices architecture; 2) There is a need to minimize the number of servers; and 3) VNFs have only one operating system. However, Kaur *et al.* [2022] also emphasize that containers suffer from security issues due to kernel sharing and are an early-stage technology.

Cziva and Pezaros [2017] compared the performance of VMs (XEN dom0, clickOS, KVM virtio, XEN domU, KVMel000) and containers. The results show that containers have: 1) The third smallest delay, with KVM and XEN presenting the lowest values; 2) A shorter initiation time, consuming 10 seconds to instantiate and start 50 VNFs, while VM XEN consumed 40 seconds to instantiate; 3) The lowest memory consumption, spending 2.21 MB per VNF, while ClickOS consumed more than twice that amount. Conse-

quently, containers emerge as an interesting approach for resource-constrained scenarios, such as at the network edge.

Mai *et al.* [2019] present a proposal to implement, manage, and protect an NDN network. Each node in the network corresponds to a containerized NFD. The solution is evaluated dynamically in a content poisoning and scaling attack scenario. The scaling is performed according to the size of the PIT. While the PIT size indirectly indicates a container's stress, it is more advantageous to scale based on CPU and memory consumption in this case. CPU and memory-based scaling is a common approach to scaling microservices in the cloud.

Lema *et al.* [2019] proposed an architecture to handle the dynamic deployment of a named function network and network services for ICN infrastructure. The developed approach is based on Topology and Orchestration Specification for Cloud Applications (TOSCA), which defines VNF behavior in VMs. Lema *et al.* [2019] highlight that the solution developed lacks more mature monitoring and automatic scaling functionalities.

The European Telecommunications Standards Institute (ETSI) developed NFV Management and Orchestration (MANO) [ETSI, 2014], an architecture for management and orchestration in the NFV context. MANO as a reference architecture inspired the development of solutions such as Open Network Automation Platform (ONAP) and Open Source MANO (OSM). However, ONAP and OSM are far from complete or stable. Furthermore, implementing solutions using these tools requires experience in cloud computing network platforms, making the process non-trivial [Yilma *et al.*, 2020; Kaur *et al.*, 2022].

This section presents works related to the use of containers for NFV, as opposed to the usual VMs approach. Containers are lighter and, therefore, more suitable for applications where resources are scarce, such as at the network edge. Furthermore, both Mai *et al.* [2019] and Lema *et al.* [2019] posit that MANO-based solutions are typically evaluated in the context of IP networks, necessitating adjustments when applied to ICN scenarios.

### 3.3 Placement Problem

In cloud environments, containerized microservices-based applications commonly utilize Kubernetes as a tool for deployment, scaling, and management [Ghorab and St-Hilaire, 2022; Abdollahi Vayghan *et al.*, 2018; Rossi *et al.*, 2020]. However, Kubernetes has some limitations. One of these limitations is that Kubernetes alone does not perform runtime operations, such as custom configurations and establishing links between microservices. As these operations depend on the use case, developing applications that work together with Kubernetes to implement the business logic is necessary.

Another problem presented by Kubernetes is its method for the placement of microservices. Kubernetes has a placement method that does not consider network conditions [Ding *et al.*, 2023; Pallewatta *et al.*, 2022]. Addressing the microservices placement problem, Pallewatta *et al.* [2022] present a scalable QoS-aware application scheduling policy for fog devices. Obtained results indicate an up to 35% improvement in makespan satisfaction and up to 70% improve-

ment in budget satisfaction. Similarly, Ding *et al.* [2023] also addresses this problem by developing a method for positioning with dynamic resource allocation to minimize the sum of resources and communication costs. Both works address network usage as a parameter, such as delay between microservices or nodes.

## 4 Micro-Chain Architecture

This section begins by presenting a scenario based on satellite systems, explaining why it is interesting to use ICN microservices. Subsequently, a conceptual definition of the Micro-Chain architecture is outlined, summarizing its modules and interrelationships. Subsequently, the implementation details of each module are presented. Finally, the core operations that enable the management of NDN microservices across a cluster are described.

### 4.1 Application Scenario

In satellite systems, the space segment, as illustrated in Fig. 2, focuses on relaying data, having limited capabilities to store and process data. Hence, satellites are part of a more extensive system in which they are in direct or indirect contact with ground stations to download data in real time.

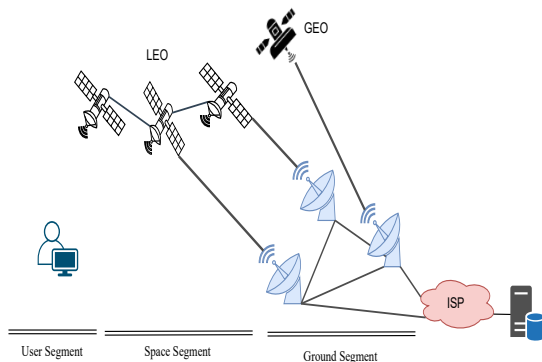


Figure 2. Ground Station as a Service Scenario.

Ground stations, however, are pretty expensive to build and operate. In such a scenario, the Ground Station as a Service (GSaaS) business model, introduced by Amazon Web Services (AWS) and Microsoft Azure Orbital, allows satellite operators to concentrate on building satellites and applications, knowing that ground station services will be available for use as needed [Velusamy and Lent, 2022].

The concept of GSaaS relies on a virtualized satellite ground station with a pay-per-usage model. Moreover, when such a virtualized approach is based on a microservice architecture, satellite operators may adjust their operations by scaling up and down the microservice configuration of ground stations according to different demands, such as changes in traffic or detection of security hazards.

AWS and Azure Orbital's ground station virtual architecture is very similar, leveraging Software Defined Radio (SDR) technologies for transporting Radio Frequency signals over IP. However, developing a virtualized architecture

based on IP extends the space domain of the mismatch between today's Internet architecture and its usage. Observing how the current Internet is, content-based networking is already the dominant paradigm (*e.g.*, YouTube, Netflix, Amazon, iTunes). The same conclusion is achieved considering other Internet services, such as the Internet of Things (IoT) and specific middleware used in industrial and mission-critical environments (*e.g.*, Data Distribution Services).

Hence, a better approach for developing a GSaaS model is to support a networking approach based on data and not host identifiers, following the ICN paradigm. ICN brings potential advantages to dynamic networks, such as satellite systems. ICN provides connectionless and topological independent communications by identifying data and not devices, allowing data to be retrieved on-demand, avoiding undesirable traffic, and potentially processing and storing in intermediary nodes, such as Ground Stations.

Having Ground Stations fetching, storing, and redistributing data to support all data-centric Internet services allows customers to focus on what is important to them: easy, non-interrupted, and reliable access to data to build their business cases. Hence, to ensure that this happens from the point in time when data arrives at the space segment until it leaves toward any consumer (*e.g.*, data center or content delivery network), via a set of ground stations. It is essential to develop ground stations based on a data-centric networking approach built upon a microservices architecture, allowing an easy adaptation of the networking services, such as the Micro-Chain ICN concept described in the next section.

### 4.2 Micro-Chain Architecture

The proposed Micro-Chain architecture to orchestrate and manage ICN microservice is depicted in Fig. 3 and includes four modules:

- **Microservices** module: it represents the microservices used in the architecture. In the Fig. 3, it is represented by rectangular boxes with the prefix "micro".
- **Monitoring** module: it acquires, stores, processes, and exposes metrics via the monitor API. As needed, another module can request the data acquired by the monitoring module to be used in some internal logic.
- **Orchestrator** module: it is responsible for deploying and ensuring the integrity of containers and/or VMs, even in the face of failures;
- **Manager** module: it is the central point of the architecture, responsible for implementing the business logic. In addition, this module also implements a web application that allows the operator to check information about the system and perform manual operations.

In addition, Micro-Chain also defines communication between modules via APIs. The architecture defines three APIs: monitoring API, micro API, and orchestrator API.

In Fig. 3, each dotted rectangle represents a ground station of the application scenario described in the previous section.

Micro-Chain extends the work of Marchal *et al.* [2018] to a local cluster, overcoming the restriction of a single local machine, which is closer to the proposed use of NDN.

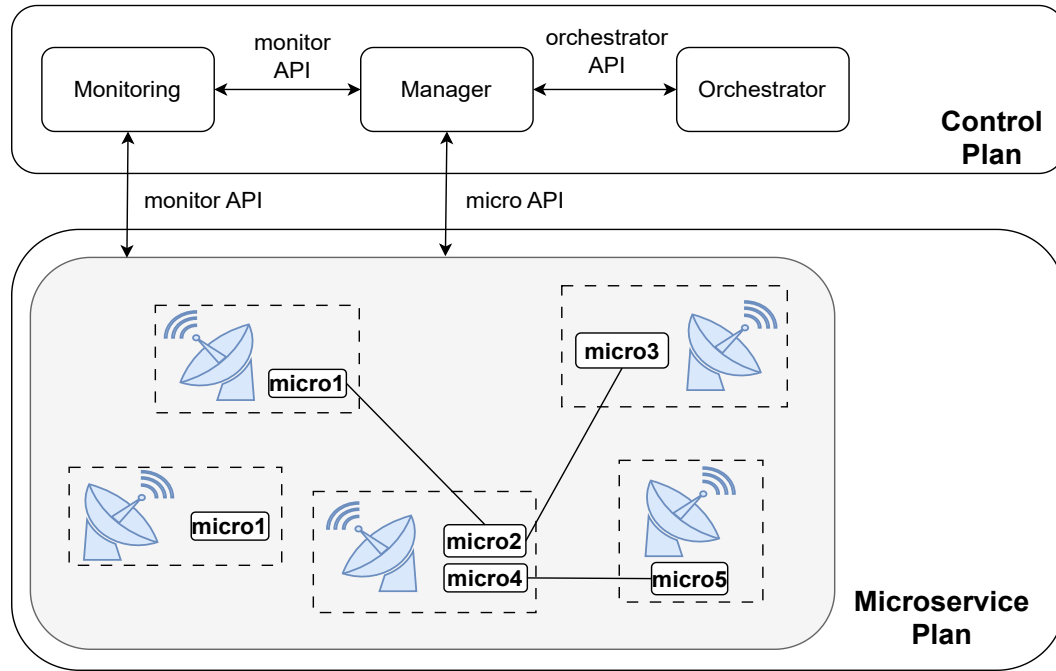


Figure 3. Micro-Chain, modules and their relationships.

In this way, several devices may use this protocol to enable communication between groups of clients and servers. Additionally, Micro-Chain introduces the Monitoring module to handle metrics, whereas Marchal *et al.* [2018] utilized the Manager for this purpose. In this context, the proposed architecture is designed to enhance the distribution of responsibilities between modules.

This work also extends what is presented in da Cruz *et al.* [2024b] by providing four key contributions: 1) details of how the architecture works; 2) outlines an application scenario where the solution can be used; 3) a placement method based on the number of hops between devices; and 4) a more comprehensive analysis of the limitations associated with the solution.

#### 4.2.1 Microservices Module

The implementation of Micro-Chain employs a modified version of the microservices developed and available online<sup>1</sup> by Marchal *et al.* [2018]. The main modifications are related to the suitability of microservices for operations with the Monitoring module. In this context, a server application was developed to respond to Monitoring requests.

To the best of the authors’ knowledge, there is no microservices-based NDN implementation other than Marchal *et al.* [2018].

For NDN communication, each microservice has egress and ingress interfaces. A microservice’s egress interfaces correspond to the interfaces where it is the source, and its ingress interfaces are the interfaces where it is the destination [Marchal *et al.*, 2018].

The microservices have two properties: cardinality and orientation. A non-oriented microservice can process both interest and data packets at its interfaces, whereas an orientated one can only process one. Conversely, cardinality refers to

the number of distinct sources or destinations a microservice can identify during forwarding. The microservice forwards packets to all its ingress or egress interfaces when its cardinality is "1" (multicast). A microservice with a cardinality of "N" can forward packets to a specific source or destination, implying that some route information is maintained [Marchal *et al.*, 2018].

The intricate relationships between cardinality and orientation present a significant challenge in deploying these microservices. To illustrate this, Fig. 4 presents an erroneous configuration to scale up BR1. In this configuration, an interest packet received by CS1 on an ingress interface would be sent to two BR instances since CS1 has an egress cardinality of N. This would lead to potential errors and the inefficient use of resources, as two identical packets of interest would pass through the network. To correct this configuration, appropriately, utilizing an SF microservice to divide the data flow is necessary. Consequently, establishing and manipulating an NDN network based on these microservices requires a comprehensive understanding of their operation. This involves identifying the configurations of these microservices to achieve specific desired characteristics.

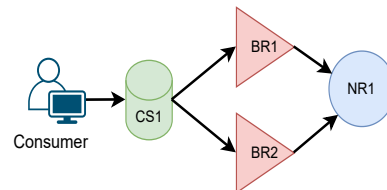


Figure 4. Microservices configuration for BR1 scaling with package duplication.

All microservices accept at least three operations: 1) the addition of a new interface, which is used to create connections between microservices; 2) the deletion of an interface; and 3) configuration changes, which depend on the type of microservice. Additionally, microservices can have

<sup>1</sup><https://github.com/Nayald/NDN-microservices>

other specific operations. The NR microservice, for example, accepts operations to configure routing to content servers, which the Manager uses to control the routes.

#### 4.2.2 Manager Module

The Manager and micro API were developed based on Marchal *et al.* [2018], with implementation modifications also being made to adjust Micro-Chain operations.

The Manager plays a pivotal role in maintaining and managing two distinct graphs: one representing the topology of microservices and other network nodes. These graphs store essential operational data, including CPU and memory usage, names, and routing information. Additionally, the Manager offers a web application interface for operators to visualize and interact with the data stored in the microservices graph, enabling manual operations as needed.

One of the Manager's primary responsibilities is the scaling process, which is subject to certain constraints. To scale, a microservice must meet specific requirements, including: 1) it cannot be located at the edge of the network to prevent broken connections outside the managed network; 2) it must be defined as scalable; and 3) it must have a CPU and memory consumption that exceeds a certain threshold, where each microservice can have customized values.

To exemplify the scaling process, consider the scaling up process of BR1 shown in Figure 5. This process involves seven operations, as follows: deployment of SF1 (1), the connection between SF1 and BR1 (2), the connection between CS1 and SF1 (3), deletion of the connection between CS1 and BR1 (4), deployment of BR2 (5), the connection between BR2 and NR1 (6), and finally, the connection between SF1 and BR2 (7). It is important to highlight that during the deployment of microservices, the Manager sets consumption limits for CPU and memory.

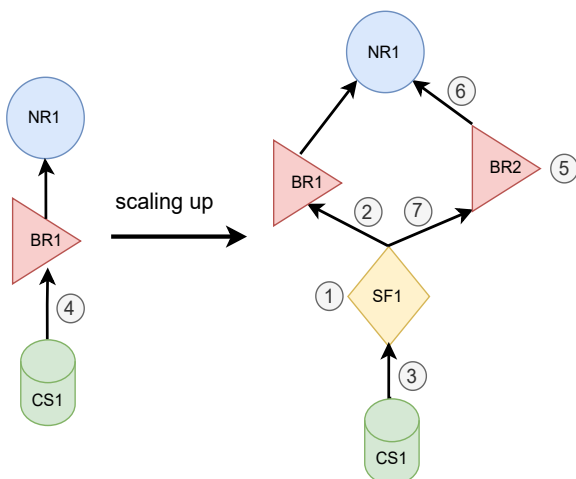


Figure 5. Steps to scale up BR1.

Microservices and the Manager communicate via User Datagram Protocol (UDP) (micro API) using the JavaScript Object Notation (JSON). Most messages transmitted between them contain two fields: "action" and "id." The "id" field is an integer the Manager generates to identify the operation. When a microservice sends a response, it includes the same "id" received from the Manager, enabling the Man-

ager to associate the response with the corresponding request. On the other hand, the "action" field is employed to map messages to a specific function, which then executes the pipeline for that action.

To exemplify the communication model between Manager and microservices, consider the message format presented in Listing 1 and Listing 2 to create a connection between CS1 and SF1 in step 3 of Fig. 5. Listing 1 shows the message transmitted by the Manager to CS1, requesting this microservice to create a new interface with SR1. This message includes the address and port of the target microservice. Conversely, Listing 2 shows the response transmitted by CS1 to the Manager, confirming the successful establishment of the connection in the status field.

Listing 1: Message sent by the Manager to request a new face.

```
{
  "action": "add_face",
  "id": 6,
  "layer": "tcp",
  "address": "172.18.0.5",
  "port": 6363
}
```

Listing 2: Message sent by the CS1 microservice in response to the Manager's request.

```
{
  "name": "CS1",
  "type": "reply",
  "action": "add_face",
  "id": 6,
  "status": "success",
  "face_id": 3
}
```

The Manager centrally manages the routes available in the network and transfers them directly to the relevant modules, such as an SDN controller. Route changes are triggered when the content provider sends a route request or leaves the network and when connections are created. This route control is performed for the NR microservice since its objective is to forward interest packets to the producer.

Implementing microservices faces the placement problem, where inappropriate placement can result in quality degradation of communication. To mitigate this problem, the Manager performs a method to select a node for deploying microservices. However, it is worth highlighting that the placement problem is not the focus of this work.

The Pseudocode 1 presents the steps to determine which node will deploy a microservice. The objective is to minimize the hops between nodes required to execute a chain of microservices. The method starts with acquiring the predecessor nodes with sufficient resources, line 1. Next, in line 2, the number of hops required for the acquired nodes is calculated, assuming the microservice deployment in each node. The nodes with the lowest number of hops are selected in line 3. Then, it is checked if more than one node was selected, lines 4 to 6. If yes, the node with more resources and the lowest number of hops is returned (line 5). If not, the node

with the fewest hops is returned (line 7). As the node with the lowest number of hops is always the predecessor node, the method tends to deploy the microservice in this node.

---

**Pseudocode 1:** Steps to select a node for deploying a microservice.

---

- 1 Gets the predecessors of the microservice with sufficient resources;
  - 2 Calculates the number of hops from the microservice to each node in the network;
  - 3 Selects node(s) with the least number of hops;
  - 4 **if** *Is there more than one node?* **then**
  - 5 | returns node with more resources and fewer hops;
  - 6 **end**
  - 7 returns the node with the fewest hops;
- 

### 4.2.3 Monitoring and Orchestrator Modules

The Micro-Chain implementation employs Prometheus<sup>2</sup> as a Monitoring module and an Hypertext Transfer Protocol (HTTP) client library<sup>3</sup> as an API monitor. In particular, Prometheus uses pull-type data acquisition methodology to actively obtain metrics from microservices.

Marchal *et al.* [2018] suggest metrics that can be monitored by the microservices they developed. These include route statistics for the NR, unsolicited data packets and interest packets retransmitted to the BR, and hit-and-miss counters for CS.

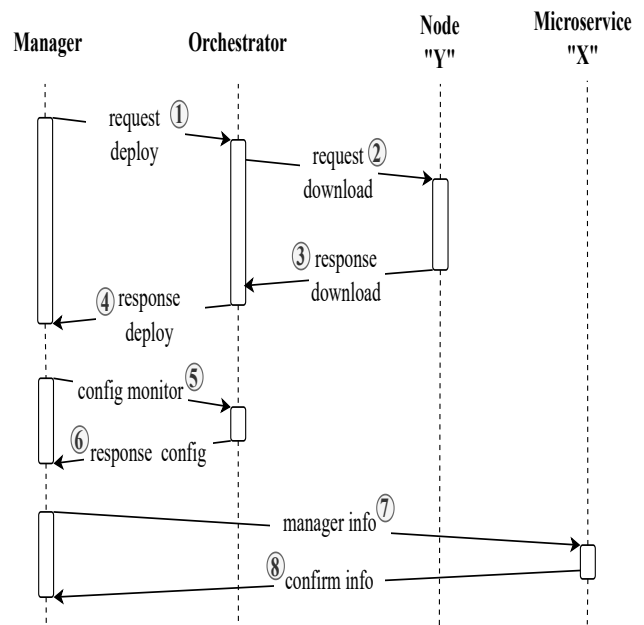
In Micro-Chain, the Monitoring module only captures, processes, and exposes the metrics, i.e., it does not perform any decision-making based on the captured metrics. The Manager performs this type of operation. Considering this context, in Micro-Chain, the metrics monitoring process can be acquired in three ways: directly through the Monitoring module, directly through the Manager, or using these two modules. The default process for capturing metrics is via Monitoring. In this case, the Manager needs to request the metrics captured by Monitoring to execute its operations, which intrinsically introduces latency. Therefore, another form of capture is directly by the Manager, which provides faster decision-making.

In addition, the Micro-Chain implementation employs K3s<sup>4</sup> as an Orchestrator module and an HTTP client library<sup>5</sup> as orchestrator API. In particular, K3s and Prometheus have an integration that allows the deployment of Prometheus in K3s, which allows their combined use. Furthermore, these tools have good tutorial support and periodic updates, accelerating development.

The Manager, Monitoring, and microservices are developed explicitly in containers and deployed within the K3s cluster. This will allow modules like Manager and Monitoring to scale on demand in the future. Furthermore, this approach leverages K3s' robustness to restart modules in case of failure automatically.

### 4.2.4 Core Operations

Fig. 6 shows the operation for deploying a microservice in the cluster. Initially, the Manager requests the Orchestrator via orchestrator API to create the microservice by providing details such as the image name, deployment node, and maximum resource usage (1). The Orchestrator then receives and communicates with the node to download and configure the container according to characteristics defined by the Manager (2). When this process is completed, the node responds to Orchestrator (3), informing if the request was successful. The Orchestrator then saves the status and communicates the operation result to the Manager (4). Additionally, the Orchestrator is responsible for keeping the container running correctly and restarting it in case of failure. If the deployment operation of a microservice is successful, the Manager saves the state locally in a graph. The process of destroying a microservice is similar to deploying it. Upon deploying the microservice (1-4), the Manager configures the Orchestrator with monitoring details for the deployed microservice (5 and 6), such as the capture port and interval. If all operations (1-6) are successful, the Manager transmits its IP address and port to the microservice via micro API (7) to enable direct communication. The microservice performs the configuration and responds, informing the operation result (8).



**Figure 6.** Sequence diagram for deploying a microservice.

When a microservice runs, it starts a server that exposes the acquired metrics to the Monitoring module. Fig. 7 shows the operations for discovering and capturing metrics using Monitoring. First, the Monitoring requests information from the Orchestrator regarding new endpoints (1 and 2), which enables the Monitoring to identify the deployment of new microservices. Upon discovery, Monitoring executes its metrics capture process via monitor API (3 and 4), as configured by the Manager.

Fig. 8 illustrates the Manager's steps for acquiring metrics from a microservice. As previously presented, this can be performed by three methods: through the Monitoring, directly by the Manager, or both. In the first method, the Man-

<sup>2</sup><https://prometheus.io/>

<sup>3</sup>[https://github.com/prometheus/client\\_python](https://github.com/prometheus/client_python)

<sup>4</sup><https://k3s.io/>

<sup>5</sup><https://github.com/kubernetes-client/python>



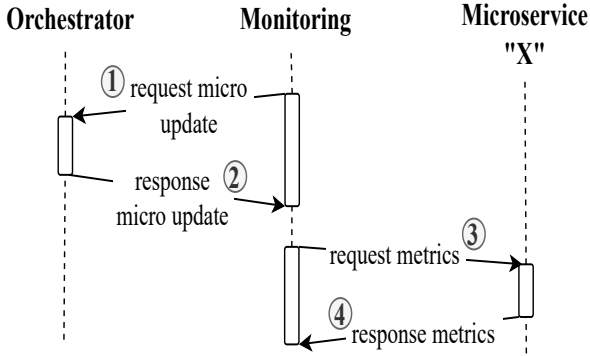


Figure 7. Sequence diagram for microservice discovery and metrics capture from the Monitoring.

ager can request the metrics stored by Monitoring via monitor API (1 and 2). Alternatively, the Manager can configure the microservice for direct metric transmission by sending a command via micro API containing essential details (3). Upon receiving this configuration request, the microservice processes it and responds with the result of the operation (4). So, the microservice sends metrics to the Manager according to the configuration (5 and 6). Direct capture by the Manager has less delay than the capture performed through Monitoring and requested by the Manager. Therefore, this strategy can be used in cases where there is a delay restriction.

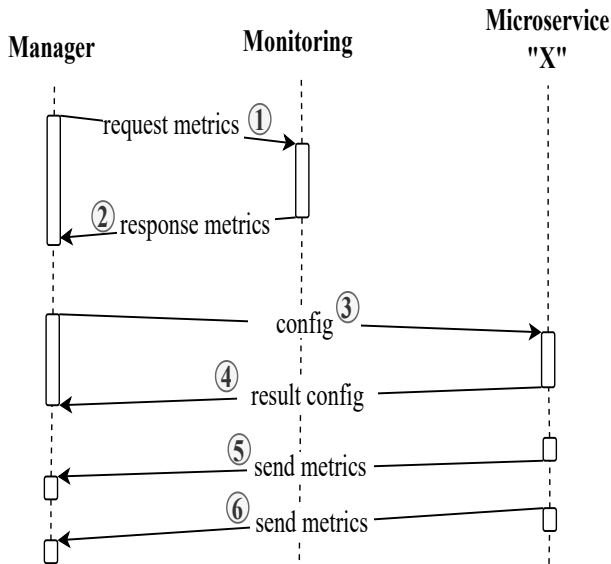


Figure 8. Sequence diagram for configuring and capturing metrics from the Manager.

Fig. 9 presents the operation to create a new link through the Manager. First, the Manager sends a message via micro API to the source microservice informing the IP addresses and port of the destination microservice (1). When the source microservice receives this message, the address is saved in a forwarding table. Like other operations, the Manager waits for confirmation of the operation (2) before storing the state in the microservice graph.

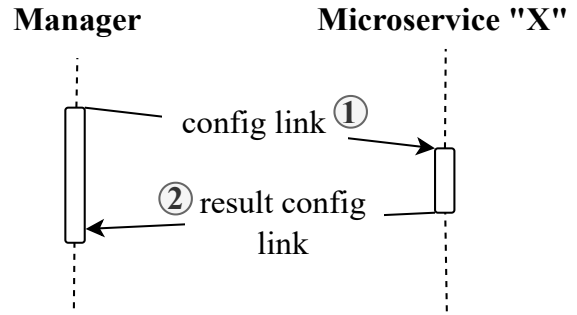


Figure 9. Sequence diagram for creating a link between microservices from the Manager.

## 5 Evaluation

This section commences with a description of the experimental setup used. Subsequently, the evaluation of an on-demand scaling scenario is presented. Finally, the experimental results and findings are provided.

### 5.1 Setup Overview

The experiments focus on operating a set of ground stations in the idealized scenario presented in Fig. 2. The experiments contain three ground stations, a client, and a server. In this way, the space segment and the Internet Service Provider (ISP) were omitted. The experiment includes three computers, each simulating a ground station.

Fig 10 shows an overview of the testbed, where all computers are on the same IP network, and the ellipses represent the software resources installed on each one. The *Computer 1* has a server K3s with Manager and the main Prometheus features installed; *Computer 2* has a K3s agent installed, and *Computer 3* has a K3s agent installed. Depending on the experiment, the NDN client and server are deployed on Computer 2 or 3.

Table 1 presents the technical specifications of each computer used in the experiments. The three computers have been configured to create a K3s cluster. First, the *Computer 1* installs an Ubuntu 22.04.3 LTS operating system and the K3s v1.26.5<sup>6</sup>, in this order. Then, the Manager Pod is deployed on the cluster. Finally, the Prometheus features are also deployed using a helm project<sup>7</sup>.

The *Computer 2* and *Computer 3* run Ubuntu 22.04.3 LTS with ndn-cxx v0.6.1. The implementation is an open-source project available in GitHub<sup>8</sup>.

Table 1. Technical specification of the nodes.

Devices	Description
Computer 1	Intel Core i3-6100 with 2 cores and 4 threads; 8 GB of memory
Computer 2	Intel Core i5-3210M with 4 cores and 2 threads; 6 GB of memory
Computer 3	AMD Ryzen 7 6800H with 8 cores and 8 threads; 16 GB of memory

<sup>6</sup><https://docs.k3s.io/quick-start>

<sup>7</sup><https://github.com/prometheus-community/helm-charts>

<sup>8</sup><https://gitfront.io/r/otavio/kGTNssbpC7wL/micro-chain/>

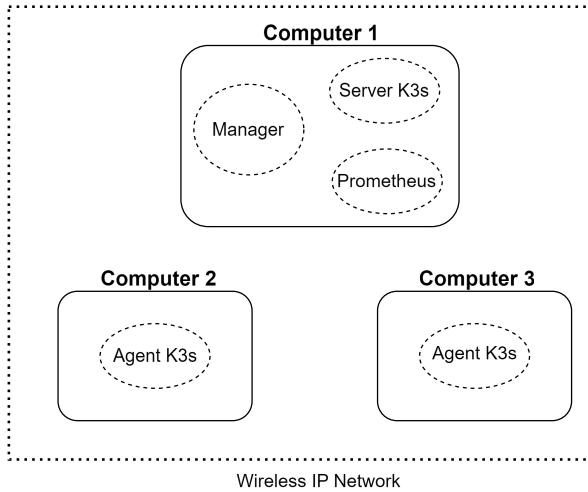


Figure 10. Experimental setup.

Regarding the placement of microservices, it was assumed that the first microservice in the chain has the client application (NDNperf client) as its predecessor. It is important to note that this assumption does not reflect reality, as the client application does not operate within the controlled network infrastructure. However, this consideration can be adapted to real-world scenarios by deploying the first microservice at one of the network’s entry nodes closest to the client. In addition, it was assumed that only agent nodes can deploy microservices (Computers 2 and 3), leaving the server node to focus on the control plane. Finally, it is considered that Computer 2 can deploy up to 1 microservices, and Computer 3 can deploy up to 4.

## 5.2 Evaluation Scenario

This use case addresses the following questions: **Q1**: Can the implemented solution automatically scale a microservice? **Q2**: What is the communication overhead introduced by the Micro-Chain control plane? **Q3**: Does the placement of microservices affect network performance?

To answer these questions, a microservices scaling scenario is addressed. All architecture modules are used to execute this process, which represents the expected operations well. In this scenario, a client varies the number of interest packets sent, which results in fluctuations in microservices resource consumption in terms of CPU and memory.

In the experiment, the microservices *CS*, *BR*, and *NR* are deployed and connected in this sequence. This choice is because these microservices correspond to the main NDN functionalities (local content storage, FIB, and PIT). Specifically, the experiment focuses on *BR* scaling, where thresholds are set for CPU and memory consumption, causing *BR* to scale up or down once these thresholds are reached.

To determine the threshold values, it was hypothesized that a value of 70% would reasonably necessitate the scale-up before reaching full capacity, allowing for some time to manage the increased demand. Conversely, a threshold of 35% was considered low enough for scale-down. Therefore, the thresholds are 70% and 70%, respectively, for CPU and memory consumption for scaling up, and 35% and 35% for CPU and memory consumption for scaling down, respectively.

## 5.3 Results

Fig. 11 presents the CPU usage during the *br1* scale experiment (in orange). Initially, all microservices have CPU consumption close to 0%. After an NDNperf client and server are started, CPU consumption of *br1* increases above the upper threshold (70%) at 90 s, which starts the scaling-up process. At 120 s, the consumption of new instance of *br1*, *br1.1* (in purple), and *br1.sr1* (in orange), an *SF* microservice, can be observed. In this configuration, *br1.sr1* divides traffic between *br1* and *br1.1*.

In Fig. 11, between 150 s and 180 s, the client stops requesting content, reducing CPU consumption in the following seconds. The scaling down process is triggered as the CPU consumption of *br1* drops below the 35%. At the end of this process, the microservices configuration is the same as at time 0 s.

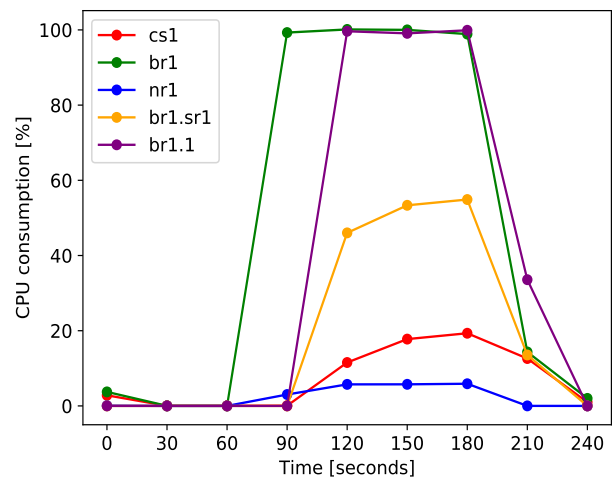


Figure 11. CPU usage at every 30 seconds.

Fig. 12 illustrates the amount of memory utilized during the same experiment, where the values for each microservice remained practically fixed. Due to low fluctuations, the system did not reach the threshold criteria required for scaling based on memory usage. In addition, since the scaling process occurred as expected, the answer to **Q1** is yes.

To demonstrate the overhead resulting from the control plane, Fig. 13 shows the communication between the Manager and the other modules: Monitoring, microservices, and Orchestrator modules. The amount of communication is calculated from the absolute number of HTTP and UDP messages the Manager sends and receives. It should be noted that the amount of communication does not consider the size of the message but rather the number of occurrences. The majority of the amount of communication is between 90 s and 100 s. At 95 s, the scale-up process generates around 44 communications, of which 22 is with K3s. During this process, the communication between the Manager and other modules is used to create two microservices and four links while also deleting one link.

Moreover, Fig. 13 shows another peak at 215 s resulting from the scale-down process, with the amount of communication equal to 30. The reduced number of messages exchanged during the scale-down process is due to the smaller

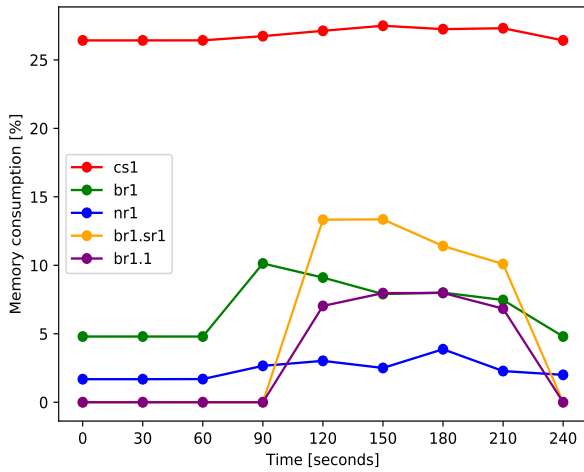


Figure 12. Memory usage at every 30 seconds.

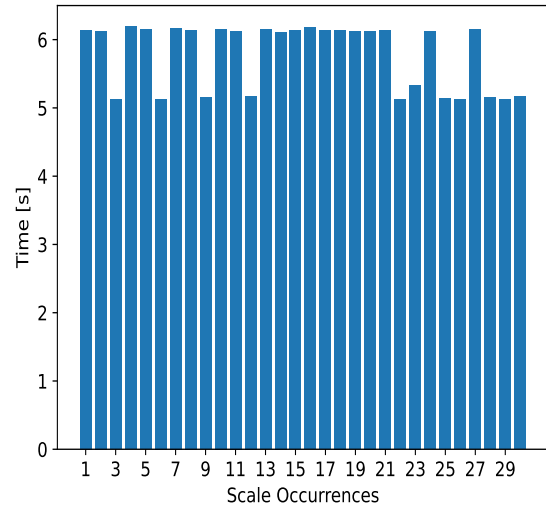


Figure 14. Time required to execute scale-up process 30 times.

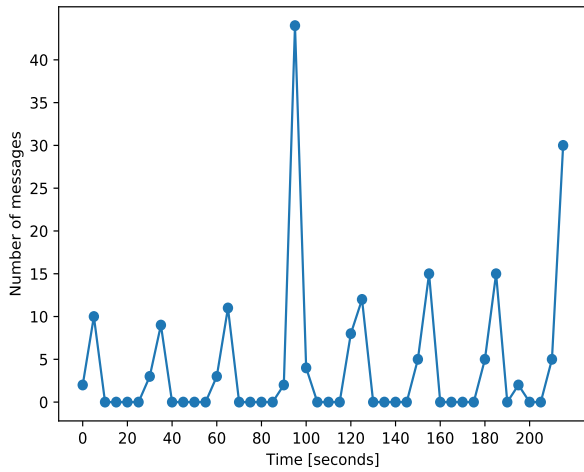


Figure 13. Number of messages at each 5 seconds.

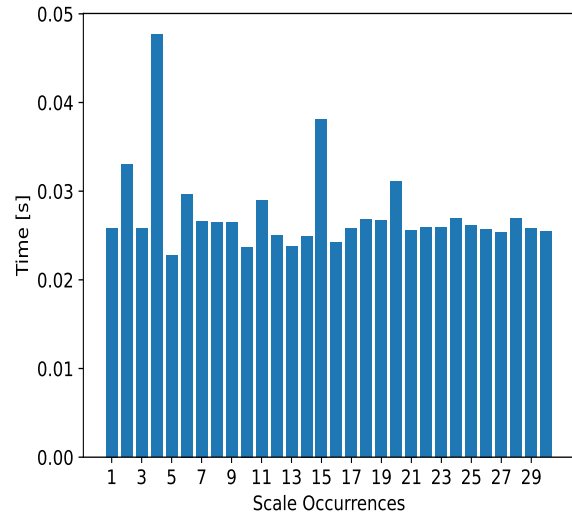


Figure 15. Time required to execute scale-down process 30 times.

number of operations. Specifically, excluding two microservices and one link creates one new link. The number of communications that occurred provided initial information about the consumption of network resources to perform the desired operations.

Answering the question Q2: a total of 29273 data packets were retrieved by the client during the experiment, while the total amount of control communications that occurred is equal to 185, representing 0.632% of the amount of data packets recovered.

Fig. 14 and Fig. 15 show the time required to perform scale-up and scale-down processes 30 times, with the image download time excluded, since each computer has the microservices image downloaded locally in advance. The average time for scale-up is 5.7815 s, while the average time for scale-down is 0.0274 s. The longer time required for scaling up is attributable to the greater number of steps performed, which encompasses all the steps presented in Fig. 5.

### 5.3.1 Placement problem

Communication performance is evaluated based on the performance experienced by the client during a scaling-up pro-

cess in two contexts, the first placing microservices manually and the second using the Pseudocode 1. Fig. 16 shows the result for the first context. Before scaling, the average throughput is 21.9436 Mbps. Then, the scaling process is completed (38 s) when the average throughput unexpectedly drops to 13.2898 Mbps.

As verified by Marchal et al. [2018], an increase in throughput was expected with the increase in BR instances. Fig. 17 shows the location and connection between microservices, in which packets require two hops between computers to go from the client to the server. This communication overhead results in lower throughput and was not identified by Marchal et al. [2018] because the experiments were performed on a single machine.

Fig. 18 shows the placement of microservices for the second context, and Fig. 19 presents the client throughput for this configuration. Before scaling, the average throughput is 23.36 Mbps. After scaling, in 34 s, the throughput increased to 35.1802 Mbps, corresponding to an increase of approximately 50%.

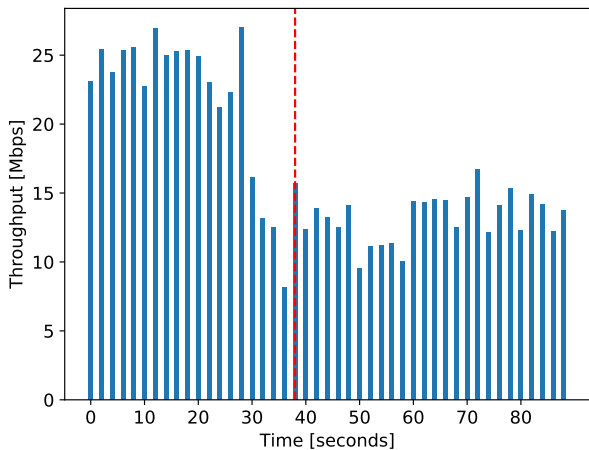


Figure 16. Client throughput at every 2 seconds for the first context.

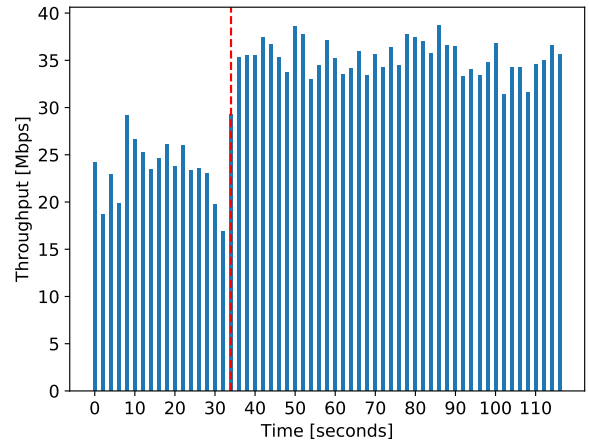


Figure 19. Client throughput at each 2 seconds for the second context.

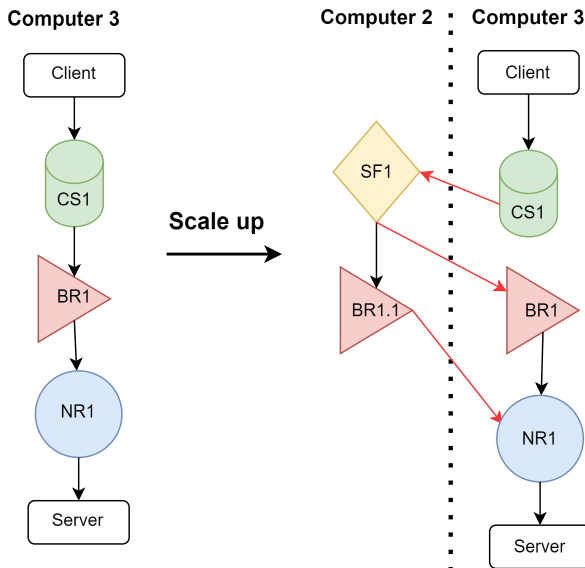


Figure 17. Positioning of microservices and their connections. The red lines represent the hops between computers.

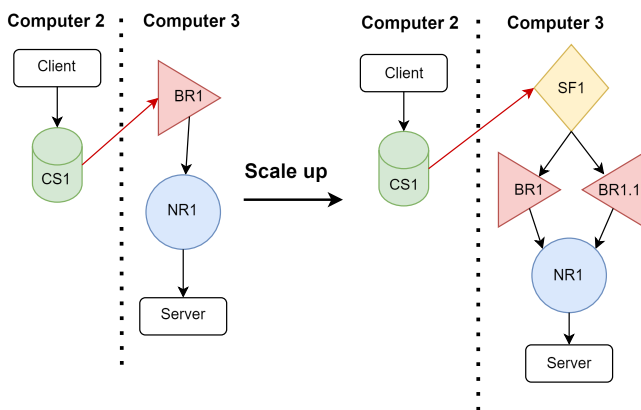


Figure 18. Positioning of microservices and their connections. It only needs a single hop.

As throughput varies depending on the placement of microservices, the answer to question Q3 is: yes, the placement of microservices affects performance.

The lower throughput in the first context can be attributed to an increase in the delay between the client and the server, where three factors stand out: (1) an increase in the number

of hops, which is supported by Pseudocode 1; (2) the insertion of a new microservice (SR) in the chain; and (3) processing speed since the computers used have heterogeneous resources.

In this work, only the hop between nodes is considered for the placement method. It should be noted that in other experiments not presented here, it was verified that the proposed positioning method (Pseudocode 1) did not improve throughput. A possible justification for this is factor (3), which was not considered. Therefore, a fair way to assess the impact of the hops on performance is to use computers with identical technical specifications or virtual machines on the same computer, where factor (3) would have no impact.

## 6 Discussions

The experiments demonstrated the flexibility of the microservices architecture, where specific parts of the application could be scaled separately without the need to scale the entire application as in the monolithic architecture. Additionally, the system can choose which NDN feature sets to implement as needed. In future experiments, it is necessary to compare monolithic and microservices implementation performance in a cluster scenario. However, to ensure a fair comparison between these architectures, it is essential to establish a method for scaling NFD.

Due to its modular characteristics, the microservices architecture can facilitate the implementation of improvements in NDN microservices. For example, new methods for CS storage decisions can be implemented by changing the CS microservice without needing to deal directly with other microservices. Another example is the development of new approaches for name lookup, which can be implemented and tested via the NR microservice.

The architecture was initially designed for an NDN microservices approach. However, the structure can be extended to be used in more generic scenarios, like NFVs or NFVs microservices. In addition, the implementation uses Kubernetes and Prometheus as Orchestrator and Monitoring modules, but the Micro-Chain architecture is not limited to them. For example, with adjustments, it should be possible to use other orchestrators such as Docker Swarm, Apache

Mesos, and Nomad.

Compared to the ETSI NFV MANO architecture, Micro-Chain includes a dedicated module for monitoring. This difference is also valid for the Marchal *et al.* [2018] solution, where the monitoring process is the exclusive responsibility of the Manager. This segregation aims to offer greater modularity, allowing for the independent development and maintenance of this Monitoring. This approach is effective in the cloud, where monitoring software has been developed and integrated into solutions.

Experiments conducted on a local machine by Marchal *et al.* [2018] indicate that throughput increases with scaling. Nevertheless, during these experiments, it was impossible to identify the microservices placement problem in scenarios with multiple nodes. To address this problem, the present work defines a method to minimize the number of hops. However, it is essential to define a placement method that considers other network parameters, such as bandwidth and delay [Ding *et al.*, 2023]. Additionally, an artificial intelligence-based solution can be designed to predict demands and prepare the network proactively [Manias and Shami, 2021].

One challenge of implementing a microservices architecture is determining the appropriate level of granularity, *i.e.*, the scope of functionality for each microservice in the solution. Evaluating and adjusting granularity can improve performance by reducing latency in the context of NDN microservices. In principle, verifying the combination of NR and BR functionalities in a single microservice is feasible, as suggested by Marchal *et al.* [2018].

For large-scale geographic scenarios with multiple devices and microservices, it would be interesting to verify the division of the network into regions based on specific parameters of interest. This strategy is expected to facilitate the positioning and management of microservices.

Another extension consists of modifying the proposed architecture to support multi-cluster context. A possible approach in this context would be to use a manager to control each cluster and implement a way to communicate between the managers. Therefore, each Manager would need to consider the local variables of their cluster and the general variables of the set of clusters when making decisions. Or, there could be a global Manager who controls the local Managers.

The proposed division for Micro-Chain can facilitate the development of improvement and maintenance, as it isolates responsibilities. For example, changing the Orchestrator module without requiring many changes to the other modules is possible. In addition, the responsibilities of the modules can be executed by existing solutions, which improves reusability.

Micro-Chain has a Manager responsible for dealing with NDN forwarding policies by sending interfaces to the NR microservices and creating connections between microservices. In the future, this function can be separated from the Manager by developing an SDN controller module to handle forwarding and routing strategies within the NDN network. This controller can be connected to Monitoring and the Manager for decision-making.

## 7 Conclusions

This work proposes Micro-Chain, an architecture to manage NDN microservices on demand. The architecture defines four modules: Manager, Monitoring, Microservices, and Orchestrator. It defines the interrelationships between these modules and the operations that enable manipulating a set of microservices according to specific interests.

Among the implemented Micro-Chain functionality, the scaling and placement of microservices stand out. The scaling decision is based on the upper and lower CPU and memory usage thresholds. A scaling up or down process is triggered when a given microservice reaches one of these values. Additionally, the Manager module employs a placement method to minimize the number of hops.

To evaluate the architecture, the experiments focused on a scale-on-demand scenario in a cluster of three nodes with three distinct microservices, where all architecture modules are used. This scenario addresses three questions: **Q1:** Can the implemented solution automatically scale a microservice? **Q2:** What is the communication overhead introduced by the Micro-Chain control plane? **Q3:** Does the placement of microservices affect network performance?

The results demonstrate that the microservices can be scaled based on CPU usage, as memory did not reach the established thresholds (Q1). Additionally, the communication overhead was evaluated based on the number of messages transmitted and received by the Manager, which represented 0.632% of the data packets recovered during the experiment (Q2).

Furthermore, the throughput is evaluated in two contexts. In the first context, microservices are placed manually, whereas in the second context, the placement method implemented by the Micro-Chain Manager is utilized. The results demonstrate that inadequate microservice placement negatively impacts performance (Q3).

In the future, it would be beneficial to investigate a placement method that considers a broader range of network parameters and can predict demand and implement preventative measures. Additionally, working in a multi-cluster context is a promising avenue for further research.

## Declarations

### Funding

This work was partly funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES) - Finance Code 001 and in part by Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brazil (CNPq) Projects 309505/2020-8 and 311773/2023-0. We also thank the Brazilian Army for the support provided via the research project S2C2, ref. 2904/20 and CEREIA Project (# 2020/09706-7) São Paulo Research Foundation (FAPESP), FAPESP-MCTIC-CGI.BR in partnership with Hapvida NotreDame Intermedica group.

### Authors' Contributions

All authors contributed to the writing of this article and read and approved the final manuscript.

## Competing interests

The authors declare that they have no competing interests.

## Availability of data and materials

Data can be made available upon request.

## References

- Abdollahi Vayghan, L., Saied, M. A., Toeroe, M., and Khendek, F. (2018). Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 970–973. DOI: 10.1109/CLOUD.2018.00148.
- Aldaoud, M., Al-Abri, D., Awadalla, M., and Kausar, F. (2023). Leveraging icn and sdn for future internet architecture: A survey. *Electronics*, 12(7):1723. DOI: 10.3390/electronics12071723.
- Alencar, D., Both, C., Antunes, R., Oliveira, H., Cerqueira, E., and Rosário, D. (2022). Dynamic microservice allocation for virtual reality distribution with qoe support. *IEEE Transactions on Network and Service Management*, 19(1):729–740. DOI: 10.1109/TNSM.2021.3076922.
- Cerny, T., Abdelfattah, A. S., Bushong, V., Al Maruf, A., and Taibi, D. (2022). Microservice architecture reconstruction and visualization techniques: A review. In *Proceedings of the IEEE International Conference on Service-Oriented System Engineering (SOSE 22)*, pages 39–48. DOI: 10.1109/SOSE5356.2022.00011.
- Chowdhury, S. R., Salahuddin, M. A., Limam, N., and Boutaba, R. (2019). Re-architecting nfv ecosystem with microservices: State of the art and research challenges. *IEEE Network*, 33(3):168–176. DOI: 10.1109/MNET.2019.1800082.
- Cziva, R. and Pezaros, D. P. (2017). Container network functions: Bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31. DOI: 10.1109/MCOM.2017.1601039.
- da Cruz, O. A. R., Pereira, C. E., da Silva, A. S., da Costa, J. P. J., Mendes, P., and de Freitas, E. P. (2024a). Dynamic deployment and control of an ndn network for military multi-uavs based surveillance applications. In *2024 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1018–1025. DOI: 10.1109/ICUAS60882.2024.10556830.
- da Cruz, O. A. R., Pereira, C. E., De Freitas, E. P., Mendes, P., do Rosário, D. L., Cerqueira, E. C., da Silva, A. A. S., and dos Anjos, J. C. S. (2024b). Micro-chain: Towards the use of ndn microservices. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 1099–1103. DOI: 10.1145/3605098.3636172.
- Ding, Z., Wang, S., and Jiang, C. (2023). Kubernetes-oriented microservice placement with dynamic resource allocation. *IEEE Transactions on Cloud Computing*, 11(2):1777–1793. DOI: 10.1109/TCC.2022.3161900.
- Dulal, S., Ali, N., Thieme, A. R., Yu, T., Liu, S., Regmi, S., Zhang, L., and Wang, L. (2022). Building a secure mhealth data sharing infrastructure over ndn. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pages 114–124. DOI: 10.1145/3517212.3558091.
- Dynerowicz, S. and Mendes, P. (2017). Named-data networking in opportunistic networks. In *Proceedings of the ACM Information Centric Networking Conference*. DOI: 10.1145/3125719.3132107.
- ETSI (2014). *Network Functions Virtualization (NFV); Architectural Framework*. Available at: [https://www.etsi.org/deliver/etsi\\_gs/nfv/001\\_099/002/01.02.01\\_60/gs\\_nfv002v010201p.pdf](https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf).
- Fang, P. and Wolf, T. (2023). Implementing virtual network functions in named data networking and web 3.0. In *Proceedings of the International Conference on Computing, Networking and Communications (ICNC 23)*, pages 117–123. DOI: 10.1109/ICNC57223.2023.10074018.
- Ghorab, A. and St-Hilaire, M. (2022). Sdn-based service function chaining framework for kubernetes cluster using ovs. In *Proceedings of the 32nd International Telecommunication Networks and Applications Conference (ITNAC 22)*, pages 347–352. DOI: 10.1109/ITNAC55475.2022.9998380.
- Kalafatidis, S., Demiroglou, V., Mamatas, L., and Tsaousidis, V. (2022). Experimenting with an sdn-based ndn deployment over wireless mesh networks. In *Proceedings of the IEEE International Conference on Computer Communications Workshops (INFOCOM WKSHPS 22)*, pages 1–6. DOI: 10.1109/INFOCOMWKSHPS54753.2022.9798224.
- Kaur, K., Mangat, V., and Kumar, K. (2022). A review on virtualized infrastructure managers with management and orchestration features in nfv architecture. *Computer Networks*, 217:109281. DOI: 10.1016/j.comnet.2022.109281.
- Khalid, A., Rehman, R. A., and Burhan, M. (2023). Cbilem: A novel energy aware mobility handling protocol for sdn based ndn-manets. *Ad Hoc Networks*, 140:103049. DOI: 10.1016/j.adhoc.2022.103049.
- Lema, J. C., Neto, A., Silva, F., and Kofuji, S. (2019). Network function virtualization in content-centric networks. In *Anais do X Workshop de Pesquisa Experimental da Internet do Futuro*, pages 31–37, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/wpeif.2019.7696.
- Mai, H. L., Aouadj, M., Doyen, G., Mallouli, W., de Oca, E. M., and Festor, O. (2019). Toward content-oriented orchestration: Sdn and nfv as enabling technologies for ndn. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 594–598. Available at: <https://ieeexplore.ieee.org/document/8717804/authors#authors>.
- Manias, D. M. and Shami, A. (2021). The need for advanced intelligence in nfv management and orchestration. *IEEE Network*, 35(1):365–371. DOI: 10.1109/MNET.011.2000373.
- Marchal, X., Cholez, T., and Festor, O. (2018).  $\mu$  ndn: an orchestrated microservice architecture for named data networking. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*, pages 12–23. DOI: 10.1145/3267955.3267961.

- Nekovee, M., Sharma, S., Uniyal, N., Nag, A., Nejabati, R., and Simeonidou, D. (2020). Towards ai-enabled microservice architecture for network function virtualization. In *2020 IEEE Eighth International Conference on Communications and Networking (ComNet)*, pages 1–8. DOI: 10.1109/ComNet47917.2020.9306098.
- Pallewatta, S., Kostakos, V., and Buyya, R. (2022). Qos-aware placement of microservices-based iot applications in fog computing environments. *Future Generation Computer Systems*, 131:121–136. DOI: 10.1016/j.future.2022.01.012.
- Qi, J. and Wang, R. (2023). R2: A distributed remote function execution mechanism with built-in metadata. *IEEE/ACM Transactions on Networking*, 31(2):710–723. DOI: 10.1109/TNET.2022.3198467.
- Rossi, F., Cardellini, V., and Presti, F. L. (2020). Hierarchical scaling of microservices in kubernetes. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 28–37. DOI: 10.1109/ACSOS49614.2020.00023.
- Singh, V. P. and Ujjwal, R. (2020). A walkthrough of name data networking: Architecture, functionalities, operations and open issues. *Sustainable Computing: Informatics and Systems*, 28:100419. DOI: 10.1016/j.suscom.2020.100419.
- Velusamy, G. and Lent, R. (2022). Ai-based ground station-as-a-service for optimal cost-latency satellite data downloading. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 2363–2368. DOI: 10.1109/GLOBECOM48099.2022.10001260.
- Wang, X., Wang, X., and Li, Y. (2021). Ndn-based iot with edge computing. *Future Generation Computer Systems*, 115:397–405. DOI: 10.1016/j.future.2020.09.018.
- Yilma, G. M., Yousaf, Z. F., Sciancalepore, V., and Costa-Perez, X. (2020). Benchmarking open source nfv mano systems: Osm and onap. *Computer communications*, 161:86–98. DOI: 10.1016/j.comcom.2020.07.013.
- Zhang, L., Afanasyev, A., Burke, J., Jacobson, V., claffy, k., Crowley, P., Papadopoulos, C., Wang, L., and Zhang, B. (2014). Named data networking. *SIGCOMM Comput. Commun. Rev.*, 44(3). DOI: 10.1145/2656877.2656887.