



MSN: A Playground Framework for Design and Evaluation of MicroServices-Based sdN Controller

Sisay Tadesse Arzo, et al. [full author details at the end of the article]

Received: 2 April 2021 / Revised: 16 September 2021 / Accepted: 17 September 2021 /
Published online: 21 October 2021
© The Author(s) 2021

Abstract

Software-defined networking decouples control and data plane in *softwarized* networks. This allows for centralized management of the network, but complete centralization of the controller functions raises potential issues related to failure, latency, and scalability. Distributed controller deployment is adopted to optimize scalability and latency problems. However, existing controllers are monolithic, resulting in code inefficiency for distributed deployment. Some seminal ongoing efforts have been proposed with the idea of disaggregating the SDN controller architecture into an assembly of various subsystems, each of which can be responsible for a certain controller task. These subsystems are typically implemented as microservices and deployed as virtual network functions, in particular as Docker Containers. This enables flexible deployment of controller functions. However, these proposals (e.g., μ ONOS) are still in their early stage of design and development, so that a full decomposition of the SDN controller is not been available yet. To fill that gap, this article derives some important design guidelines to decompose an SDN controller into a set of microservices. Next, it also proposes a microservices-based decomposed controller architecture, foreseeing communications issues between the controller sub-functions. These design and performance considerations are also proven via the implementation of the proposed architecture as a solution, called Micro-Services based SDN controller (MSN), based on the Ryu SDN controller. Moreover, MSN includes different network communication protocols, such as gRPC, WebSocket, and REST-API. Finally, we show experimental results that highlight the robustness and latency of the system on a networking testbed. Collected results prove the main pros and cons of each network communication protocol and an evaluation of our proposal in terms of system resilience, scalability and latency.

Keywords Software defined networking · Network function virtualization · Microservice-based decomposition architecture · ETSI management and orchestration · Docker container · 5G

1 Introduction

Software-defined networking (SDN) is a networking paradigm that aims to give a definitive solution to break the limitations of traditional network infrastructure [1]. It breaks the vertical integration by separating the network control logic (i.e., the control plane) from the underlying routers and switches that forward the traffic (i.e., the data plane). With the separation of control and data planes, network switches have become simple forwarding devices, while the control logic is logically centralized in a controller entity, thus simplifying policy definition and network (re)configuration and evolution [2].

In particular, the SDN architecture consists of three layers [3]: data plane, control plane, and the application plane as an additional layer sitting atop them. Moving from the uppermost to the lowest layer, the *application plane* contains software applications to provide network services and performs ranges of functionalities such as Quality of Service (QoS), advanced security, and advanced routing. The *control plane* is the central agent which interfaces the application and data plane to implement applications network requirements: it communicates through the northbound interface to the applications and via the southbound interface to the forwarding devices. The *data plane*, is responsible for handling and forwarding packets and contains a group of data plane resources that can forward and manipulate packets. These resources include forwarding devices that have physical/logical interfaces to receive the incoming packets and forward them to an outgoing interface(s). The controller communicates with forwarding devices using several network communication protocols, in most of the cases the OpenFlow protocol [4].

The main issues of the centralized control plane range from latency constraints to fault tolerance and load balancing, to tackle those challenges, the distribution of the SDN controllers has been proposed to reduce typical issues of centralized controllers [5]. However, existing controllers are implemented as monolithic entities, even in the case of distributed deployments. In particular, in the case of distributed SDN controllers, there are replicas of the SDN controller, which means all SDN sub functionalities are replicated even if not all are necessary. For instance, Ryu SDN Controller [6], an open-source SDN controller implementation, provides a single piece of code installable on heterogeneous operating systems that enables the machine (or virtual machine) to act as an SDN controller. At the current time, all opensource and proprietary releases of SDN implementations adopt a monolithic software approach, which include ONOS [7], OpenDayLight [8], and Floodlight [9]. The main issue of monolithic implementations is that it does not allow network administrators and developers to choose SDN components and/or functionalities to (de-)activate for having the SDN controller functionalities according to SDN deployment and application needs in different scenarios. This results in limited flexibility in the network and creates multiple problems in terms of scalability, fault isolation, and latency. In particular, future 5G network infrastructures will leverage the network softwarization and network slicing concepts using SDN and Network Function Virtualization (NFV) in 5G

[10]. However, in some scenarios system constraints can be very strict, such as in Industrial 4.0 and 5G Tactile Internet, which require a high rate of reliability and low latency communications [11, 12], and therefore, the adoption of a monolithic SDN deployment may result not suitable.

At the same time, the legacy definition of the SDN reference architecture does not mandate the internal composition, implementation, and design of an SDN controller [3]. Thus, the SDN controller can be decomposed and implemented as a set of software components, running in a distributed manner. Specifically, it is possible to design the SDN controller as a composition of logical sub-functions, sharing the network service load and creating a robust system against failures. These sub-functions are loosely coupled units that can be executed in different and distributed computing platforms [13]. The possibility of decomposing the monolithic SDN controller and designing the controller as loosely coupled provides a possibility of flexible controller deployment.

Accordingly, some research efforts have been started to decompose an SDN controller into microservices. For instance, the ONOS project proposes μ ONOS, which is the next-generation architecture for the Open Network Operating System controller [14]. μ ONOS adopts a microservices-based architecture disaggregating the controller and the core itself as an assembly of various subsystems. However, μ ONOS has been specialized mainly for cloud datacenter scenarios by employing a service orchestrator, Kubernetes, to manage microservices that are realized as Docker containers.

Even if μ ONOS, faced the issues discussed above, the implementation is in an early stage that needs more works to provide a playground framework. In addition, their approach has some limitations: first, is limited to certain technologies, not all 5G compliant, for instance, Kubernetes instead of ETSI MANO or containers instead of VNFs. Second, inter-functionalities communication is limited to Google Remote Procedure Call (RPC), which does not give a fair degree of flexibility in certain scenarios. Finally, the implementation is not completed yet and that hinders the possibility to thoroughly test it.

To overcome all those limitations, we propose a novel microservices-based SDN controller decomposed architecture based on Ryu SDN Controller called MSN that has been specifically designed for next-generation 5G RAN Edge deployments and shows several original elements. First, it shows original design guidelines for implementing a microservices-based SDN controller; second, presents a novel decomposition architecture for SDN controller by showing the use of REST-API or gRPC or WebSocket as different possible interfaces between the decomposed and virtualized/containerized functions of the controller; third, it presents an implementation proposal using Ryu SDN Framework that is completely agnostic to particular technologies and is 5G compliant (e.g., ETSI MANO and Virtual Machines or Docker Containers); fourth, it presents an evaluation of the proposed implementation, which indicates the robustness of the system and the low latency achieved by showing a comparison of communication interfaces such as REST-API, gRPC, and WebSocket; finally, an open-source version of our proposed framework is available for the community at the link: https://gitlab.com/dscotece/ryu_sdn_decomposition/. In particular, the contribution of this article is summarized as follows:

- to analyze and discuss the possibility of decomposing an SDN controller;
- to propose a microservice-based SDN controller decomposing architecture;
- to decompose the Ryu controller and to deploy it in a docker container as a proof of concept;
- to perform evaluation for the decomposed controller from functionality, resiliency, scalability, and latency perspectives.

The rest of the paper is organized as follows. Section 2 discusses the motivation and background of decomposing an SDN controller. Section 3 presents the design guidelines for decomposing an SDN controller into microservices. Section 4 illustrates the proposed architecture and the performance evaluation for functionality, reliability, scalability and latency. Section 5 presents the literature review. Finally, we conclude by presenting our conclusion, while discussing future research direction in Sect. 6.

2 Background and Motivation

In this section, we briefly review the existing standard architectures for softwarized networks such as SDN and NFV by analyzing the large numbers of synergies between them. Once clarified the standard monolithic SDN controller, we present the motivational factors for decomposing the SDN controller. Finally, we provide a list of SDN functionalities and we motivate the benefits for distributing these functionalities as microservices.

2.1 Background

SDN and NFV are network softwarization paradigms that are transforming the network management and design approaches. Network softwarization is the mapping of hardware-based network functions into software. Network softwarization enhances the possibilities of innovation due to flexibility, programmability, virtualization, and slicing. SDN and NFV enable the traditional static network to be flexible paving the way for network innovation.

What we have discussed above implies the physical separation of the network control plane from the forwarding data plane. NFV is an architecture proposed by ETSI for network function softwarization. In other words, it is a softwarized implementation of network functions. The functions are traditionally implemented in preparatory hardware [15] such as firewall, load-balanced, deep packet inspection (DPI), and network address translator (NAT). The architecture of NFV contains three main components.

- Network function virtualization infrastructure (NFVI) consists of the hardware and software that host different virtual network functions (VNFs).

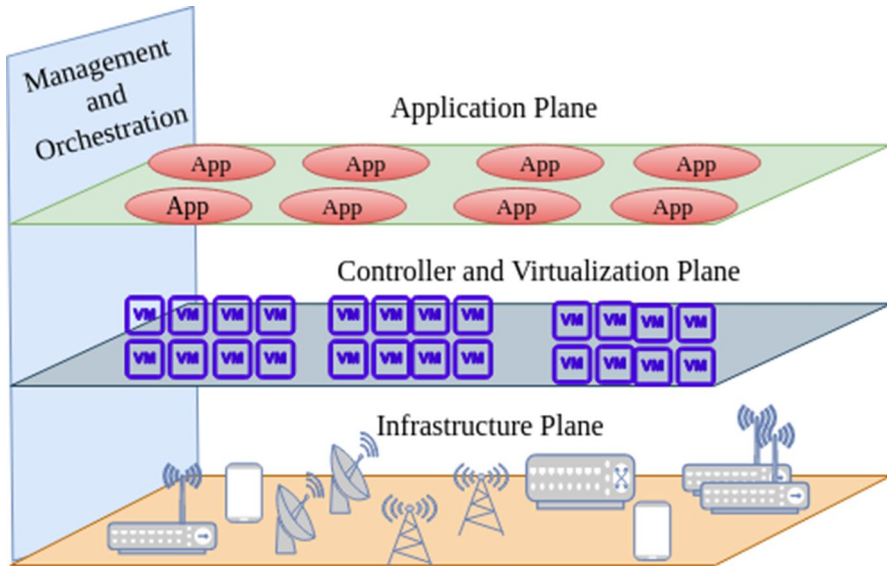


Fig. 1 A unified architecture for SDN and NFV

- VNFs are softwarized network functions such as firewall, network address translation (NAT), packet/serving-gateway (P/S-G), and baseband unit (BBU). These functions could be deployed in a containerized environment such as Docker.
- NFV management and network orchestration (MANO) is the place where management and orchestration of VNFs are implemented.

SDN and NFV are complementary technologies that can support each other for better and computer network softwarization and management. An attempt to unify and find a single architecture considering the two paradigms is done in [16]. The SDN controller provides the possibility of programming the network to have a virtualized network that NFV could use to orchestrate virtual functions that are deployed in a data center or distributed environment. Whereas, NFV could provide a virtualized SDN controller that can be deployed in a cloud. Such possibility provides flexibility and full network function softwarization. Figure 1 shows the unified SDN and NFV architecture.

2.2 Overview of SDN Controller Components

Existing legacy SDN controllers are typically implemented as the composition of various function modules and libraries in a single monolithic system [6–9]. For instance, Ryu controllers have internal components such as event distributor, topology discovery, and firewall and libraries such as Netconf, NetFlow, and sFlow. The Open Network Foundation (ONF) defined the basic elements and conceptual framework for an SDN controller design [3]. As defined by ONF, the internal components

of the SDN controller typically contain the following basic network elements and basic functions. The very basic network elements that an SDN controller has to manage are:

- Devices are units such as switches, routers, ports, and other physical networking units.
- Links are physical or wireless interconnections that connects one physical/logical port to another physical/logical port(s).
- Hosts represent the end devices such as computers.
- Packets/flows are the fundamental units of information in user and management services, at the network layer.

An inventory of these elements is registered and their state is updated in the controller's database. The common functions are:

- Topology management is managing a topology and determining which nodes and edges are present in the topology.
- Device and link discovery and management is a mechanism to configure and incorporate new devices into the network system.
- Route management is determining the path for a packet/flow to route through the network from the source to the destination. It computes the path for a given packet of flow based on the packet information.
- Routing/forwarding rule-setting enables the packet to route from the source to the destination based on the computed path.
- Performance monitoring is the mechanism of ensuring the performance of a network such as QoS for a given service.
- Network-state management is the management of the network information such as links status, available path, the available device along with their status, etc.

Depending on SDN controller types, the internal components of a controller may vary. Figure 2 depicts the ONF's SDN architecture. The functionalities mentioned above are confined in the Control Plane as part of the SDN controller. In addition, there are possible to have a set of sub-functions that we can consider as additional/high-level functions. These functions in most existing SDN controller implementations are implemented as external applications confined in the Application Plane. In particular, the SDN controller supports a set of APIs (via the North Bound Interface) that make it possible to implement external network services. The most common SDN-based external applications are:

- Virtualization and slicing;
- Tenant creation and tunneling;
- Traffic flow measurement and statistics (telemetry);
- Performance monitoring;
- Firewall and security;
- Network address translation;
- Load balancing.

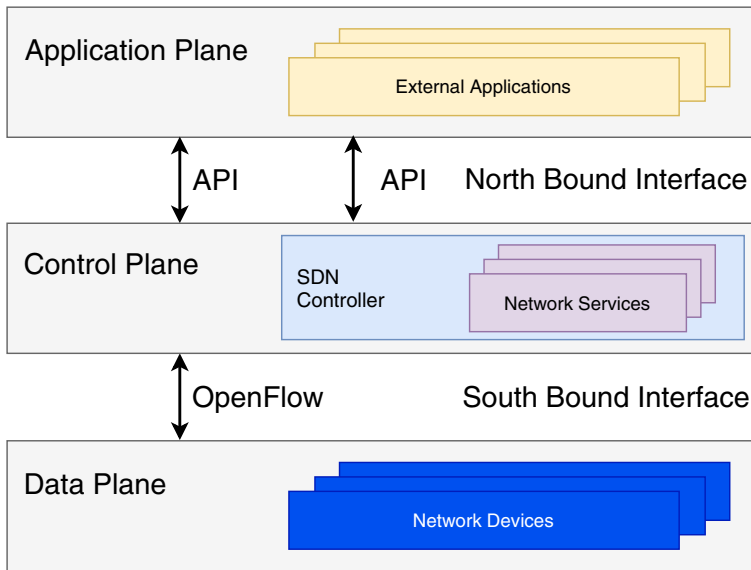


Fig. 2 ONF's software-defined network architecture

2.3 Ryu SDN Framework

Due to its simplicity and components-based architecture, the Ryu SDN framework is the starting point for our MSN implementation. In particular, a Ryu application consists of a Python script that extends the `RyuApp` base class and implements the `Observable` and `Observer` interface. These interfaces allow the application to interact with the event-based communications in the Ryu framework. For instance, the `OFPPacketIn` event is the event generated when the switch sends the packet to the controller. This event invokes the subscribed functionality that can process the packet and can create the OpenFlow rule. First, the Ryu Framework starts the Application Manager that loads all the applications and registers the associated events. The most important application in the Ryu Framework is `ofp_handler` which allows the framework to interact with OpenFlow protocol (`OpenFlowController` class). To efficiently communicate with switches, Ryu Framework creates a virtual representation of switches called *Datapath*.

Figure 3 shows the logical architecture of the monolithic Ryu SDN Framework. The `ofp_handler` operates as the event dispatcher of the Framework and manages the Datapaths. In particular, it manages the *Hello* and *Echo* messages and updates the status of the ports when necessary. The Ryu apps, once invoked by an event can reply directly to the Datapaths.

By default, if nothing is specified, Ryu starts with only `ofp_handler` as application. The `ofp_handler` is the core-fundamental component of the Ryu SDN framework as it contains all the basic SDN functionalities.

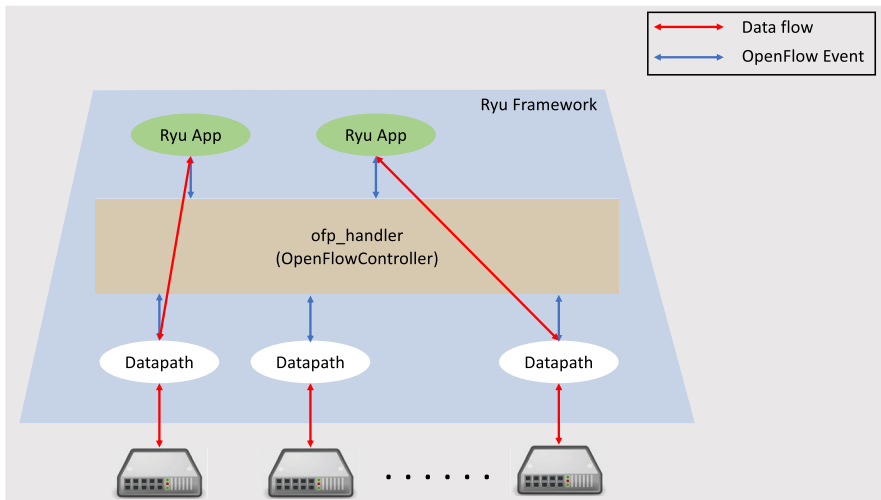


Fig. 3 Ryu internal monolithic logical Architecture

2.4 Motivation for SDN Controller Decomposition

Recently, the application of distributed SDN controllers has been widely studied in the literature, from different deployment perspectives. The main efforts were on the applications IoT device for smart city, disaster management [17–19], and Industrial Internet of Things (IIoT). In the following of this paper, we focus on the application of distributed deployment for IIoT. This because is a challenge scenario due to very strict requirements in terms of latency, reliability, and scalability. In particular, IIoT is a new paradigm in Industry 4.0 and it consists of the remote operation of machines, computers, and robots enabling intelligent industrial operations. Moreover, it is aimed at complete automation of the manufacturing process, from the raw material input to manufacturing, storage, distribution, and end-user marketing. In such scenarios, various heterogeneous devices and users are involved [20]. IIoT network requires real-time controlling, e.g., to control robots. These applications require, resilient, dynamic, and autonomous networking with low-cycle times (around 100 ms), and a high-reliability rate (close to 99,99%) [11, 12]. These requirements are very difficult to achieve with the existing SDN centralized controller deployment, especially for a large network. It is because SDN deployment incurs in propagation latencies as they have to be deployed at the center, which could be at a significant distance from the network device or forwarding switches to be managed. Therefore, despite the expected benefits of centralized controller design, it raises many challenges, including scalability and reliability.

The existing technique proposed to alleviate this problem is the hierarchical deployment of the SDN controller. Using the physical decentralization of the control plane approach, it is expected to address the scalability and latency problems [20]. However, such physically distributed, but logically centralized, systems bring an additional set of challenges. First, the distributed deployment of a monolithic

SDN controller requires (unnecessary) replication of code so to deploy the whole SDN controller at each location. That means whenever a new controller is deployed the whole SDN system has to be replicated in the distributed location. In other words, the monolithic system further provides a granular level challenge in terms of increasing the required functionalities. As the SDN system's internal modules are tightly coupled, it is difficult to dynamically increase the serving capability of the controller without adding a new SDN controller. In other words, the whole SDN has to be replicated in response to the workload demand that could have been performed by increasing just the required functional modules of the SDN controller. Furthermore, in terms of resilience, the monolithic system also has further disadvantages as a single controller fails in a given local area, it has to use the central controller which is further away from the center creating latency and congestion [21]. This is because instead of instantiating the specific function that leads to failure, either it has to instantiate the whole SDN controller or contact the nearby controller. See Table 1 for comparison between monolithic systems and microservice system.

However, if we can decompose the monolithic SDN controller into sub-functions, we could deploy only the required functionality in the required location as per network size and network management workload demands. Moreover, dynamic response to dynamic network demand is possible by dynamically instantiating the required functions of the controller components to meet the service and network demands. This means dynamically scaling the controller capability with the dynamic service demand.

Therefore, a decomposed SDN controller deployment could provide a flexible and efficient local deployment of required controller's functionalities, while deceptively scaling controller's unctions on demand. This would potentially be advantageous in terms of latency and reliability, due to reduced code size and the flexible scaling of resources and controller functions. It would become possible to dynamically increase the number of functions and resources, which could be a horizontal and vertical extension of functions deployed as virtual network functions (VNFs) to meet the service demands. A typical application of a decomposed SDN scenario is depicted in Fig. 4. The SDN controller with the minimum required functionality could be hosted in an edge data center.

2.5 Microservice Architecture

The monolithic SDN controller is easier to develop and deploy. However, since a monolithic SDN controller is built as a single and indivisible unit, updates or changes are very difficult because they require the replacement of the whole stacks of the control system. Moreover, handling a huge codebase, adopting new technology such as AI, dynamic scaling, deploying, and implementing changes is very difficult. This is a big disadvantage in the era of functions *containerization* and *cloudification* where features of loose-coupling, distributed deployment, and dynamic scaling of resources are required. In general, the monolithic SDN controller has disadvantages of scalability, reliability, and reusability.

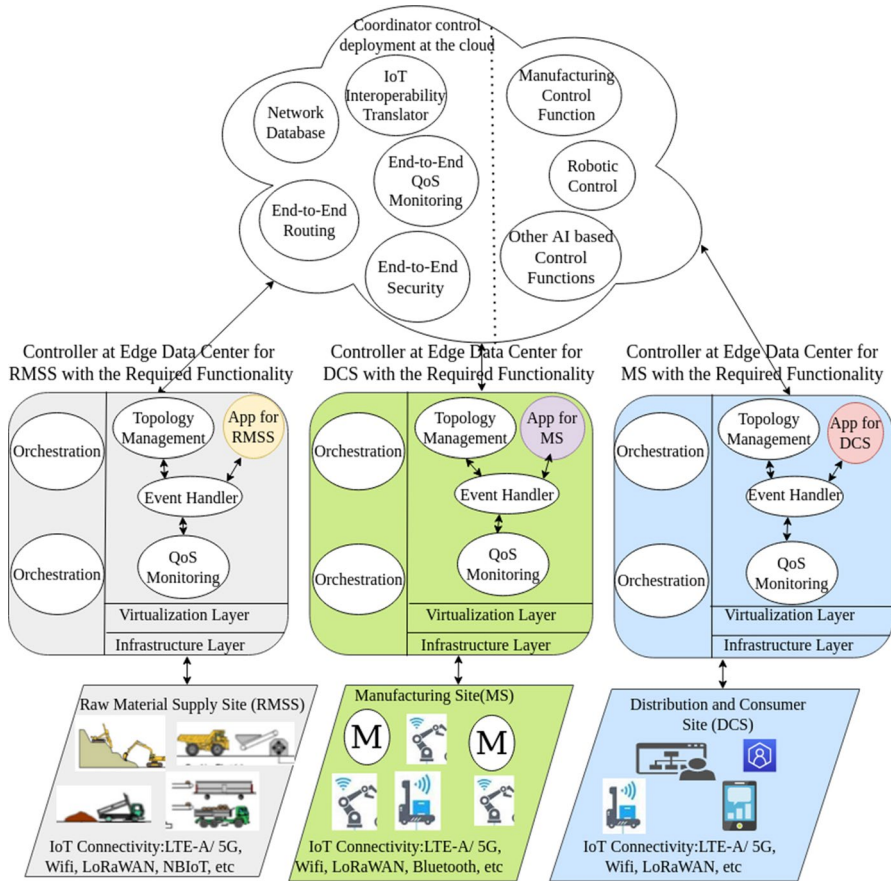


Fig. 4 Microservices-based SDN deployment blueprint in IIoT Scenarios

In principle, monolithic SDN controllers have pros and cons compared to microservices. Microservices are a means of creating loosely-coupled sub-functions or sub-services, replacing a large software system. So far we have mentioned its disadvantages. An alternative approach that we have indicated so far is microservices. A multi-agent system is also another alternative to have a decomposed management system. Since we are focusing on a microservice-based decomposition technique, we will limit our discussion to only microservices instead of multi-agent approaches, which is discussed in detail in [22].

Microservice is a variant of the service-oriented architecture (SOA) structural style in software development [23]. It arranges an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained decoupled functions. The protocols interconnecting the decoupled services are lightweight. The architecture describes a particular way of designing software applications as suites of independently deployable services. While there is no precise

Table 1 Comparison between SDN Monolithic architecture and Microservices-based SDN architecture [21]

Design principles	SDN controller as monolithic system	SDN controller as microservices
Scalability	<p>Monolithic SDN controller is difficult to scale because it requires replicating overall SDN controller for smaller increase in demands. This is because tight coupling of internal structure of the Monolithic SDN controller. Moreover, it is very hard to upgrade as it requires working on the big system instead of replacing or maintaining just the failed functions.</p> <p>Monolithic SDN controller is bulky to deploy the system in a containerized environment.</p>	<p>Microservices-based SDN controller can be placed in a container or any virtual environment. Using function sequencing, dynamic instantiate of containerized services, and function orchestration, it is possible to recreate the SDN functionality, which can easily scale by adding only those components that require additional resources.</p>
Cloud readiness	<p>Monolithic SDN controller is bulky to deploy the system in a containerized environment.</p>	<p>Microservices can easily be deployed and orchestrated in containerized environment. Therefore, a microservice-based design of SDN controller enables easy deployment in a distributed environment such as cloud</p>
Loose coupling	<p>Monolithic SDN controller's internal modules are tightly coupled which prevents it to be deployed in a distributed environment without replicating the controller. Wherever and/or whenever SDN controller functions are needed, the whole system must be deployed instead of the required functions only</p>	<p>On the contrary to Monolithic based controller, SDN controllers based on microservices have loosely coupled components which enables flexible deployment in a distributed environment for dynamic scaling along with the dynamic service demand</p>
Maintenance	<p>If a Monolithic SDN controller internal component fail, locating the problem to make changes is difficult and may take a lot of time. This is because the SDN controller is a big complex set of code that are tightly coupled</p>	<p>Instead, in the case of microservice-based design of SDN controller has enough decoupling of the functional components to identify, isolate, and replace with new function of that particular function without requiring to replace or work on the whole system. I means, If one part of the system fails, mostly only a single component of the SDN controller function may damaged, and it can be isolated and replaced/fixd. This could be by instantiating a new container for example</p>
Component reuse	<p>For a monolithic based SDN controller, it is difficult to reuse the sub-components as every function or internal modules are part of a single system</p>	<p>However, microservice-based components of an SDN controller functions could be reused and orchestrated to be deployed with other functions for dynamic response to the workload increase. So microservice-based SDN controller functions reusable as components are loosely coupled, independently implemented and deployed</p>

definition of this architectural style, there are certain common characteristics around an organization, business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

As per the above definition, microservices-based systems provide the possibility of building a system from small heterogeneous components. A lot of existing tools could be geared toward microservices such as communication interfaces. Microservices-based systems have the advantages of scalability, reusability, flexibility, and agility. A comparison of microservices and monolithic systems is provided in Table 1. In the table, we made three important considerations. First, the SDN functions that we are proposing to be designed as microservice inherit the general microservice properties. Second, the microservice-based SDN function could be considered as separate functions that could be placed in a container and cloud environment. Third, by instantiating and sequencing the necessary functions, we can recreate the decomposed SDN controlled system with only the important and required SDN functions hosting them in the nearest edge computing data center. We discuss these points further in the next subsection

3 Microservices-Based Decomposed SDN Controller

This section provides the proposed decomposed architecture of an SDN controller, presenting a functional definition of components and interfaces. The proposed SDN controller decomposition architecture shows the decoupling of internal components of the SDN controller to be deployed as a microservice.

3.1 SDN Controller Internal Components as a Microservice

The main principle to retain in decomposing an SDN controller is that the network information and state should be synchronized and self-consistent providing a global view of the network. That allows an independent implementation and components reuse. We consider a decomposition of SDN controller, as depicted in Fig. 5 which shows a decomposed three-layer SDN architecture reflected in an NFV architecture [15].

Figure 6 depicts our proposed deployment architecture of the decomposed SDN controller, based on microservice architecture. The main principle behind the architecture is the use of microservice based functions to replace the monolithic SDN controller and develop it as a lossless coupled composition of containerized services. As discussed in Sect. 2.2, we identified and defined the internal components of the SDN controller that could be developed independently as a microservice. This requires delineating the system based on specific service functions. In other words it should be possible to specifically define a function that could independent be developed as microservice and deployed in a virtual environment or container.

This also means that the control layer core sub-functions are decomposed into sub-functions and implemented as microservice and deployed as VNFs in containers. Each sub-function is developed as a microservice and creates an independent and

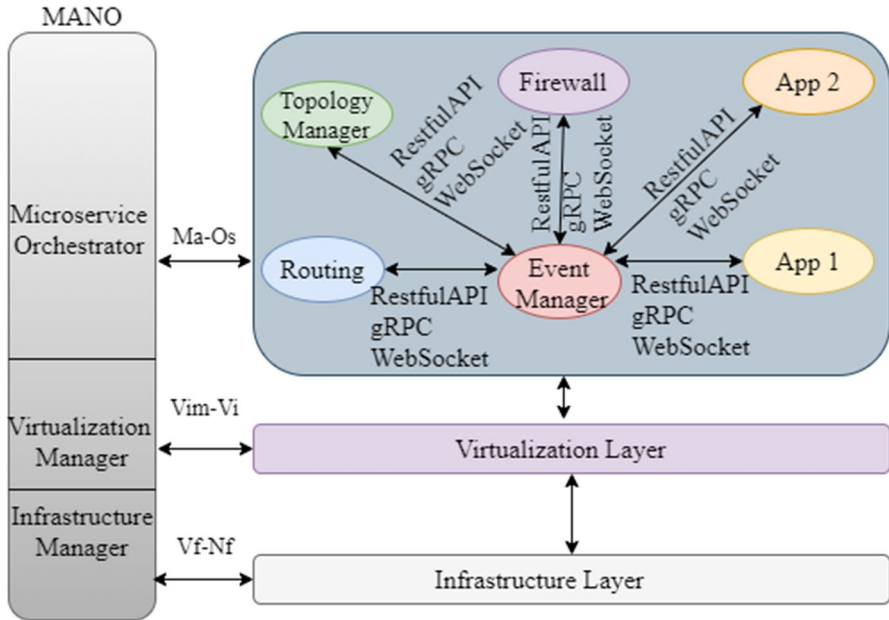


Fig. 5 Architectural overall of proposed microservices-based SDN Controller decomposition

SDN as Microservice: Simplified Deployment Architecture

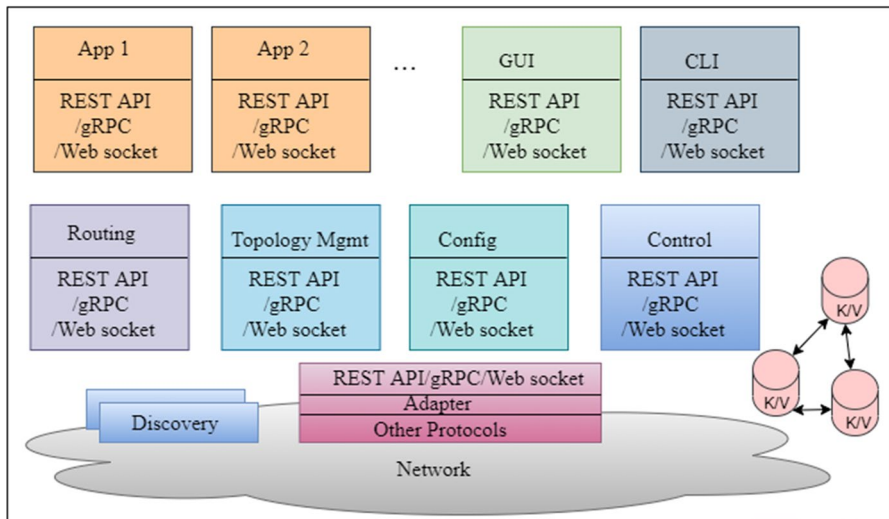


Fig. 6 Proposed microservices-based SDN controller sample deployment architecture

autonomous service unit. These microservices are able to independently perform the required service function. The independence of microservices would provide the possibility of deploying them in a distributed environment while scaling them as per the service workload. This may require instantiating multiple instances of the same service or new instances of additional microservices.

Once the necessary functions are developed as microservices, the SDN system requires a service aggregation to deliver a final functional system. Therefore, the final controller system becomes the organization of the independently developed microservices to create the equivalent SDN controller functions. As indicated above, the controller's components can be executed on arbitrary computing platforms on distributed and virtualized resources such as virtual machines (VMs) or containers in data centers. The loosely-composed system can be viewed as a black box, defined by its externally-observable behavior, emulating the original monolithic SDN controller. However, a distributed implementation must consider maintaining a synchronized and self-consistent view of network information and states. The independently developed microservices based functions could be orchestrated by a standard orchestrator, such as MANO. This would create a service function chain to equivalently perform the legacy SDN controller's functions.

As can be seen from Fig. 5, the upper layer is a pool of independently implemented microservice based SDN components such as topology manager, event handler, and other applications. Each component performs a specific function such as traffic routing, topology management, and even handling. It is possible to categorize functions as basic SDN controller functions and additional functions or applications. Basic SDN controller functions are mandatory to provide the minimum possible function of the SDN controller.

3.2 External Applications as a Microservice

In addition to the network services, external applications could be incorporated to extend the controller basic functionality [3, 14]. Depending on the network to be controlled, various types of applications could be implemented in the application layer such as additional QoS service, traffic predictions, traffic classification, slicing, firewall, and novel deep packet inspection, see Fig. 6. All these functions can be developed based on microservice and could be considered as VNF in container which can be deployed in a distributed environment. In doing so, we are effectively eliminating the traditional delineation between the control layer and the application layer. This is interesting concept to notice as the legacy architecture of the SDN controller has three layers, which are forwarding, control, and application, see Fig. 2. However, in the proposed MSN framework there is no apparent difference between an SDN internal function and network application that are deployed as VNF in containers. This is because all components could be implemented as loosely coupled microservices, and running in a container that can be deployed anywhere. Our implementation is a testbed showing this by splitting the Ryu controller into two separate functions that are deployed in a docker container.

3.3 Communication Interface Between Decomposed Services

As indicated above, in our proposed MSN framework, the components can be deployed as VNF in a distributed environment as a web service. A web service is a service that can be called by an application. Therefore, the decomposed SDN controller sub-functions could be considered as web services, which can separate programs that are independent of other applications and can be run on different machines. Such functions communicate with each other or with the event distributor, such as sending and receiving event notifications through the communication interface. These communication interfaces between the decomposed and containerized applications are based on open-source communication interfaces. These generic communication interfaces are tested in our implementation which are REST, WebSocket, and gRPC. Each of them has its pros and cons in terms of latency for web-based services.

3.3.1 Communication Interface Between the Decomposed Microservice Based Controller Functions

Our proposed MSN framework uses RESTful API for as the communication interface. RESTful is an application program interface that uses HTTP requests to GET, PUT, POST, and DELETE data [23, 24]. It provides interoperability between different network application developers of the SDN controller sub-functions. These APIs can be used to facilitate efficient microservices-based function orchestration and automation of the network to align with the needs of different applications. RESTful API is a stateless architecture for data transfer. We chose REST for multiple reasons such as performance, scalability and, most important, is the standard declared in the 3GPP white-paper about Release 15 of 5G networks [25]. RESTful API also allows the support of large numbers of components and interactions among them which makes it ideal for IIoT deployment scenario indicated above. Moreover, RESTful API has a uniform interface which simplifies and decouples functions making it suitable for a microservice-based SDN function communication.

As a comparison with RESTful API while testifying our hypothesis on how the use of REST has advantages for SDN controller decomposition, we used gRPC and WebSocket [24]. gRPC is an open-source remote procedure call (RPC) system initially developed. It uses HTTP/2 for transport, Protocol Buffers as the interface description language, and provides features such as authentication, bidirectional streaming and flow control, blocking or non-blocking bindings, and cancellation and timeouts. gRPC is roughly seven times faster than REST when receiving data and roughly ten times faster than REST when sending data for this specific payload. This is mainly due to the tight packing of the Protocol Buffers and the use of HTTP/2 by gRPC. Moreover, WebSocket is another communications protocol that provides a full-duplex communication channel between the servers and the clients, using a single TCP connection. It was standardized by the IETF as RFC 6455 in 2011 [24]. It provides real-time communication between a client and the server.

Finally, we would like to indicate that the drawback of an SDN controller decomposition and deploying it as a distributed system could create an additional challenge

of synchronization between components. Even if the decomposition has advantages compared to a centralized SDN architecture in terms of availability, resilience, and flexibility for a reconfigurable system, the distribution of functions imposes a continuous network state synchronization challenge. The network state database could be centralized or distributed. In other words, the network state is replicated or distributed between the controllers requiring repeated synchronization. In each case, maintaining synchronization is a challenge. However, the problem of synchronization of the database in a distributed system is a long-studied subject that could be considered for the case of decomposed SDN controller [26, 27]. For example, the existing controller synchronization strategies developed for distributed controllers improve joint controller decision making for inter-domain routing. Given existing solutions in the literature, in this work, we consider the proposed MSN system precisely synchronized. This assumption is made reasonable by the system's characteristics. The various modules, composing the distributed SDN controller, are virtual containers placed in and running on servers and, more in general, on network computing hardware. The containers get the synchronization from the clocks of their hosting hardware. In fact, this network hardware is accurately synchronized via well-known standardized synchronization protocols like IEEE 1588 Precision Time Protocol (PTP) [28, 29], which has already been used to achieve a synchronization accuracy in the order of tens of nanoseconds.

4 TestBed Implementation and Performance Results

For the evaluation of the MSN implementation, we proposed an implementation based on Ryu SDN controller [6] due to its component-based characteristics that blend well with the microservices-based SDN controller perspective. The following subsections introduce first the fundamentals of our proposed microservices-based decomposition framework and, finally, we present our experimental environment and the performance results of the MSN implementation.

4.1 Decomposing Ryu SDN Controller

As theoretically described in the Sect. 3, we identified in the Ryu implementation the essential modules that describe a basic SDN system:

- Event Handler System Management: this module is in charge of catching an OpenFlow event and forwarding it to the destination. This module works reactively and may be considered as the core module for a decomposed SDN implementation.
- Routing System: this function is used to generate Flow rules to allow the network to exchange packets among nodes and switches.
- South-bound Management: this module allows the system to interact with the underlying system with several protocols.

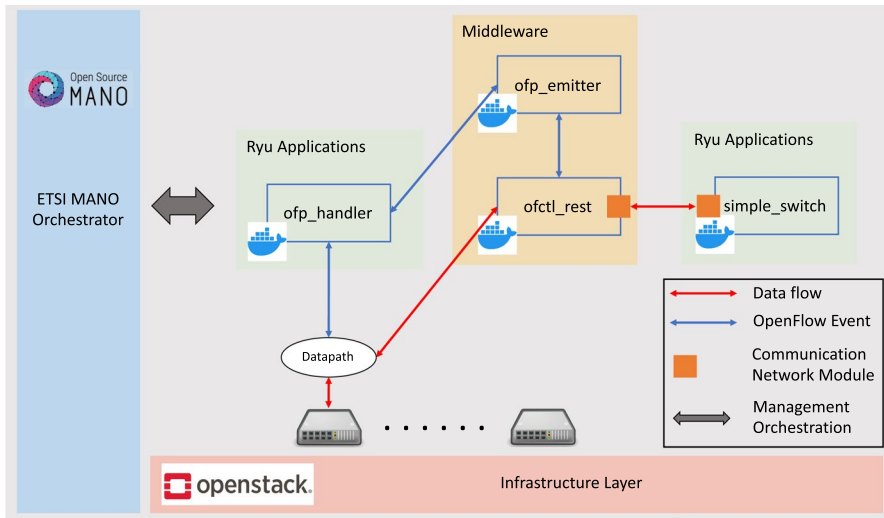


Fig. 7 Ryu-based MSN Implementation Architecture

The starting point for the microservices-based SDN decomposition is the characterization of the core part of the SDN system that allows the communication from network components to the applications (i.e.: from the control plane to the data plane). In particular, the externalization of that SDN core part allows the network to be observable and manageable from external processes. The proposed MSN implementation follows that principle and its implementation is provided to demonstrate the feasibility of using a middleware that allows the interaction between the core of the SDN and external processes.

First, we isolated the event emitter from the core of Ryu Framework and we created a support middleware module (the yellow block in the Fig. 7), incorporating the REST APIs block with the emitter to be able to transform events in REST calls. The middleware is the fundamental block for a microservices-based SDN decomposition, precisely because connects the legacy SDN environment with external microservices. Second, we turned each Ryu App in a separate block (i.e., microservices) external to the Ryu Framework which can communicate with the Framework via REST APIs through the middleware. In this way, we transform an SDN functionality into an atomic block (microservice) releasing it from the whole SDN Framework. For implementation purpose, we leverage the already existing REST-based APIs in the Ryu framework, precisely the *ofctl_rest* module. Figure 7 shows the resulted Ryu-based MSN implementation architecture. The described approach can be used for different network technologies, not only REST-based, such as gRPC, WebSocket, RPC, and so on. What is changing is the block internal to the middleware (the *ofctl_rest* block in the Fig. 7) module that connects to external microservices.

Once demonstrate the feasibility of the MSN framework, to improve reliability and scalability is important to leverage a solution like virtualization and/or containerization that results easy to orchestrate via an orchestrator. In our solution, we adopted Docker

Container as containerization ecosystem, Open Source MANO for the orchestration and OpenStack as the infrastructure layer. We produced different Dockerfile for reproducing the architecture shown in Fig. 7. In particular, we created a Docker container for our middleware that incorporates the *ofp_emitter* and *ofctl_rest* blocks inside and another Docker container for the event handler functionalities such as the *ofp_handler* block. Finally, Ryu Apps are considered as separated Docker Containers that include SDN functionalities including routing functionality or Firewall. For major details, we have all codes available at repo source: https://gitlab.com/dscotece/ryu_sdn_decomposition/.

4.2 Experimental Environment

Our performance evaluation aims to prove the feasibility of the MSN framework and evaluates its performance by proposing a benchmark of several communication technologies to enable needed interconnections and interoperability across microservices. The tests show results in terms of reliability, scalability, and latency of the system. To achieve this, we separated our testbed into two different parts: first, we tested the system latency introduced by splitting the SDN controller in microservices for different network interconnection technologies; finally, we tested our system to calculate the improvements in terms of reliability and scalability.

The performance of the MSN framework has been evaluated in a simulated testbed environment. We used the implementation details described in the Sect. 4.1. In addition, the OpenFlow protocol will be used for the forwarding plane of switches. However, it should be noted that the vision of the MSN framework is completely agnostic to any specific SDN implementation. Our testbed consists of a Linux workstation (Ubuntu Server 18.04 LTS) equipped with a 2x AMD Opteron(tm) Processor 6376 3.2GHz 16 cores processor and 32 GB 1600MHz DDR3 memory. We employ the following software, used to implement and test the proposed architecture:

- The Microstack version of the Openstack which plays the physical infrastructure of our testbed
- The docker community edition version 19.03.7 for dockerized microservices
- The open-source MANO release 8 as the system orchestrator
- Ryu SDN Framework
- Python 3
- Mininet for creating a virtual network

We evaluated the feasibility and performance of the solution for three different network communication technologies such as WebSocket, gRPC, and REST. Finally, all results obtained in the testbed are an average of 30 runs that exhibited a limited variance of under 5%.

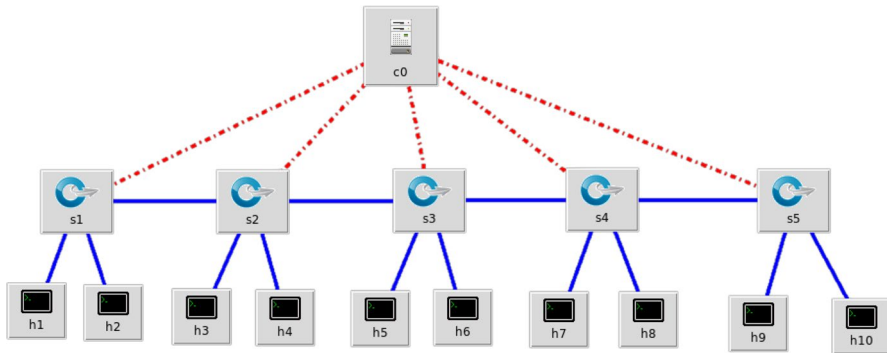


Fig. 8 Mininet Topology for Experimental Testbed

4.3 Benchmark of Network Communication Protocols

First, the testbed calculates the overhead introduced by the microservices-based approach in-terms of response time. To ensure this, we calculated three different latency: the response time of the first packet, the response time of the normal flow, and the average response time of rule updating packets. The response time for the first packet means, the time needed to send the first packet from one node to another. In an SDN network, with a reactive approach, the controller adds rules to the data-plane when it receives a new packet. This generates latency for the first packet of the flow. Once that is done, the flow can reach the destination node without passing through the SDN controller. Sometimes, during the flow, there are some updating packets to update, for instance, the expiration time of a rule. These packets generate latency because as in the case of the first packet, the flow must reach the SDN controller before. We repeated these experiments for different nodes of the network (H1 and H3 first, and then H1 and H10). As it can be seen from Fig. 8, there are two switches between H1 and H3, and there are five switches between H1 and H10. To calculate these latencies, we sent a video streaming across the network and we kept the average round trip time. We repeated this test for each network communication technology.

The results, shown in Fig. 9, show that the major delay is on the first packet latency. REST protocol appears to be seven times slower than WebSocket technology, in H1 and H3 scenario, whereas gRPC protocol provides performance like the REST protocol but a little faster. The performance further degrades in the H1 and H10 scenario for all protocols. In particular, the REST protocol results around ten times slower than WebSocket technology. This is due to the multiple connections between switches. Finally, the performance of all protocols during the normal flow was omitted due to the very low latency time (0.01 ms average around all protocols) but proves that all protocols are consistent and similar to each other.

In conclusion, we note that the REST protocol has a high response time for the first packet and rule updating packets compared to WebSocket and also to standard Ryu. However, the response time during the normal flow remains the same for all

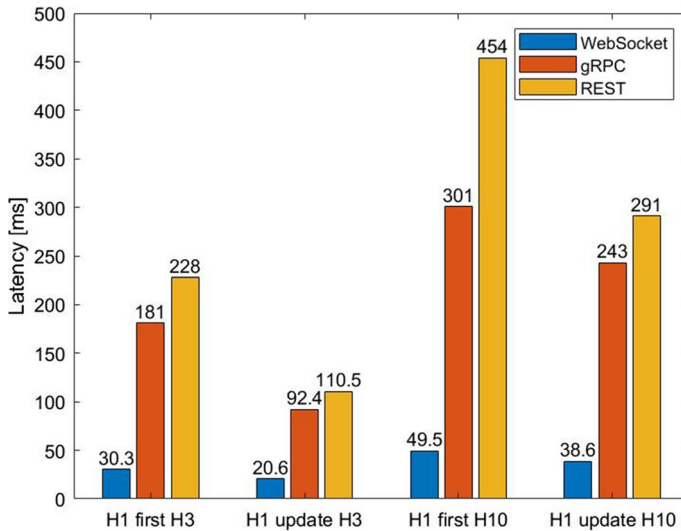


Fig. 9 Performance results

protocols. Therefore, it is apparent that the benefits of the microservice-based SDN model need to be balanced with any trade-offs incurred. On the one hand, despite the WebSocket protocol proves to be faster, it strongly depends on the Socket concept which means rely on the IP address and the Port number of the services. On the other hand, the gRPC protocol could become dominant in the future thanks to the adoption of the HTTP/2 protocol and to the use of Protobuf as the payload format. Furthermore, factors such as scalability and reliability (or availability) should be taken into account when deciding whether to use standard SDN or the microservices-based one. Moreover, the best choice of the right communication protocol depends on many factors including the context. For instance, a heterogeneous and ultra-reliable industrial scenario may require REST as a communication protocol to guarantee high connectivity among devices.

4.4 Resilience and Scalability Test

Finally, the testbed is designed to calculate the improvements of our solution in terms of the reliability and scalability of the system. To achieve these features, we used the ETSI MANO standard such as the open-source MANO implementation. The microservices-based approach allows the system to develop a horizontal scalable to easily adapt to the dynamics of the input workload and to tolerate potential run-time faults. Indeed, the OSM Autoscaling functionality automatically scales VNFs based on available metrics such as CPU and memory consumption, packets received, packets sent, and so on. Each SDN component is a Docker container that will be encapsulated in a VNF. Therefore, there will be single or multiple VNFs that represent SDN applications or functionalities. As discussed before, the OSM Autoscaling function provides an automatic solution for fault tolerance management. This

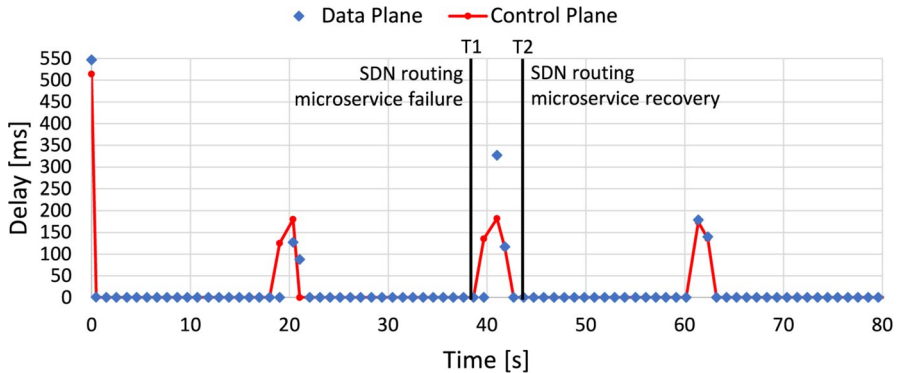


Fig. 10 Resilience Test: average message delay

is possible through a scaling descriptor, that is part of the Virtual Network Function Descriptor (VNFD) which specifies the metrics and thresholds to be monitored.

4.4.1 Resilience Test

To test the resilience of the system, a node sends a continuous flow of packets. This test simulates the fault for an SDN routing path component and the possibility of instantiating a new instance as a backup. In this test, the communication between VNFs is via REST protocol and the MANO orchestrator helps to manage them by automatic scaling process. In particular we used a video streaming simulation, from a source to a sink node while the SDN routing microservice goes down. Figure 10 shows the average message delay recorded during the experiment. Each observation shows the delay in the control plane and the data plane. Since there is no message lost between time T1 and T2 but only small glitches in both the average message delay in the control plane and data plane at two instants prove the robustness of the system. The other small peaks correspond to the controller's rule updating packets at the datapath. The first delay is the delay introduced by the first packet, see the next experiments. This experiment shall be understood as a way to highlight the robustness achieved by the SDN system in its microservices-based deployment. Therefore, the standard SDN Controller, if some issues occur, is not able to react unless it is used in a distributed way. However, this means having two or more SDN Controllers deployed at the same time.

4.4.2 Scalability Test

We evaluate the scalability of the system in two different scenarios: multiple network service, and single network service. ETSI MANO defines network services (NSs) as a composition of VNFs that specifies a service such as an SDN controller. The first scenario—multiple NS—relates to the scalability of the entire NS that is comparable to a distributed SDN scenario. Figure 11 shows the related scenario in the OpenStack and Open Source MANO platforms. The OpenStack

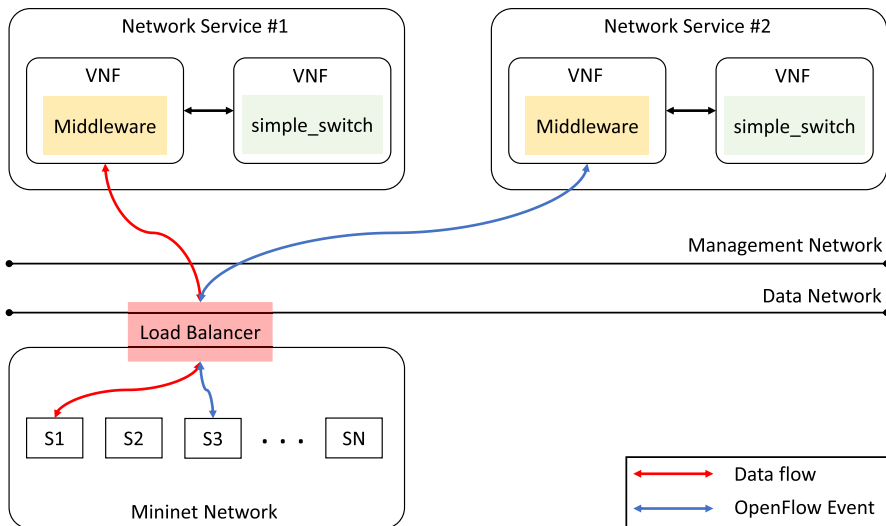


Fig. 11 Scalability test: scenario 1 (multiple NS)

provides a management network and a data network to correctly connect entities to each other, in particular, the Mininet network and the SDN system. In this scenario, we instantiate different NS to provide scalability. The load balancer in the mininet network provides a distributed control plane where each controller is in charge of a sub-set of the switches.

In the other scenario—single NS—we simulate the scalability of VNFs for instance the routing capabilities of the MSN prototype by keeping a single NS. Figure 12 shows the related scenario in the OpenStack and Open Source MANO platforms. The SDN system is composed of a single NS in which each micro-services (as VNFs) can be instantiated multiple times to provide robustness and scalability. The autoscaling feature of the Open Source MANO allows to define the scaling descriptor as a part of the VNF definition. In particular, is possible to define several metrics to monitor and a load balancer (included in the middleware) that redirects requests to the routing service following a balancing strategy such as Round Robin and so on.

We calculated the average latency time for the first packet and for the normal flow by considering different host at different distances. The reference topology shown in Fig. 13 is composed of 16 hosts and relates to a hierarchical network topology that is the most widely used in real datacenters [30]. In the Fig. 14 we show the average latency time for the first packet in both scenarios with 2 and 3 replicas, while the average time for normal flow is depicted in Fig. 15. In conclusion, we tested two different scenarios in the OpenStack and Open Source MANO for the scalability concern, and we noted that replicating the entire NS performs better than a single NS. This because in our test the middleware is the bottleneck for the switches while splitting the management of the network to more middleware is more efficacy. However, in the second scenario, it is possible to replicate the middleware as well.

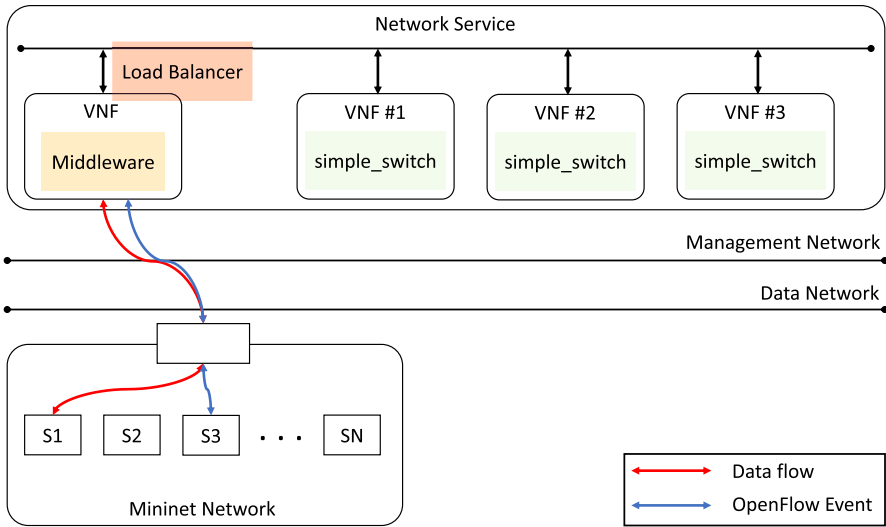


Fig. 12 Scalability test: scenario 2 (Single NS—multiple VNF)

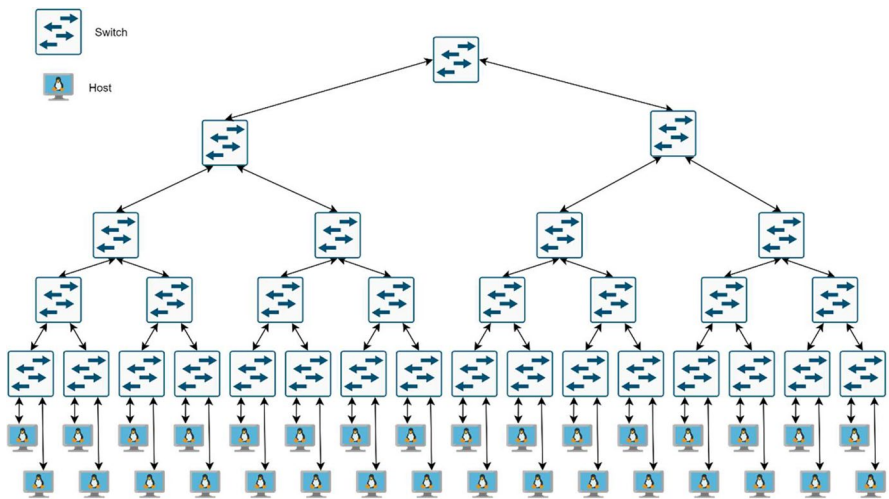


Fig. 13 Scalability test: network topology

5 Related Work

In recent years, there has been a lot of interest in SDN-based mobile networks, and several papers are proposing SDN-based mobile network architectures and listing the benefits they can bring to the mobile industry [20, 31–33]. In particular, the high numbers of researches and the high interest in this topic have led to an evolution of the traditional SDN architecture and due to the widespread of

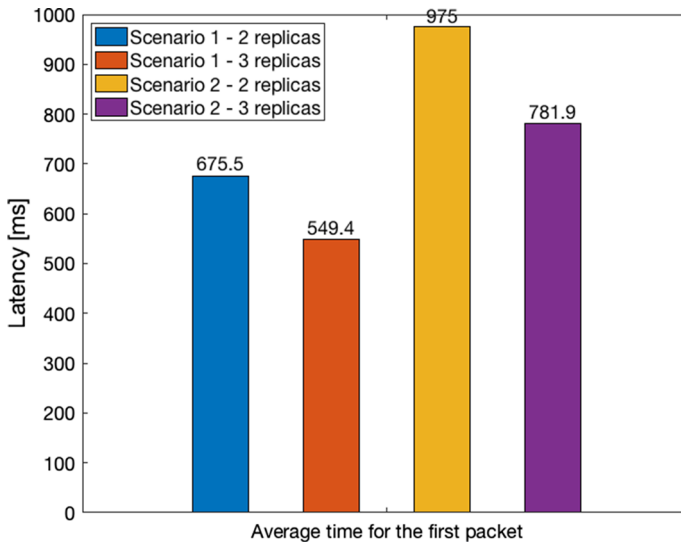


Fig. 14 Scalability test: average latency time for the first packet

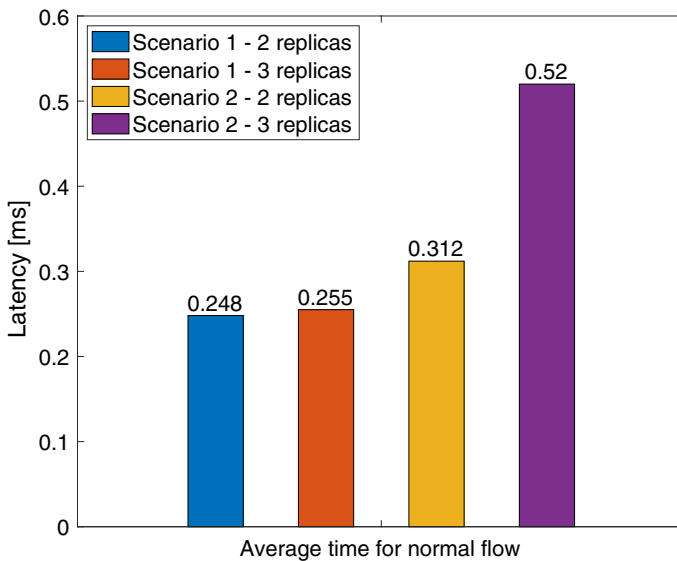


Fig. 15 Scalability test: average latency time for normal flow

the IoT, the SDN paradigm has started to be used to manage the IoT in several domains including smart city [34], smart home [35], smart health [36] and so on. The SDN paradigm helps the IoT networks to challenge several issues such as latency, reliability, privacy, flow control, etc. However, it still has some issues

caused by the logical centralization of the SDN controller, the main of them being scalability and robustness [37].

There are also numerous SDN controller implementation both from the open-source community and from commercial vendors. Typical examples are NOX, POX, Ryu, OpenDayLight, and ONOS. Even if the core principle of all SDN controllers is the same, each of them has a slightly different implementation approach. NOX is the first OpenFlow-based SDN controller written in C++ [38]. In the early exploration of the OpenFlow and SDN space, NOX has been the basis for many research and development projects. The NOX internal components mainly contain event handlers to receive and dispatch events such as incoming packets. POX is similar to NOX with a Python-only implementation. It is considered as a general, open-source OpenFlow controller [39]. Ryu is also a component-based SDN controller [6]. OpenDayLight is a collaborative open-source controller [8]. It is a modular, extensible, scalable, and multi-protocol SDN controller deployment. ONOS is also an open-source SDN controller [7]. The ONOS SDN controller software is written using Java and it provides a distributed SDN application platform atop Apache Karaf OSGi container. The controller has been designed to operate as a cluster of nodes that are similar in terms of their software stack and can endure the failure of individual nodes without causing disruptions in its ability to control the network operation.

All of above SDN implementations are based on a modular approach. However, all of them are based on monolithic architecture. NFV has changed the way we deploy network functions [15]. It enables easier, flexible, and dynamic deployment of a given network functions. SDN and NFV are complementary technologies [3, 15], but SDN controller function could be considered as a network function and it can be deployed as a VNF in a container [15]. However, the overall controller is a cumbersome and monolithic process. ONF specification indicated a possibility of implementing the SDN controller as either monolithic or decomposed in microservices [3]; however, only a few recent efforts have started exploiting this second possible design choice.

In wireless cellular networks, a recent article [40] showed how to split BaseBand units (BBU) of wireless RAN deployment. The authors propose to split it into different configurations; in each configuration, the functional units of BBU are split to be deployed as a virtualized functions. The BBU is virtually stored in a network cloud and accessible, as a shared resource. Similar to other network functions, such as BBU in wireless networks, SDN controller could be decomposed [41].

The first work showing an externalization of packet processing in SDN is presented in [41]. As an extension of this work, the author in [42] provided steps that are required to migrate from a monolithic to a microservice-based architecture. The functional components are distributed as microservices and a gRPC is used to communicate between the core modules and external components or applications.

μ ONOS is the latest solution proposed towards a standard architecture for distributed and split control plane. The μ ONOS project aims at creating a new generation of SDN architectures based on ONOS, splitting it into a set of microservices. These split functionalities are deployed as Docker containers and managed by Kubernetes orchestrator. The μ ONOS project is relatively young (it started in October 2019) and it is based on the P4/P4runtime [43] protocol which is a different control protocol

compared to the standard de-facto for the SDN paradigm, namely, the OpenFlow protocol. P4 protocol can emulate the behavior of OpenFlow; in addition, the communication between functionalities is via gRPC-based protocols, including gRPC Network Management Interface (gNMI) for network management interface configuration and gRPC Network Operations Interface (gNOI) for network command operations. However, the μ ONOS implementation is still in its infancy and there is still no available implementation to play with. Moreover, and most important, some implementation choices are not compliant with ongoing 5G standards. First, μ ONOS has limited integration with and support for ETSI-NFV standards because it neither implements the OpenFlow abstraction nor that of legacy network elements. In particular, the use of Kubernetes does not allow an easy integration within the 5G Edge architecture which requires ETSI-based protocols such as ETSI-MEC and ETSI-MANO. On the contrary, MSN framework exploits ETSI-NFV standard to define its architecture and ETSI-MANO for managing microservices, which allows MSN to easily operate within 5G-based networks. Second, in μ ONOS the communication between microservices is based on gRPC, a Remote Procedure Call framework developed by Google, which allows network entities to communicate (once defined an agreement) by serializing data with the Protocol Buffers, another Google solution. This results in a not so easy interaction with most available third-party applications which are using more open protocols such as REST-APIs. Because of that, in our MSN solution, we defined a generic communication module between microservices that allow users to choose the network communication technologies according to scenario requirements. This operation can be done at developing time, but, thanks to the dynamicity of containers/VMs orchestration via ETSI-MANO, it is possible to change dynamically the network communication technology. Finally, μ ONOS is based on the ONOS SDN system that forces users to have previous knowledge on it. Also, this may result inefficient in several scenarios where network entities are not powerful enough to run the (rather heavy) ONOS system such as the Industrial IoT scenario. To overcome this, MSN framework provides general guidelines to decompose an SDN system that is completely agnostic to specific SDN software and communication means.

In general, to the best of the authors' knowledge, the MSN solution is the first seminal work implementing complete guidelines for the SDN controller decomposition in microservices that fit 5G requirements including an implementation based on VNFs, Docker containers, and ETSI-MANO.

6 Discussion and Conclusion

In this paper, the microservices-based decomposition architecture is proposed for the SDN paradigm to improve agility, scalability, and reliability. The SDN features including dynamic flow control and the possibility to reconfigure the network according to application needs make it an enabler for the 5G next-generation IIoT networks. However, most SDN controllers are deployed as a monolithic block, and that can make them not efficient enough to cover the requirements of the IIoT networks such as scalability and robustness.

The MSN framework, proposed in this paper, paves the way to a new generation of microservices-based approaches for the next generation 5G-ready SDN networks. The use of microservices represents a big step ahead for the Cloud Continuum also in the vision of the Edge/Cloud hybrid architectures [44]. This paper inspects the use of microservices in the SDN paradigm presenting pros and cons of this novel paradigm when employed in this specific domain. The results presented in this paper are focused on the delay for the first packet, which means the latency introduced by the decomposition and distribution of microservices. Obtained results demonstrate the feasibility of applying microservices-based architecture to the SDN paradigm, which offers a wide range of benefits including deployment agility, scalability, and robustness that can be granted for each different SDN controller functionality. In the proposed solution, we take into account more ways to interconnect microservices with each other by analyzing different protocols such as REST, gRPC, and WebSocket. That allowed to compare not only the introduced delay by the mentioned technologies, but also the benefits that each protocol could bring. The tests show significant improvement in terms of reliability of the system in the case of a microservice become unavailable. Moreover, tests on the scalability show how to achieve scalability on the MANO orchestrator with two different scenarios. Therefore, orchestrating microservices for managing fault tolerance or for distributing the load will be a task for the MANO orchestrator. We also provide to the community working in the field our implementation.¹

Boosted by obtained results, we are now working along different ongoing work directions. First, we are using a reactive approach in the evaluation, we are working on implementation also a proactive approach so to further improve some aspects of latency. In fact, a reactive approach allows a system to react when something happens, for instance when a fault of a specific functionality occurs. In particular, MSN reacts by instantiating a new instance of the functionality as an VNF via the MANO orchestrator. On the contrary, using a proactive approach for the orchestrator would guarantee less downtime service by leveraging intelligent algorithms to predict fault at functionalities. Second, we are considering the possibility to add intelligence at the orchestration level to dynamically and proactively manage network entities according to network behavior. For instance, if the network is not performing as expected (e.g., because of congestion), an intelligent orchestrator can predict that and can allocate useful functionalities to overcome the congestion. This vision can be extended to all system components including every single SDN (sub-)functionality, by bringing intelligence at functionality level. So, each functionality can set up its behavior to fit network requirements. Third, we are working on a resource provisioning strategy, where a plan is needed for the careful identification of network nodes, VNFs, and services that will fulfill the application requirements. In particular, the provisioning strategy must consider both applications and control network functionalities as resources to manage. Following the plan instructions, the orchestrator will deploy and configure all resources needed by the application.

¹ For more details on the implementation and to reproduce the presented tests, we provide the entire project at https://gitlab.com/dscotece/ryu_sdn_decomposition/.

Acknowledgements This work has been partially funded by NATO Science for Peace and Security (SPS) Programme in the framework of the project SPS G5428 “Dynamic Architecture based on UAVs Monitoring for Border Security and Safety.”

Funding Open access funding provided by Alma Mater Studiorum - Università di Bologna within the CRUI-CARE Agreement.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Schenker, S.: The future of networking, the past of protocols, Mar. 2021. <https://youtu.be/YHeyuD89n1Y>
2. Kim, H., Feamster, N.: Improving network management with software defined networking. *IEEE Commun. Mag.* **51**(2), 114–119 (2013)
3. Open Networking Foundation: Software-Defined Networking: The New Norm for Networks (2012)
4. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: Enabling innovation in campus networks in *SIGCOMM Comput. Commun. Rev.* **38**(2), 69–74 (2008)
5. Bannour, F., Souihi, S., Mellouk, A.: Distributed SDN control: survey, taxonomy, and challenges. *IEEE Commun. Surv. Tutor.* **20**(1), 333–354 (2018)
6. Ryu: Ryu SDN Framework. 2021. <https://ryu-sdn.org/>
7. ONOS: Open Network Operating System. 2021. <https://opennetworking.org/onos/>
8. OpenDaylight: OpenDaylight and Open Networking ecosystem. 2021. <https://www.opendaylight.org/>
9. Floodlight: Floodlight Controller. 2021. <https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview>
10. Barakabitze, A.A., Ahmad, A., Mijumbi, R., Hines, A.: 5G network slicing using SDN and NFV: a survey of taxonomy, architectures and future challenges. *Comput. Netw.* **167**, 106984 (2020). <https://doi.org/10.1016/j.comnet.2019.106984>
11. Maier, M., Chowdhury, M., Rimal, B.P., Van, D.P.: The tactile internet: vision, recent progress, and open challenges. *IEEE Commun. Mag.* **54**(5), 138–145 (2016). <https://doi.org/10.1109/MCOM.2016.7470948>
12. 5G-PPP: White paper: 5g and the factories of the future. 2021. <https://5g-ppp.eu/wp-content/uploads/2014/02/5G-PPP-White-Paper-on-Factories-of-the-Future-Vertical-Sector.pdf>
13. Afolabi, I., Taleb, T., Samdanis, K., Ksentini, A., Flinck, H.: Network slicing and softwarization: a survey on principles, enabling technologies, and solutions. *IEEE Commun. Surv. Tutor.* **20**(3), 2429–2453 (2018)
14. μ ONOS: the next-generation architecture for the Open Network Operating System Controller. 2021. <https://docs.onosproject.org/>
15. ETSI NFV: NFV Technology. 2021. <https://www.etsi.org/technologies/nfv>
16. Leonardo, Á., Caraguay, V., Ludeña-González, P.J., Vicente, R., Tandazo, T., Isabel, L., López, B.: SDN/NFV architecture for IoT networks. In: *Proceedings of the 14th International Conference on Web Information Systems and Technologies V.1: ITSCO*, pp. 425–429, Sept. (2018)
17. Bera, S., Misra, S., Vasilakos, A.V.: Software-defined networking for internet of things: a survey. *IEEE Internet Things J.* **4**(6), 1994–2008 (2017)

18. Wu, D., Nie, X., Asmare, E., Arkhipov, D.I., Qin, Z., Li, R., McCann, J.A., Li, K.: Towards distributed SDN: mobility management and flow scheduling in software defined urban IoT. *IEEE Trans. Parallel Distrib. Syst.* **31**(6), 1400–1418 (2020)
19. Mouradian, C., Jahromi, N.T., Glietho, R.H.: NFV and SDN-based distributed IoT gateway for large-scale disaster management. *IEEE Internet Things J.* **5**(5), 4119–4131 (2018)
20. Okwuibe, J., Haavisto, J., Harjula, E., Ahmad, I., Ylianttila, M.: SDN enhanced resource orchestration of containerized edge applications for industrial IoT. *IEEE Access* **8**, 229117–229131 (2020). <https://doi.org/10.1109/ACCESS.2020.3045563>
21. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC), pp. 583–590 (2015)
22. Arzo, S.T., Bassoli, R., Granelli, F., Fitzek, F.H.P.: Multi-agent based autonomous network management architecture. *IEEE Trans. Netw. Serv. Manag.* (2021). <https://doi.org/10.1109/TNSM.2021.305975>
23. Kookarinrat, P., Temtanapat, Y.: Design and implementation of a decentralized message bus for microservices. In: 2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 1–6 (2016). <https://doi.org/10.1109/JCSSE.2016.7748869>
24. Chamas, C.L., Cordeiro, D., Eler, M.M.: Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: an energy cost analysis. In: 2017 IEEE 9th Latin-American Conference on Communications (LATINCOM), pp. 1–6 (2017). <https://doi.org/10.1109/LATINCOM.2017.8240185>
25. 3GPP: 3GPP TR 21.915 version 15.0.0 Release 15. 2021. https://www.etsi.org/deliver/etsi_tr/121900_121999/121915/15.00.00_60/tr_121915v150000p.pdf
26. Das, T., Gurusamy, M.: Controller placement for resilient network state synchronization in multi-controller SDN. *IEEE Commun. Lett.* **24**(6), 1299–1303 (2020). <https://doi.org/10.1109/LCOMM.2020.2979072>
27. Schiff, L., Schmid, S., Kuznetsov, P.: In-band synchronization for distributed SDN control planes. *SIGCOMM Comput. Commun. Rev.* **46**, 37–43 (2016). <https://doi.org/10.1145/2875951.2875957>
28. Lévesque, Martin, Tipper, David: A survey of clock synchronization over packet-switched networks. *IEEE Commun. Surv. Tutor.* **18**(4), 2926–2947 (2016). <https://doi.org/10.1109/COMST.2016.2590438>
29. IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008), IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems (2020)
30. Kliazovich, D., Arzo, S.T., Granelli, F., Bouvry, P., Khan, S.U.: e-STAB: energy-efficient scheduling for cloud computing applications with traffic load balancing. In: 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, pp. 7–13 (2013). <https://doi.org/10.1109/GreenCom-iThings-CPSCom.2013.28>
31. Gudipati, A., Perry, D., Li, L.E., Katti, S.: SoftRAN: software defined radio access network. In: Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN '13). Association for Computing Machinery, New York, NY, USA, 25–30 (2013). <https://doi.org/10.1145/2491185.2491207>
32. Raza, S.M., Kim, D.S., Choo, H.: The proposal for SDN supported future 5G networks. In: Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems (RACS '14). Association for Computing Machinery, New York, NY, USA, pp. 180–185. <https://doi.org/10.1145/2663761.2664237>
33. Coronado, E., Khan, S.N., Riggio, R.: 5G-EmPOWER: a software-defined networking platform for 5G radio access networks. *IEEE Trans. Netw. Serv. Manag.* **16**(2), 715–728 (2019). <https://doi.org/10.1109/TNSM.2019.2908675>
34. Gharaibeh, A., et al.: Smart cities: a survey on data management, security, and enabling technologies. *IEEE Commun. Surv. Tutor.* **19**(4), 2456–2501 (2017). <https://doi.org/10.1109/COMST.2017.2736886>
35. Stojkoska, B.L.R., Trivodaliev, K.V.: A review of Internet of Things for smart home: challenges and solutions. *J. Clean. Prod.* **140**, 1454–1464 (2017). <https://doi.org/10.1016/j.jclepro.2016.10.006>
36. Salahuddin, M.A., Al-Fuqaha, A., Guizani, M., Shuaib, K., Sallabi, F.: Softwarization of Internet of Things infrastructure for secure and smart healthcare. *Computer* **50**(7), 74–79 (2017). <https://doi.org/10.1109/MC.2017.195>

37. Oktian, Y.E., Lee, S., Lee, H., Lam, J., Yustus Eko Oktian: Distributed SDN controller system: a survey on design choice. *Comput. Netw.* **121**, 100–111 (2017). <https://doi.org/10.1016/j.comnet.2017.04.038>
38. Nicira Networks, NOX Controller. <https://github.com/noxrepo/nox>
39. Open Source Development Platform, POX Controller. <https://github.com/noxrepo/pox>
40. Bonafin, S., Bassoli, R., Granelli, F., Fitzek, F.H.P., Sacchi, C.: Virtual baseband unit splitting exploiting small satellite platforms In: 2020 IEEE Aerospace Conference, pp. 1–11 (2020)
41. Comer, D., Rastegarnia, A.: Externalization of packet processing in software defined networking. *IEEE Netw. Lett.* **1**(3), 124–127 (2019)
42. Comer, D., Rastegarnia, A.: Towards disaggregating the SDN control plane in CoRR (2019). [arXiv: abs/1902.00581](https://arxiv.org/abs/1902.00581)
43. P4: P4 Consortium. 2021. <https://p4.org/>
44. Bittencourt, L., Immich, R., Sakellariou, R., Fonseca, N., Madeira, E., Curado, M., Villas, L., DaSilva, L., Lee, C., Rana, O.: The Internet of Things, Fog and Cloud continuum: integration and challenges. *Internet Things* **3–4**, 134–155 (2018). <https://doi.org/10.1016/j.iot.2018.09.005>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Sisay T. Arzo is a PhD student at the University of Trento, Italy where he received his MSc. in Telecommunication Eng. He received his BSc. from Hawassa University., Ethiopia. He has more than five years of industrial experience in Telecom industry. His research interest included network softwarization, Software Defined Networking, Network Function Virtualization, IoT and Network Automation.

Domenico Scotece is a postdoc researcher at the University of Bologna. He received his Ph.D. and his M.Sc. degrees in Computer Science Engineering from University of Bologna in 2020 and in 2014, respectively. His research interests include pervasive computing, middleware for fog and edge computing, IoT, and management of cloud computing systems.

Riccardo Bassoli is a senior researcher with the Deutsche Telekom Chair of Communication Networks at the Faculty of Electrical and Computer Engineering, Technische Universität Dresden (Germany). He received his B.Sc. and M.Sc. degrees in Telecommunications Engineering. from University of Modena and Reggio Emilia (Italy) in 2008 and 2010 respectively. Next, he received his Ph.D. degree from 5G Innovation Centre (5GIC) at University of Surrey (UK), in 2016. Between 2011 and 2015, he was also a Marie Curie Early Stage Researcher at Instituto de Telecomunicações (Portugal) and a visiting researcher at Airbus Defence and Space (France).

Daniel Barattini received his B.Sc. in Electronic and Computer Engineering. from the University of Pavia, Italy (2018). He is currently pursuing his M.Sc. in Computer Engineering. at the University of Bologna, Italy. His areas of interest include Mobile Systems and AI.


Fabrizio Granelli is Associate Professor at the Deptment of Information Eng. and Computer Science (DISI) of the University of Trento, Italy. From 2012 to 2014, he was Italian Master School Coordinator in the framework of the European Institute of Innovation and Technology ICT Labs Consortium. He was IEEE ComSoc Distinguished Lecturer for 2012-15, IEEE ComSoc Director for Online Content in 2016-17 and IEEE ComSoc Director for Educational Services in 2018-19. He is coordinator of the research and didactical activities on computer networks. He is author/co-author of more than 250 papers published in international journals, books and conferences on networking.

Luca Foschini is Associate Professor of Computer Engineering. His interests span from integrated management of distributed systems and services to edge computing, from management of cloud computing systems to Industry 4.0. He is a voting member of the IEEE ComSoc EMEA Board.

Frank H.P. Fitzek is a Professor and Head of the Deutsche Telekom Chair of Communication Networks at Technische Universität Dresden (Germany), also coordinating the 5G Lab Germany. He is the spokesman

of the DFG Cluster of Excellence CeTI. He received his diploma (Dipl.-Ing.) degree in electrical eng from the University of Technology - Rheinisch-Westfälische Technische Hochschule (RWTH) - Aachen, Germany, in 1997 and his Ph.D. (Dr.-Ing.) in Electrical Engineering from the Technical University of Berlin, Germany in 2002 and became Adjunct Prof. at the University of Ferrara, Italy in the same year. In 2003 he joined Aalborg University as Associate Professor and later became Professor.

Authors and Affiliations

Sisay Tadesse Arzo¹ · Domenico Scotece²  · Riccardo Bassoli³ · Daniel Barattini² · Fabrizio Granelli¹ · Luca Foschini² · Frank H. P. Fitzek^{3,4}

✉ Domenico Scotece
domenico.scotece@unibo.it

¹ Department of Information Engineering and Computer Science (DISI), University of Trento, Trento, Italy

² Department of Information Engineering and Computer Science (DISI), University of Bologna, Bologna, Italy

³ Deutsche Telekom Chair of Communication Networks, Institute of Communication Technology, Faculty of Electrical and Computer Engineering, Technische Universität Dresden, Dresden, Germany

⁴ Centre for Tactile Internet with Human-in-the-Loop (CeTI), Cluster of Excellence, Dresden, Germany