IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 25 November 2024, accepted 31 December 2024, date of publication 8 January 2025, date of current version 15 January 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3527224

RESEARCH ARTICLE

Hybrid-Hierarchical Synchronization for Resilient Large-Scale SDN Architectures

ALESSANDRO PACINI[1], DAVIDE SCANO[1], ANDREA SGAMBELLURI[1], LUCA VALCARENGHI[1], (Senior Member, IEEE), AND ALESSIO GIORGETTI[2]

[1]TeCIP Institute, Scuola Superiore Sant'Anna, 56124 Pisa, Italy
[2]Department of Information Engineering, University of Pisa, 56122 Pisa, Italy

Corresponding author: Alessandro Pacini (alessandro.pacini@santannapisa.it)

This work was supported in part by European Commission Horizon Europe Smart Networks and Services (SNS) Joint Undertaking (JU) DESIRE6G Project under Grant 101096466; in part by European Union (EU)—Next Generation EU under Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, Partnership on "Telecommunications of the Future" (PE00000001—Program "RESTART") under Grant CUP: J53C22003120001; and in part by Italian Ministry of Education and Research (MUR) in the Framework of the FoReLab Project (Departments of Excellence).

ABSTRACT Interest in hierarchical Software-Defined Networking (SDN) controllers is growing recently due to their ability to address the challenges associated with the SDN paradigm, such as responsiveness and scalability. This design enables efficient domain control separation, which uses different child instances to manage large-scale networks. Parent controller computational resources can be dedicated to cross-domain decision making, exploiting network views provided by its children. In this context, the correctness of the process fully relies on the network view synchronization mechanism, which should be fast and resilient. This paper presents a hybrid synchronization model combining a hierarchical design with established resilient cluster mechanisms. In this way, high-level control over large-scale networks can be guaranteed even with failures affecting every level of the management plane. Specifically, two applications are developed for the ONOS controller to share topology events using low-latency channels from child clusters to parent clusters. The performance of both applications is measured under different cluster configurations, topology sizes and number of generated topology updates. The results show that the proposed approach offers high performance while being fully compliant with the platform for which it is designed. This makes the solution easily extendable to heterogeneous child controllers. In fact, events are propagated from children to parents using gRPC, achieving end-to-end latency of less than 10ms under normal conditions and 40-60ms under high-rate event conditions. Consistency of network views is also guaranteed by strong event ordering and delivery mechanisms. Failures are handled seamlessly at both cluster levels (i.e. parent and child controllers) with a maximum synchronization delay of 2 seconds, which is quickly recovered.

INDEX TERMS Cluster, efficient, gRPC, hierarchical, hybrid, ONOS, software-defined networking, synchronization, resiliency.

I. INTRODUCTION

The Software Defined Networking (SDN) paradigm has revolutionized the way networks behave. With a centralized control plane, the network can be seamlessly and automatically configured. This increased flexibility allows network administrators to quickly adapt and meet new business requirements. At the same time, SDN controllers bring new challenges to the reliability and scalability of the architecture. Indeed, the very nature of these controllers can lead to single points of failure. For this reason, two options are typically considered in SDN architectures [1]: centralized and distributed. Centralized architectures, which use a single SDN controller instance, are generally adopted for research and education purposes. As well as being prone to failure, they have scalability issues as the network grows.

Distributed architectures solve these problems by using multiple controller instances. More specifically, a common

The associate editor coordinating the review of this manuscript and approving it for publication was Nafees Mansoor.

approach is to split large networks into multiple SDN domains, each managed by a dedicated controller instance. However, this separation should still allow the different domains to be managed in a logically centralized manner, ensuring the ability to control the entire network end-to-end. As also reported by [2], distributed architectures can be implemented using two different designs: flat and hierarchical.

The flat design interconnects domain controllers as peers, with each controller sharing its own topology view with the others. In this way, each controller manages a subset of network devices while participating in the global view. An example of this design is the *cluster* configuration, where multiple controller instances work together to manage the same network domain. However, a flat design typically requires strict synchronization between all controllers to ensure consistency of the network view and fast fail-over. It is therefore not suitable for controlling geographically distributed networks where such strict synchronization constraints cannot be met.

The hierarchical design vertically partitions the control plane to further increase scalability [3]. Each network domain is managed by a dedicated *child* controller, with each child sharing its network view only with the *parent* controller. The global network view is therefore only available to the latter. Since only children are directly connected to network devices (e.g. using the OpenFlow protocol), the parent can use its computing resources to make high-level decisions.

Even though the flat design generally increases the workload of each instance, it potentially allows the management of the entire domain from any of them. On the other hand, the hierarchical design requires a dedicated instance to implement the parent entity, but allows for role-based workload balancing. This paper describes a hybrid synchronization that combines flat and hierarchical approaches. The aim of the resulting architecture is to further increase the resilience of a child-parent design, making it compatible with a per-domain flat design (i.e. clusters). The proposed implementation, preliminarily demonstrated in [4] and exploited in [5], provides a fast and reliable gRPC-based synchronization model where child clusters reactively exchange network events with the parent cluster.

The solution is implemented and tested on top of the Open Network Operating System (ONOS) [6] SDN controller, which is widely considered to be the most reliable implementation of an open source SDN controller. It also provides a clustering design that supports technologies typically used in geographically distributed transport networks (e.g. disaggregated optical networks) [7]. Two applications, to be deployed on parent and child controllers respectively, are designed, developed and evaluated under different cluster setups and workload conditions. The applications communicate using low-latency gRPC-based channels [8] and can therefore be easily extended to support other controllers on the child side. This allows the parent controller to extend its control to domains based on different controllers and technologies (e.g. radio access networks and disaggregated optical networks). Both applications are released as open source projects, enabling further development and community collaboration [9], [10]. Therefore, the implemented system enables a hierarchical design which is capable of interconnecting heterogeneous domains while using the resiliency of the cluster design. More specifically, this work provides:

- A reference synchronization model for resilient large-scale SDN networks, mixing cluster and hierarchical designs.
- A topology sharing model based on gRPC, propagating events from multi-instance children to multi-instance parents.
- An open-source implementation of the presented system, developed for the ONOS SDN controller.
- Ad-hoc benchmarks for the proposed solution and similar, testing the resiliency and the scalability of an SDN architecture in terms of topology view synchronization.

The paper is organized as follows. Section II provides an overview of the existing cluster, flat and hierarchical designs for SDN controllers, and then focuses on the ONOS controller architecture. Then, Section III presents the proposed solution, showing details of the implemented applications and their behavior. Section IV describes the experimental evaluation, where several tests on the scalability and resilience of the system are performed. Finally, in Section V, conclusions are drawn, highlighting the advantages of the presented solution and considerations for future work.

II. BACKGROUND AND RELATED WORKS

As presented in [1], several SDN controllers have been proposed in recent years, each with its own features and performance. Among those with a flat design, ONIX [12] and Hyperflow [13] are the first to introduce this type of architecture. ONIX instances use a distributed store to replicate the Network Information Base (NIB) between them. It implements two consistency options (strong/eventual), thus optimizing performance according to the data to be stored. On the other hand, Hyperflow uses a publish-subscribe system to share network views using topology events. In this way, instances recombine and align their views by reorganizing the events generated in each local domain, providing eventual consistency.

OpenDayLight (ODL) [14] and ONOS are currently the most widely used open-source SDN controllers supporting a multi-instance, flat design architecture. They are both production oriented and therefore support a wide range of network devices and protocols (e.g. NETCONF, OpenFlow, P4). ODL implements a strong consistency model based on the RAFT [15] consensus protocol to share information across the cluster. ONOS combines a distributed database and a gossip protocol to achieve both strong and weak consistency models, ensuring performance and coherent network views. Specifically, local topology events are optimistically replicated to other instances, and occasionally aligned with the

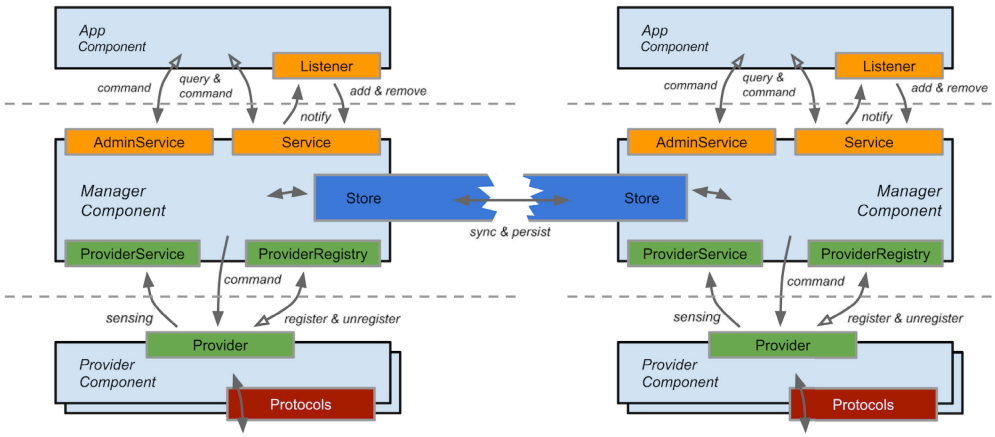

FIGURE 1. ONOS SB-NB subsystems relationship, as in ONOS documentation [11].

gossip protocol. This allows failure scenarios to be handled smoothly at both data and control plane levels.

Strong consistency models provide up-to-date data across all nodes, but inevitably have high latency and poor scalability. For this reason, the implementation of mixed consistency models supported by alignment mechanisms is a good compromise. However, even in this case, strict latency requirements must be guaranteed to ensure good performance and correct behaviour within the cluster, making the clustering solution inapplicable if the controller instances need to be geographically distributed (e.g. in the case of wide area networks).

To this end, a specific application is developed for ONOS that supports a flat architecture and relaxes the low latency requirements, i.e. ICONA (Inter Cluster ONOS Network Application) [16], allowing remote and separate ONOS clusters to share information about their managed networks using the Hazelcast [17] publish-subscribe platform. It also implements configuration policies between clusters, providing full control over services and events in different domains. Each ONOS cluster running ICONA shares the local topology with remote clusters and receives abstracted network views back from the others. In this way, ICONA preserves the consistency performance of ONOS clusters while extending their capabilities to the wide area network scenario.

On the other hand, regarding hierarchical design, Kandoo [18] implements a two layer structure where selected network events are propagated from child controllers to the parent. However, fault tolerance is not implemented in this project, so consistency is not fully addressed.

A similar concept is used by Orion [19], Google's SDN controller. It uses a hierarchy of instances that communicate using RPC and exchange protobuf messages [20]. Like Kandoo, it prioritises the isolation of failures at both the data and control planes by assigning a single Orion instance to each domain. Conversely, Google's private WAN intra-domain SDN controller, called B4 [21], uses the same two-tier architecture but focuses on robustness. It implements a combination of ONIX-based controllers for the lower-tier sites and a global SDN gateway/TE server at the root, implementing fault-tolerance mechanisms at both layers. More specifically, it deploys a cluster at each site, managed by Paxos [22], which selects primary and backup instances. At the global level, the SDN gateway and TE server are replicated across the WAN sites to ensure availability.

In these last years, a new SDN controller has emerged as an open source project sponsored by ETSI, namely Teraflow (TFS) [23]. TFS differs from previously presented projects in that it is based on micro-services that delegate the scalability and reliability of the controller to the underlying orchestration tool (i.e., Kubernetes [24]). Preliminary work presented in [25] demonstrated a hierarchical deployment of TFS, with a parent instance acting as a network orchestrator and optical network controller, and a child instance acting as an IP layer controller. The reported results do not include an analysis of the achieved network view consistency. However, this work, which involves major telecom operators and vendors, reflects the significant interest in providing SDN controllers with a hierarchical design that introduces a high degree of flexibility, enabling the control of domains using different technologies. Another paper demonstrating the usefulness of hierarchical design is [26], which shows how such a design can offer advantages in terms of end-to-end path computation time compared to a flat design. Therefore, these designs can be beneficial for scaling heterogeneous and complex scenarios, such the one presented in [27].

This paper proposes a *hybrid* synchronization solution that enables a two-level hierarchical architecture between ONOS clusters, which in turn use a flat design for local redundancy. In this way, within each domain, scalability and reliability are automatically guaranteed by the ONOS multi-instance cluster architecture, while at the network-wide level these features are guaranteed by exploiting the hierarchical structure, which also introduces the flexibility required to control

heterogeneous wide area networks. Since communication between parent and child controllers is implemented using gRPC and relies on easily extensible protocol buffers/JSON to serialize structured data, our work also provides a solid foundation for connecting non-ONOS-based child controllers to our ONOS-based parent controller.

A. ONOS CORE AND CLUSTER DESIGN

This work uses ONOS as a controller as it already provides a reliable multi-instance cluster architecture and several tools necessary for the development of the hybrid hierarchical architecture (e.g. defined network device description in protobuf). Furthermore, compared to other SDN controllers, ONOS achieves the best performance in terms of message processing, network topology change detection and reactive path provisioning [28]. Finally, it is well established in the open source SDN community. It emerged from an Open Networking Foundation (ONF) initiative and is now part of two Linux Foundation projects, AETHER [29] and P4 [30].

Fig. 1 shows the ONOS architecture, which natively supports multi-instance clusters. A brief description of this architecture is given below, as it forms the basis of our development work and helps to better understand our contribution. Specifically, the inter-tier boundaries in Fig. 1 represent the northbound and southbound APIs. At the very bottom, a *Provider* component implements a protocol (e.g. OpenFlow) used by the controller to manage the specific network devices. This *Provider* is registered in an important core component of ONOS called the *Manager*. The role of the *Manager* is to receive information from one of the various Providers and pass it on to Services (e.g. the *Topology* Service) and Applications. In fact, an application can receive updates from a specific service by implementing a dedicated listener. The *Manager* also interacts with the *Store* component. The *Store* indexes all relevant information and makes it persistent by writing it to Atomix [31], a distributed and scalable database. Sharing the same Atomix between multiple ONOS instances is a way of sharing the network state and is therefore necessary for implementing a cluster. Furthermore, this framework uses a variety of replication protocols (e.g. RAFT, Gossip) to share state across multiple nodes, allowing it to be deployed as a cluster. In fact, an ONOS cluster running on top of an Atomix cluster achieves the best results in terms of both resilience and scalability, as its state is both partitioned and replicated across more than one Atomix node.

III. PROPOSED HYBRID-HIERARCHICAL SYNCHRONIZATION MODEL

This section provides a detailed description of the proposed hierarchical synchronization solution applied to SDN controller clusters. The key idea is to propagate topology events from child controllers to a parent controller while exploiting the resiliency of the cluster configuration.

In this work, the proposed approach is applied to ONOS through the development of specific Java-based applications. These applications make use of a set of APIs exposed by ONOS, allowing their coordination in the distributed environment. By taking full advantage of this configuration, it is then possible to increase the resilience and possibly the scalability of the child-parent synchronization. The basic structure of the applications is based on the open source ONOS application Kafka-Integration [32], which is extended and optimized for the proposed scenario.

Two applications are developed: *Hierarchical Sync Child (HSC)*, running on the ONOS child clusters, and *Hierarchical Sync Parent (HSP)*, running on the parent cluster. These applications, which are specifically designed to be resilient, communicate via gRPC channels. The choice of gRPC is driven by the need to share highly structured data in an efficient way, such as ONOS Java objects. In fact, the system developed works by reactively forwarding network event objects. These events, generated on the children, are seamlessly received by the parent through gRPC channels embedded in the applications. In this way, the parent is able to maintain the global topology view with updates coming from its children.

This mechanism allows each child controller to focus on managing the underlying devices by delegating inter-domain functions to the parent controller. This approach enables a number of further optimizations to be applied to the proposed scenario. For example, each child could specialize in a particular domain (e.g. packet or optical) or protocol, thus reducing the overall workload by concentrating its functionalities. Furthermore, each child controller can choose the level of network aggregation used to abstract its domain to the parent controller, for example the full topology can be exported, or it can be abstracted as a limited number of nodes (e.g. edge nodes), or even abstracted as a single node to preserve domain confidentiality [33].

On the other hand, the parent is relieved of the role of managing the devices and can thus exploit its computational resources by making cross-domain decisions (e.g. inter-domain path computation). Moreover, the proposed approach allows the development of a new set of applications at the parent site, taking advantage of the global network view. This is possible to the extent that the system is designed to be fully transparent and compliant with the ONOS architecture.

The hierarchical architecture allows easy integration of technologically heterogeneous domains, as any device supported by a child controller can be easily imported into the parent view. For example, at the time of writing, the architecture is tested with Open vSwitch devices (controlled using Openflow protocol), Reconfigurable Optical Add-Drop Multiplexers (ROADMs) and transponders (NETCONF protocol), and BMv2 switches (P4Runtime through gRPC). In this context, a variety of device types can be considered at the parent level, but communication protocol support to such devices is only required at the child controllers, facilitating the integration of heterogeneous domains.

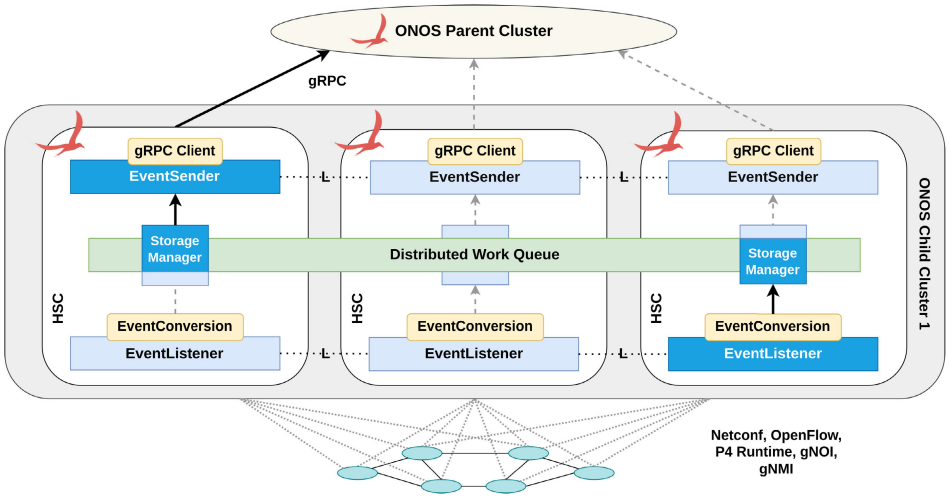

FIGURE 2. Hierarchical Sync Child application logic, deployed on a 3-instance ONOS child cluster.

Therefore, the resulting architecture from the proposed synchronization model allow a modular management for large and heterogeneous networks. Indeed, this work serves as a starting point for the management of complex networks by increasing both the number of child clusters and the number of hierarchies. However, this approach can add complexity to the system and waste many resources just by communicating between different levels. The flexibility of the data model in the proposed solution also enables the transmission of abstracted/aggregated topology. Combining the synchronization solution with these abstraction strategies (e.g. limiting the propagation of certain events or aggregating them) at child side, allows to limit the complexity of the management of the parent network.

A. HIERARCHICAL SYNC CHILD

The Hierarchical Sync Child application has two main functions:

- It listens for network events of its local cluster and temporarily store them;
- It retrieves the events from the store and send them to the parent via gRPC.

As the application is intended to be installed on a cluster, these two functionalities are programmed to behave in a coordinated way between ONOS instances. More specifically, ONOS exposes some APIs that allow consensus-based leadership to be run on top of an Atomix distributed primitive called *topic*. This means that the same functionality, replicated on each controller instance application, can run for leadership of the same topic. In this way, only one of them becomes the leader, and thus actually runs it. The other candidates can instead be programmed to take over if the previous one withdraws for any reason (e.g. failure). In this way, all application functions continue working as long as there is at least one ONOS child running in the cluster.

Fig. 2 shows the logical structure of the HSC application, replicated on three ONOS instances that form the child cluster. Each instance runs the services that implement the above functionality (primary services in blue and secondary services in yellow). As an example of the management concept, the primary services in darker blue represent the *active* services, which are therefore selected with a leadership process (represented in the figure by the dotted line with the *"L"* between services of the same level).

At the bottom, the very first service is the *EventListener*. Its role is to capture all topology events that occur on the managed devices. In ONOS, events are generated at the instance that has mastership over the device that generates them, and then dispatched to the others. For this reason, only one *EventListener* needs to be active for the entire cluster to avoid redundant events. There is therefore a leadership process between these replicated services. The events then pass through the *EventConversion* service: here they are converted into Proto objects using the classes and methods implemented for the μONOS project. In this way, the application is able to seamlessly adapt to possible changes or additions to the ONOS event objects by the open source community. The converted events are then encapsulated into an intermediate format called *Encapsulated ONOS Event (EOE)*. The EOE is the actual entity that travels from child to parent instances and can be used to carry additional information across the different services.

The active *EventListener* instance (i.e. the one on the right in the Fig. 2) then calls the *StorageManager* service after the event is converted. Its purpose is to interact with the Distributed Work Queue that resides in the Atomix database. More specifically, the lower part of the *StorageManager* is used to put EOE items into this queue, while the upper part is used to retrieve them. This approach has many advantages, but the main one is that it allows the upper/lower services to be decoupled. As shown in the figure, even the upper service

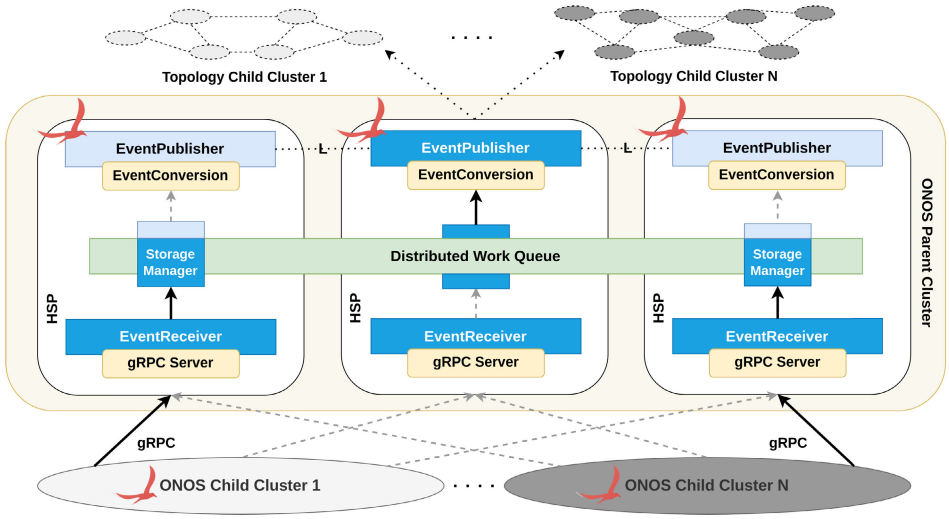

FIGURE 3. Hierarchical Sync Parent application logic, deployed on a 3-instance ONOS parent cluster with N children.

EventSender uses a leadership negotiation that is independent of that of *EventListener*. This means that these two services could be active in different ONOS instances, thus balancing the application workload within the cluster. The shared queue is therefore necessary to move EOEs from *EventListeners* to *EventSenders* as all cluster controllers share access to it. At the same time, it allows services to renegotiate leadership (e.g. due to an instance failure), so that service leadership can be passed from one instance to another without interruption.

On the upper part, the *EventSender* leader (i.e., the one on the left in Fig. 2) implements a listener for the Atomix queue using the *StorageManager*: as soon as a new EOE becomes available, it is consumed from the queue and processed by the service. If the service is somehow unable to complete the processing, the current EOE is returned to the queue. This Atomix feature guarantees that all items popped from the queue are actually processed and not lost due to some failure. The *EventSender* service handles this "completed" feedback part towards the *StorageManager*, and implements some support mechanisms for the *gRPC Client* service. It is configured to be aware of all the IP addresses of the parent cluster instances: this allows some EOEs to be quickly redirected to another controller if something goes wrong. In fact, as will be discussed later, all HSP applications are equipped with an available gRPC server. This means that if a parent gRPC server instance becomes unreachable or the latency to send a single EOE exceeds 30ms, the *EventSender* reconfigures the gRPC client to a different parent instance. In addition, this service initially takes a random selection from the configured parent IPs. This helps to avoid multiple children clustering to send EOEs to the same gRPC server among those available at a parent. If the EOE is correctly acknowledged by the parent gRPC server, it considers it processed. Otherwise it retries until it succeeds.

The entire application enforces a transparent event forwarding, preserving the order in which events are generated. It also takes advantage of the cluster configuration to provide resilience. Therefore, by increasing the number of instances in the cluster, the management plane and the proposed system itself can tolerate more failures at once.

B. GRPC COMMUNICATION AND MANAGED EVENTS

Before introducing the parent application, there are some features of the implemented gRPC channels and managed events that require further discussion. Applications use synchronous and unary Remote Procedure Calls (RPCs): this means that the client (child) sends a single request and waits for the server (parent) to reply with a blocking call. This enforces event delivery and ordering while sacrificing some performance in high load scenarios. The message structure is defined as it follows:

- EOE: the Protobuf event object;
- ClusterID: a unique identifier for each child cluster.

Thanks to the ClusterID, the parent application is able to group events coming from the same child.

Furthermore, this work only handles topology events, but it can be further extended to support core level information (e.g. intents, applications). All events related to up/down/updates of ports, devices or links are specifically managed, except those related to port statistics. These are handled at the *EventListener* level, which can easily filter or abstract most of them to achieve higher system scalability. It is important to note that using gRPC with protocol buffers enforces a rigid data structure. This can make data object modelling more difficult and reduce flexibility compared to systems using formats such as JSON. At the same time, this models enable several functionality which are implemented by default (e.g. data type checking).

C. HIERARCHICAL SYNC PARENT

On a parent cluster, the Hierarchical Sync Parent application implements the following functions:

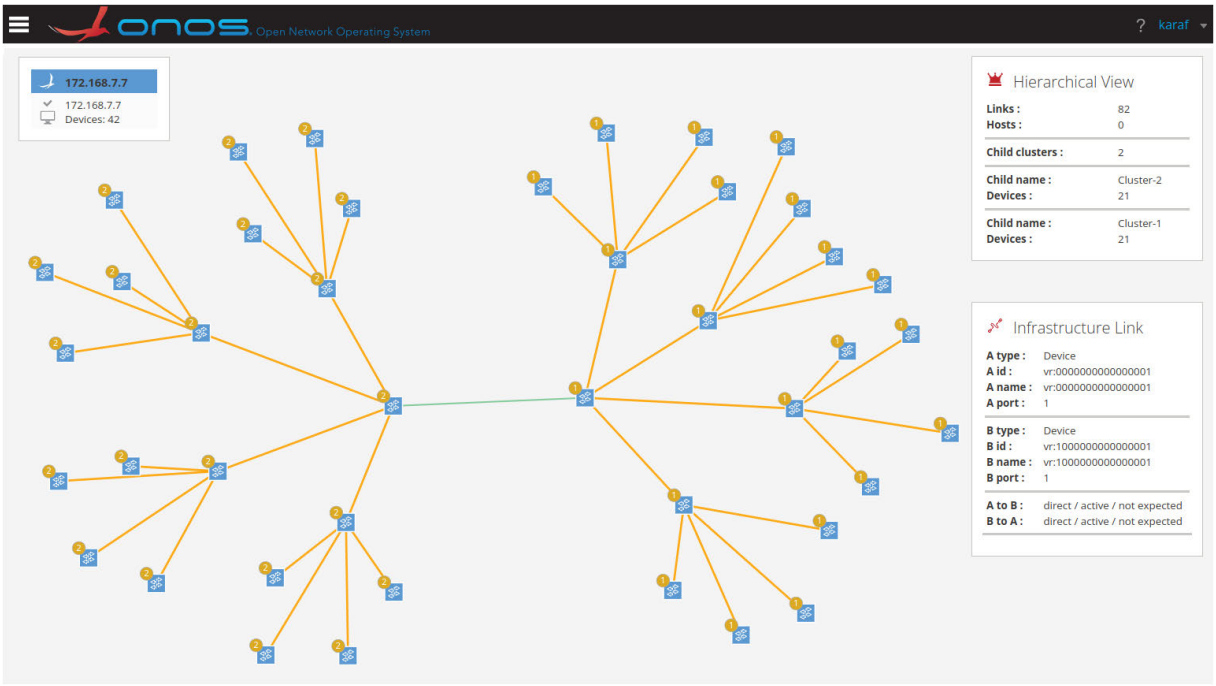

FIGURE 4. Hierarchical Sync Parent application GUI, showing the view of a single instance ONOS cluster with two imported and interconnected topologies, each managed by two child clusters.

- It receives events via gRPC from the child clusters and temporarily stores them;
- It retrieves events from the store and publishes them to its local view.

Figure 3 shows the application logic on a 3-instance ONOS parent cluster, synchronizing events from N different children. The bottom service is represented by the *EventReceiver*, which implements a *gRPC Server*. As previously anticipated, HSC can autonomously switch between parent IPs to avoid possible failures or major delays. For this reason, this service must be running in all application instances. In fact, as shown in the figure, no leadership process is configured between *EventReceivers*, and more than one is collecting data from child clusters (all are active, so coloured dark blue).

The EOE, once received by the *gRPC Server* service, is placed in a dedicated Distributed Work Queue. This queue has the same purpose as the one in the child application, but belongs to the Atomix parent cluster. A similar *StorageManager* service is therefore used to interact with it.

On the upper part, a leadership mechanism is implemented between the *EventPublisher* services. As for the HSC, the leader instance (i.e. the central one in the Figure 3) activates a listener via *StorageManager* to retrieve and process new EOEs added to the queue. Finally, the *EventConversion* service converts the encapsulated events into ONOS event objects using the reverse conversion functions of μONOS.

Finally, the events are pushed to the local topology view of the parent cluster, following the order of generation of the children. This process actually requires a set of mechanisms to allow the imported views to be handled correctly. Events cannot simply be pushed into the view, but must be "translated" into actual actions (e.g. a *DeviceAdded* update requires a new device instance to be inserted into the view) using a combination of *AdminServices* and *Providers* services (see Figure 1).

As devices and links cannot be directly accessed by the parent instances, they can only be recombined as an abstract representation using the information carried in the events (e.g. device properties, link type). This is made possible by the AdminServices component: it allows elements to be manually inserted into the topology view while preserving core ONOS functionality (e.g. intra-cluster event propagation).

A custom provider is then required to prevent ONOS from managing the devices as real devices. Since providers implement management protocols and thus implement keep-alive mechanisms, a real provider would mark the abstract devices as unreachable after a few seconds. Instead, by implementing a custom provider, each child device can be handled without problems: no other provider can manage them, and so only imported events can affect the parent view involved. Furthermore, the implemented provider also avoids possible mastership issues between cluster instances on the imported child devices. In fact, the instance with the HSP application that has the leadership for the *EventPublisher* service is forced to be the only one in charge for the considered devices. As a consequence of these mechanisms, there are no compatibility issues on the child managed topologies: all SDN domain types (i.e. optical or packet) natively supported by ONOS can be imported at the parent site without any loss of information.

In addition, since the aim of this work is to allow the parent clusters to deploy applications that take advantage of the imported views, an additional concept from ONOS is used: *Regions*. They enable the logical separation of elements within the same topology: in this case, a new region is implemented for each child attached to the parent. Using the ClusterID embedded in each received gRPC message, the HSP sends the event to a different region. In this way, new parent applications can implement listeners for one or more regions, and thus react to events that occur only on specific child views.

Finally, a GUI overlay is available at the parent application. This overlay uses regions to graphically display useful information in the ONOS topology view. For each child (or region) it shows the number of devices or links currently available and their original information. It is also able to highlight network elements belonging to the same child by applying a label showing their ClusterID. Fig. 4 shows a screenshot of the HSP GUI in a single instance parent cluster connected to two child clusters (i.e.; Cluster-2 on the left topology page and Cluster-1 elements on the right). You can also see that an inter-child link is highlighted in green. This is done automatically by the HSP application, which colours all inter-domain elements to emphasize their presence.

This is a crucial aspect of the hierarchical architecture: the parent rebuilds and aggregates information that is partially available in the children. Specifically, the link in the figure is not shown in any child view, as the two switches at the ends of the link do not belong to the same domain/cluster. Instead, the parent is able to reconstruct it because it has knowledge of both, even if they are in different regions. Therefore, the HSP application uses the ONOS framework itself natively to achieve this result.

As for the HSC application, the application is meant to run on top of a cluster. The leadership processes across same applications components enforce a resilient behaviour in case of a failure in the parent cluster, thus avoiding a single point of failure scenarios. Additional cluster instances can be added to increase the number of simultaneous failures.

D. SYNC BEHAVIOUR

The synchronization system performs a continuous exchange of events from the child topologies to the parent topology. It therefore requires an initial alignment between these views. This behaviour is implemented at the HSC: when the application is installed on the child cluster, it artificially generates *Added* events for all the elements currently present in the network, as if they are connected to the controller. This allows the HSP to retrieve the current view, which can now be updated according to the real topology events. Implicitly, this means that the parent application must be the first to run on the scenario. After that, new children can be added in a plug-and-play fashion. Once the HSP is uninstalled, imported views are automatically deleted. If this is the case, all running HSCs continue to enqueue events from their topologies while active-waiting for an HSP to receive them.

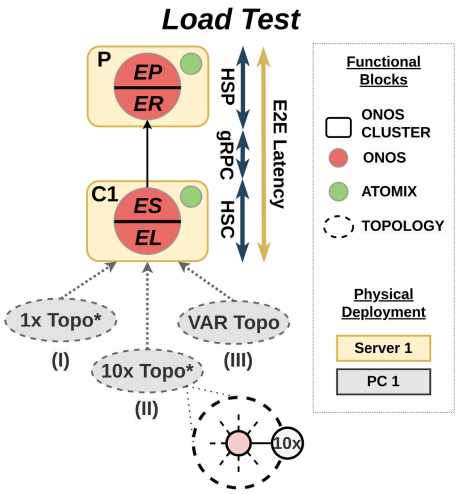

FIGURE 5. Load Test logical and physical deployment. Two single-instance ONOS clusters, one child and one parent, are used to evaluate the impact on the system of increasing the number of events and varying the workload of the ONOS child instance. Topologies I and II are used to generate different numbers of events, while the child workload is changed by increasing the size of Topology III.

IV. EXPERIMENTAL EVALUATION

This section presents the scenarios and the results of the tests performed to evaluate the overall correctness, scalability and resilience of the proposed solution. Event processing latency, defined as the time required by each application to process an event, is used as the Key Performance Indicator (KPI) across the different scenarios.

The main services in each application are programmed to insert a timestamp into each EOE carried at the following moments:

- Event captured by HSC *EventListener*;
- Event successfully sent by HSC *EventSender*;
- Event received by HSP *EventReceiver*;
- Event published to parent view by HSP *EventPublisher*.

In this way, all timestamps are available at the parent, allowing the different latency contributions of HSP and HSC processing, as well as gRPC transmission latency, to be estimated when possible.

A. SCENARIO

Figures 5, 7 and 9 describe the logical organization of the tests performed. More specifically, these figures show how the ONOS clusters and their respective applications are configured and installed on five different machines. Up to three equivalent servers (Server 1-3) are dedicated to the deployment of the controller instances, each equipped with Intel Xeon E5-2643v3, 6-core 3.40 GHz clock, 32 GB RAM and Ubuntu 18.04. They are physically connected via a Gigabit connection through a physical switch. On the other side, two PCs with Ubuntu 18.04, Intel i5, 16GB RAM and a Gigabit interface are used to emulate the network topologies composed of OpenFlow switches using Mininet [34].

Each ONOS cluster is configured to be deployed using Docker containers [35], with a variable number of ONOS (in

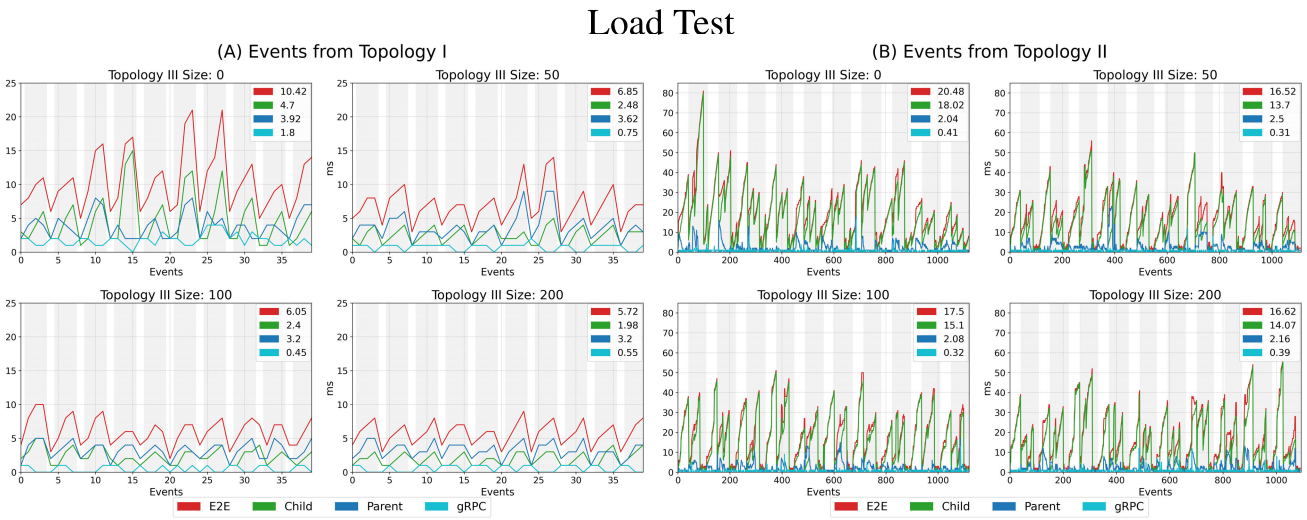

FIGURE 6. Load Test results showing the end-to-end and per-app processing latency for each event generated within Topology I (A) and Topology II (B) and propagated to the parent. Events are generated by repeatedly turning the central switch of the affected topology off (white background) and on (gray background) while increasing the size of Topology III.

red) and Atomix instances (in green), and that are part of the same L2 domain. Cluster deployment follows a simple rule: same cluster instances belong to the same machine. In this way, even when using multi-instance clusters, the correctness of event latency can be guaranteed at least per application. In fact, despite the different tests, it is always possible to retrieve the latency contributions for each event traversing HSC and HSP applications respectively.

Furthermore, the ONOS configuration changes according to the type of cluster it belongs to: child clusters require additional applications to run in order to manage the OpenFlow switches (e.g. openflow, lldpprovider), while parent clusters do not. In HSC and HSP, the leadership of the main functions is controlled to create load-balanced scenarios. For this reason, the ONOS instances in the figures indicate how the running functions are distributed according to the cluster instance type:

- HSC's *EventListener* (EL in the figures);
- HSC's *EventSender* (ES in the figures);
- HSP's *EventReceiver* (ER in the figures);
- HSP's *EventPublisher* (EP in the figures).

This only applies to multi-instance clusters, as single-instance clusters have to implement all the functions of the installed HSC/HSP applications.

Scalability is assessed by generating different network sizes, producing a variable number of topology events. Mininet is used to generate custom star topologies only: by switching the central switches up and down, ONOS itself generates a burst of events, mainly related to links and port status, coming from the edge switches. In this way, it is possible to place a much higher load on the system compared to the Mininet network generation process. Topology producing events are shown in the figures with an asterisk next to their name.

B. LOAD TEST

The first test (Fig. 5) considers a simple scenario with a single-instance child cluster (*C1* in Figure) and a single-instance parent cluster (*P* in Figure). Both use the same configuration, with one ONOS instance and one Atomix instance. The child and parent controllers run on the same machine (i.e. Server 1) to keep them synchronized. The aim of this test is to verify how the two applications perform with a variable number of events and how this is affected by the overall workload of ONOS. For this reason, a specific network consisting of three subtopologies is attached to the child:

- I: Single switch topology;
- II: 10x star topology (1+10 switches, 10×2 links);
- III: Variable-size star topology.

Topology I and II are used to generate a different number of events while increasing the workload on ONOS with additional OpenFlow devices (the devices included in Topology III). The test is therefore split into two parts:

- (A): turn on and off the switch of Topology I;
- (B): turn on and off the central switch of Topology II.

In both cases, the experiment is performed under 4 different workload conditions, mainly determined by the size of Topology III (0, 50, 100, 200 switches). Each experiment is repeated 10 times for each workload scenario, with an intermediate delay between the up and down phases. All other events related to the setup of the topologies are filtered out.

The results of this test are presented in Fig. 6, showing the latency affecting the event propagation from child to parent clusters according to the scenario involved. In this particular case, it is possible to obtain some additional information other than the application latencies (child/HSC in green, parent/HSP in dark blue) for each event. As both child and parent run on the same server, the timestamps

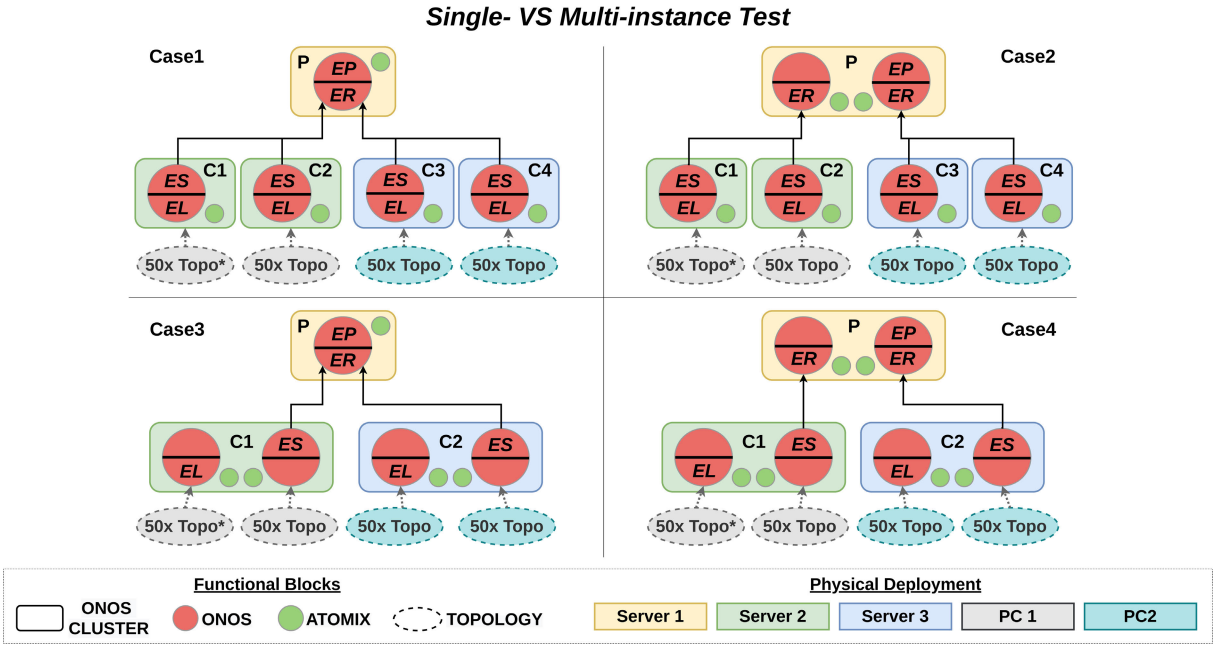

FIGURE 7. Single- VS Multi-Instance Test logical scheme, along with physical deployment. Four different 50x star topologies, generating a fixed number of events, are used to evaluate HSC performance within a four single-instance child cluster setup (cases 1-2) against a two dual-instance one (cases 3-4). The same is done for the HSP application, but comparing a single-instance parent setup (cases 1-3) against a dual-instance setup (cases 2-4).

of the different applications are coherent. This also allows us to estimate the latency of the gRPC communication (in light blue) and therefore the overall end-to-end (E2E) communication (in red). The average latencies for each component are also shown in the upper right corner of each plot. In addition, events generated by the up and down phases of the central switch are distinguished in both plots by a different background color: the white background represents the down phases, while the grey background represents the up phases.

For the Load Test (A) (Fig. 6.A), the Topology I generates events. As this is a single switch topology, the down phase generates a single event (device down), while the up phase generates 3 events (device up and port updates). After 10 repetitions, the total number of events is 40. The latency peaks visible in figure are mainly from the up phases, with an overall maximum of 21ms. Although the average latencies of the parent application are fairly balanced over the tests, those of the child application appear to be inversely proportional to the increase in workload. In fact, as the size of the Topology III increases, latencies of the child applications decrease. This could be due to some optimizations that ONOS may apply to manage a larger set of devices. At the same time, some services may also run in steady state condition, avoiding sleeps and thus reducing the processing time of the ONOS instance.

The Load Test (B) (Fig. 6.B) shows a general increase in the average latencies. This is due to the number of events processed, which is drastically higher compared to (A). Indeed, the total number of events, now generated by Topology II, from down phases (40 each, including link down and port updates) and up phases (approximately 75 events each) is of almost 1150. In these cases, both up and down phases originates peaks, due to the burst of events generated. The increasing latency that leads to each peak is a symptom of a queuing delay, mainly determined by the ONOS child instance itself handling the new topology updates. Indeed, it is possible to see that the E2E latency is quite close to the HSC processing latency. On the other hand, the HSP application shows negligible values: a small peak is visible at the beginning of almost every phase, but it quickly disappears as the HSC reduces its transmission rate. The child application again confirms that it is not affected by the increase in Topology III size. Although there is a higher number of events compared to the previous case (Load Test A), the highest E2E latency is still recorded when the Topology III size is 0. For bigger topology sizes, the latency shows slightly lower values compared to size 0. This may be further evidence of some existing mechanisms that optimize processing of ONOS as the number of devices handled increases. At the same time, there is not a smooth decreasing trend as before: the average E2E latency with size 100 is slightly higher compared to size 50. For this reason, we may conclude that this optimization does not offer consistent benefits with topology III with size higher than 50. Indeed, the biggest latency gap can be found among size 0 and size 50 in both test A and B. It is still worth to mention that the worst peak in E2E is less than 100ms (around 80ms) with Topology III of size 0, with an average of 21ms and mainly dictated by the child application.

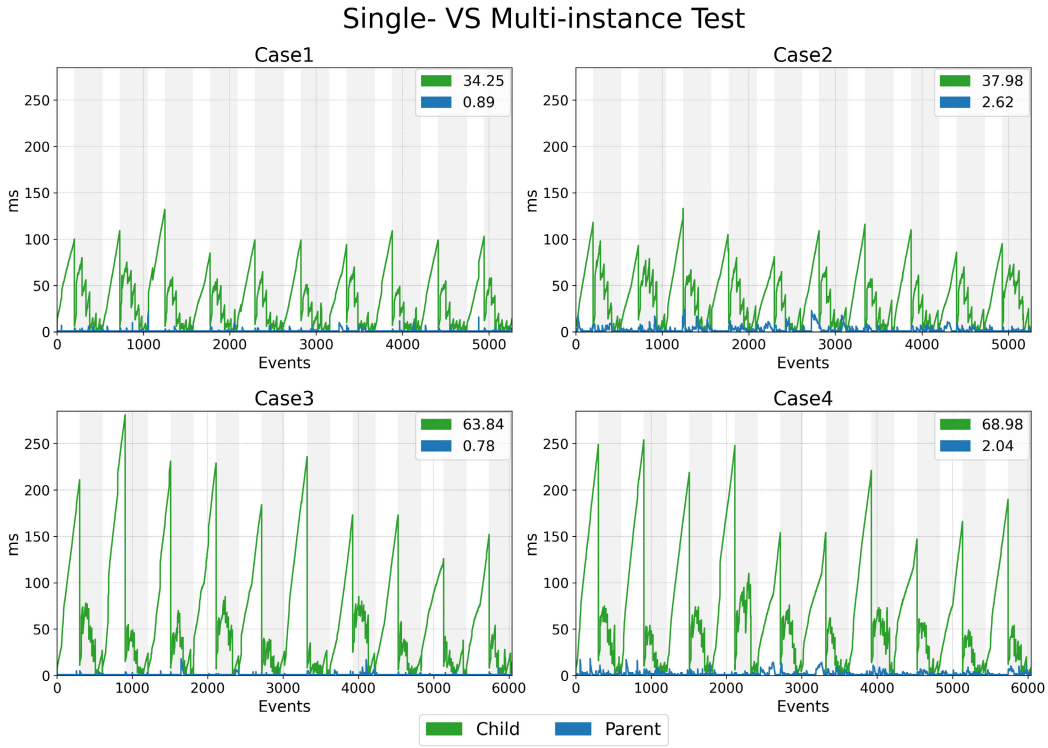

FIGURE 8. Single- VS Multi-instance Test results, showing per-application processing latency for each event generated by a 50x star topology and propagated to the parent. An equal number of events are generated by repeatedly turning the central switch of the involved topology down (white background) and up (grey background) while changing the configuration of the child and parent clusters. The performance of single and multi-instance application deployment is therefore compared for both HSC (cases 1-2 vs cases 3-4) and HSP (cases 1-3 vs cases 2-4).

C. SINGLE- VS MULTI-INSTANCE TEST

The second test compares the performance of applications running on single-instance clusters against multi-instance clusters.

The four different configurations shown in Figure 7, involving both child and parent controllers, are used while managing the same network topology (e.g. handling the same number of events). Each application is installed on two different configurations:

- A single-instance ONOS cluster, with one Atomix instance;
- A dual-instance ONOS cluster, with two Atomix instances.

According to Fig. 7, the parent configuration changes horizontally (cases 1-3 vs cases 2-4) while the child changes vertically (1-2 vs 3-4). To evaluate the performance of the child application, four ONOS instances are used, each equally loaded with the same 50 star topology size. Therefore, depending on the specific scenario, they are properly rearranged into 4 single-instance clusters (C1-C4 in Figure, with 4 separate HSC applications) or 2 dual-instance clusters (C1 and C2 in Figure, with 2 replicated HSC applications) with the same number of Atomix instances per server. On the other hand, the load on the parent cluster (P in Figure) is kept constant: regardless of the single or multi-instance configuration, it is responsible for managing the global 200-switch topology view imported from the children. In addition, whatever application is installed on a multi-instance cluster, its functionalities are forced to be distributed across its cluster instances.

The topology attached to the cluster C1 is the one that generates the events in the different cases. Therefore, the results collected refer to the performance of the applications running on the child cluster C1 and the parent cluster P, both in single or multi-instance configuration. The children running on the Server 3 are only used to load the system with additional topologies to be managed.

The results of this test are shown in Figure 8. In this case, the child and parent clusters run on two different servers, so measuring gRPC transmission and E2E latency requires very accurate clock synchronization. For this reason, only the latency contributions of the child and parent applications are shown for each event. Even in this case, events are divided into up (grey background) and down (white background) phases, which are generated by switching the central switch of the topology up and down.

It is possible to see that the latency measured on the parent application is still negligible compared to that measured on the child controllers. However, in this specific case, the

child latency peaks from the down phases show a significant increase with respect to the up phases. This is due to the 50x star topology used for this test, resulting in bursts of 200 and 325 events for down and up phases respectively. Therefore, down phases appear to have a greater impact on the instance workload for higher event rates.

Let's start by comparing the performance of the child application running on a single-instance and on a two-instance cluster. So we need to compare the cases vertically (Case1 vs Case3, Case2 vs Case4) for the same parent configuration. In the first two scenarios (1-2), cluster C1 runs both HSC components on its single instance. In the other two (3-4), C1 instead balances the child application components between its two instances. The results show that the event processing latency increases by about 30ms in both comparisons, from an average latency of ±36ms in cases 1-2 to ±65ms in cases 3-4. This may be due to the overhead introduced by the HSC components sharing events between distinguished instances. In fact, even though the child application workload is distributed, the performance is slightly worse than the single instance scenarios. On the other hand, this configuration guarantees the isolation of component failures, which can be handled better than in the single instance cases. It is also important to note that in the multi-instance cases (3-4), more events are generated by ONOS (±800), evenly distributed in the up/down phases. This is due to some redundant events that this configuration may need to share in order to ensure coherent behaviour across its instances. Indeed, the correctness of the parent view is still achieved.

On the other hand, the performance of the parent application in a single/multi-instance scenario can also be evaluated by comparing cases horizontally (Case1 vs Case2, Case3 vs Case4) for the same child configuration. In Case1 and Case3, HSP runs on a single-instance ONOS cluster, while in Case2 and Case4 it runs on a two-instance cluster. In the latter cases, the application components are still distributed among the cluster instances.

Although the latency at the parent level is negligible with respect to HSC (±1.5ms on average), a similar trend of performance degradation is slightly visible in the multi-instance scenario (latency more than doubled, ±2ms difference from cases 1-3 to cases 2-4). Since HSC and HSP applications share the same logical architecture, it is clear that the overhead of the multi-instance setup is affected by the instance workload. Indeed, this overhead is much more pronounced on HSC than on HSP, since the former is burdened by topology management.

Finally, from the HSP point of view, there are no significant differences when changing the HSC configuration (cases 1-2 vs cases 3-4) and vice versa for the HSC (cases 1-3 vs cases 2-4).

D. FAILURE TEST

The third test aims to verify the resiliency and correctness of the proposed implementation. More specifically, it measures the impact of a failure in terms of event processing latency on each application component, both at the child and parent level. To this end, four different scenarios are considered, as shown in Figure 9.

Each case has the same initial condition, with a 50x star topology connected to a dual-instance child cluster C1 running HSC components in a distributed fashion. The Mininet switches are equally balanced between the two instances from a mastership point of view. The parent cluster P shares the same configuration as the child, so each instance runs one HSP component. The topology then generates a constant stream of events. This is achieved by disabling one of the 100 interfaces on the central switch every 5ms. Halfway through the script, after the 50th interface is disabled, an ONOS instance is killed. This forces the remaining instance in the cluster to manage the dead functionality through the leadership mechanism.

The four cases considered cover all possible combinations of application functionality running on the killed instance:

- Case1: child instance failure, running HSC's *EventListener*;
- Case2: child instance failure, which runs HSC's *EventSender*;
- Case3: parent instance failure, handling HSP's used *EventReceiver*;
- Case4: parent instance failure, which executes HSP's *EventPublisher*;

In addition, a special Case0 with no failure is considered and used as a baseline. In this way it is possible to determine how much the failure of a particular application component affects the performance of the system.

The results of this test are shown in Figure 10. Again, only the latency contributions of the parent and child applications can be measured. In Case0 it is possible to see how the 400 events are processed with very low latency from both sides. In fact, the average latency is less than 3ms for both HSC and HSP, with no samples exceeding 10ms. Case1 takes into account the failure of the child instance managing the *EventListener*. The event latency reflects that of Case0 for both parent and child, but then increases for the latter after the failure (200th event). It starts with an initial peak, but then increases up to 75ms. This may be due to the fact that half of the topology devices is handed over to the new instance, which now has to manage them. Indeed, after a failure, events are received by the new instance, which is now in charge of the switches. It is also important to note that fewer events are processed during this transition (80 less, pink area in the graph). The inconsistency is not due to the *EventListener* leadership process, but to the child itself. Although ONOS handles this type of scenario autonomously by periodically checking for inconsistencies with each device. Once found, events are generated and the state becomes consistent.

Case2 takes into account the failure of the child instance managing the *EventSender* component. In this scenario, the delay due to the renegotiation of the leadership becomes more evident. As shown in the Figure, there is an immediate peak

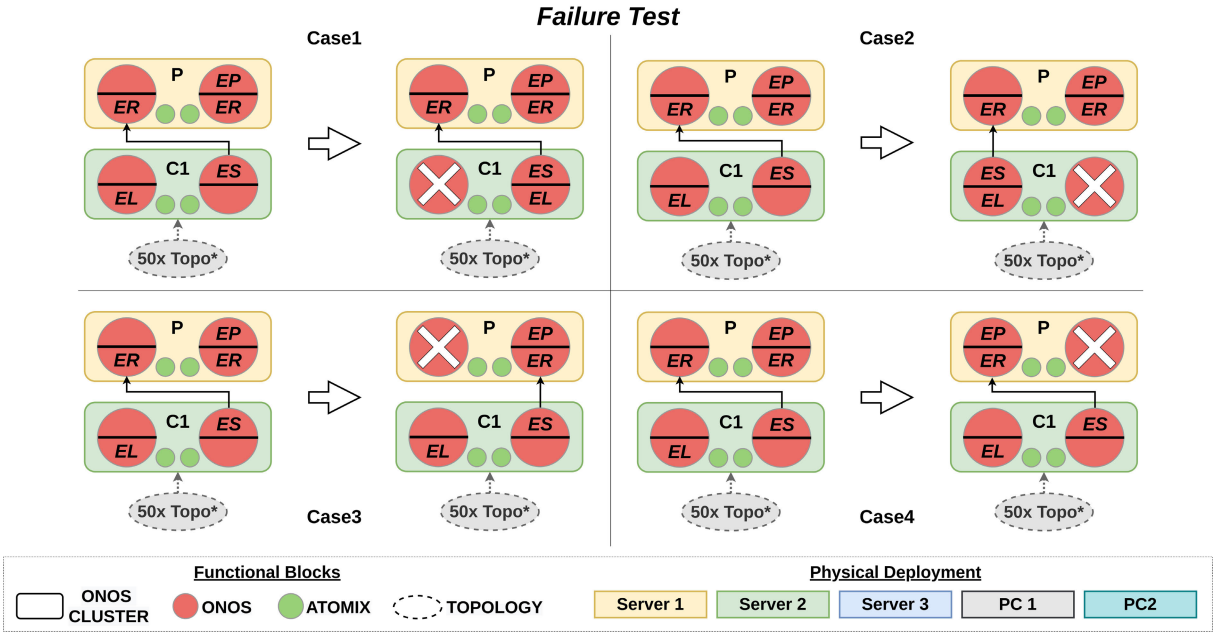

FIGURE 9. Failure Test logical scheme, along with physical deployment. The test considers a same starting setup composed of two dual-instance ONOS clusters, one child and one parent. A continuous stream of events is then generated by a 50x star topology while killing a different ONOS instance handling a different app functionality. Case1 and Case2 considers a failure affecting one of the child cluster instances, thus HSC. Case3 and Case4 does the same at the parent one, and so HSP.

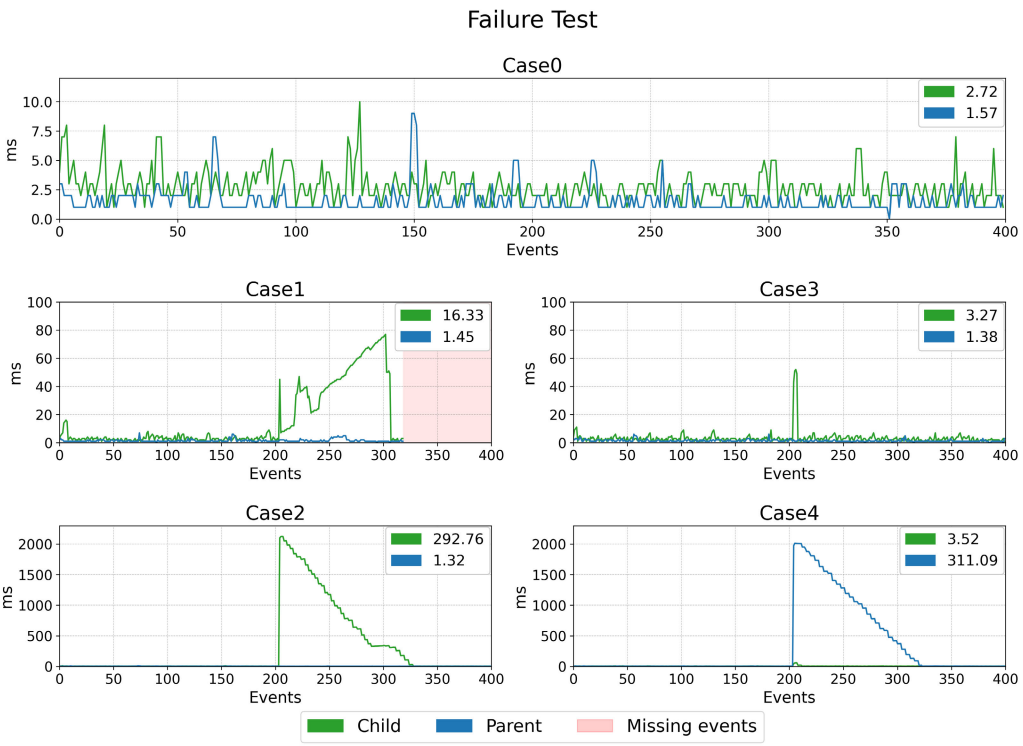

FIGURE 10. Failure Test results showing per-app processing latency for each event of a continuous stream generated by a 50x topology. Halfway through this event stream (200th event), an ONOS instance handling an application component is killed. This forces the missing functionality leadership to migrate to the other active cluster instance. Case0 represents the baseline with no failure. Case1 and Case2 kill an ONOS instance on the child cluster C1 handling one of the HSC application components, *EventListener* and *EventSender* respectively. Case3 and Case4 do the same, but on the parent cluster P running HSP's *EventReceiver* and *EventPublisher* components.

in latency (±2s) in the child application after the failure. This is because events have to wait in the Atomix queue before being picked up by the newly elected *EventSender* on the other instance. Indeed, the latency then continues to

decrease as the queued events are processed and the new ones are processed until the system returns to a steady state (±130 events). On the other hand, the performance of the parent application is similar to Case0, as expected.

In Case3, the parent instance that handles the *EventReceiver* used to receive the event is killed. However, as shown in the figure, this scenario does not have a significant impact on the performance of the system. In fact, only a small peak in processing latency is registered, but on the child application. As mentioned earlier, the *EventReceiver* component is the only one in the architecture that is not managed by a leadership process. Therefore, multiple *EventReceiver* components run in parallel on each parent instance. After the failure of the one receiving the events, the child *EventSender* component realizes that the gRPC channel is no longer available (30ms deadline). So it redirects the pending events to the other *EventReceiver* instance that was ready to receive the data. Events in the middle of this process reflect this delay on the child side, as the timestamp of the HSC output is overwritten with each attempt. The first event after the failure is therefore the only one to experience increased processing latency (peak of approximately 50ms).

Finally, Case4 considers the failure of the parent instance running the *EventPublisher* component. It is immediately apparent that the results of this scenario are almost identical to those of Case2, but reflected on the HSP application. Indeed, they have in common that the component alive has to retrieve events that are waiting in the queue during the leadership renegotiation process. It shows the same immediate peak of ±2s after the failure, which still decreases rapidly until reaching the normal behaviour (±130 events).

V. CONCLUSION

This work presents and thoroughly evaluates a hybrid-hierarchical synchronization solution for SDN architectures. The resulting system achieves a strong two-level resilience model, exploiting cluster configurations on both the child and parent sides. It propagates events through gRPC channels, enabling seamless integration with other child controllers, and has demonstrated fast and responsive view alignment from child to parent from a performance perspective.

The implementation on the ONOS controller shows excellent performance. The developed applications achieve view synchronization with an end-to-end latency averaging less than 10 ms in low event rate scenarios and around 40-60 ms in burst scenarios. This performance is verified while increasing the number of devices managed by the controller, with no noticeable impact on system performance. The applications are also tested in both single and multi-instance clusters, with a maximum overhead of 30ms for the child application in the latter case. This replicated configuration improves fault tolerance, with failures effectively managed at both levels. In fact, component failures affect the synchronization latency by up to 2 seconds, after which the system recovers quickly. In addition, the implemented applications are released as open source projects, enabling further development and community collaboration.

Another major strength of the proposed solution is its complete transparency with respect to the different technologies and communication protocols used in the child domains (e.g. OpenFlow, P4, NETCONF, etc.). Consequently, the resulting architecture facilitates the development of new applications at the parent level, using network views imported from the child controllers, while completely delegating communication with devices to the child controllers. This further emphasizes the potential of the proposed solution to enable a range of new scenarios, including those with multiple hierarchical levels, where the management of increasingly complex networks can be simplified.

REFERENCES

[1] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN control: Survey, taxonomy, and challenges," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 1, pp. 333–354, 1st Quart., 2018.

[2] T. Hu, Z. Guo, P. Yi, T. Baker, and J. Lan, "Multi-controller based software-defined networking: A survey," *IEEE Access*, vol. 6, pp. 15980–15996, 2018.

[3] Y. Liu, A. Hecker, R. Guerzoni, Z. Despotovic, and S. Beker, "On optimal hierarchical SDN," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2015, pp. 5374–5379.

[4] A. Pacini, D. Scano, L. Valcarenghi, A. Sgambelluri, and A. Giorgetti, "Enabling event-based hierarchical synchronization in SDN ONOS clusters," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2022, pp. 92–93.

[5] A. Pacini, A. Sgambelluri, C. Centofanti, A. Marotta, E. Paolini, A. Giorgetti, and L. Valcarenghi, "Hierarchical software-defined control for coordinated RAN and PON-based transport scaling," in *Proc. IEEE Netw. Oper. Manage. Symp.*, May 2024, pp. 1–3.

[6] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an open, distributed SDN OS," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.* New York, NY, USA: Association for Computing Machinery, Aug. 2014, pp. 1–6, doi: 10.1145/2620728.2620744.

[7] A. Giorgetti, A. Sgambelluri, R. Casellas, R. Morro, A. Campanella, and P. Castoldi, "Control of open and disaggregated transport networks using the open network operating system (ONOS) [invited]," *J. Opt. Commun. Netw.*, vol. 12, no. 2, pp. A171–A181, Feb. 2020.

[8] Google. *gRPC*. Accessed: Jan. 2025. [Online]. Available: https://grpc.io/

[9] *Hierarchical Sync Child Repository*. Accessed: Jan. 2025. [Online]. Available: https://github.com/Network-And-Services/HierarchicalONOS-Child

[10] *Hierarchical Sync Parent Repository*. Accessed: Jan. 2025. [Online]. Available: https://github.com/Network-And-Services/HierarchicalONOS-Parent

[11] Open Networking Foundation. *ONOS Documentation*. Accessed: Jan. 2025. [Online]. Available: https://api.onosproject.org/2.7.0/apidocs/

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proc. OSDI*, 2010, vol. 10, no. 1, p. 6.

[13] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in *Proc. Internet Netw. Manage. Conf. Res. Enterprise Netw.* Berkeley, CA, USA: USENIX Association, Apr. 2010, p. 3.

[14] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," in *Proc. IEEE Int. Symp. World Wireless, Mobile Multimedia Netw.*, Jun. 2014, pp. 1–6.

[15] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 19–20.

[16] M. Gerola, F. Lucrezia, M. Santuari, E. Salvadori, P. L. Ventre, S. Salsano, and M. Campanella, "ICONA: A peer-to-peer approach for software defined wide area networks using ONOS," in *Proc. 5th Eur. Workshop Softw.-Defined Netw. (EWSDN)*, Oct. 2016, pp. 37–42.

[17] *Hazelcast*. Accessed: Jan. 2025. [Online]. Available: https://github.com/hazelcast/hazelcast

[18] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw.* New York, NY, USA: Association for Computing Machinery, Aug. 2012, pp. 19–24, doi: 10.1145/2342441.2342446.
[19] A. D. Ferguson et al., "Orion: Google's software-defined networking control plane," in *Proc. 18th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, Apr. 2021, pp. 83–98. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/ferguson
[20] Google. *Protocol Buffers—Google's Data Interchange Format*. Accessed: Jan. 2025. [Online]. Available: https://github.com/protocolbuffers/protobuf
[21] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Hong Kong, China. New York, NY, USA: Association for Computing Machinery, 2013, pp. 3–14, doi: 10.1145/2486001.2486019.
[22] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proc. 26th Annu. ACM Symp. Princ. Distrib. Comput.*, Aug. 2007, pp. 398–407.
[23] R. Vilalta, R. Muñoz, R. Casellas, R. Martínez, V. López, O. G. de Dios, A. Pastor, G. P. Katsikas, F. Klaedtke, P. Monti, A. Mozo, T. Zinner, H. Øverby, S. Gonzalez-Diaz, H. Lønsethagen, J.-M. Pulido, and D. King, "TeraFlow: Secured autonomic traffic management for a tera of SDN flows," in *Proc. Joint Eur. Conf. Netw. Commun. 6G Summit (EuCNC/6G Summit)*, Jun. 2021, pp. 377–382.
[24] *Kubernetes*. Accessed: Jan. 2025. [Online]. Available: https://kubernetes.io/
[25] L. Gifre, R. Vilalta, J. C. Caja-Díaz, O. G. de Dios, J. P. Fernández-Palacios, J.-J. Pedreno-Manresa, A. Autenrieth, M. Silvola, N. Carapellese, M. Milano, A. Farrel, D. King, R. Martinez, R. Casellas, and R. Muñoz, "Slice grouping for transport network slices using hierarchical multi-domain SDN controllers," in *Proc. Opt. Fiber Commun. Conf. Exhib. (OFC)*, Mar. 2023, pp. 1–3.
[26] M. A. Togou, D. A. Chekired, L. Khoukhi, and G.-M. Muntean, "A hierarchical distributed control plane for path computation scalability in large scale software-defined networks," *IEEE Trans. Netw. Service Manage.*, vol. 16, no. 3, pp. 1019–1031, Sep. 2019.
[27] T.-C. Huang, C.-Y. Huang, and Y.-C. Chen, "Real-time DDoS detection and alleviation in software-defined in-vehicle networks," *IEEE Sensors Lett.*, vol. 6, no. 9, pp. 1–4, Sep. 2022.
[28] A. M. D. Tello and M. Abolhasan, "SDN controllers scalability and performance study," in *Proc. 13th Int. Conf. Signal Process. Commun. Syst. (ICSPCS)*, Dec. 2019, pp. 1–10.
[29] *AETHER*. Accessed: Jan. 2025. [Online]. Available: https://aetherproject.org/
[30] *P4*. Accessed: Jan. 2025. [Online]. Available: https://p4.org/
[31] J. Halterman. *Atomix*. Accessed: Jan. 2025. [Online]. Available: https://github.com/atomix/atomix-archive
[32] *Kafka-Integration ONOS App*. Accessed: Jan. 2025. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Kafka+Integration
[33] B. Yan, Y. Zhao, X. Yu, Y. Li, S. Rahman, Y. He, X. Xin, and J. Zhang, "Service function path provisioning with topology aggregation in multi-domain optical networks," *IEEE/ACM Trans. Netw.*, vol. 28, no. 6, pp. 2755–2767, Dec. 2020.
[34] *Mininet*. Accessed: Jan. 2025. [Online]. Available: http://mininet.org/
[35] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, Mar. 2014.

ALESSANDRO PACINI received the bachelor's degree in computer science from the University of Camerino, in 2018, and the joint master's degree in computer science and networking from the University of Pisa and Scuola Superiore Sant'Anna, in 2021. He is currently pursuing the Ph.D. degree in emerging digital technologies with the Scuola Superiore Sant'Anna. During this time, he won a one-year research fellowship at SSSA focused on building a scalable and reliable monitoring architecture for optical networks. His research interests include next-generation software-defined networks, with a particular focus on reusing existing network architectures to move toward a zero-touch paradigm.

DAVIDE SCANO received the B.S. degree in telecommunication engineering from the University of Pisa, in 2017, and the joint M.S. degree in computer science and networking from the University of Pisa and Scuola Superiore Sant'Anna, in 2019, with a research thesis on SDN for guaranteeing QoS in network slicing. He is currently pursuing the Ph.D. degree in emerging digital technologies with Scuola Superiore Sant'Anna. In 2020, he got a Research Scholarship at Scuola Superiore Sant'Anna, Pisa. His research interests include software defined networking, next generation software defined networking, optical networks, and disaggregated networks.

ANDREA SGAMBELLURI has been an Assistant Professor with Scuola Superiore Sant'Anna, Pisa, Italy, since 2019. He has published around 100 papers (source Google Scholar, May 2021) in international journals and conference proceedings. His research interests include control plane techniques for both packet and optical networks, including software defined networking (SDN) protocol extensions, network reliability, industrial ethernet, switching, segment routing application, YANG/NETCONF solutions for the dynamic management, telemetry, (re)programming, and monitoring of optical devices. In March 2015, he won the grand prize at the 2015 OFC Corning Outstanding Student Paper Competition with the article "First Demonstration of SDN-Based Segment Routing in Multi-Layer Networks."

LUCA VALCARENGHI (Senior Member, IEEE) has been an Associate Professor with Scuola Superiore Sant'Anna, Pisa, Italy, since 2014. He has published almost 300 papers (source Google Scholar, May 2020) in international journals and conference proceedings. He received a Fulbright Research Scholar Fellowship in 2009 and a JSPS "Invitation Fellowship Program for Research in Japan (Long Term)" in 2013. His research interests include optical networks design, analysis, and optimization; communication networks reliability; energy efficiency in communications networks; optical access networks; zero touch network and service management; experiential networked intelligence; and 5G technologies and beyond.

ALESSIO GIORGETTI received the Ph.D. degree from Scuola Superiore Sant'Anna (SSSA), Pisa, Italy, in 2006. In 2007, he was a Visiting Scholar with the Centre for Advanced Photonics and Electronics, University of Cambridge, U.K. From 2008 to 2020, he was an Assistant Professor with SSSA. From 2020 to 2024, he was a Researcher with the Institute of Electronics and Information Engineering of the Italian National Council for Research (IEIIT-CNR). He has been an Associate Professor with the University of Pisa, Italy, since 2024. He is the author of about 200 international publications, including journal articles, conference proceedings, and patents. He is an active software contributor to Open Network Foundation projects. His research interests include optical network architectures and control plane, industrial networks design, software-defined networking, and quantum communications.