JOURNAL OF
Optical Communications
and Networking

Reliable and scalable Kafka-based framework for optical network telemetry

ANDREA SGAMBELLURI,[1,*] ALESSANDRO PACINI,[1] FRANCESCO PAOLUCCI,[2] PIERO CASTOLDI,[1] AND LUCA VALCARENGHI[1]

[1]Scuola Superore Sant'Anna, Pisa, Italy
[2]CNIT, Pisa, Italy
*Corresponding author: andrea.sgambelluri@santannapisa.it

Received 9 March 2021; revised 3 May 2021; accepted 10 May 2021; published 7 June 2021 (Doc. ID 424639)

Telemetry data acquisition is becoming crucial for efficient detection and timely reaction in the case of network status changes, such as failures. Streaming telemetry data to many collectors might be hindered by scalability issues, causing delay in localization and detection procedures. Providing efficient mechanisms for managing the massive telemetry traffic coming from network devices can pave the way to novel procedures, speeding up failure detection and thus minimizing response time. This paper proposes a novel Kafka-based monitoring framework leveraging the telemetry service. The proposed framework exploits the built-in scalability and reliability of Kafka to go beyond traditional monitoring systems. The framework allows a continuous monitoring of optical system data and their distribution through simple compressed text messages to a large number of consumers. Moreover, the proposed framework keeps a limited history of the monitored data, easing, for example, root cause failure analysis. The implemented monitoring platform is experimentally validated, considering the disaggregated paradigm, in terms of functional assessment, scalability, resiliency, and end-to-end message latency. Obtained results show that the framework is highly scalable, supporting up to around 4000 messages per second (and potentially more) with low CPU load, and is capable of achieving an end-to-end (i.e., producer–consumer) latency of about 50 ms. Moreover, the considered architecture is capable of overcoming the failure of a monitoring framework core component without losing any message.

https://doi.org/10.1364/JOCN.424639

1. INTRODUCTION

The current era is characterized by a data deluge [1]. Data are becoming increasingly important in many fields, and telecommunications networks are not second to any of them. Network telemetry [2] is rapidly emerging as a means for collecting data to support network control [3], failure and anomaly detection [4,5], and many other functions.

In the vision of a fully automated network [6], such as the one envisioned by the European Telecommunications Standards Institute (ETSI) in the Zero Touch Network and Service Management architecture [7] and by the Open Radio Access Network (O-RAN) Alliance for what concerns the mobile radio access network [8], the effectiveness in collecting, storing, and making available all the data is of fundamental importance for closed-loop control performance [9].

In addition, the software defined networking (SDN) control plane and the advent of disaggregation in optical network solutions have opened the way to the availability of unified models [i.e., Yet Another Next Generation (YANG) models] to define, configure, and monitor the state of optical nodes, devices, and components of multiple vendors without the need to rely on closed proprietary platforms. This novel paradigm has been enhanced with the utilization of optical telemetry, enabling online streaming data related to signal quality, tributary transmitter and receiver metrics [e.g., optical signal-to-noise ratio (OSNR)], and line optical devices (e.g., amplifiers) input/output power values. The capability to configure, activate, and collect monitoring data to be consumed by central big data analytics platforms is part of the current optical telemetry system design hot topics.

The studies related to telemetry applied to optical networks are several, as reported in Section 2. However, few of them focus on the capabilities of the telemetry platform to provide reliability and scalability.

This study proposes an optical network telemetry framework based on Apache Kafka and exploiting Kafka Streams, Zookeeper, and Telegraf. The proposed framework exploits the built-in scalability and reliability of Kafka to go beyond traditional monitoring systems. Monitoring messages are exchanged by simple text messages that can be formatted according to desired models without requiring specific protocol for message exchange. Kafka provides many libraries and tools to integrate

data producers from different sources, and it allows a continuous monitoring of the considered metrics. Each device produces a single stream of data, leaving the task of distributing them to different consumers, ensuring both scalability and reliability. Last but not least, in the proposed framework, the cluster can keep a history of the messages retrieved from the producers, making it possible, upon a failure of the optical network, to retrieve previous monitoring parameters for performing, for example, root cause failure analysis.

2. RELATED WORK ON OPTICAL TELEMETRY

In the last five years, there have been significant research activities related to telemetry and monitoring architectures and solutions, specifically addressing optical networks, mainly driven by the disaggregation framework and the advent of artificial intelligence (AI). The main research directions are based on telemetry protocols, platforms, architectures, and novel disaggregated optical devices offering open interfaces and hardware drivers for physical parameter data monitoring interfaces.

The work in [10] is one of the first proposals of YANG extensions for on-demand optical equipment data acquisition, in the context of the Internet Engineering Task Force (IETF) Abstraction and Control of TE Networks (ACTN) virtual network models.

The study reported in [11] proposed the use of data analytics platforms based on continuous monitoring of Internet Protocol Flow Information Export (IPFIX) observation points related to a sliced optical network based on the Spark platform. The work was extended in [12] to support disaggregation and perform telemetry of data sourced by an open optical spectrum analyzer (OSA) aiming at inferring optical filter malfunctioning. Multilayer extensions were covered in [13].

The work in [14] presents two architectural implementations of telemetry based on partial and full disaggregation degrees, using the generalized Remote Procedure Call (gRPC) protocol, showing good scalability performance and highlighting the effectiveness of the gRPC framework, able to support efficient encoding and native push streaming mode, more efficient with respect to notification mechanisms based on Network Configuration Protocol (NETCONF). Focusing on the optical device agent, the work in [15] adopts an extended agent performing threshold-based telemetry streaming for network verification and data analytics. The work in [16] proposes an online gRPC telemetry of physical fiber parameters (i.e., bending) affecting coherent receiver OSNR, combined with an AI-based convolutional network.

A number of recent works report the implementation of telemetry agents co-located with optical devices having high frequency rates and resolutions [17]. For example, the work in [18] proposes a power monitor blade with a 400 μs telemetry streaming period and control plane using direct memory access to support AI engines. In addition, the work reported in [19] details an open disaggregated reconfigurable add and drop multiplexer (ROADM) including a filterless add/drop module equipped with photodetector tap arrays with gRPC telemetry servers including NETCONF notifications, capable of 20 Hz sample update frequency.

In the framework of multilayer networks, a combined per-layer telemetry approach has been proposed in [20]. The authors propose to combine optical telemetry related to light-path health with in-band telemetry (INT) of tributary flows handled by a P4 switch, analyzing the benefits of a combined approach. In the context of packet-switched layer INT, a preliminary evaluation of a Kafka-based monitoring system handling INT values has been carried out in [21], proposing an event-triggered mechanism to produce telemetry reports to the Kafka system from aggregate statistics related to specific data plane flows.

The work in [22] exploits a telemetry-based workflow to assign the proper transmission operational modes to transponders (including proprietary modes not disclosed explicitly to the SDN controller) during connection provisioning in partial disaggregation. The work also evaluates the impact of the proposed telemetry workflow in a multilayer network upon soft failure recovery affecting both the optical and packet-switched layers.

Optical telemetry may be extended not only to support isolated online network monitoring, but also to enrich data correlation related to optical network security, such as online video analytics [23].

Agile telemetry systems were proposed in the critical use case of disaster recovery, addressing high reconfigurability. In particular, the system is robust to unstable control plane connectivity and failure recovery to auxiliary low bandwidth network segments [24].

To the best of our knowledge, there are a number of aforementioned works targeting telemetry scalability for disaggregated optical networks; however, there are not specific works addressing the joint requirement of scalability and reliability of an optical network telemetry system. In particular, the issue of a reliable optical telemetry system is currently undiscussed in the literature.

3. INTRODUCTION TO KAFKA CONCEPTS AND FEATURES

Apache Kafka [25], originally developed by LinkedIn, is an open-source stream processing platform, highly scalable, distributed, and fault tolerant. Based on a publish–subscribe paradigm, Kafka makes it possible to handle streams of messages grouped into topics, with a number of publishers and many subscribers for the same topic. Figure 1 shows an example of the main functional blocks involved in a Kafka system. The core of the system is based on a Kafka *cluster*, which typically consists of several *brokers*, where each broker receives, temporarily stores, and re-transmits the data streams. In addition to the brokers, one or more Apache Zookeeper instances (represented as zookeepers in Fig. 1) are utilized for the coordination of the different operations required to maintain the cluster (i.e., cluster's leader election, heartbeat requests transmission, and recording of the cluster's information). Those instances are decoupled from the brokers and can be deployed in different machines. Clients are connected to the Kafka cluster in two possible operation modes: as (i) Kafka producer instances (*KProducer*), which send data using one or more topics within the cluster, or (ii) Kafka consumer instances

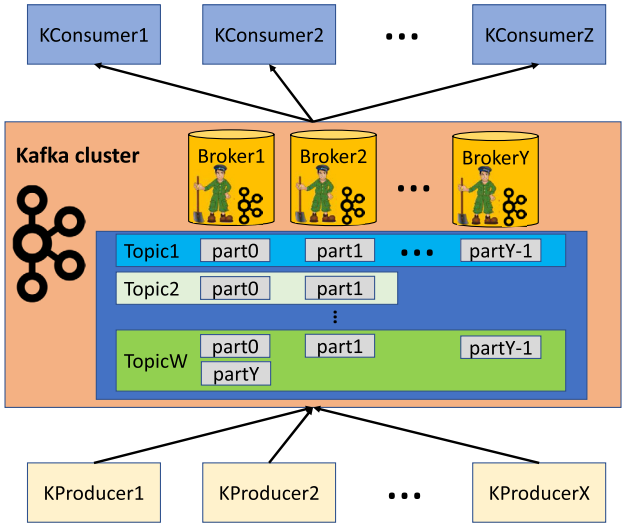

Fig. 1. Kafka cluster architecture example.

(*KConsumer*), able to subscribe and retrieve the data from one or more topics of the cluster.

This system has been natively designed to provide scalability. Indeed, each topic can be partitioned and spread across the cluster brokers, making it possible to distribute the workload over the available brokers and to increase the overall aggregate data retrieval speed. As shown in Fig. 1, the partitioning can be configured in different ways: (i) one partition per broker (i.e., the case of Topic1), (ii) limited partitions over the available brokers (i.e., the case of Topic2), or (iii) some broker leads more than one partition (i.e., the case of TopicW, with broker1 leading both partition0–part0 and partitionY–partY). Each Kafka consumer retrieves data from every partition subscribed to. Moreover, multiple consumers can be bound together in a consumer group and subscribe to the same topic, so that they can retrieve in parallel the data from the different topic's partitions. Thus, each consumer reads from one of the topic partitions, but if more consumers are defined with respect to partitions, the surplus consumers are not used, but remain waiting in case someone active is stopped.

When messages are sent to a topic by a producer, the partition in which the message is stored can be (i) directly specified with the partition ID, (ii) automatically derived from the optional key field of the message, or (iii) not defined and thus sent in round robin over the available partitions. In general, Kafka preserves the order of messages received on the same partition, but it does not perform this operation in the case of multiple partitions. Thus, if messages sent over a topic with multiple partitions must be ordered, they must be sent to the same partition. This objective can be achieved by specifying the same partition ID or message key for messages that must be ordered.

In addition, the Apache Kafka platform provides an embedded resiliency mechanism, called *replica*. As shown in Fig. 2, each topic's partition, served by one cluster broker (partition leader), represented on top of the partition list with solid bold lines (e.g., part0), can be replicated across one or more of the other available brokers (partition followers) (e.g., R_part0). The replication strategy enables the system to quickly recover

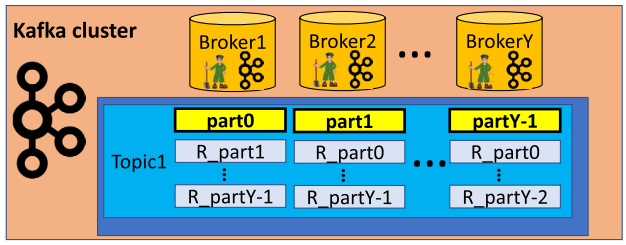

Fig. 2. Cluster topic's partition and replication example.

the affected partitions in the case of a broker failure. The replica configuration consists of two main parameters: (i) the replica count, which defines the number of replicas to be activated for each partition (at most as the number of brokers), and (ii) the number of in-sync replicas, which identifies the minimum number of replicas, among the ones configured, to be updated at every received message. In this way, no messages will be lost in the stream, if an in-sync replica will be used as the active topic partition. The replica configuration can also be considered at the producer side, waiting until all the in-sync partitions are updated before considering a message delivered. New topics with all their relative properties (partitions, replicas, etc.) can be created on demand by using specific instances, called KafkaAdmin, to apply changes and retrieve information to/from the cluster. However, some default configuration values, such as the default number of replicas per topic, can be defined on the broker side.

The Kafka cluster maintains the history of the latest received messages per consumer for the different topics' partitions, according to a broker's log configuration. In this way, the cluster administrator can limit in size (i.e., storage in GB) or in time (i.e., time in hours) the history that can be read by a Kafka consumer. Moreover, consumers can also specify the message index from which it wants to start reading. This temporary local storage is the main advantage of using Kafka with respect to RabbitMQ or other message-queue-based systems. This makes it possible to implement "opportunistic" consumers that can retrieve messages from a topic even if they were not subscribed to it earlier. In other systems, such as RabbitMQ, messages are not preserved once consumed, and so newly added consumers cannot retrieve them. The approach of Kafka enables scenarios of context-dependent monitoring, which completely relies on the temporary storage of Kafka to read subranges of data just when needed. This allows a certain dynamicity of the monitoring architecture, without affecting any other continuous consumption operation.

4. PROPOSED ARCHITECTURE

This section reports the detailed description of the proposed monitoring platform, enhanced with a new generation telemetry feature exploiting the features provided by Apache Kafka. The considered architecture is shown in Fig. 3. According to the partial disaggregation concept, it includes nodes of two types: (i) xPonder nodes that control the optical cards and (ii) optical line system (OLS) agents, responsible for control of the transmission infrastructure (i.e., the optical amplifiers). Each node is equipped with a Kafka producer (i.e., Telegraf

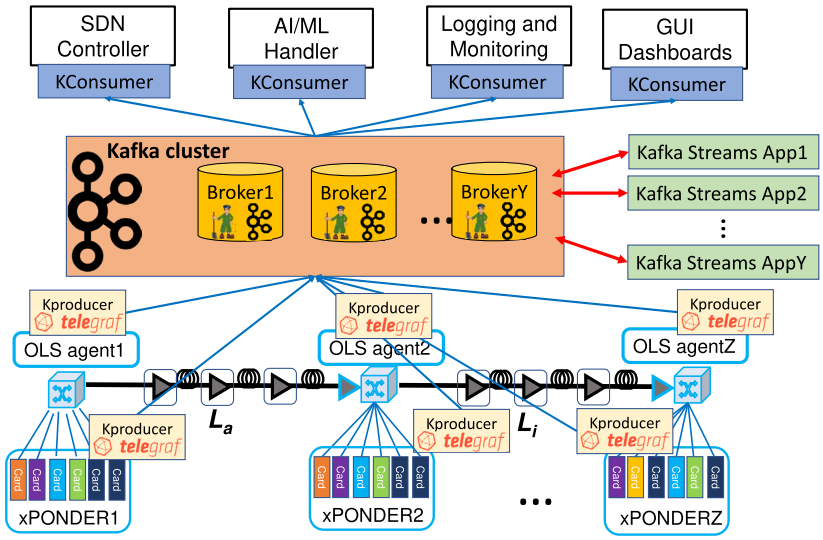

Fig. 3. Proposed Kafka-based architecture.

[26]), to stream the metrics related to all the controlled components (i.e., xPonder nodes report the metrics related to all the controlled cards and related optical ports, while OLS agents report metrics related to all the controlled amplifiers spread over the different optical links). All the streams are managed by the Kafka cluster, consisting of different hosts, each one running a Kafka broker along with a Zookeeper instance. Each topic has multiple partitions, with a number of replicas and at least two in-sync replicas. In this way, the failure of one host (i.e., both the broker and the Zookeeper instances) does not impair the correct functioning of the system. As shown in Fig. 3, specific Kafka Streams [25] applications have been developed, to allow the online processing/filtering of the messages related to a topic, producing additional topics, streamed to the subscribed consumers. This feature can be useful for filtering the data related to a specific lightpath or a link. More details will be provided in the next section. As already mentioned, if a client would like to retrieve the data from a topic, it may resort to a consumer group, composed by three consumers, being capable of retrieving data in parallel from the three partitions of that topic. As shown in Fig. 3, examples of consumers can be the SDN controller, an AI/machine learning (ML) handler, a full logging and monitoring system (for long-term storage of data) and a short-term data visualizer with a graphical user interface (GUI) and dashboards.

At the producer level, data can be collected at different endpoints (i.e., sensors, network devices, custom probes . . .). By instantiating a Kafka producer, agents write the input data (i.e., metrics) into one or more Kafka topics. The considered producers are based on Telegraf. Telegraf is a plugin-driven server agent for collecting and sending metrics and events from databases, systems, and Internet of Things (IoT) sensors, and it is a mature example of a producer. Telegraf can collect metrics from a wide array of input sources (called input plugins) and write them into a wide array of output destinations (called output plugins). Kafka is one of the supported output destinations. By editing the Telegraf configuration file, it is possible to define the plugins, desired metrics, input collection interval, output flush interval, and even filtering and routing metrics across its input/output plugins. One of the most important functionalities of Telegraf is that, in the case that a specific output plugin's destination is not available, a Telegraf agent saves the actual batched data and retries to send it in the next flush. This means that, even if the entire cluster is not reachable for a certain period (i.e., the cluster goes down or network connectivity fails), data are not lost. In addition, each input/output plugin can also specify its data format and also include the generation timestamp. Moreover, at the consumer side, in the case of network problems, each consumer will retry polling messages from the subscribed topics until the cluster is reachable again. If a consumer is forced to stop, once restarted, it will be able to retrieve all the messages, starting from the point where it was stopped.

The adoption of the aforementioned architecture as a monitoring platform for optical networks paves the way to novel and advanced functionalities. At the basis of our choice, the scalability, resiliency, and distributed approach are fundamental features, well demonstrated in many important production messaging systems that are not always considered/addressed in traditional monitoring platforms. Another strong point of the proposed system is that, being Kafka-based, it considers the exchange of text messages. By formatting the messages according to specific models, using JSON formatting, many fields of application can be exploited, without requiring additional/specific protocols (i.e., gRPC, thrift, IPFIX) and the related definitions. A Kafka-based system offers many libraries and tools to integrate data producers from different sources (i.e., Telegraf) over different programming languages (Python, Java, Golang). In this way, the Kafka producers, acting as telemetry servers, can be easily integrated in different platforms, including network devices. In a traditional telemetry-enabled monitoring system, metrics are collected with a relaxed heartbeat (i.e., 15 min or 1 h), but specific, and sometimes limited, metrics can be retrieved by subscribing to each specific stream of interest, generated directly by the device (i.e., a telemetry stream). In the proposed approach,

all network devices are equipped with a Kafka producer (i.e., Telegraf) to be able to report continuously and efficiently all the metrics, related to the node, including specific node identifiers and intra-node components' tags to be used for filtering purposes (according to the JSON model for node representation/abstraction). Considering traditional telemetry solutions, each subscription request includes one or more collector points, generating a data stream for each collector point (e.g., if 10 collectors are included in the request, 10 data streams will be activated in parallel from each considered device). In the proposed approach, each device presents only one data stream to the Kafka cluster, always active. The distribution of the data is performed at the cluster, where the data are replicated and sent to all the subscribed consumers. Moreover, in traditional monitoring systems, the telemetry functionality is activated on-demand, when a failure is detected, to retrieve real-time data with low polling times, being able to detect possible fluctuations of parameters on the considered devices. However, activating the telemetry in response to a detected failure can be a sub-optimal solution, since the system can miss the transient of the affected metrics, by collecting the streams of real-time metrics. In the proposed approach, the cluster can be configured (considering storage time or size) to keep a history of the messages retrieved from the producers for different topics. When a new client (i.e., a Kafka consumer) performs the subscription to a topic, it can potentially receive all the data included in the history available on the cluster for that topic, being able to detect metric fluctuation and/or transience also in the time period before the subscription. As a possible advanced application of the proposed architecture, the Kafka Streams applications can be configured to create a new topic (i.e., Lightpath_ID) for each installed lightpath. If the main AI/ML handler, analyzing the data of the main topic, including the metrics of all the optical devices, detects a performance degradation at one line port [e.g., by noticing a decrease in the OSNR and an increase in the pre-forward-error-correction bit error rate (fec-BER)] a self-managed procedure, leveraging a dedicated agent, can identify the lightpath of interest (i.e., lightpath X, where the card acts as RX) and then perform the subscription to the topic Lightpath_X, as a new consumer, being able to receive all the metrics related to all the traversed devices with the configured history. Further details will be provided in the next section on the activation of new topics with the Kafka Streams application.

5. IMPLEMENTATION

This section provides the implementation details of the proposed framework. In Fig. 4(a) is reported a simple scenario to explain the proposed approach.

Starting from the dataplane devices (i.e., bottom-up approach), we enhanced the commercial optical equipment in our laboratory (i.e., Ericsson SPO) with agents and custom application programming interfaces (APIs) to enable both the configuration and monitoring. In particular, a Python-based representational state transfer (REST) server, running in a dedicated host, is devoted to configure and monitor the optical devices. This operation is performed by adopting proprietary libraries on an embedded console. This console allows direct access to all the parameters exposed by the optical devices considering both configuration and monitoring features. The Python REST server implements a dedicated monitoring job (i.e., thread), invoked via an API call, that instantiates a connection with the requested optical device and periodically queries the metrics of interest (i.e., OSNR and pre-fec BER for each optical port or input and output power for amplifiers).

Then, a Telegraf agent is locally configured to collect the data sent by the aforementioned monitoring thread [listen to a specific User Datagram Protocol (UDP) port]. At each flush interval, it sends the collected metrics in a JSON format. The payload structure is the same one used by InfluxDB, since it was natively supported by Telegraf and pretty adaptive to any context. An example is provided in Fig. 4(b). To preserve a per device temporal order, the field device_id within each metric is used to route the messages to the same partition.

Three hosts are used to run the Kafka brokers and the Zookeeper instances. Each topic has three partitions, allocated one per broker, with three replicas and two in-sync replicas. All messages sent are gzip compressed by the Kafka producer and decompressed at the consumer side. Topics' max retrieval logs are limited to 72 h.

An OpticalNetwork topic is created to store all the messages related to the monitoring of all the optical controlled devices. Since an optical network typically consists of tens of optical

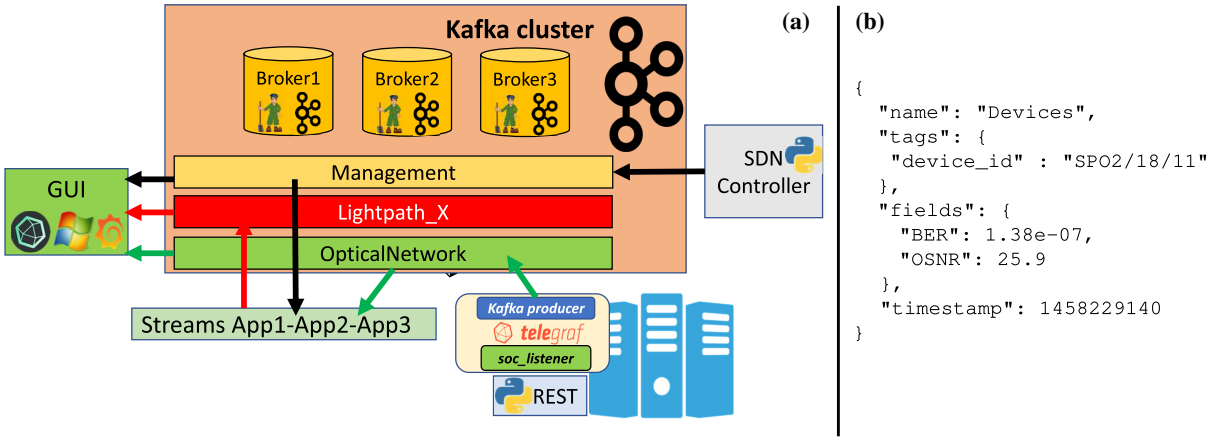

Fig. 4. (a) Optical network's monitoring flow. (b) JSON model example for transponder metrics.

nodes, each including tens of both controlled optical ports and controlled amplifiers, this topic could contain metrics of up to hundreds of components, potentially causing scalability issues. Indeed, even to monitor a single lightpath, all the metrics in the topic have to be consumed. To overcome this limitation, specific Kafka Streams applications have been developed. Kafka Streams allows processing on topics' messages by also choosing a different output topic for the modified streams. In this way, the metrics in the main OpticalNetwork topic, coming from all the network devices, can be filtered, according to specific policies (i.e., the metric of all the devices traversed by a specific optical lightpath) and can be sent to a new dedicated output topic. To provide a good level of dynamicity, a specific management topic has been conceived in the cluster enabling the communication among the SDN controller/orchestrator and the Kafka Streams applications. In particular, each application, by reading a message for the newly created Ligthpath_X1 from this topic, generates a new output topic (i.e., called Lightpath_X1) that will include only the metrics of the devices composing the lightpath (declared within the management message), filtered from the main topic OpticalNetwork. In the current implementation, a simple Python script is used to generate those management messages. Since a Kaflka Streams application acts basically as a consumer for the main OpticalNetwork topic, it can be run in multiple instances (i.e., three) by composing a consumer group. In this way, each instance filters the metrics from one of the three topic's partitions. Moreover, since the messages' keys are preserved from the ingress topic to the egress one, the temporal ordering of the metrics is maintained during the process. In addition, Streams applications have been designed and implemented to preserve the status of the active lightpaths, so that, in the case of the failure of an application, the filtering operation can be restored without re-processing all the messages exchanged over the management topic.

By relying on the aforementioned setup, a client application (i.e., Kafka consumer) is able to receive exclusively the metrics related to a specific lightpath, without performing any further data filtering/processing, by simply subscribing to the related topic. This functionality provides good flexibility in exploiting the available metrics, with a good level of resource abstraction, and without affecting the other topics (e.g., the main topic OpticalNetwork) that remain available for heavy data processing applications.

A specific topic has also been conceived to monitor the status of the cluster (i.e., cluster performance), enabling the collection and streaming of many metrics related to the cluster's hosts. Also in this case, a Telegraf application has been adopted to track the Kafka broker's performance, due to its flexibility. Figure 5 shows the considered scenario. Indeed, brokers natively provide their own metrics through Management Beans, which are then configured to be exposed with a Jolokia agent. Mbeans can be queried via HTTP in a JSON format, directly from a specific Telegraf input plugin, called jolokia2. Thus, each broker's Telegraf agent has the following input plugins:

- jolokia2, which retrieves a broker's specific metrics from the jolokia agent;
- procstat, which reads the Telegraf's process CPU/memory impact;
- execd, which starts a deamon handling a long-term running Python script to send an additional cluster's information.

Considering the latter input plugin, the Python script is able to read specific cluster metrics using the KafkaAdmin instance (i.e., examples of metrics are the list of topics, the replicas' positions, in-sync replicas, and more). To avoid redundant messages, a specific control procedure has been designed and implemented, running the script only when the local broker is the cluster's leader. At the other brokers, the control is periodically repeated, reacting to possible cluster leadership changes. The metrics collected by Telegraf using the different input plugins are sent using the topic KafkaMetrics, enforcing temporal ordering of the messages for each broker.

To summarize, at execution time, the cluster presents the following topics:

- OpticalNetwork, which includes the telemetry metrics coming from all optical devices;
- KafkaMetrics, which includes the metrics related to the performance of the cluster;
- management, used by the SDN controller/orchestrtor to inform that a new lightpath has been installed, listing the device IDs that compose it;
- Lightpath_X, one for each installed lightpath, to store the metrics of the devices traversed by lightpath X.

At the consumer side, a data visualization GUI, based on Grafana, has been implemented. This system exposes REST APIs to create custom dashboards (defined in a JSON format) that plot the data read from different possible data sources (i.e., an InfluxDB time series database). A dedicated Python-based agent has been designed and implemented to generate a consumer group (i.e., reading from all the available partitions) for each of the topics to be written into a per topic dedicated InfluxDB database. Grafana dashboards read the data from the database and display the selected metrics on the proper

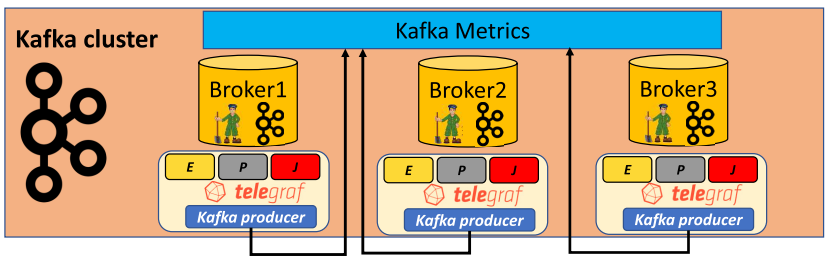

Fig. 5. Kafka cluster's monitoring flow.

panel of the dashboard, including both the Kafka cluster and the network metrics of the considered topics (i.e., aggregated or per lightpath). Moreover, another Python-based agent has been used to read messages from the management topic to create custom dashboards on Grafana for the newly advertised lightpaths. In the future, an additional application of the management topic may also be used to distribute configuration commands to the subscribed consumers, but this option is still under investigation.

6. PERFORMANCE EVALUATION

This section summarizes the aspects related to the experimental validation of the proposed architecture, highlighting details of the considered evaluation scenario and the obtained results in terms of overall functional assessment, scalability, resiliency, and end-to-end latency.

A. Evaluation Scenario

To validate the performance of the proposed solution, we considered the cloud environment, depicted in Fig. 6. The Kafka cluster consists of three Ubuntu 18.04 virtual machines in an ESX Environment (e.g., Broker1, Broker2, and Broker3), each with four Intel Xeon vCPUs with 2.3 GHz and 8 GB of RAM, running the Kafka broker along with a Zookeeper instance. All of them have 1 GB of Java Virtual Machine (JVM) committed memory. The three Kafka Streams applications (e.g., Stream App1, 2, 3) run in three Ubuntu 18.04 physical machines, with a quad-core Intel with 1.5 GHz, 4 GB of RAM, and 1 GB of JVM committed memory. Two Ubuntu 18.04 servers (e.g., Producer Server1 and Producer Server2), each equipped with an octa-core AMD Epyc 7262 with 3.2 GHz and 64 GB of RAM, are used to run the Python-based Kafka producers, including 20 xPonders and 12 OLS agents. Another Ubuntu 18.04 server (e.g., Consumer Server), equipped with two octa-core Intel Xeon Gold 6244 with 3.60 GHz and 64 GB of RAM, is used to run the Kafka consumers to consume, in parallel, the messages received over the different topics. An Ubuntu 18.04 virtual machine (e.g., KtoDB) in the ESX Environment, with four Intel Xeon vCPUs with 2.3 GHz and 8 GB of RAM, receives all the metrics and injects these data to the time-series InfluxDB database. Finally, a Windows physical machine (e.g., GUI), equipped with a dual-core Intel Core i5 with 3.1 GHz and 8 GB of RAM, is used to execute an instance of the InfluxDB time-series database and to run the Grafana-based GUI. This machine also runs the Telegraf producer, using a Python-based driver as an input plugin, to retrieve metrics from the real optical devices (as explained in Section 5).

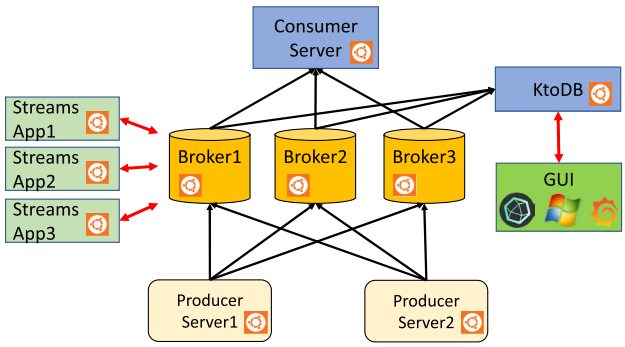

Fig. 6. Experimental testbed.

All the considered servers have been interconnected using a dedicated L2 Ethernet network at 1 Gb/s.

B. Functional Assessment

Initially, several tests were performed to verify the different functionalities of the proposed architecture and optimize the configuration of the involved functional blocks, including the Kafka brokers, Telegraf-based Kafka producers, and Kafka consumers. One of the Telegraf-based producers was connected to the real optical devices available in the laboratory, being able to periodically collect data from the muxponder ports (i.e., OSNR and pre-fec-BER) and from the connected amplifiers (i.e., input-power and output-power levels) and to send it over the OpticalNetwork topic. Since the traffic load of the Telegraf application controlling few optical devices available in the laboratory was very low (i.e., less than 1 kb/s), other Python-based instances of producers were activated. By streaming multiple times the same metrics collected on real devices, it was possible to increase the system load. Those producers, differently from Telegraf, send continuously the messages as soon as a metric is available. As already mentioned, to maintain the time order of the metrics, each producer pushes the data over the same partition of the topic.

Considering the Python-based consumer instances, each was configured to use three threads, reading data from the three partitions of each topic in parallel, speeding up the retrieval procedure. Kafka-to-DB is a Python script designed and implemented to retrieve the metrics from the partitions of selected topics (i.e., KafkaMetrics, Lightpath_X) and push it into distinct InfluxDB time-series databases to be plotted by Grafana-based dashboards. The three Kafka Streams applications were configured to optimize their jobs. They act as consumers of the OpticalNetwork topic and remain listening to the management topic for commands coming from the SDN controller/orchestrator. When a new command is received, which includes the lightpath ID and the list of all traversed devices, a new streaming process is activated, including only the metrics related to the listed devices. The management topic is also consumed by another Python script, running on the same machine of Kafka-to-DB, called DashbordManager. It creates, for each of the advertised lightpaths, a specific dashboard reading from the proper database instance that includes only the filtered metrics of the lightpath.

All those functionalities have been tested with success.

C. Scalability

Apache Kafka has been adopted in numerous production networks, including LinkedIn and Twitter where millions of users are efficiently served. To our knowledge, this is the first time that Kafka is adopted for the monitoring of an optical network. For this reason, we performed several tests to validate the scalability of the proposed solution.

Table 1. Scalability Test Results with Varying the Number of Consumers, Producers, and Active Topics

Name	#Producers	#Messages/s	#ConsumerGroups	Topics	ThroughputP1 (Mb/s)	ThroughputP2 (Mb/s)	ThroughputC (Mb/s)
Test1	32	3920	10	ON	1.7	1.7	48
Test2	16	1960	10	ON	1.7	–	27
Test3	32	3920	10	ON+10Ls	1.7	1.7	70
Test4	32	3920	1	ON+10Ls	1.7	1.7	7.2

At the producer side, each server runs 20 xPonder nodes, equipped with 20 cards each, presenting four line ports on each card and 12 OLS agents controlling 30 amplifiers each. The resulting scenario, considering the two producer servers, presents 3200 line ports and 720 amplifiers, reporting their metrics each second (i.e., 3920 messages per second are generated by the producers and spread across the three available partitions, while maintaining partition consistency).

We configured 10 consumer groups, composed of three consumers each, to read in parallel from the three partitions from the selected topics (i.e., 30 threads active). Those consumers were configured to manually commit every 100 ms the last message read, just to further stress the architecture.

Table 1 summarizes the results obtained during the scalability tests performed by varying the numbers of producers, consumers, and active topics.

In test1, 32 producers (i.e., all the available producers, including 20 xPonders and 12 OLS agents) were configured to produce 3920 messages per second, with 10 consumer groups receiving only the messages over the OpticalNetwork topic. Exchanged packets were captured at both producer and consumer sides to detect the throughput of the streaming traffic. On each producer, an average throughput of 1.7 Mb/s was registered, while the 10 consumers registered an average throughput of 48 Mb/s (i.e., around 4.8 Mb/s per consumer).

In test2, only 16 producers were considered producing 1960 messages per second, with the 10 consumer groups still receiving messages only for the OpticalNetwork topic. The same throughput was registered on the producer server (i.e., 1.7 Mb/s average). The reduction in the producer is reflected on the average throughput registered by the 10 consumer groups (i.e., aggregate 27 Mb/s and around 2.7 Mb/s per consumer).

During test3, the scenario of test1 was repeated with the addition of 10 new topics, related to Lightpaths read by the 10 consumer groups. In this case, the traffic generated by the producers is the same (the Lightpath topics are generated at the cluster level by the Streams applications processing the already existing metrics of the OpticalNetwork topic). The 10 consumer groups are subscribed to the OpticalNetwork topic and to the 10 Lightpath topics (Lightpath_1, …, Lightpath_10) generated by the Kafka Streams application. The registered average throughput is around 70 Mb/s (i.e., 7 Mb/s per customer). Figure 7 shows the Wireshark I/O graph of the 10 consumers aggregate traffic collected on the Ethernet interface used to communicate with the Kafka cluster during test3.

During test4, the same scenario of the test3 was considered (OpticalNetwork and 10 Lightpath topics active), with only one consumer group instance active. As expected, the registered average throughput was 7.2 Mb/s, and no changes were detected at the producer side.

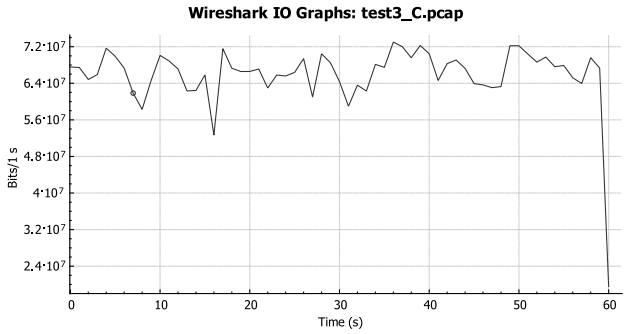

Fig. 7. Ten consumers aggregate throughput at the consumer server in b/s with OpticalNetwork and 10 Lightpath topics.

The obtained results show the effectiveness of the proposed solution. The worst case considers 10 consumer groups running on the same machine. Each consumer receives around 4000 messages per second from the main topic and around 60 messages per second (i.e., 10 lighpaths with six involved devices each) from the 10 active Lightpath topics: each consumer receives an average throughput of around 7.2 Mb/s.

Moreover, in this same case, we reached the highest CPU usage for each of the broker's processes, which was still under 50%.

D. End-to-End Latency and Resiliency

To validate the performance of the system in terms of time required to deliver a message from a producer to a consumer (i.e., end-to-end latency experienced by the messages), the test3 scenario was considered, with 10 consumer groups active and 32 producers generating 3920 messages per second. We instantiated an additional consumer group and three additional producers, connected to the management topic. Both the consumers and the producers were running on the same machine to preserve time synchronization, adopting the same clock reference. The producers were sending messages, including the generation timestamp. At the consumer side, the elapsed time was computed (considering the difference between reception time and generation time). The end-to-end latency performance of the system was evaluated under three conditions: normal operation (i.e., three active brokers), failure of a cluster's non-leader broker, and failure of the cluster's leader broker.

1. Normal Operation

To evaluate the performance of the system in steady state condition, each producer generated 10 messages per second (i.e., one message each 100 ms) to one of the three partitions, resulting in an aggregate rate of 30 messages per second. The consumer group consisted of three consumers connected to the three partitions, receiving all the generated messages in parallel. The test was repeated 10 times, collecting the performance per partition (i.e., per broker) considering 600 messages (i.e., test duration of 60 s). No message loss was detected in any of the executed tests. Figures 8–10 show the distribution of the end-to-end latency respectively collected for partition0, partition1, and partition2. The three distributions are quite similar, with around 90% of the samples in the range of 20–140 ms; average end-to-end latency of 53.88 ms, 53.48 ms, 53.92 ms, respectively, for partition0, partition1, and partition2; and confidence interval at the 95% confidence level in the range of [4.85 ms, 5.32 ms].

Obtained results show that the transmission of the messages through the Kafka cluster will produce an average delay of around 53 ms on the communication between the producer (i.e., network device) and consumer (i.e., collector) with no difference regarding the traversed broker. To evaluate the strict latency contribution introduced by the Kafka system, we measured the latency experienced by a standard gRPC telemetry system over the same infrastructure. The experienced baseline latency is around 5 ms. Hence, the added Kafka contribution is around 48 ms average, mainly due to buffering, transmission, and synchronization processing at the cluster level.

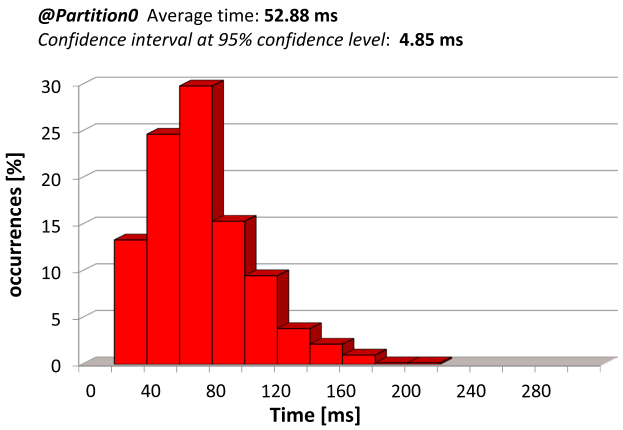

Fig. 8. End-to-end latency distribution for messages handled by partition0.

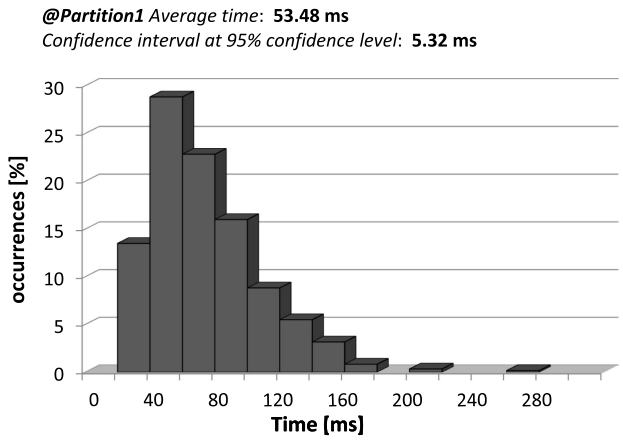

Fig. 9. End-to-end latency distribution for messages handled by partition1.

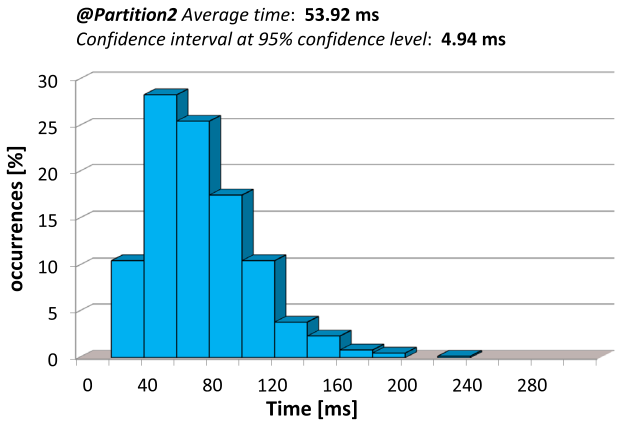

Fig. 10. End-to-end latency distribution for messages handled by partition2.

2. Non-Leader Failure

The same test was repeated, generating the failure of one of the brokers, with no cluster leadership role. In this way, the cluster capability to recover from a failure was validated, detecting possible effects on the end-to-end latency. Producers and consumers presented the same configuration described in the previous section.

Figure 11 shows the evolution of the end-to-end latency experienced by the messages over the three partitions, served by the three brokers, during the test. Before the failure, B1 was the leader of partition0, B3 of partition1, and B2 of partition2. Then B1 failed, and partition0's leadership moved to B3, devoted to both partition0 and partition1. The inset of Fig. 11 shows that the failure did not affect the messages related to partition0, the one served by the failed broker. Partition2 is slightly affected by the failure, with few messages presenting a higher end-to-end latency, while partition1, whose leader took responsibility of partition0 (since it was its in-synch replica), is the most affected by the failure. This may depend on a Kafka policy that could give priority to the new partition to be served,

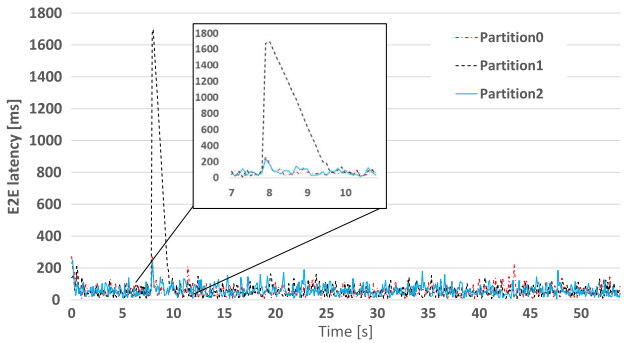

Fig. 11. End-to-end latency evolution during the test in the case of non-leader broker failure.

with respect to the one already served. The effect on partition1 presents a peak of 1.6 s and lasts for 1.7 s with a decreasing impact. After that, all the partitions are served with the same performance observed before the failure. No messages were lost during the experiment.

3. Leader Failure

The previous test was repeated, generating the failure of the broker acting as a cluster leader. Also in this case, producers and consumers presented the same configuration described in the normal operation.

Figure 12 shows the evolution of the end-to-end latency experienced by the messages over the three partitions, served by the three brokers, during the test. Before the failure, B1 was the leader of partition0, B3 of partition1, and B2 of partition2. Then B3, acting as the cluster's leader, failed, and partition1's leadership was moved to B2, devoted to both partition1 and partition2. The inset of Fig. 12 shows that the failure affected the messages related to all the partitions. Partition0 and partition1 are slightly affected by the failure, with a few messages presenting a higher end-to-end latency, while partition2, whose leader took responsibility for partition1 (since it was its in-synch replica), is the most affected by the failure. The effect on partition2 presents a peak of 714 ms and lasts for 0.9 s with a decreasing impact. After that, other peaks, with lower intensities, are registered on partition0 and partition1, caused by the cluster leadership negotiation. After the failure, B2 became the new leader of the cluster. Even in this case, no messages were lost during the experiment.

As a final consideration on the reported results, the proposed Kafka-based system has been demonstrated to provide high flexibility and efficiency while enhancing the resiliency and scalability of the telemetry communication in optical networks. Two main possible side effects have been identified, i.e., the increment of latency messages and the overall system configuration burden. However, the disaggregated optical network use case, in the latest implementations resorting to telemetry sample granularity of the order of seconds, appears to be a suitable candidate for the deployment of the proposed system.

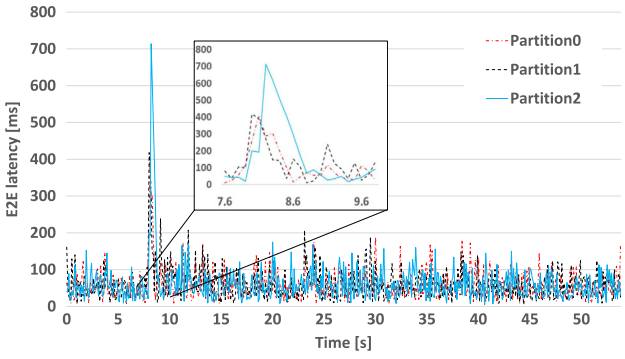

Fig. 12. End-to-end latency evolution during the test in the case of leader broker failure.

7. CONCLUSION

In this study, a novel monitoring framework, exploiting a Kafka-based messaging system, has been designed, implemented, and experimentally validated. The proposed framework is devised to efficiently handle the continuous telemetry traffic from the devices present in an optical network, also encompassing disaggregated concepts. The efficiency and flexibility of the proposed solution has been shown to enhance commercially available optical devices, not natively supporting the telemetry. However, the scalability performance has been evaluated in an emulated environment, enabling the telemetry of a high number of devices. The performance evaluation conducted in an emulated optical network testbed has shown that the framework is highly scalable supporting up to around 4000 messages per second, generated by 32 producers and received by 10 consumer groups. Results also showed that the architecture can be further stressed. Moreover, the framework was capable of achieving an end-to-end (i.e., producer–consumer) latency of about 50 ms. Finally, the considered architecture based on replicated brokers was capable of overcoming the failure of a cluster broker without any impact on the collected data.

Funding. European Commission (876967); Italian Ministry of Education, University, and Research.

Acknowledgment. This work has received funding from the ECSEL Joint Undertaking (JU) BRAINE Project, under grant agreement no. 876967. The JU receives support from the European Union's Horizon 2020 research and innovation programme and from the Italian Ministry of Education, University, and Research (MIUR). The work was also supported by the Department of Excellence in Robotics and Artificial Intelligence funded by the MIUR to Scuola Superiore Sant'Anna.

REFERENCES

1. K. Slavakis, G. B. Giannakis, and G. Mateos, "Modeling and optimization for big data analytics: (statistical) learning tools for our era of data deluge," IEEE Signal Process. Mag. **31**(5), 18–31 (2014).
2. H. Song, F. Qin, P. Martinez-Julia, L. Ciavaglia, and A. Wang, "Network telemetry framework," IETF draft-ietf-opsawg-ntf-07 (2021).
3. A. Sgambelluri, J. Izquierdo-Zaragoza, A. Giorgetti, L. Gifre, L. Velasco, F. Paolucci, N. Sambo, F. Fresi, P. Castoldi, A. C. Piat, R. Morro, E. Riccardi, A. D'Errico, and F. Cugini, "Fully disaggregated ROADM white box with NETCONF/YANG control, telemetry, and machine learning-based monitoring," in *Optical Fiber Communication Conference (OFC)* (2018), paper Tu3D.12.
4. F. Paolucci, A. Sgambelluri, M. Dallaglio, F. Cugini, and P. Castoldi, "Demonstration of gRPC telemetry for soft failure detection in elastic optical networks," in *European Conference on Optical Communication (ECOC)* (2017).
5. S. Nam, J. Lim, J. H. Yoo, and J. W. K. Hong, "Network anomaly detection based on in-band network telemetry with RNN," in *IEEE International Conference on Consumer Electronics–Asia (ICCE-Asia)* (2020).
6. D. Rafique and L. Velasco, "Machine learning for network automation: overview, architecture, and applications [Invited Tutorial]," J. Opt. Commun. Netw. **10**, D126–D143 (2018).
7. ETSI, "Zero-touch network and service management (ZSM); reference architecture," ETSI GS ZSM 002 V1.1.1 (2019).
8. O-RAN Alliance, "O-RAN architecture description," O-RAN.WG1.O-RAN-Architecture-Description-v03.00, Technical Specification (2021).
9. V. R. Chintapalli, K. Kondepu, A. Sgambelluri, A. Franklin, B. R. Tamma, P. Castoldi, and L. Valcarenghi, "Orchestrating edge- and

cloud-based predictive analytics services," in *European Conference on Networks and Communications (EuCNC)* (2020), pp. 214–218.
10. Y. Lee, R. Vilalta, R. Casellas, R. Martínez, and R. Muñoz, "Scalable telemetry and network autonomics in ACTN SDN controller hierarchy," in *19th International Conference on Transparent Optical Networks (ICTON)* (2017).
11. L. Velasco, L. Gifre, J.-L. Izquierdo-Zaragoza, F. Paolucci, A. P. Vela, A. Sgambelluri, M. Ruiz, and F. Cugini, "An architecture to support autonomic slice networking," J. Lightwave Technol. **36**, 135–141 (2018).
12. L. Velasco, A. Sgambelluri, R. Casellas, L. Gifre, J. Izquierdo-Zaragoza, F. Fresi, F. Paolucci, R. Martínez, and E. Riccardi, "Building autonomic optical whitebox-based networks," J. Lightwave Technol. **36**, 3097–3104 (2018).
13. L. Gifre, J. Izquierdo-Zaragoza, M. Ruiz, and L. Velasco, "Autonomic disaggregated multilayer networking," J. Opt. Commun. Netw. **10**, 482–492 (2018).
14. F. Paolucci, A. Sgambelluri, F. Cugini, and P. Castoldi, "Network telemetry streaming services in SDN-based disaggregated optical networks," J. Lightwave Technol. **36**, 3142–3149 (2018).
15. A. Sadasivarao, S. Syed, D. Panda, P. Gomes, R. Rao, J. Buset, L. Paraschis, J. Brar, and K. Raj, "Demonstration of extensible threshold-based streaming telemetry for open DWDM analytics and verification," in *Optical Fiber Communication Conference* (Optical Society of America, 2020), paper M3Z.5.
16. T. Tanaka, S. Kuwabara, H. Nishizawa, T. Inui, S. Kobayashi, and A. Hirano, "Field demonstration of real-time optical network diagnosis using deep neural network and telemetry," in *Optical Fiber Communication Conference (OFC)* (2019).
17. A. Sadasivarao, S. Jain, S. Syed, K. Pithewan, P. Kantak, B. Lu, and L. Paraschis, "High performance streaming telemetry in optical transport networks," in *Optical Fiber Communication Conference (OFC)* (2018), paper Tu3D.3.
18. K. Ishii, S. Yanagimachi, A. Tajima, and S. Namiki, "Submilisecond control/monitoring of disaggregated optical node through a direct memory access based architecture," in *Optical Fiber Communication Conference (OFC)* (2019), paper Tu3H.5.
19. J. Kundrat, O. Havlis, J. Radil, J. Jedlinsky, and J. Vojtech, "Opening up ROADMs: a filterless add/drop module for coherent-detection signals," J. Opt. Commun. Netw. **12**, C41–C49 (2020).
20. A. Sgambelluri, F. Paolucci, A. Giorgetti, D. Scano, and F. Cugini, "Exploiting telemetry in multi-layer networks," in *22nd International Conference on Transparent Optical Networks (ICTON)* (2020).
21. J. Vestin, A. Kassler, D. Bhamare, K. Grinnemo, J. Andersson, and G. Pongracz, "Programmable event detection for in-band network telemetry," in *IEEE 8th International Conference on Cloud Networking (CloudNet)* (2019).
22. A. Sgambelluri, A. Giorgetti, D. Scano, F. Cugini, and F. Paolucci, "OpenConfig and OpenROADM automation of operational modes in disaggregated optical networks," IEEE Access **8**, 190094–190107 (2020).
23. J. E. Simsarian, M. N. Hall, G. Hosangadi, J. Gripp, W. van Raemdonck, J. Yu, and T. Sizer, "Stream processing for optical network monitoring with streaming telemetry and video analytics," in *European Conference on Optical Communications (ECOC)* (2020).
24. S. Xu, Y. Hirota, M. Shiraiwa, M. Tornatore, S. Ferdousi, Y. Awaji, N. Wada, and B. Mukherjee, "Emergency OPM recreation and telemetry for disaster recovery in optical networks," J. Lightwave Technol. **38**, 2656–2668 (2020).
25. Apache, "Apache Kafka and Kafka Streams," https://kafka.apache.org/.
26. InfluxData, "Telegraf," https://www.influxdata.com/time-series-platform/telegraf/.