Received February 25, 2022, accepted April 22, 2022, date of publication May 9, 2022, date of current version May 12, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3173434

TEMPOS: QoS Management Middleware for Edge Cloud Computing FaaS in the Internet of Things

ANDREA GARBUGLI, (Student Member, IEEE),
ANDREA SABBIONI, (Student Member, IEEE),
ANTONIO CORRADI, (Life Senior Member, IEEE),
AND PAOLO BELLAVISTA, (Senior Member, IEEE)
Department of Computer Science and Engineering (DISI), University of Bologna, 40136 Bologna, Italy

Corresponding author: Andrea Garbugli (andrea.garbugli@unibo.it)

ABSTRACT Several classes of advanced Internet of Things (IoT) applications, e.g., in the industrial manufacturing domain, call for Quality of Service (QoS) management to guarantee/control performance indicators, even in presence of many sources of ''stochastic noise'' in real deployment environments, from scarcely available bandwidth in a time window to concurrent usage of virtualized processing resources. This paper proposes a novel IoT-oriented middleware that i) considers and coordinates together different aspects of QoS monitoring, control, and management for different kinds of virtualized resources (from networking to processing) in a holistic way, and ii) specifically targets deployment environments where edge cloud resources are employed to enable the Serverless paradigm in the cloud continuum. The reported experimental results show how it is possible to achieve the desired QoS differentiation by coordinating heterogeneous mechanisms and technologies already available in the market. This demonstrates the feasibility of effective QoS-aware management of virtualized resources in the cloud-to-things continuum when considering a Serverless provisioning scenario, which is completely original in the related literature to the best of our knowledge.

INDEX TERMS Edge cloud computing, FaaS, Internet of Things, interoperability, middleware, QoS management, serverless.

I. INTRODUCTION

The continuous advancement of network and computing infrastructures is pushing the digital transformation and evolution of many diverse industrial sectors, asked to become ''smarter'', including industrial manufacturing, healthcare, and even tourism [1]. The types of applications that are requested to run on these distributed infrastructures are very differentiated and with very differentiated requirements, from latency upper-bounds to maximum allowable downtime and reliability; moreover, the ICT infrastructures hosting them include very heterogeneous resources and tend to employ more and more cloud continuum virtualized resources, which are typically positioned close to IoT sensors and actuators for greater efficiency [2].

The associate editor coordinating the review of this manuscript and approving it for publication was M. Anwar Hossain.

In addition, in these scenarios, depending on the industrial sector and the specific kind of application, the severity of effects due to provisioning failures can range from negligible to critical. Those elements push for considering novel approaches based on new forms of cloud computing belonging to the family of Serverless computing, such as Function as a Service (FaaS), where the complexity of the ICT infrastructure is demanded from the provider because they can enable application-domain experts to concentrate exclusively on the development of domain-specific solutions.

The need for control and compliance with QoS specifications in cloud-to-things environments for industrial manufacturing is widely recognized. But similar needs are present more and more in other application domains, which are increasingly benefiting from IoT-empowered technologies. For example, in the Smart Hospitality domain, modern accommodation facilities are integrating more and more connected sensors and actuators to provide an increasingly

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

digitized and personalized experience for their customers [3]. IoT devices, along with digital services, work alongside staff promising to help manage and create a more engaging experience, while also achieving accessibility and reduced environmental impact goals. Hence the need to provide differentiated QoS levels in the delivery and processing of information from different devices. For instance, IoT devices embedded in smart doors or management systems (e.g., SPA temperature controllers) require low latency and small variation in response time; at the same time, the growing number of AI and Virtual/Augmented Reality technologies embedded in both guest rooms and hotel gyms [4], while requiring high bandwidth and benefiting from localization of computational resources, can tolerate small performance degradation. Even if it is widely recognized the need for more advanced QoS management features in the cloud-to-things continuum [5], to the best of our knowledge there is no state-of-the-art solution in the current literature that works to control/guarantee differentiated QoS levels for FaaS-based IoT applications in the cloud continuum, by exploiting coordinated mechanisms and strategies spanning the whole set of employed and virtualized networking/processing resources.

To fill this relevant gap, we propose TEMPOS: a Time-Effective Middleware for Priority Oriented Serverless for IoT applications in the cloud continuum. TEMPOS can exploit and coordinate the different QoS mechanisms available over different technologies across the stack of virtualized FaaS invocations in the cloud continuum to properly manage end-to-end QoS in terms of jitter, latency, and enqueuing time. While preserving the general aspect of our study, and without loss of generality, the paper presents the current status of the implementation of a TEMPOS prototype, which primarily exploits Linux real-time scheduling, differentiated MOM priorities, and Time-Sensitive Networking (TSN) as the underlying system-level mechanisms to enforce the QoS-aware TEMPOS abstractions (TEMPOS QoS-aware topics) for QoS management. Our prototype has already been tested under different load conditions to simulate the behaviours of different and realistic use case scenarios, thus allowing us to collect a large set of experimental measurements about TEMPOS performance indicators, which are originally presented in the paper; those experimental results quantitatively show both the feasibility of the proposed approach and the efficiency of the achieved implementation of TEMPOS.

In short, we believe that this paper significantly advances the state-of-the-art literature in the field by proposing the following primary original contributions: i) introducing support for an edge cloud FaaS offer for IoT applications guaranteeing differentiated quality of services ii) holistic QoS management of different aspects of FaaS distributed invocation over virtualized resources iii) original architecture based on layers (Logical, QoS, and System) and slices (Bridging, Delivery, and Processing), to abstract the aspects related to the technologies used and compartmentalize the different tasks iv) an original implementation of the proposal based on MOM prioritization, Linux real-time scheduling, and TSN v) in-the-field experimental results over simple deployment cases that quantitatively show the feasibility of the proposed approach and the efficiency of the implemented middleware.

II. BACKGROUND

Edge cloud computing (or edge cloud) is a recent form of distributed cloud computing in which virtualized processing, networking, and storage resources are hosted at nodes at the edge of the considered network [6], [7]. The introduction of edge cloud technologies in this sense has opened to the development of many new smarter and QoS-sensitive services that can better meet latency, jitter, and scheduling-delay constraints thanks to the locality of data producers and processing devices [8]. The adoption of edge cloud solutions can offer multiple advantages but suffers from limited resource availability at edge hosts if compared with the more traditional cloud. Thus, sharing resources among multiple services and applications becomes essential in many scenarios of the so-called cloud-to-things continuum [9].

In this context, the definition and efficient usage of prioritization mechanisms become necessary to meet the different QoS demands of different types of IoT applications, composed of multiple tasks competing for the same virtualized resources. In addition, single mechanisms are not sufficient: the coordination of different prioritization technologies, across the full invocation stack (possibly including virtualized processing, invocation messaging, and communications) is needed to meet constraints of end-to-end jitter, latency, and queuing delays [10]. However, such coordination and orchestration of distributed resource reservation and invocation prioritization, while maximizing the efficiency of resource utilization, is recognized as a challenging task. This is further exacerbated by the huge heterogeneity of devices, operating systems, communication mediums, and protocols that are present in edge cloud-enabled IoT environments [11].

On the other side, Serverless computing is emerging as a new and successful paradigm promoting total absence of control, for the customer, over deploy and execution of services. Belonging to this family, FaaS [12] is a model in which customer-defined logic are put into execution dynamically, triggered by events. FaaS platforms are characterized by fine-grained per-request scaling of resources: at any time the number of resources employed by the FaaS platform automatically tends to be proportional to the number of requests issued. This scaling mechanism also leads to zero-scaling capability, which allows services and applications not in use to consume almost no resources, with evident advantages in practical deployment scenarios, in particular for environments with a high density of differentiated applications or with relatively limited resources such as for edge cloud nodes.

A typical FaaS platform consists of three main architectural components: *Trigger*, *Controller*, and *Executor Nodes* depicted in Fig. 1. The Trigger is the logical entity responsible for receiving or sensing external information. It converts them into internal events, manageable by the FaaS platform and

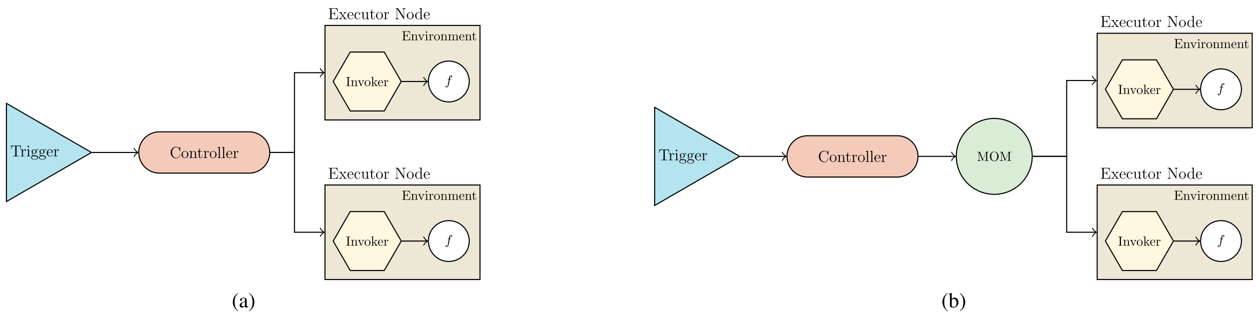


FIGURE 1. High level FaaS architectural approaches. (a) Architecture employing a direct invocation scheme enacted by the controller. (b) Architecture introducing a MOM system to decouple the controller and the functions.

processable by functions. Triggers typically receive requests from heterogeneous sources and the interaction with them can employ different protocols and formats.

Events generated by triggers are managed by a *controller*. The controller is the logically centralized entity that manages the configuration of other components in the FaaS platform, including function life-cycle or trigger setup. In addition, the controller takes care of the process of distribution of internal events generated by triggers; events are distributed to computing nodes where they are processed. For event distribution, the controller can interact with other components either realizing synchronous service call invocation (Fig. 1a) or exploiting asynchronous pub-sub-based communication (Fig. 1b). The former requires that each received event is first passed to the controller and processed there to decide the next hop in the invocation chain. In the latter, the controller typically exploits pub-sub Message-Oriented Middleware (MOM) to deliver events to nodes.

Executor Nodes are agents installed on the different participating physical nodes and are in charge of hosting and putting into execution functions (typically the business logic), for each incoming event. In particular, the process responsible for waiting for events and for starting function invocation is the so-called *Invoker* or *Watchdog* [13]. Functions are executable codes expressed in one of the supported languages, e.g, Java or Python, and uploaded by customers ahead of time in conjunction with a configuration. The configuration is the set of information exploited by the FaaS platform to know to which external events the execution of a function is associated and in which execution environment. The execution environment does not only specify the language framework but also the possible dependencies, the operating system, and the architecture on which the uploaded function code has to be put into execution. The combination of a function and its configuration constitutes a specific *workflow* deployed on the FaaS platform and externally invocable through a trigger.

III. THE TEMPOS MIDDLEWARE FOR QOS-AWARE FAAS INFRASTRUCTURES

In this paper, we originally propose TEMPOS, a novel middleware, specifically designed and optimized for advanced QoS management in FaaS infrastructures, that hides the possible heterogeneity and complexity of edge deployment environments while providing a strong QoS separation among the workflows put into execution. For this purpose, the TEMPOS orchestrator coordinates and composes different technologies of prioritization and reservation available across the full support stack associated with the different virtualization layers involved in FaaS infrastructures deployed over the edge cloud continuum. Thanks to its complexity hiding, TEMPOS can be exploited in multiple diverse scenarios such as Smart Tourism, Industry 5.0, or Smart Agriculture. TEMPOS abstractions require only to define the business logic in form of a workflow with an associated QoS level (among the available ones). It is the TEMPOS middleware that asynchronously checks the QoS support of the targeted resources in the deployment environment components and updates the configuration of the single FaaS components accordingly. The current implementation of the TEMPOS middleware assumes that essential middleware and FaaS components are already installed and configured; the integration with existing orchestrator features for QoS-aware dynamic deployment is currently under research, as better detailed in Section VIII.

With easy extensibility and flexibility as guiding design principles, in order to cope with different application scenarios, deployment sites, and installed technologies, the TEMPOS architecture consists of three functional slices, namely Bridging, Delivery, and Processing; in addition, one TEMPOS component, called Controller, is used to orchestrate them (Fig. 2). Slices realize a different functional aspect of our solution and are designed to interact the one with the other only through a set of agreed interfaces (e.g., on predefined UDP ports). This creates a contract among slices and enables each one to be independent of the specific technologies or protocols (e.g., network stacks or process scheduling implementations) employed by the other slices.

A. CONTROLLER

The TEMPOS Controller configures and orchestrates both the TEMPOS slices and the activation/maintenance of customer-defined workflows. In particular, the Controller represents the endpoint, addressable by application

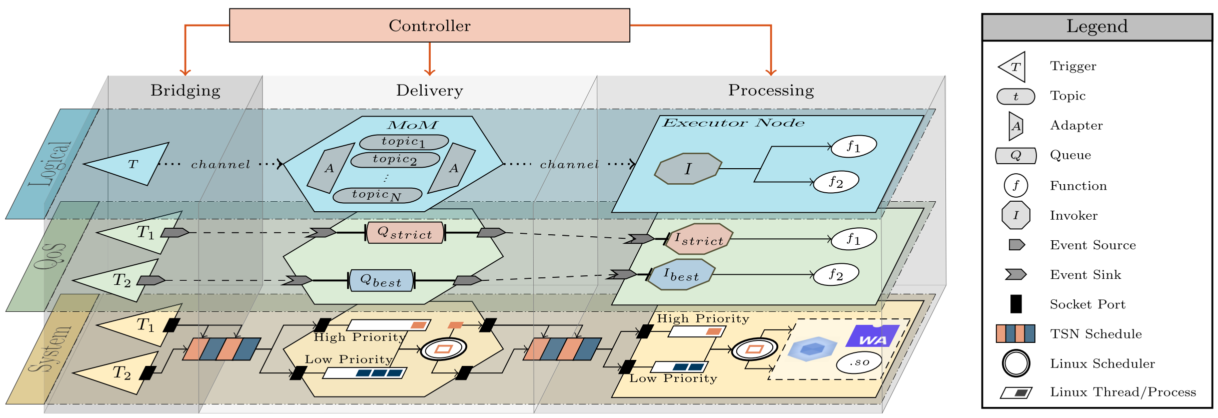


FIGURE 2. Multilevel representation of TEMPOS architecture foreseeing the three slices, namely Bridging, Delivery and Processing, and the three conceptual layers, i.e., Logical, QoS, and System.

developers, that exposes applicable configurations in a simplified and facilitated way. For this purpose, the Controller maps and matches the QoS requested by an application developer/deployer to the quality levels supported by the different slices and exposed through well-defined interfaces. More specifically and practically, the Controller handles two different configuration steps. On the one hand, it receives the configuration for all components of the TEMPOS infrastructure from the developer/deployer, then remaps this configuration to specific commands sent synchronously to each of the different slices. The goal of this phase is to configure Channels and Topics with the different QoS levels offered by our middleware (see Section III-B). On the other hand, the Controller receives the set of workflows initially defined by the developer/deployer, along with the QoS expressed with per-flow granularity (not for single invocation) and that will have to be mapped to the underlying infrastructure. The developer/deployer can also request the configuration of workflows at runtime, as the Controller exposes APIs for deploying new workflows or modifying existing ones. As in the other cases, these reconfiguration events are handled by the Controller, which interacts synchronously with the TEMPOS slices to preserve the QoS required for the whole application.

B. DELIVERY SLICE

The *Delivery* slice realizes QoS-aware event distribution among TEMPOS components. TEMPOS event distribution process is achieved through the interworking of different communication technologies and protocols, along with services in the duty of orchestrating and composing them. A series of abstractions are then introduced to easily extend the set of supported technologies and to provide developers/deployers with a simplified view. The core part of this slice is a novel MOM capable of dynamically exploiting different mechanisms and technologies to achieve QoS differentiation.

The introduction of our MOM decouples, in space and time, the interactions among TEMPOS components and enables advanced mechanisms such as load balancing and automatic fault tolerance. Our MOM introduces a transparency feature that allows adding and/or scaling dynamically the deployment of TEMPOS middleware components. The synergy between the TEMPOS MOM and Controller completely hides the internal complexity of our middleware from the application developers' perspective, thus achieving an essential feature of Serverless computational models.

Application/middleware components can connect to our MOM either to send or receive a message, through the creation of a *Channel*. The TEMPOS Channel is the abstraction that we offer to define a connection between any pair of TEMPOS components. Since the Delivery slice potentially covers several communication environments, a Channel is characterized by a specific communication protocol and, if supported, a prioritization or reservation technique. Therefore, to employ our delivery notion in highly heterogeneous contexts, the MOM adopts a mechanism based on the concept of Adaptor. Adapters allow to support a considerable number of Channels and interact with them seamlessly and simultaneously. Messages received by a specific Channel are processed in priority order through the use of "priority queues". Through these queues, the TEMPOS MOM processes events in parallel, prioritizing those associated with higher QoS.

To provide our middleware with a consistent end-to-end quality abstraction, we introduced the concept of *QoS-aware Topic* defined as:

$$T = \left(\begin{bmatrix} C_{in1} \\ C_{in2} \\ \vdots \\ C_{inN} \end{bmatrix}, Q \begin{bmatrix} C_{eg1} \\ C_{eg2} \\ \vdots \\ C_{egN} \end{bmatrix} \right)$$

where

$$T = \text{Topic},\ Q = \text{Priority Queue},$$
$$C_{in} = \text{Channel Ingress},\ C_{eg} = \text{Channel Egress}$$

The topic is the reference construct in TEMPOS for coordinating and abstracting the different QoS levels made available by the channels and associated with the priority queues of the MOM. Each topic is then associated with a specific QoS, which can be derived from the performance of the two channels with the worst input and output performance, respectively, and the processing performance associated with the processing slice. Thanks to the Topic and Channel constructs, it is thus possible to provide application developers with a single and transparent view, even if the platform is leveraging different QoS-sensitive technologies, such as TSN, 5G slicing, or Wi-Fi 6 prioritization, as detailed in the following.

To allow for differentiated message distribution policies among the subscribers of a TEMPOS Topic, we also inherited, from classical MOM solutions, the concept of Subscription Groups. In the classical pub-sub model, each subscriber of a topic receives all messages sent through it. Through the mechanism of Subscription Groups, each message is sent to only one of the subscribers in each group, thus realizing a load balancing feature among the subscribers. Indeed, load balancing is an essential capability of a MOM in the context of FaaS platforms as it enables the distribution of workflow requests across multiple executor nodes.

C. BRIDGING SLICE

The bridge slice is the abstraction responsible for unifying requests sent by external entities that want to utilize services distributed through the TEMPOS platform. The Bridge slice employs components and mechanisms that transform external events into an internal representation of the FaaS platform so that they can be managed by other slices and transparently processed by functions. In particular, users, devices, and services interact with the workflows through the central component of the Bridging slice, the *Trigger*. The main responsibility of the Trigger is so to forward to MOM every information sensed or received after having adapted and encapsulated them under the form of events. Trigger behaves as a bridge between the external world and TEMPOS, by adapting external protocols, representations, and QoS levels to internal ones. Moreover, *Trigger* is the first TEMPOS component that can differentiate and characterize event quality by exposing a different endpoint for each supported QoS level. Thanks to the location transparency introduced by the MOM, the deployment of triggers can adapt to different scenarios and needs: in particular, we designed and implemented three deployment options for Trigger, depending on closeness to either the MOM or the external source.

In the first case (Fig. 3 case A), the trigger is co-located with the event source. This case allows us to simplify the support of delivery quality between the source and the trigger as they are co-located on the same host. As a drawback, this pattern prevents the simultaneous use of the trigger for multiple sources and also requires that the source device has enough resources to host the trigger execution.

In the second pattern, i.e., Fig. 3 case B, the trigger runs in the middle between external sources and the MOM. In this scenario, we have the certainty that the delivery of information between the source and the trigger is feasible with

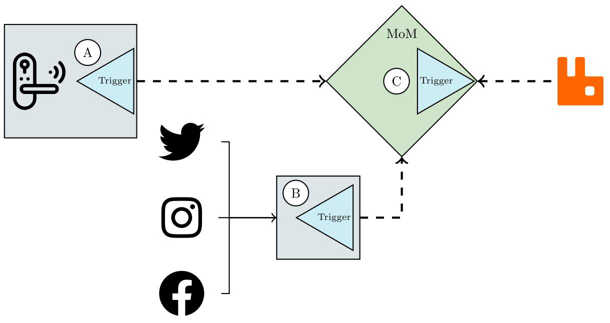


FIGURE 3. Different deployment options of Trigger: A) locally to source, B) in the middle between multiple sources and the MOM C) locally to the MOM as bridge to other event systems.

adequate quality. The trigger can be located in any node reachable by both MOM and sources and can behave also as a gateway between different networks. In this configuration Trigger is addressable by multiple sources, thus maximizing resource usage but also potentially causing conflicts. Of course, incoming events belonging to the same quality class can incur conflicts in case of concurrent transmission; a fine-grained distribution and allocation of Triggers are so advisable to avoid situations of quality degradation.

In the last case, Fig. 3 case C, the external source already provides QoS concepts and exchanges information in the form of events. This scenario embraces use cases where TEMPOS is deployed as part of an existing infrastructure that already leverages some form of event exchange, such as an Enterprise Service Bus infrastructure. In this context, Trigger is placed within the TEMPOS MOM and acts as a connector to external sources. Finally, Trigger is still tasked with mapping arguments, queues, and qualities of an external system onto internal ones.

D. PROCESSING SLICE

Processing is the last abstraction slice in our TEMPOS middleware and is responsible for processing events forwarded by the Bridging slice and then handled by the Delivery slice. Another task of the Processing slice is to take the burden off the customer of knowing both the characteristics of the processing environment and the computing resources used to execute a specific workflow. This slice allows the customer to define both the business logic and QoS requirements, without knowing how the platform implements the support that can satisfy them. Specifically, the processing is done through user-defined business code that is loaded in advance.

The main component responsible for the Processing slice behavior is the *Invoker*. The Invoker is the terminal part of each output channel and waits for the arrival of events to be processed by functions. At each event arrival, Invoker instantiates the associated function, ahead of time through the user-provided configuration, and then takes care of forwarding the event to the function.

No TEMPOS components, except the controller, are aware of how an event will be processed by a specific Invoker;

Invoker is, therefore, the component in charge of managing the life cycle, the execution environment, and the invocation of the functions in such a way as to reach the target QoS for that workflow. TEMPOS Invokers can be specialized to exploit different function invocation methods and execution environments depending on deployment scenarios and achieving better performance or resource-saving.

This specialization can take advantage of the opportunistic composition of different technologies available either in the environment of execution of the Invoker, e.g., Operating System, Hypervisor, or realized by the Invoker itself.

So, for its execution, the same workflow can exploit different technologies and optimizations at the same time, e.g. concurrent usage of an execution environment for 2 different functions or re-usage of the same function instance for subsequent requests. About QoS-aware processing, Invoker can employ both its internal techniques and the ones possibly present at the Executor Node, e.g, Operating System prioritization. Thanks to the Invoker abstraction, TEMPOS is capable of executing heterogeneous functions while employing different QoS mechanisms and policies, without causing side effects on other components or executor nodes.

IV. TEMPOS IMPLEMENTATION

This section presents the primary implementation insights about how we realized the TEMPOS architecture described in Section III, first focusing on the *QoS Level* and then on the *System Level* (Fig. 2).

A. QOS LEVEL

The current implementation of the TEMPOS middleware provides application developers with two distinct QoS levels. On the one hand, we define a *Best-effort Quality (BQ)* and assume its use in case of communication and function invocation with no strong latency and jitter constraints. On the other hand, we specify the *Strict Quality (SQ)* to support the execution of functions that require more stringent and soft real-time QoS.

B. CONTROLLER

The Controller is the module that manages all the other TEMPOS entities. We chose to develop the Controller as a Linux daemon process, which, once started, performs a first configuration of the entire TEMPOS middleware. For this initialization to be successful, the application developer/deployer must simply provide a specific configuration file, currently based on the TOML configuration file format, containing: i) all the information needed by the Controller to interact with all other entities, i.e., MOM, Triggers, and Invokers, and ii) the specification of the requested QoS levels for the connection between components and function execution at each targeted node. Then, at the end of the configuration phase, the Controller waits for reconfiguration/management requests from the developer/deployer, thus making both the Controller and the entire middleware reconfigurable and modifiable at run-time. The current implementation exposes the Controller functionality through REST APIs. In particular, every time the Controller receives a request, it performs the possibly needed reconfiguration by interacting synchronously with the entities involved in each slice. The latter, in turn, exposes specific management interfaces and manages these configuration requests in an ad-hoc process outside the interactions of the TEMPOS workflows defined by the developer/deployer. As already planned future work, we are starting to implement this configuration mechanism through a special configuration topic on the MOM where the different TEMPOS components can subscribe to receive updated configurations. Finally, the Controller maintains an internal representation of all TEMPOS components, which is updated with each request, thus allowing a centralized and updated view of the entire middleware deployment environment. Therefore, in the configuration phase, there is no need of direct interaction between the different TEMPOS components because the communication is mediated by the MOM to guarantee a strong decoupling between our infrastructure entities.

C. MESSAGE-ORIENTED MIDDLEWARE

Following the architecture proposed in Section III, the second TEMPOS macro-component is a Message-Oriented Middleware with two different queues, for SQ and BQ, respectively. We developed these two queues using two different network sockets and two threads. The sockets separate messages into two separate queues, while each thread acts as a priority queue processor since both are scheduled according to the real-time Linux scheduler [14]. The first thread handles all messages labeled with strict quality and runs with a higher priority than the best-effort thread. Since the priorities provided by the scheduler range from 0 to 99, as default, we use the lowest priority (0) for BQ, while we associate the highest priority (99) with SQ. Moreover, an application developer can specify to use the Controller to choose the type of Linux real-time scheduler, e.g., Round Robin or FIFO, and set different values for the priorities of the threads associated with the queues. That makes the MOM more flexible and, in the future, opens up to the easy introduction of additional queues with intermediate quality.

A significant aspect of our MOM is its transparency of the protocols used by the underlying network; this property is achieved thanks to the introduction of TEMPOS middleware elements called Adapters (Section III) and realized via a plugin-based mechanism within the MOM. A plugin represents a set of well-defined interfaces, which specify how to: i) open a connection, i.e., create a Channel, ii) configure the QoS level of a newly created connection, iii) send messages through the Channel, and iv) safely close the connection. In addition, the association between one or more channels connected to the MOM and one of the queue processors realizes the concept that we name TEMPOS Topic. Since the TEMPOS components, including the MOM, are entirely implemented by using a compiled language such as Rust, we based the plugin system on the dynamic library loading mechanism [15]. The realized plugins must be compiled and distributed as shared libraries, which are then loaded at run-time by the MOM according to the configuration received from the Controller.

At the time of writing, we completed the implementation of a TSN-based plug-in. Specifically, we based our implementation on the IEEE 802.1Qbv standard, which aims to support the combination of best-effort and real-time traffic within TSN networks [16]. The standard presents the notion of time-triggered communication windows, often called time-aware traffic windows, thus defining a mechanism to support different types of time-critical flows. In practice, a window is divided into multiple time slots associated with selected classes of traffic and repeated cyclically. This makes it possible to minimize the interference of best-effort traffic with priority traffic (i.e., real-time traffic), which we refer to as strict communication QoS level. This mechanism is achieved by inserting a so-called guard band before the scheduled traffic window, which forces the buffering of packets belonging to traffic classes not to be transmitted. From the developers and implementation-oriented perspectives, windows and slots are expressed through a Gate Control List (GCL) that identifies the time instants when packets can be transmitted on the medium [17].

D. TRIGGER

The Trigger is the TEMPOS component responsible for the forwarding of events issued by one or more sources to the MOM, thus defining the core part of the *Bridging Slice*. To provide a unique implementation for the different deployment scenarios presented in Section III-C, we developed the trigger as an always running Linux process listening on a network socket. Thus, an event issued by a source (e.g., a sensor) is received by the trigger via the network, or if possible, taking advantage of an IPC mechanism. This can be applied to optimize communication in the case of the co-located deployment scenario. Once executed, the Trigger first receives the configuration, containing the QoS level to be used, from the Controller, and then opens a second connection, i.e., the *Channel* used to communicate with the MOM. Different from the MOM, we consider the implementation of each Trigger as limited to a single protocol, be it TSN, Wi-Fi 6, or any other protocol providing a priority-based communication mechanism. At the moment, we have completed the implementation of the co-located trigger model by exploiting TSN-based communication.

E. INVOKER

Invoker is implemented as a multi-threaded TEMPOS component, which spawns two main threads. The first one manages configuration requests from Controller, while the second handles actual invocations. Once started, Invoker receives its configuration from Controller and sets its QoS level based on the application developer's specification. Moreover, we developed three distinct invocation mechanisms, and the developer must express which one to use in the configuration request. The three invocation methods are designed to support different use cases, and are defined as follows:

1) DLF (DYNAMICALLY LOADED FUNCTION)

We based our DLF mechanism on the *dynamic library loading* technique. This is generally used to combine several functions into a single unit shared by multiple processes at run-time, thus saving disk space and RAM. Although the library code can be used by multiple processes at the same time, its variables remain isolated. Our DLF employs the POSIX standard APIs to handle the dynamic library loading [18]. When a function invocation occurs, Invoker opens (`dlopen`) the requested shared object file and, subsequently, loads the symbol (`dlsym`) related to the main library entry. For this to happen, the application developer must expose the function within the library with the name and arguments we expect. Consequently, the loaded function is first executed and finally unloaded (`dlclose`) after its termination. Due to the mechanism involved, this invocation method is suitable for executing functions with high performance and strict requirements, thus primarily aimed at our strict QoS level [19].

2) WASMF (WASM FUNCTION)

The WASM invocation method is similar to DLF, as it adopts the same underlying loading mechanism. The TEMPOS invoker integrates a complete WASM engine, i.e., using the Wasmer library [20], initialized in the startup phase. When Invoker receives a request, the engine dynamically loads the shared library containing the requested function. To ensure a correct loading, the library must be compiled using a WASM code generator that translates a target-independent intermediate representation into executable machine code, e.g., Cranelift [21]. Once the function is loaded and executed, the engine removes the WASM code on its internal store. An advantage of this invocation method is the possibility of the application developers to implement functions in the programming language of their choice, still providing good results in terms of performance and levels of quality.

3) FSPAWN (FUNCTION SPAWN)

The last invocation mechanism, called Function Spawn (FS), follows the classic Unix idiom of `fork()` followed by `exec()` to execute a different program in a child process. If the invoker is deployed on a node supporting the `posix_spawn` API, the latter is used instead of the `fork()` and `exec()` scheme to achieve better performance in case the parent process has a larger size or memory layout [22]. Due to the flexibility and standard nature of the mechanism employed in this invocation method, it is possible to execute the function in arbitrary environments. In particular, as a first implementation, we leveraged FSpawn to execute function, deployed in the form of an executable program, directly as a Linux user-space process. Alternatively, a function can be spawned inside an already started Docker

TABLE 1. Specifications of the nodes used for the evaluation testbed.

Node Tag	Model	CPU	Memory	TSN driver
A	Custom Workstation	AMD Ryzen 3700X 8/16 CPU	32 GB	1 × Intel I211
B	Dell Optiplex 3010	Intel Core i5-3470 4/4 CPU	10 GB	-
C	UP Core Plus board	Intel Atom E3950 4/4 CPU	8 GB	4 × Intel I210
D,E	UP Xtreme board	Intel Core i3-8145UE 2/4 CPU	8 GB	4 × Intel I210

container providing all the dependencies needed to execute the code.

V. EXPERIMENTAL TESTBED DESCRIPTION

To quantitatively evaluate and validate the effectiveness of TEMPOS, we developed a series of testbeds to analyze the behaviours of several of its primary components. The tests described below aim at demonstrating that the TEMPOS middleware can support differentiated end-to-end QoS levels while offering to application developers a simplified interaction and instrumentation interface. Special attention was given to demonstrating the ability of TEMPOS to orchestrate and compose different mechanisms to achieve highly differentiated QoS for different workflows. Nevertheless noteworthy, these testbeds also show the validity and feasibility of the achieved TEMPOS implementation stack under very different load conditions.

Of course, the performance results achievable by TEMPOS in absolute terms depend on the characteristics of resources in the targeted deployment environment. Therefore, the following series of testbeds can also constitute the first step to calibrate resources in target deployment scenarios with similar technological stacks. Our testbeds are designed to simulate a worst-case where the number of concurrent requests is putting under stress the TEMPOS middleware. In particular, we organized our testbeds in three cases, aiming each one to stress alternatively the event-delivery process, the event processing, or the overall middleware.

In the first case, we test the behaviour of TEMPOS event-delivery under different load conditions, thus emulating diverse resource competition scenarios of workflows. The specific goal is to demonstrate the ability of our middleware to chain different QoS mechanisms while maintaining guarantees about latency and jitter. The second case aims at demonstrating the TEMPOS ability of hiding heterogeneity while still providing a strong differentiation of QoS. For this reason, in this testbed case, we trigger the execution of a complex and computational heavy function, representative of many common workloads (Algorithm 1), while employing all the different function invocation methods currently supported in our TEMPOS prototype.

In the last testbed case, instead, we aim at verifying the ability of TEMPOS of composing mechanisms for QoS at different TEMPOS slices, to achieve configurable and complete end-to-end QoS over different workflows.

We have implemented and configured two workflows, invoking the same function (Algorithm 1) and configured one

Algorithm 1 Pseudo Code Showing the Operations Performed by the Function Used in the Tests: Deserialization, Count of Occurrence in Text, and Repetitions of Operations of Square Root and Power Based on the Index Value

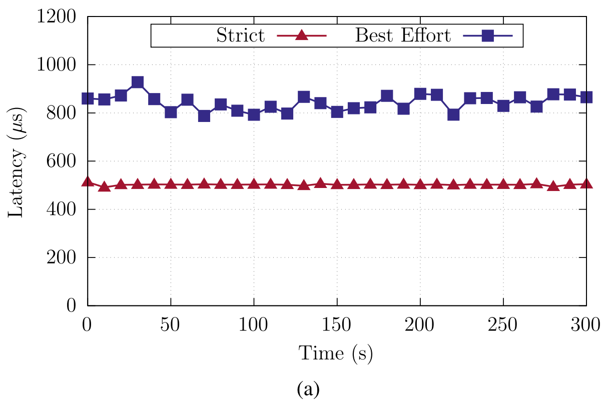
```
1:  function main($e$ : $Event$)          ▷ The function entry point
2:      $message, pattern \leftarrow$ deserialize($e$)
3:      $occur \leftarrow$ count_occurence($message, pattern$)
4:      $res \leftarrow 0$
5:      for $i \leftarrow 0, occur$ do
6:          if $i \bmod 2 = 0$ then
7:              $res \leftarrow res +$ pow($i$)
8:          else
9:              $res \leftarrow res +$ sqrt($i$)
10:         end if
11:     end for
12:     output($res$)
13: end function
```

with BQ level and the other with SQ. All tests foresee an increasing number of requests for each workflow, to show the TEMPOS behavior in presence of challenging dynamicity in the supported service load. The reported results are discussed and analyzed by presenting the overhead quotas introduced by single TEMPOS components.

To assess and validate TEMPOS feasibility over edge cloud deployment environments, we have decided to conduct our test on nodes with limited computational resources (Table 1). As the edge hosts, we have employed three TSN-enabled nodes. In particular, Node A and Node B have been introduced and exploited only during the second testbed case to verify how invocation methods performance would variate in correlation with node performance.

We opted to co-locate Triggers and data Producer on Node E, thus emulating a practical case where two edge nodes cannot communicate by employing differentiated QoS mechanisms. The choice of assigning one of the two resource-rich nodes to these TEMPOS components is mainly due to the need to generate high and precise loads to stress our middleware. The second most performant board, node D, hosts an instance of the TEMPOS MOM. In addition, we decided to deploy all the invokers on the node with fewer resources to emphasize concurrent resource requests and potential QoS conflicts in the processing slice, which is a practically recurrent situation.

The three nodes of our testbed are connected through a Relyum RELY-TSN-BRIDGE Ethernet switch, configured



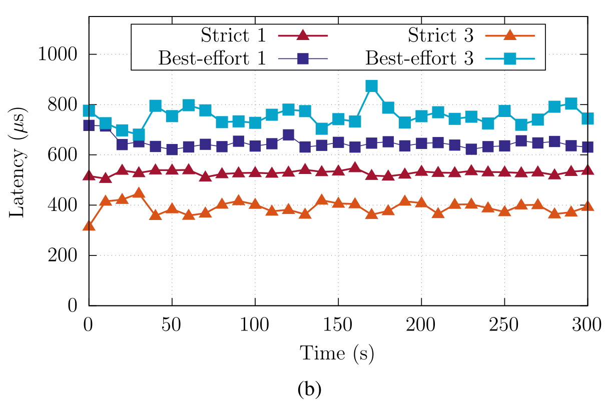


FIGURE 4. First testbed section showing performance of Delivery slice. (a) Average end-to-end latency of best effort and Strict effort traffic when executed in separated environment. (b) Average end-to-end latency of best effort and Strict effort traffic with 1 and 3 concurrent best effort producer and 1 strict effort.

with a TSN setup realizing differentiated QoS channels for the ingress/egress of topics. To set up the selected QoS mechanism for TEMPOS best effort and strict effort levels, we use two new *qdiscs* queuing disciplines built into the Linux kernel: i) *taprio* (Time-Aware Priority Shaper) implementing a simplified version of the scheduling defined by IEEE 802.1Qbv and ii) *etf* (Earliest TxTime First) qdisc that allows applications to set a transmission time for each packet (this information is then used by the scheduler to de-queue the packet and forward it over TSN). Note that applications based on the IEEE 802.1Qbv standard must rely on a single time reference: to this purpose, an autonomous standard called IEEE 802.1AS is specified in the TSN context, which defines a specific profile of the IEEE 1588 standard by extending the Precision Time Protocol (PTP). This extension, called generic Precision Time Protocol (gPTP), defines two main entities, namely the Clock Master (CM) and the Clock Slave (CS), associated with the devices in the network [16]. About our testbed synchronization, each TEMPOS node participates to elect a controlling entity, determined by the Best Master Clock Algorithm (BMCA): this controlling node is called the PTP grandmaster [17]; the grandmaster sends clock information to each of the Clock Slaves connected to it; once all TEMPOS devices are synchronized, we have what is effectively a time-aware network of nodes, i.e., a ready-to-use gPTP domain.

In our testbed, we created two time-aware TSN windows of 1 ms, i.e., between Trigger and the MOM, and between the MOM and Invoker. Each window is divided into two time slots, one for SQ and one for BQ, each of 500 µs; the first SQ slot is scheduled in the first half of the first window, where the second SQ slot is skewed of 300 µs concerning the starting time of the MOM-invoker window; this configuration enables strict TEMPOS traffic to find the gate open at each step, with no additional delays. Finally, to gather monitoring statistics and to evaluate TEMPOS performance, in each TEMPOS component in the testbed we introduced the logging of any received event id, associated with its synchronized timestamp; those logs are collected and analyzed only offline at the end of the tests, not to perturb the performance of workflow execution.

VI. IN-THE-FIELD EXPERIMENTAL PERFORMANCE RESULTS

A. EVENT DELIVERY

In the first testbed case, we aim at demonstrating the TEMPOS ability to prioritize event delivery based on workflow QoS. In particular, in the first test, we submitted a constant rate of 1000 events per second to Trigger for a time-lapse of 5 minutes. Then, we measured the difference between the timestamp corresponding to event creation at Trigger and the one reported at its delivery. We alternate the activation of SQ and BQ workflows to observe the behaviours of the two in a scenario with no perturbation due to concurrency. The results in Fig. 4a show that the events belonging to the SQ workflow are characterized by a lower end-to-end latency and jitter when compared with those of the BQ workflow. In particular, end-to-end latency for SQ workflow events settles to 501 µs on average, thus showing that TEMPOS is compatible with very challenging contexts that call for less than 1 ms response time, like soft real-time ones.

Note that a clear differentiation between TEMPOS QoS levels is possible thanks to the combined exploitation of prioritization mechanisms acting at the network layer and the event processing layer. In particular, the lower jitter is mainly due to the strict scheduling of events and the synchronization of TSN windows in ingress and egress of the topic. In fact, the maximum latency that we measured throughout all tests for each hop is 223 µs for the delivery of one event to the TEMPOS MOM, 57 µs for event processing, and 299 µs for event delivery to Invoker. Overall, once transmitted by Trigger, a packet reaches Invoker in no more than 700 µs, in full compliance with what is configured as the QoS request in the testbed setup.

Finally, let us observe that the high priority assigned to the queue processor for SQ events prevents other applications in the user space running at the edge node (such as the MOM control thread) to steal resources to event processing; of course, this does not happen for BQ. Note also that these measurements could be considered as the baselines for event delivery by TEMPOS in the ideal case of absence of perturbations.

In the second test of this testbed case, we investigate how the TEMPOS event-delivery mechanisms behave when multiple workflows are active and in competition for resources. This test consists of two rounds: i) 1 best QoS and 1 strict QoS workflows are concurrently active and ii) the number of active best QoS workflows is increased to 3. We decided to increase only the number of best-QoS workflows in this test because the configuration of the current testbed makes impossible the simultaneous sending of more than 1000 strict messages per second, moreover, in most practical scenarios, most events tend to belong to the Best-quality type. We submitted for 5 minutes a constant rate of 1000 events per second per each active flow and again we measured the time-lapse between the creation of the event in Trigger and its delivery at Invoker. The results in Fig. 4b demonstrate that the strict-quality latency is not penalized by the concurrent execution of one or more best-effort workflows even when compared with the baselines of Fig. 4a. In both rounds, the latency constantly remains under the threshold of 600 µs. As a second-level observation, note that in the first round, despite the concurrent presence of two active flows of event delivery, the jitter is negligible but a noticeably increase is observable in the second round: this is mainly due to our usage of new API (NAPI), a device driver packet processing extension to improve the networking performance; in particular, NAPI implements a mechanism of interrupt mitigation for network devices. This mechanism allows the network card driver to exploit two different packet reception modes: i) interrupt request (IRQ) issued for each incoming packet, and ii) a polling [23] based mechanism. Since the IRQ-based implementation can be very inefficient in high-speed networks as it constantly interrupts the kernel, NAPI introduces the polling mechanism that allows the kernel to periodically check incoming network packets without being interrupted. When incoming packet datarate is sufficiently high for NAPI, then it automatically switches to polling-based mode, thus motivating the observed behavior [24].

The periodic activation of the NAPI polling mode allows us to achieve a substantial acceleration in terms of latency, at the expense of jitter and CPU utilization. In the case of workloads more sensitive to jitter than latency, the disabling of this feature is recommendable; TEMPOS can perform this disabling transparently for application developers thanks to its abstractions.

B. PROCESSING

After validating the QoS-constrained delivery features of TEMPOS, here we present a series of tests to show the TEMPOS performance in terms of event processing. The following tests are therefore implemented by considering the Processing Layer only, with local-to-nodes function triggering.

The first test is focused on how different methods of invocation and execution environments perform when run over heterogeneous hardware. To this purpose we consecutively invoked the same function (Algorithm 1), programmed in a compiled language, for 2 minutes when invoked with the mechanism of i) *DLF*, ii) *WASMF* and iii) *FSpawn*. We next repeated the test with the *FSpawn* mechanism but with two different versions of the same function implemented in two different interpreted languages, i.e., Python and JavaScript. These tests are repeated on nodes A, B, and C (Table 2) as representative of three very different cases of resource availability on edge hosts. All the results show that i) startup and execution times are sensibly influenced by the employed hardware and ii) latency minor than 1 ms is easily achievable on medium-top class hardware. *DLF* with execution duration near to 100 µs qualifies as the fastest mechanism to invoke functions; this opens up to the application of TEMPOS in many challenging and latency-sensitive use cases where sub-millisecond end-to-end latency is needed; however, DLF restricts the usable programming languages to the only ones compatible with the generation of shared libraries.

The *FSpawn* execution, on the contrary, showed maximum flexibility, being able to run every language executable in a Linux environment. However, it exhibited the worst performance in terms of total execution time, with latency up to hundreds of milliseconds, in particular when running non-compiled languages (Fig. 5). This qualifies FSpawn as a good mechanism to adopt in a FaaS platform given its flexibility, but its measured performance makes it infeasible

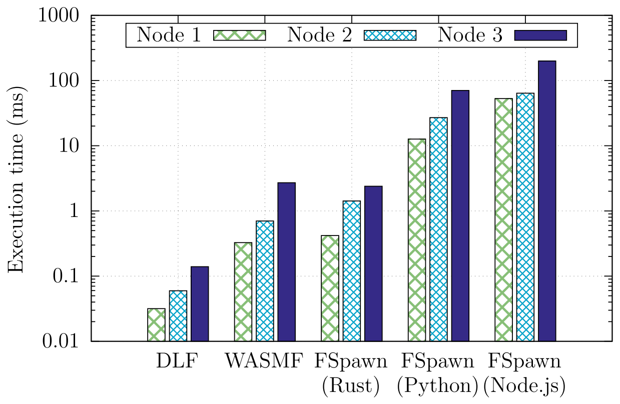


FIGURE 5. Mean execution times for the different invocation methods gathered in a run of 5 min. Each run repeated on nodes A, B, and C.

TABLE 2. Number of invocations executed by the different invocation methods during the processing test (5 min. run).

Invocation Mode	Node A	Node B	Node D
DLF	1884×10^3	1004×10^3	428×10^3
WASMF	183×10^3	85×10^3	22×10^3
FSpawn (Rust)	142×10^3	42×10^3	25×10^3
FSpawn (Python)	5×10^3	2×10^3	855
FSpawn (Node.js)	1×10^3	940	303

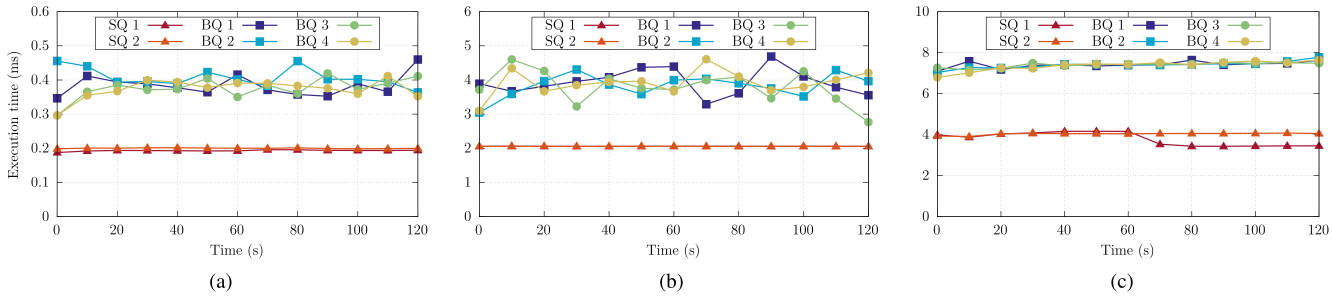


FIGURE 6. **Testbed results of concurrent invocation of functions configured with different QoS. (a) Test results using the *Dynamic Loaded Functions* invocation model. (b) Test results using *Function Spawn* invocation model. (c) Test results using *WASM Function* invocation model.**

to use in deployment scenarios where end-to-end latency needs to be below the 1 ms threshold.

The execution through *WASMF* performed one order of magnitude worst than *DLF* and only slightly better than the execution of a compiled function with *FSpawn*, with an execution time of the order of 1 ms. However, this mechanism showed the potentiality of sensibly reducing the execution and startup time of many non-compiled languages.

Table 2 shows that the choice of the right invocation mechanisms also results from a trade-off between the freedom in implementation language selection and the number of executable functions on given hardware infrastructure. Also, in this case, the results of this first test work also as a baseline for the successive results because the first test was conducted without concurrency among workflows.

In the second test, we separately experimented again with the three invocation mechanisms (i.e., DLF, WASMF, FSpawn), but this time with concurrent invocations of 6 functions, 2 executed with SQ setup, and 4 with BQ. The level of parallelism selected is motivated by the number of cores available on the used nodes (Node C Table 1) and the need of creating challenging resource conflicts among workloads in our tests. As shown in Fig. 6 our queuing mechanism can well prioritize strict-quality when resource conflicts occur: the time of execution of SQ functions is almost half of the BQ ones. In other words, Invoker demonstrates to be capable of correctly applying the requested prioritization even with heterogeneous mechanisms and in different execution environments.

In addition, the reported results highlight a negligible variability in execution time in the case of SQ functions, as opposite to BQ. BQ functions showed a significant variation of the execution time of the order of hundreds to thousands of ms depending on the method used; therefore, the usage of SQ functions enables, not only to achieve a more reduced latency but also for stricter predictability of processing time. Invoker showed to be capable of transparently executing heterogeneous workloads while exploiting diverse technologies present in infrastructure nodes.

C. FULL STACK

This section report results about the TEMPOS ability to coordinate and concatenate different QoS mechanisms available in each slice to achieve the targeted end-to-end quality for the workflows. We deployed on node E two data producers and two triggers configured with the two BQ and SQ levels; on node B, instead, we deployed 3 invokers with SQ configuration and 3 with BQ.

We then create and deployed two workflows executing the same function and triggered by the same event, but configured one with BQ and one with SQ. Next, we linearly increased the number of events submitted to the triggers until reaching 1000 events per second for each workflow. The experiment is repeated firstly with only one active workflow, then with both workflows concurrently active.

As predictable from the results of the previous sub-section, in an isolation case with only one workflow active per time, the SQ end-to-end latency is considerably better, with an average of 3.34 ms than BQ, which settled to an average of 3.96 ms, as also shown in Fig. 7a. It is also noteworthy that this behaviour is maintained for the entire duration of the test, with different rates of requests, thus demonstrating the elasticity of the TEMPOS middleware. In the concurrent scenario, with both workflows active and competing for resources, the two workflows coexist and do not affect each other's performance until reaching the critical threshold of 500 messages per second. Until this threshold, we can also observe that both workflows behave similarly as in the previous experiments where they were executed separately. Over the critical threshold, we can observe that conflicts among workflows become critical and the BQ workflows progressively degrade their performance. Note that the latency performance of SQ workflows remains consistently approximately of 3.1 ms despite the constrained hardware adopted and the concurrency with other workflows.

Zooming in on the performance behaviour of some single TEMPOS components, we can observe (Fig. 7b) how QoS mechanisms are correctly applied across all the hops of the technological stack. In fact, we can observe how, in each trait of the invocation stack, SQ performs almost identically when executed in concurrency with other workflows, while BQ workflows degrade their performance when competing with other active workflows. Let us finally note that in Fig. 7b the ‘‘Best Conc’’ MOM-Invk bar is almost the same as the ‘‘Invok’’ bar because the time is taken as the difference between invoker function invocation instant and the

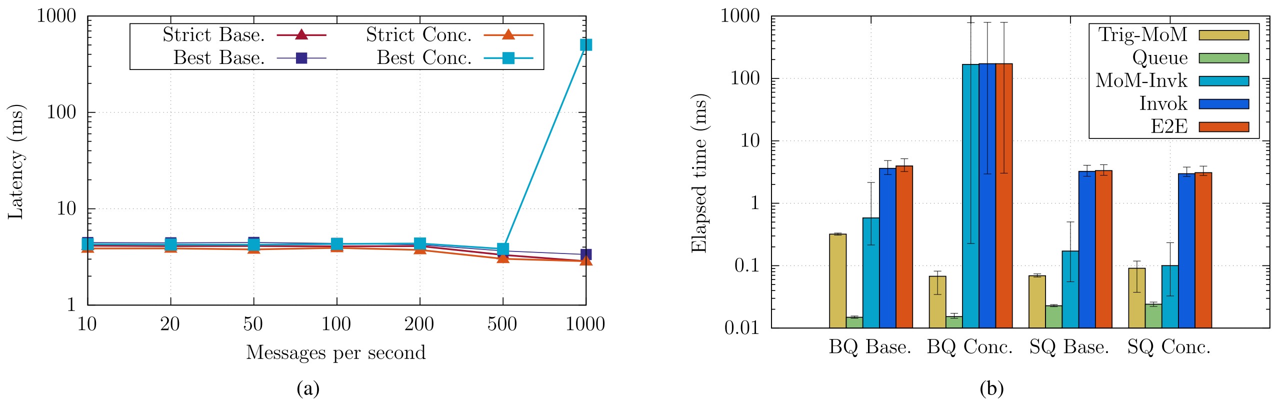


FIGURE 7. **End-to-end test performance of the TEMPOS platform operated in two different scenarios: isolated execution of workflows with different QoS (1 BQ and 1 SQ), and concurrent execution of them. (a) Comparison of end-to-end latency averages for BQ and SQ traffic executed both separately and simultaneously with increasing number of messages/seconds (from 10 to 1000). (b) Zoom-in on end-to-end latency results showing single contributions of TEMPOS components (execution time) to the overall response times.**

sending message instant from the MOM: given that the invoker reception is sync-blocking, that message anyway waits in the invoker socket until the previous invocation is completed.

VII. RELATED WORK

To the best of our knowledge, the design and implementation of a middleware able to exploit and coordinate different QoS mechanisms across the stack of virtualized FaaS invocations for the cloud continuum are completely novel in the existing literature. However, several works have proposed solutions to some of the challenges addressed by our proposal. Many of them not only paved the way for the development of TEMPOS but also have inspired some architectural and technological choices that we adopted. The following concise section aims only to be a representative excerpt of the most influential published research papers related to that.

The opportunistic usage of edge cloud resources to improve latency and jitter has been extensively discussed in [25], [26] and also represents one of the key factors pushing for wide adoption of this computing model [27].

The coordination and coupling of different prioritization mechanisms is not a recent issue but, with the recent advent of next-generation networking, it has gained an increasing research interest. The need for concatenation of mechanisms present at different levels of the stack has been considered a primary problem since the earliest distributed systems. To tackle resource orchestration and partitioning while guaranteeing QoS levels at the edge, [28] proposes DRAGON: that paper describes some implementation insights about DRAGON and evaluates its performance benefits if compared with traditional orchestration approaches.

The introduction of middleware for the concatenation of QoS-aware composition mechanisms is a frequent design pattern applied in the literature to reduce complexity. In [29] the authors propose a technique to couple priority and reservation-based OS and network QoS management mechanisms through Distributed Object Computing middleware, with adequate performance results. In [30] the authors present a middleware built on CORBA for providing distributed soft real-time applications with a uniform API to reserve heterogeneous resources with real-time scheduling capabilities in a distributed environment: that solution introduced uniform interfaces to support the reservation of CPU, disk, and network bandwidth on Linux systems.

Even if Serverless computing and in particular FaaS platforms are relatively novel, some platform improvements have already been proposed in the literature to achieve better FaaS performance and in particular latency reduction. Some papers have proposed the deployment of serverless platforms on edge nodes to achieve better QoS [31], such as in TEMPOS. The usage of different invocation methods to speed up function startup has been proposed as the exploitation of cross-compiling to achieve faster executable. For example, in [32] the authors propose *Faaslets*, an isolation abstraction that exploits WebAssembly to achieve good isolation and fast function startup; they also propose an additional optimization with a mechanism to restore from already initialized snapshots, thus improving platform throughput and tail latency. In the proposed project Catalyzer [33] the authors propose a serverless sandbox system to enhance function startup and isolation. To provide fast startup, Catalyzer exploits a checkpoint mechanism to skip initialization and a new OS primitives to reuse the state of the running sandbox; this results in a relevant reduction of the startup time of function invocations, up to less to 1 millisecond in the best cases.

VIII. CONCLUSIVE REMARKS AND FUTURE WORK

In this work, we proposed TEMPOS a QoS-aware middleware for serverless platforms which employs and coordinates different QoS mechanisms provided by individual technologies. Leveraging on virtualized FaaS invocation stack in the cloud continuum, TEMPOS is capable of properly managing end-to-end QoS in terms of jitter, latency, and enqueuing time. Therefore, to evaluate the validity of TEMPOS, we presented a series of real testbeds to extensively assess the state-of-the-art implementation of a TEMPOS prototype. The latter mainly exploit Linux real-time scheduling, a novel MOM with differential priorities, and Time-Sensitive Networking (TSN) protocols as underlying system-level mechanisms to apply TEMPOS QoS-aware abstractions (TEMPOS QoS-aware topics) for QoS management.

The results show that TEMPOS strongly differentiates workflows based on the assigned QoS level. Specifically, TEMPOS allows the SQ workflow to maintain an end-to-end latency that is 1 millisecond lower than the BQ workflow, throughout the isolated test. In addition, the SQ flow maintains a stable latency of 3 ms even during concurrent testing, while the BQ averages 600 ms under heavier workloads.

QoS awareness is preserved across the entire invocation stack with the delivery layer able to achieve nearly twice the performance for event delivery leveraging SQ workflows compared to BQ workflows, even under concurrent execution. Finally, the TEMPOS processing slice leverages multiple invocation methods seamlessly, ensuring that higher priority (SQ) workflows execute twice as fast as lower priority (BQ) workflows.

As future work, we are planning to integrate TEMPOS with a novel resource orchestrator for the full cloud continuum chain, e.g., up to 5G micro-datacenters and traditional geographically distant cloud datacenters, able to fully handle both network [34] and computing [35] resources. In addition, we aim to introduce new levels of QoS considering not only latency and jitter differentiation but also semantic of delivery and throughput while expanding support to resources not considered in this work such as storage or hardware accelerators.

REFERENCES

[1] Z. Ke, S. Leng, Y. He, S. Maharjan, and Y. Zhang, "Mobile edge computing and networking for green and low-latency Internet of Things," *IEEE Commun. Mag.*, vol. 56, no. 5, pp. 39–45, May 2018.

[2] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The Internet of Things, fog and cloud continuum: Integration and challenges," *Internet Things*, vols. 3–4, pp. 134–155, Oct. 2018.

[3] D. Buhalis and R. Leung, "Smart hospitality—Interconnectivity and interoperability towards an ecosystem," *Int. J. Hospitality Manage.*, vol. 71, pp. 41–50, Apr. 2018.

[4] (2019). *Indoor 5G Scenario Orientated White Paper—Huawei—GSA*. [Online]. Available: https://gsacom.com/paper/indoor-5g-scenario-orientated-white-paper-huawei-2019/

[5] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 4, pp. 2557–2589, 4th Quart., 2021.

[6] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[7] *Information Technology-Cloud Computing-Edge Computing Landscape*, ISO/IEC, Geneva, Switzerland, 2018.

[8] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan, "Quantifying the impact of edge computing on mobile applications," in *Proc. 7th ACM SIGOPS Asia–Pacific Workshop Syst.*, Aug. 2016, pp. 1–8, doi: 10.1145/2967360.2967369.

[9] L. Tang and S. He, "Multi-user computation offloading in mobile edge computing: A behavioral perspective," *IEEE Netw.*, vol. 32, no. 1, pp. 48–53, Jan. 2018.

[10] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications," *IEEE Access*, vol. 5, pp. 6757–6779, 2017.

[11] M. Agiwal, A. Roy, and N. Saxena, "Next generation 5G wireless networks: A comprehensive survey," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 3, pp. 1617–1655, 3rd Quart., 2016.

[12] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019.

[13] *Watchdog-Openfaas*. Accessed: Apr. 2022. [Online]. Available: https://docs.openfaas.com/architecture/watchdog/

[14] C. Scordino and G. Lipari, "Linux and real-time: Current approaches and future opportunities," Anipla, Scuola Superiore Sant'Anna, Pisa, Italy, Tech. Rep. str09-011, 2006. [Online]. Available: http://feanor.sssup.it/~lipari/courses/str09/lipari.pdf

[15] W. W. Ho and R. A. Olsson, "An approach to genuine dynamic linking," *Softw., Pract. Exper.*, vol. 21, no. 4, pp. 375–390, Apr. 1991.

[16] L. Lo Bello and W. Steiner, "A perspective on IEEE time-sensitive networking for industrial communication and automation systems," *Proc. IEEE*, vol. 107, no. 6, pp. 1094–1120, Jun. 2019.

[17] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. El Bakoury, "Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G Ull research," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 88–145, 1st Quart., 2019.

[18] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA, USA: Starch Press, 2010.

[19] C. Jung, D.-K. Woo, K. Kim, and S.-S. Lim, "Performance characterization of prelinking and preloadingfor embedded systems," in *Proc. 7th ACM IEEE Int. Conf. Embedded Softw. (EMSOFT)*, Sep. 2007, pp. 213–220.

[20] *Wasmer_Engine_Dylib—Rust*. Accessed: Apr. 2022. [Online]. Available: https://docs.rs/ wasmer-engine-dylib/2.0.0/wasmer_engine_dylib/

[21] *Wasmtime/Cranelift at Main. Bytecodealliance/Wasmtime*. Accessed: Apr. 2022. [Online]. Available: https://github.com/bytecodealliance/wasmtime/tree/main/cranelift

[22] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe, "A fork() in the road," in *Proc. Workshop Hot Topics Operating Syst.*, 2019, pp. 14–22.

[23] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *Proc. 5th Annu. Linux Showcase Conf. (ALS)*, 2001, pp. 88–95.

[24] *Networking: Napi Wiki*. Accessed: Apr. 2022. [Online]. Available: https://wiki.linuxfoundation.org/networking/napi

[25] C. Avasalcai, B. Zarrin, and S. Dustdar, "EdgeFlow—Developing and deploying latency-sensitive IoT edge applications," *IEEE Internet Things J.*, vol. 9, no. 5, pp. 3877–3888, Mar. 2022.

[26] J. Pan and J. McElhannon, "Future edge cloud and edge computing for Internet of Things applications," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 439–449, Feb. 2018.

[27] Y. C. Hu, M. Patel, D. Sabella, and V. Young. (2015). *ETSI White Paper #11 Mobile Edge Computing—A Key Technology Towards 5G*. [Online]. Available: www.etsi.org

[28] G. Castellano, F. Esposito, and F. Risso, "A distributed orchestration algorithm for edge computing resources with guarantees," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2019, pp. 2548–2556.

[29] R. E. Schantz, J. P. Loyall, C. Rodrigues, D. C. Schmidt, Y. Krishnamurthy, and I. Pyarali, "Flexible and adaptive qos control for distributed real-time and embedded middleware," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.* Cham, Switzerland: Springer, 2003, pp. 374–393.

[30] M. Sojka, P. Píša, D. Faggioli, T. Cucinotta, F. Checconi, Z. Hanzálek, and G. Lipari, "Modular software architecture for flexible reservation mechanisms on heterogeneous resources," *J. Syst. Archit.*, vol. 57, no. 4, pp. 366–382, Apr. 2011.

[31] L. Baresi, D. F. Mendonça, and M. Garriga, "Empowering low-latency applications through a serverless edge computing architecture," in *Proc. 6th Eur. Conf. Service-Oriented Cloud Comput. (ESOCC)*, 2017, pp. 196–210.

[32] S. Shillaker and P. R. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 419–433.

[33] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, H. Chen, and C.-G. Qin, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 467–481, doi: 10.1145/3373376.3378512.

[34] A. Garbugli, A. Bujari, and P. Bellavista, "End-to-end QoS management in self-configuring TSN networks," in *Proc. 17th IEEE Int. Conf. Factory Commun. Syst. (WFCS)*, Jun. 2021, pp. 131–134.

[35] A. Bujari, C. Bergamini, A. Corradi, L. Foschini, C. E. Palazzi, and A. Sabbioni, "A geo-distributed architectural approach favouring smart tourism development in the 5G era," in *Proc. 6th EAI Int. Conf. Smart Objects Technol. Social Good*, Sep. 2020, pp. 12–17.

ANDREA GARBUGLI (Student Member, IEEE) received the M.Sc. degree *(cum laude)* in computer science engineering from the University of Bologna, Italy, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include industrial IoT systems, end-to-end QoS management, and real-time communication protocols and middleware.

ANDREA SABBIONI (Graduate Student Member, IEEE) received the M.Sc. degree *(cum laude)* in computer science engineering from the University of Bologna, Italy, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include cloud, fog and serverless computing, middleware, and architectural approaches for smart tourism applications.

ANTONIO CORRADI (Life Senior Member, IEEE) received the Graduate degree from the University of Bologna, Italy, and the M.S. degree in electrical engineering from Cornell University, USA. He is currently a Full Professor in computer engineering with the University of Bologna. His research interests include distributed systems, middleware for pervasive and heterogeneous computing, and infrastructure for services and network management.

PAOLO BELLAVISTA (Senior Member, IEEE) received the Ph.D. degree in computer science engineering from the University of Bologna, Italy, in 2001. He is currently a Full Professor with the University of Bologna. His research interests include middleware for mobile computing, QoS management in the cloud continuum, infrastructures for big data processing in industrial environments, and performance optimization in wide-scale and latency-sensitive deployment environments. He serves on the Editorial Boards of IEEE COMMUNICATIONS SURVEYS AND TUTORIALS, IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, IEEE TRANSACTIONS ON SERVICES COMPUTING, *ACM CSUR*, *ACM TIOT*, and *PMC* (Elsevier). He is the Scientific Coordinator of the H2020 IoTwins Project (https: www.iotwins.eu).