DIFFUSE: a DIstributed and decentralized platForm enabling Function composition in Serverless Environments

Andrea Sabbioni, Lorenzo Rosa, Armir Bujari, Luca Foschini, Antonio Corradi

University of Bologna, Italy

Abstract

Serverless computing is an emerging proposition in the cloud offering landscape that promotes a higher level of abstraction, further decoupling software operations from the underlying hardware. Often recognized as an economically driven computational approach, the serverless model relies on the execution of short-lived stateless functions, enabling a fine-grained accounting and control of resources. In this context, function composition represents an appealing feature, allowing the composition of two or more functions to create tailored processing pipelines, incentivizing modularity and reusability of functions, while paving the way to application-specific run-time optimizations. This work presents DIFFUSE: a DIstributed and decentralized platForm enabling Function composition in Serverless Environments. DIFFUSE embodies an innovative infrastructural support, enabling the efficient and transparent composition of functions by relying on pluggable middleware support, serving as a conveyor of messages among the platform components. Broadening the deployment spectrum of our proposal, we assess different middleware solutions, each presenting distinct delivery profiles, evidencing the tradeoffs that emerge.

Keywords: serverless computing, function composition, shared memory, middleware, latency, distributed system

Email addresses: {andrea.sabbioni5}@unibo.it (Andrea Sabbioni), {lorenzo.rosa}@unibo.it (Lorenzo Rosa), {armir.bujari}@unibo.it (Armir Bujari), {luca.foschini}@unibo.it (Luca Foschini), {antonio.corradi}@unibo.it (Antonio Corradi)

1. Introduction

The increasing demand for ICT services has driven cloud providers to further differentiate their offerings, lowering the barrier of entry to services to better match the emerging market needs [1, 2]. As a result, we see an increasing trend toward solutions that guarantee a faster time to market by making the cloud easier to program and more economically viable [3].

In this context, Function as a Service (FaaS) embodies a recently new cloud computing model, tasking the customer only with the creation of the business logic, while transparently offloading to the platform all aspects such as scaling, deployment, monitoring, security [4] etc. The unit of execution in FaaS consists of a stateless function that is instantiated and executed in response to an incoming event, e.g., user request. This capability allows for fine-grained control and autoscaling of resources so that utilization closely matches the applications' demand.

On the other side, the ephemeral nature of functions poses several issues when fine-grained state sharing needs arise, demanding efficient, scalable and cost-effective storage solutions. At the same time, the opaqueness of functions makes it more challenging to implement efficient mechanisms for process coordination, such as broadcast, aggregation and shuffling, which are common communication primitives in distributed systems. This aspect is particularly relevant when considering machine learning and big data analytics workloads [3]. Despite these open issues, major hyperscalers have embraced the paradigm and alredy provide managed FaaS offerings like e.g., Microsoft Azure, Google Cloud Platform, Amazon AWS, etc., and many open-source projects are under active development as well [4, 5].

One novel concept rapidly developing across these platforms is the capability of composing (ready to use) functions to create application-specific processing pipeline(s). By decoupling complex functionalities into simpler ones, function composition enables smarter management of complex tasks and improved multiplexing capabilities. Moreover, function composability promotes reusability, thus further reducing the development burden, hence the time to market.

Unfortunately, current function chaining solutions exhibit some performance issues: response latencies can materialize not only from a bad user-defined chaining logic but also from inefficient infrastructural support to

function composition [6]. At the same time, FaaS solutions are not optimized to handle bursts of short-lived functions, an inherent property of this increasingly popular approach, that can amplify the overhead in the function invocation path [7].

In addition, no current FaaS platform adopts any resource-aware optimizations to exploit the specificity of the underlying hardware and/or software resources. This is becoming an increasingly important feature when considering that functionalities can be deployed over a continuum of heterogeneous resources [8]. These optimizations do not only benefit single-host FaaS deployments, but also distributed scenarios where one knows in advance the underlying resource capabilities and the application resource graph. In fact, this is not an uncommon situation, and the multi-host FaaS scheduler can be tasked to handle the resource-aware placement of functions [9].

Moving a step closer to addressing the issues, in [10] we presented a shared-memory middleware solution, exploiting function co-locality to improve on function-to-function communication performance in single-host scenarios. The approach was shown to vastly improve performance when compared to traditional application-layer constructs, however, system throughput is bound by the resources of a single-host environment. In this work, we generalize our approach and present DIFFUSE: a DIstributed and decentralized platForm enabling Function composition in Serverless Environments. DIFFUSE relies on pluggable middleware support acting as a conveyor of data among functions. Generalizing the prior communication mechanism, we propose the Distributed Shared Memory Queue (DSMQueue), a middleware solution embodying a similar operational semantic to the local counterpart. The scheme exploits modern network hardware support to enable the same zero-copy interaction that characterizes local shared-memory communication [11, 12]. The deployment of DIFFUSE is not confined to modern environments and can be easily mapped to commodity hardware by using traditional messages queue systems. To this end, we present comparison analysis contrasting the distributed shared memory middleware to alternative commodity configurations. The evaluation is carried out in a virtualized testbed environment employing different synthetic workloads, evidencing the tradeoffs that emerge.

The remainder of the paper is organized as follows. Section 2 provides a concise overview of FaaS, discussing the main approaches to function composition. Following is a brief synthesis of main elements of a distributed shared memory communication mechanism and associated system design

challenges. Section 3 compares our proposal with the state-of-the-art, outlining the main benefits and potentials. Section 4 describes our architectural approach to function composition and the use of shared memory for efficient inter-host communication. Section 5 presents an experimental assessment used to validate the system as-a-whole, evidencing the performance tradeoffs that emerge. Finally, in Sec. 6 conclusions are drawn and some directions for future work are discussed.

2. Background

This section provides an overview of the architectural schemes adopted by state-of-the-art serverless platforms, along with a concise taxonomy to function composition approaches. Next, follows a concise overview of some fundamental features of zero-copy inter-process communication and associated implementation challenges.

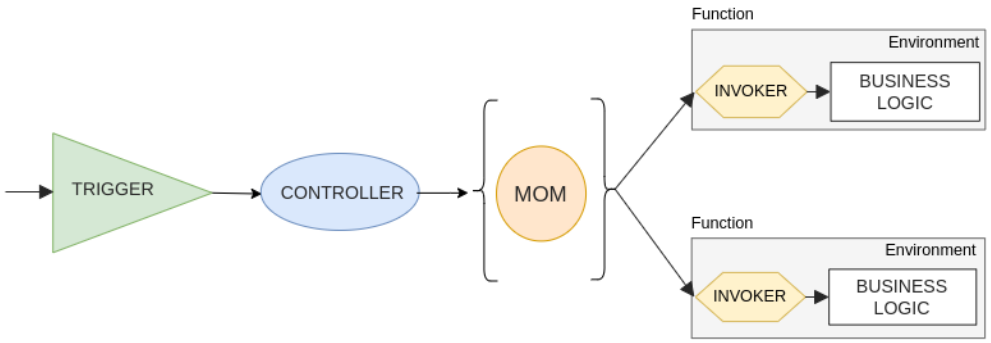


Figure 1: High level FaaS architectural approach. In a *MoM-based* setting, a middleware decouples the controller and the function, whereas in the *direct invocation* scheme the function invocation is enacted by the controller.

2.1. Function as a Service

FaaS is a novel paradigm of cloud computing where code, representing customer business logic, is dynamically put into execution in response to an incoming event. A FaaS platform adheres to the so-called *serverless computing* model, hiding to the users the infrastructure and the management thereof, tasking the consumers only with the creation of the business logic functionality.

As a consequence, the resulting programming model is inherently stateless as there is no guarantee that subsequent invocations of the same function will be executed in the same environment, or that the resources allocated for the function execution are not reclaimed after its termination. In this one-to-one mapping between functions and triggering events, FaaS platforms achieve finer-grained scalability: at any moment, the computational resources allocated to handle user requests match the ingress load.

In Figure 1 are depicted the typical components of a FaaS platform. The first element, directly interfacing with the incoming event, e.g., end-user request, is the *trigger*. This component receives external events from heterogeneous sources, via potentially different protocols, and converts them to local events for the FaaS platform. The events generated by the trigger are then managed by a *controller* which, based on configuration parameters provided by the user, forwards the event(s) to the proper function. Overall, the controller is tasked with the function lifecycle management.

Function constitutes the unit of execution in FaaS, encapsulating the business logic used to process specific events. A function is composed by an *environment*, e.g., Java, an *invoker*, and the *business code*. The invoker, also called *watchdog* [13], receives events, injects the business code deployed by the end-user, and successively launches the function to execution. The code is always executed inside a proper *environment* comprising all the required dependencies, e.g., system libraries.

Mainstream FaaS platforms implement the direct function invocation scheme either through a client/server pattern or through a pub/sub approach by exploiting a Message oriented Middleware (MoM) as an additional component of the architecture (Figure 1). Reliance on a pub/sub scheme allows the controller to be relieved from part of its responsibilities such as e.g., load balancing, event delivery etc., delegating them to the MoM [14].

2.2. Function Composition

Function composition, also referred to as function chaining in other research domains, is a pattern inherited from functional programming, whereby the output resulting from the execution of a function is forwarded as an input to another one, thus creating complex processing from the composition of simpler functions.

An efficient function chaining mechanism can encourage a decomposition of logical functions into simpler ones, enhancing code modularity and

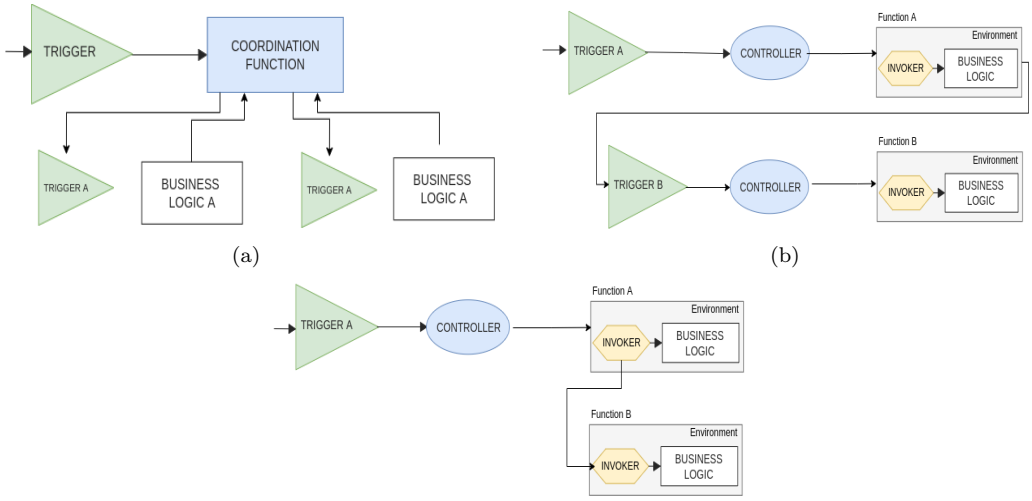


Figure 2: Function composition approaches. (a) Reflective invocation: a third entity coordinates the function invocation and forwarding of the result to the successive function in the chain. (b) Continuous Passing at the Business Layer: the business code directly invokes the next function in the chain through the associated trigger (c) Continuous Passing at the Infrastructural layer: the invoker is tasked to forward the output of the business logic to the next function in the chain.

reusability, while improving performance. From the literature, two main patterns emerge for implementing the composition logic in FaaS. The *reflective invocation* pattern relies on the use of a third entity to encapsulate the logic of function composition. The entity it tasked with the function execution, wait for its result, forwarding it to the next function in the chain, until all the processing units have been executed (Fig: 2a). In the *continuous passing* composition pattern, instead, the function can locate and name the next function in the pipeline, invoking it by forwarding the necessary input (Fig 2b) [6].

FaaS platforms implement these two patterns either at the business layer or as a built-in capability at the infrastructural layer. The implementation at the business layer foresees that each function has at least one trigger associated, exposed to the outside and addressable from other functions. In this setting, it is the responsibility of the developer to create both the logic of the invocation and implement the protocols needed to allow the composition of and message passing between functions (Figure 2b). While this approach can offer the best expressiveness and dynamicity, it also hinders function re-

usability and modularity as the business logic is bound to a pre-determined composition policy.

On the contrary, in the infrastructure layer approach, the business logic of each function is decoupled from the composition logic. In this case, upon function completion, it is up to the FaaS platform to decide whether and where to redirect the output, as well as the actual protocol used to forward it (Fig 2c). In this approach, there is the separation between policy and mechanism, leading to easier implementation and overall better performance. Moreover, some optimizations can be exploited by the infrastructure to improve the processing pipeline performance, e.g., function co-location, result caching or optimized function-to-function communication protocols.

Our proposed solution goes in this latter direction, aiming to implement an efficient function composition mechanism. In specific, we chose to follow the continuous passing pattern implemented at the infrastructure layer, exploiting pluggable middleware solutions for efficient function-to-function communication leveraging heterogeneous hardware resources.

2.3. Low Latency Middleware Support

The continuous passing pattern empowers the infrastructure provider to adopt optimized mechanisms for function-to-function communication. For instance, inter-process communication (IPC) via shared memory mechanisms is widely adopted in computer systems, enabling zero-copy transfer of information, consequently speeding up computation by reducing redundant CPU usage and memory bandwidth expenditure. However, a similar mechanism is not readily available for processes located on different machines, and a common practice is to implement IPC through high-level abstractions such as message-based communication channels.

Remote Direct Memory Access (RDMA) is a networking technique that allows a process on one host to directly access the memory of a process on a remote machine, thus achieving the same kind of zero-copy transfer that characterizes local shared-memory exchanges. To this end, RDMA needs specific hardware support, allowing it to push the entire networking stack to the network interface controller (NIC). Under this assumption - in the communication setup phase - an agent can configure a portion of memory to be shared with a remote peer, and via dedicated primitives, it can instruct the NIC to send the memory chunk to the remote one [11].

This approach allows achieving near-to line rate throughput and sub-microsecond latencies, without even involving the remote CPU in the transfer

process: the receiver NIC directly writes incoming data to the target process address space. Recognizing these benefits, RDMA-enabled NICs increasingly implement RDMA over Converged Ethernet (RoCE), a protocol that enables RDMA on regular Ethernet networks [15]. Hence, a growing number of solutions have been proposed to leverage these advanced capabilities, while retaining compatibility with existing network deployments.

Unfortunately, direct use of RDMA is not a trivial task, as it requires developers to cope with low-level APIs, as well as designing the entire system under different assumptions when compared to the traditional TCP/IP stack [16]. Our objective is to design a distributed shared-memory middleware solution embodying the same semantic and access interface as that of its local counterpart, the POSIX *mqueue* [17]. This allows us to broaden the scope of our prior proposal, embodying advanced deployment capabilities, leveraging both intra and inter-host optimizations [10]. In addition, adding to the deployment spectrum, we assess and compare the use of traditional message queue systems such as Apache Kafka and Redis, evidencing the tradeoffs that emerge [18, 19]. The details on the chosen middleware solutions, configurations and capabilities of the approaches are discussed in Sec.4.

In the following, we review state-of-the-art serverless platforms with an emphasis on the function composition feature, discussing their differences and assumptions.

3. Related Work

Despite the relative novelty of FaaS platforms, several solutions, both commercial and academic proof-of-concepts, address the topic of function composition. Our survey is confined to proposals that present system-level novelties and optimizations, in line with our overall objective.

On the commercial front, Microsoft Azure Cloud has recently introduced Azure Durable Functions as an extension of Azure Functions [20]. This solution enables a user to define stateful workflows by writing special orchestration functions, whose state is managed by the platform. Amazon adopts a slightly different approach with their AWS Step Function offering, which behaves as a finite state machine controlling the execution of AWS Lambda functions composition [21]. AWS Step Function allows the definition of a series of checkpoints in the pipeline, used to enable fault tolerance capabilities, such as error handling and retry logic.

On the academic front, the authors of [6] identify some formal properties of function composition schemes, proposing a taxonomy of possible approaches. According to the proposed classification, they also discuss an infrastructural approach for function composition based on the OpenWhisk [22] platform.

The above offerings implement the function composition feature following the *reflective invocation* approach, which violates the fine-grained (zero) scaling feature, requiring an always-on entity to enact the composition logic. Moreover, the solutions seem to lack any form of optimization for inter-function communication, which is crucial for an efficient composition solution. On the contrary, we adopt a *continuous passing* composition pattern which does not require the instantiation of a third component and guarantees the best inter-function communication performance, avoiding intermediary entities.

SAND [23] is a serverless computing platform that combines a novel execution environment and fast inter-function communication. SAND promises lower latency, better resource efficiency and more elasticity than existing serverless platforms by leveraging on an application-level sandbox and a two-level message bus. SAND makes use of a local communication bus that enables efficient function-to-function transfers. However, the proposal executes multiple functions of an application in a single container, violating a core property of FaaS, which is to run and manage code written in different languages and with different dependencies. Moreover, the local bus is implemented as a custom process, lacking a standard interface, interoperability features, and the extensibility and flexibility that DIFFUSE embodies.

FAASM [5] introduces *Faaslet*, an isolation abstraction based on WebAssembly that leverages shared memory regions for communication between functions in the same address space. Faaslets execute in the FAASM runtime, which takes care of isolating other system resources using standard Linux *cgroups*. FAASM achieves better performance compared to container-based solutions in terms of memory usage, function instantiation time and overall throughput. Although FAASM could have been a good baseline for our work, its architecture does not provide any support for efficient function composition. Moreover, even though WebAssembly as an execution environment is an appealing research direction, especially if combined with other existing technologies, it still lacks stability [24] and some features required by production-grade systems.

In [10] we present a distributed FaaS architecture equipped with innova-

tive infrastructural support for function composition adhering to the *continuous passing* scheme. Function-to-function communication is supported via a single-host shared-memory middleware approach, relying on standard, interoperable system-level primitives such as the Linux *mqueue*. In this work, we generalize our approach, extending the platform capabilities to distributed and decentralized environments, allowing us to exploit inter and intra host software/hardware optimizations. Adding to the deployment spectrum of our proposal, we assess and contrast the distributed shared-memory approach with state-of-the-art messaging systems discussed in the following.

4. Middleware-centric Function Composability for FaaS

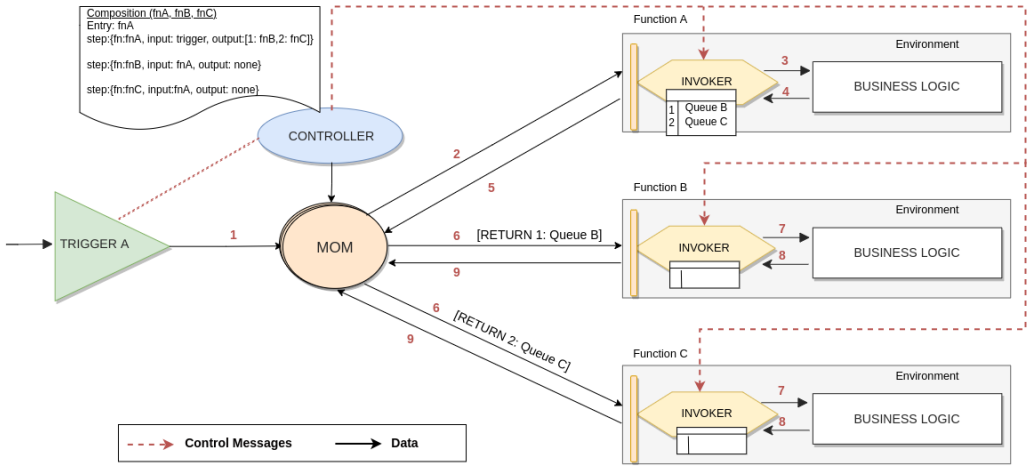


Figure 3: DIFFUSE relies on a MoM-based approach for function-to-function communication; invokers retrieve function invocation requests directly from the MoM triggering function execution.

In this section, we present a detailed view of DIFFUSE, outlining its main components, interaction flow(s), and the mechanism at the basis of the function composition feature. As anticipated, our proposal relies on a middleware-based approach, acting as a conveyor of messages among the components, serving user requests. Extending our prior work, we present the design of a distributed shared-memory middleware capable of exploiting modern hardware capabilities, embodying a similar semantic to the in-host counterpart. Concluding is a brief discussion on the characteristics of the

supported middleware frameworks, evidencing the broad spectrum of deployment options currently supported.

4.1. Function Composition Architecture

The composition mechanism comprises two layers: (i) the configuration and coordination layer instrumenting the FaaS platform components, and (ii) the function-to-function communication layer serving as a conveyor of messages between components.

Configuration and coordination. Our proposal provides the user with the capability to define custom processing pipelines expressed via association rules, residing outside the functions' business logic. Currently, the association rules are shipped to the controller in a JSON-based format, and allow the definition of generic, graph-shaped processing pipelines whereby the pipeline continuation is determined by the output of the executed function. This also allows us to introduce run-time modifications and updates to the processing pipeline, adding to the flexibility of the approach.

Indeed, at instantiation time and periodically, the FaaS components - trigger and invoker - can request the necessary configuration from the controller and participate in advancing the execution of the processing pipeline (Figure 3). These asynchronous updates allow moving the FaaS *controller* outside the chain invocation mechanism, enabling us to reduce the function response times when compared to the approach where the controller is involved in each function invocation (*reflective invocation*, Figure 1b).

Without loss of generality, in Fig. 3 is depicted an example of a processing pipeline that is composed of three functions, namely A, B and C. In a hypothetical scenario, the execution of the pipeline is triggered because of a user issued request, calling function A into execution. Upon function A termination, depending on its returned output inspected by the invoker component, the continuation of the pipeline will be either the execution of function B or C. It is important to note that in contrast to the *reflective invocation* mechanism where the (control) burden is offloaded to the controller entity, in our approach the control logic is decentralized and distributed to invoker entities.

Function-to-function communication mechanism. Once a pipeline configuration file is pushed to the platform, the components establish a series of communication queues (topics) used to exchange application and control data among them. In particular, the function-to-function communication mechanism relies on a MoM-based (Fig. 1) approach, adhering to a pub/sub

paradigm, enabling the transparent invocation of the next function in the pipeline. Returning to our prior example, once function A terminates the execution, the invoker consults the output and depending on the value, forwards the output either to Queue A or Queue B, consequently triggering the execution of function B or C, respectively. This mechanism provides the ability to dynamically scale the number of *invoker* instances, thus increasing the level of parallelism.

In these settings, the MoM acts as a conveyor for all messages and events, hence it is of paramount importance that the solution be efficient and able to gracefully scale with the number of requests. At the same time, it is desirable the platform be agnostic and decoupled by the specificities of the underlying MoM solution, promoting portability and openness to future extensions. To this end, we have introduced an abstraction layer (vertical orange box near the Invoker and Controller entities, Fig.3) decoupling the components from the specific MoM APIs by implementing a series of high level abstractions such as group (communication channel) creation, send and receive of messages, etc.

Finally, addressing the issue of the cold start phenomena [4], that is minimizing function bootstrap time whose effects are further exacerbated in a composition setting, the invoker adopts a simple pluggable optimization. In the current implementation, the provisioned strategy does not reclaim the resources allocated to the function whenever the request inter-arrival time is lower than the average function bootstrap and execution time. In the experimental setting, this simple, yet effective, logic removes any potential bias in the measurements of the different system configurations.

In the following, we present the design of a distributed shared memory middleware, allowing us to exploit modern hardware, guaranteeing low-latency and high-bandwidth communication. Next, we discuss the other MoM alternatives and overall characteristics, adding to the deployment spectrum of our platform.

4.2. DSMQueue: a Distributed Shared-Memory Queue

Herein, we present a zero-copy transfer mechanism for use in distributed multi-host deployments. The Distributed Shared-Memory Queue (DSMQueue) exploits modern networking hardware to build a zero-copy, $delete - after - read$ data transfer mechanism embodying a similar semantic to its local counterpart, the Linux kernel *mqueue* primitive. This approach enables a transparent load balancing mechanism among subscribers (Invokers), whereby a

read operation triggers the message removal from the queue. This way, the same function execution request is never executed more than once.

More in detail, DSMQueue is a distributed queue that exposes a *push* (send) and a *pop* (receive) operation. This queue, currently implemented as a ring buffer replicated among a group of processes (shared state), may be configured to offer different semantics, such as FIFO (default) or LIFO. The send/receive operations can be either blocking or unblocking. In a blocking configuration, when the queue is full, the process issuing a send (push) goes into a blocking state; when the queue is empty, the process issuing a receive (pop) blocks on it. The specific configuration depends on the scenario requirements. For example, in a context where data freshness is important, an unblocking FIFO configuration allows the sender to overwrite the data in the buffer according to a specific queue management policy.

Considering we are dealing with shared state in a distributed environment, one needs to rely on synchronization primitives to preserve consistency. To address this issue, we decided to adopt a well-known model for state sharing: the State Machine Replication (SMR) [25]. In SMR, every group member - the queue instance associated with processes representing the Invokers - holds a replica of the state, and all are bound to apply the same operations in the same order to maintain the overall consistency. Guaranteeing a strong consistency model requires implementing a form of *atomic multicast*, which imposes that any message sent by any group member is broadcasted to the others and delivered in the same order to all the members (total ordering) even in case of failures.

To fulfil all the above requirements, we implemented our solution on top of the Derecho open-source library [12]. The library enables point-to-point and multicast communication and supports total ordering, failure atomicity, and optional durable message logging. An optimal hardware mapping for RDMA enables Derecho to efficiently support even the strongest consistency properties, such as the SMR model, while guaranteeing high performance in terms of data throughput and latency. At the same time, to preserve compatibility when suitable hardware is not available, Derecho can execute on top of the TCP/IP stack without requiring any modifications to existing applications.

More in detail, Derecho allows users to define distributed services as "replicated objects", i.e., a set of state variables having an associated set of operations. Processes holding replicas of such an object will form a process group. Each state update can then be forwarded as an *atomic multicast*

Table 1: MoM properties.

	Delivery semantic	Delivery Order	Load Balancing
Kafka	Exactly-Once, At-Least-Once, At-Most-Once	Within single partition total ordering	Producer-side: Static, Round-Robin
Redis Stream	At-Least-Once At-Most-Once	Total Ordering	Consumer-Side: First come First served
DSMQueue	Exactly-Once	Total Ordering	Consumer-Side: First come First served

to all the group members and performed by all replicas. On an RDMA network, Derecho offers a zero-copy, lock-free critical path among remote applications, leading to ultra-low latency and exceptionally high bandwidth utilization. DSMQueue builds on Derecho and provides some higher level blocking primitives to send/receive messages to/from, e.g., Invoker components. Specifically, we implement DSMQueue as a Derecho replicated object, where the shared state is the ring buffer. The operations we define on that state are API calls that allow components to create and subscribe to specific message queues, acting as conveyor of data among components.

In the following, we present the other MoM solutions already integrated in our framework, discussing their characteristics and tradeoffs that emerge.

4.3. MoM Deployment Considerations

Adding to the deployment spectrum of our proposal, we identified two other state-of-the-art MoM solutions, namely Apache Kafka and Redis Stream [18, 19]. In specific, Kafka is a highly scalable, open-source event streaming platform, while Redis Stream is a streaming abstraction built on top of the widespread persistent Redis database.

Table 1 provides a summary of some characteristics the different MoM solutions embody. All the options offer advanced state replication and consistency mechanisms for improved load distribution and fault tolerance. In particular, Kafka exploits a multi-broker mechanisms with a configurable level of topic (channel) replication, while Redis employs a classical Driver-

Worker active replication scheme. Similarly, our DSMQueue proposal replicates queue state (data) exploiting RDMA to guarantee the highest possible performance. DSMQueue, which is based on the Derecho library, adopts the same active replication pattern of Redis, but the logic is completely decentralized, thus eliminating the need for a driver node on the critical data path. In this setting, all the nodes are equal peers that agree on the same shared state, thus achieving the maximum possible degree of parallelism.

Concerning the delivery semantics, DSMQueue offers an exactly-once semantic, while Redis offers an at-least-once embodying less overhead in synchronization when compared to DSMQueue. This behavior may lead to a lower use of the network resources, but does not guarantee the consistency of the shared state in case of failure of one or more nodes, which DSMQueue is always able to guarantee. Kafka is the only one of the three solutions that, thanks to its deep integration with Apache Zookeeper, allows choosing among all the three delivery semantics at-most-once, at-least-once, and exactly-once at a topic granularity.

5. Performance Evaluation

This section presents an experimental evaluation, assessing our proposal as-a-whole while varying the underlying MoM support. In particular, we evaluate and compare the capabilities of DSMQueue with the other traditional MoMs, identifying possible deployment tradeoffs.

To this end, the three emerging configurations are evaluated under three representative workloads: (i) a constant-rate stream of requests, (ii) a stream of incoming requests issued at an increasing rate, and (iii) a large batch of requests submitted to the system in a small amount of time. The first workload aims to assess the properties of our serverless platform in a steady regime of incoming requests, whereas the second and the third scenarios reproduce a typical traffic pattern that arises when a high number of concurrent events need to be processed in batch (e.g., process all the tweets with a specific hashtag).

For this evaluation, we employ a lightweight, short-lived business logic with an execution time of about 60 µs. Upon termination, the last function of each pipeline appends a timestamp to its output, later on used to compute the different metrics. Response times are measured as the time-lapse between the moment the request is issued and the termination timestamp of that function (*end-to-end* latency). We define as *throughput* the number of

satisfied requests per unit of time. We examine how these metrics, as well as the total execution time, vary under an increasing composition length. In this assessment, we vary the number of composable functions from 2 to 5, and each function is packaged as a distinct container, although embodying the same business logic. Also, the application graphic is a linear path, hence no branching logic is considered.

The experiments are conducted on two identical machines, each equipped with a 4-core i5-3470 CPU @ 3.20GHz, 10GB RAM, running Ubuntu 20.04. The two nodes are directly interconnected by a 100Gbps Mellanox ConnectX-6 DX NIC, which supports both standard Ethernet traffic and RDMA networking. On each machine, we run a single instance of the function invoker, which has access to the code of the function to be executed in the experiment. On one of the nodes, we also run a traffic generator process, which we use as a trigger to simulate different ingress traffic patterns: the trigger forwards the invocation requests to the invokers using the MoM.

We configure Kafka with at-least-once semantic to avoid the overhead introduced by transactions in an exactly-once mode, and for the same reason, the number of partitions in the topic is set equal to the number of nodes with a replication factor of 1. Redis was deployed as a single instance in one of the two nodes and set-up in order to create a Redis Stream with one single active group, i.e., function invokers cooperate to consume a different portion of the same stream of messages. Finally, we configure DSMQueue to replicate the shared-memory queue across a group of three processes, the trigger and the two invokers. We configure the underlying Derecho library to enforce strong consistency across the replicas, and to keep the shared state in volatile memory, with no persistence support.

In the following, we discuss the experimental results and the trade-offs that emerge.

5.1. Constant-rate stream of incoming requests

In this first experiment, we would like to investigate the system behavior under a steady regime. Hence, the trigger issues a fixed number of requests at a constant rate of 1.000 requests/second, and the experiment is run by varying the length of the function composition from 2 to 5.

Figure 4 shows the end-to-end latency of each execution as a function of the composition length. Note the logarithmic scale in the y-axis, which magnifies the length of the whiskers for the smaller values. We can observe

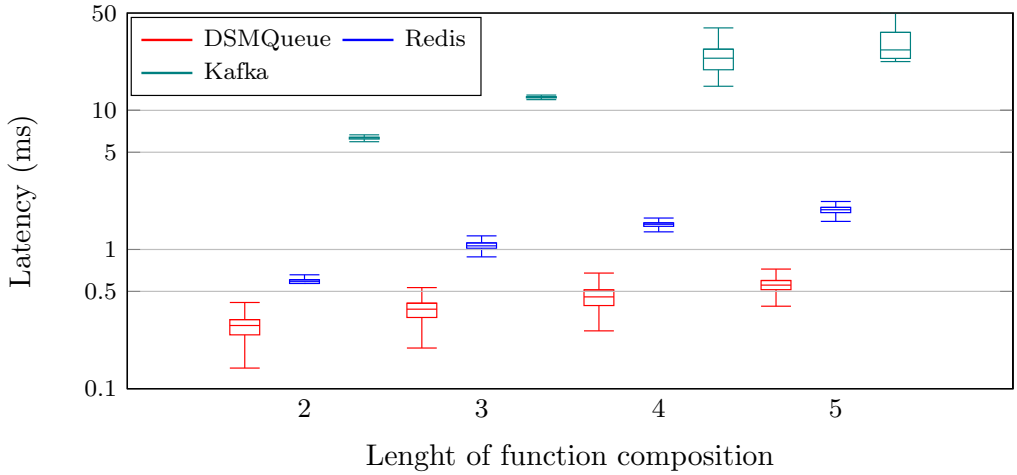
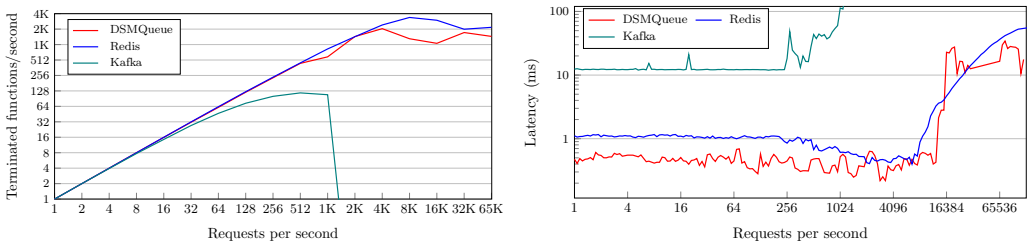


Figure 4: End-to-end latency at a steady regime.

two important trends. First, as expected, the latency increases with the composition length. This increment is generally attributed to the time taken to execute more functions, and the time spent in the (de)queuing operations. At this request rate, the MoMs can sustain the traffic with little to no queueing effects, hence the delay contribution is mainly to be attributed to networking and synchronization of concurrent requests. In all configurations, the latency increment is linear, but there are important differences. For DSMQeueue, the median latency shows a 2x increment when switching from 2 to 5 functions: much of it is the function execution time, whereas only 33% is caused by additional middleware operations. This increment is more evident in Redis, which demonstrates a higher (3x) latency increment between 2 and 5 functions. As the function execution time is constant, the additional latency time is caused by the middleware operations, which in this case account for the 86% of the total increment. Kafka exhibits similar behavior, but with an even higher increment factor (4x).

The second consideration is about the relative performance of the different middleware solutions. For the simplest case of two functions in the composition, DSMQueue shows the best median latency (284 µs). While Redis is able to keep up (2x slower), Kafka demonstrates an order of magnitude higher latency (22x slower). The amount of these latency gaps increases as the composition length increases: for a composition of length 5, Redis is 3.2x worse than DSMQueue (554 µs), and Kafka is again out of scale (49x higher latency).



(a) End-to-End throughput

(b) End-to-End latency

Figure 5: End-to-end latency and throughput with a varying rate of ingress traffic.

In conclusion, DMSQueue outperforms the other alternatives, which rely on traditional TCP/IP networking and higher-layer constructs to implement advanced capabilities. Kafka adopts a similar semantic to the other MoMs (see Section 4.3), yet it shows the worse performance by far, and this is to be attributed to its default message/topic persistency support. DSMQueue and Redis have a more similar architecture, but Redis is between two and three times slower in this context.

5.2. Incremental rate stream of incoming requests

Herein, we would like to assess system scalability by subjecting the platform to an increased rate of incoming requests, varying from 1 to 65K requests/seconds. In this scenario, the composition length is kept constant to 3, representing a common option in real-world scenarios e.g., simple map-reduce operations etc. Figure 5 shows the end-to-end throughput and latency (y-axis in log scale) of the proposal under a varying rate of incoming requests.

Figure 5a shows that the different configurations gracefully scale up the resources to keep up with demand, and throughput increases linearly up to a certain inflection point before starting to decline. This critical point corresponds to the maximum input rate that a middleware can sustain without queuing any request: after a threshold, new requests begin to queue up, competing with the existing invocation requests, using up the available resources. The critical rate is similar for DSMQueue and Redis, which start to queue requests between 8K and 12K requests/second, whereas this behavior emerges much earlier in Kafka, at about 240 requests/second.

As one may expect, the competition for resources between incoming and enqueued requests has a direct effect on latency (Figure 5b). Up to the critical input rate, DMSQueue shows a better end-to-end latency than Redis,

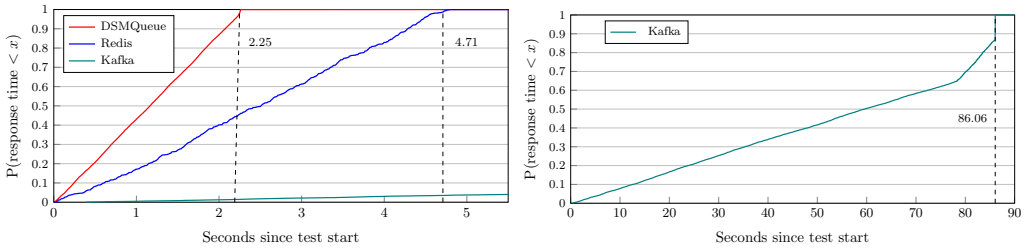
averaging about 500 µs versus 1 ms. Shortly after the critical rate, DSMQueue shows an increasingly oscillatory effect, whereas, surprisingly, Redis exhibits a decline in latency. Finally, between 8K (Redis) and 16K (DSMQueue) requests/second, performance degrades rapidly and reaches a similar regime of much higher latency (tens of ms), although DSMQueue still demonstrates a much better behavior. Finally, we observe that Kafka, even in low request regimes, demonstrates an order of magnitude higher latency than both Redis (10x slower) and DSMQueue (20x slower).

Overall, DSMQueue performs well in terms of latency (2x) and has comparable throughput to Redis. This behavior remains constant up to a critical ingress rate, as well as for the highest input rates, while the behavior of both systems becomes unstable during the transition between those two phases. In addition to the motivations provided in Sec. 5.1, it is noteworthy to point out that the DSMQueue zero-copy approach fully manifests its benefits as the message size grows. This leads to extra spare time, not spent on copying data [26].

On the other hand, the poor performance of Kafka is to be attributed to the MoMs consistency mechanism used to maintain a distributed, structured and durable commit log of events: any request - ingress data to functional components of the chain - must be acknowledged prior to serving successive ones. Considering the high ingress load and the additional load generated by intermediate results of function executions, the topics acquire an increasing backlog of requests (events) subject to the dynamics of the commit log. As a consequence, the invoker entities tasked with the execution of functions and output serialization to Kafka are subjected to ever-increasing waiting times, expecting an acknowledgement from the broker. This in turn results in a lower end-to-end chain throughput with a respective spike in terms of latency. The specific interval where the phenomenon manifests itself is tied to the current testbed characteristics (CPU, RAM etc.): to mitigate it, could rely on the topic partitioning feature of Kafka, distributing the load among cluster nodes according to design-time criteria. It is noteworthy to point out that in our current setting, Kafka is configured with an "at-least-one" semantic, more optimistic in terms of performance with respect to an "at-most-one" semantic.

5.3. Burst of incoming requests

In this experiment, we assess the behavior of the platform when subjected to a sudden burst of concurrent requests. To this end, our trigger produces



(a) MoM comparison for a composition of length 3 (b) Kafka performance in isolation for a composition of length 3

Figure 6: Response time CDF with varying MoM support.

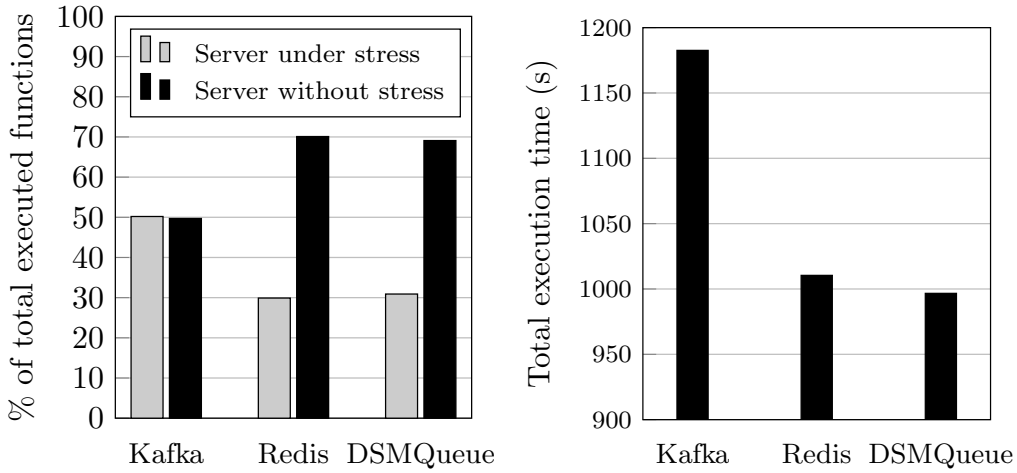
a burst of 10K invocation requests at the highest possible sending rate. This way, we intentionally exacerbate the queuing effect described in Section 5.2: the message queue will fill up with invocation requests, as the invokers will not be able to consume them at the same rate. We keep the composition length constant to 3 functions: this further stresses the queue, as per our architecture each pipeline execution requires the invokers to produce and consume new requests to and from the queue.

In this setting, we are interested in the total time the system takes to consume the entire batch of concurrent requests. Figure 6a breaks down the total execution time by plotting the Cumulative Distribution Function (CDF) of the pipeline execution time.

In this case, the different behavior of the considered systems depends on the different waiting times between the execution of two consecutive functions. Such waiting time is determined by the different communication overhead introduced by each solution, which directly affects the speed at which they process the backlog of requests. In particular, the trend that we observe is the same we described for the previous experiment. The shared memory approach of DSMQueue is the fastest in processing the request batch (2.25 seconds), Redis takes about twice that time (4.71 seconds), and Kafka is an order of magnitude slower (86.06 seconds) as shown in Figure 6b.

5.4. MoM enabled load balancing

In this last experiment, we investigate how the different properties of the three MoMs (see Sec. 4.3) impact the distribution of the pipeline workload across the available nodes. Indeed, one driving motivation for this work is to enable DIFFUSE to scale across a varying number of hosts and to efficiently use all the available resources. To effectively measure such efficiency, we are



(a) Distribution of functions across the hosts

(b) Total test execution time

Figure 7: Load balancing behavior of the different MoMs

interested in how the workload is distributed when the available machines are subject to different load conditions.

To this end, we run the same experiment discussed in Sec. 5.3, but this time the adopted function is more computationally expensive than the one in the prior experiments, totalling an average execution time of 60 ms. Also, to better highlight the differences between the MoMs, one of the two available hosts executes a background application, saturating its computing, memory, and disk resources. This way, we expect that the same function will take a different execution time depending on the host it is executing on: in one case, the function will compete with the background application to acquire the necessary resources, whereas those will be immediately available on the other host. We want to understand if and how each MoM takes the server load condition into account when deciding, transparently to the user, how to distribute the incoming workload.

Figure 7a shows the results. As expected, the same function on the two hosts takes a significantly different amount of time to complete: on average, 35 ms on the idle one, and more than twice, about 85 ms, on the other. We observe that Redis and DMSQueue execute about 30% of the workload on the saturated server, leaving almost 70% of it to the idle machine. On the contrary, Kafka assigns the same number of functions to both hosts. This different behavior is directly linked to the way each MoM implements the queue abstraction (Sec. 4.3). In DSMQueue and Redis Stream, the queue

is a (logically) single FIFO buffer that processes compete to access, either when producing or consuming new data. In our setting, these processes correspond to the two invokers. Since the function execution on the idle host takes approximately half of the time taken on the saturated one, the invoker on the idle host ends up consuming more than twice the number of functions than the invoker on the saturated host: an indirect form of load balancing induced by the load on each node. Kafka, instead, blindly follows a round-robin scheme, assigning an equal number of functions to each host. While this approach eliminates the need for coordination among the invokers - which no longer need to compete to access the queue - it also does not take the actual server load into account. As a consequence, many more functions are scheduled on the saturated host, leaving unused resources on the idle host: this is clearly inefficient, and it results in a significant increase in the time needed by Kafka to process the function batch (Figure 7b).

Even though Redis and DSMQueue already provide an implicit form of load balancing, an explicit mechanism could lead to faster function execution times, and, as a consequence, to improvements on the overall system throughput. The development of a new load balancing mechanism requires the introduction of an observability layer, providing real time information on the resource usage.

6. Conclusion

Serverless computing is a model of cloud computing that promotes a higher level of abstraction, allowing users to focus on the development of the business logic, while offloading all resource management duties to the platform provider. An emerging and appealing feature for serverless is the capability of composing functions to create complex processing workflows, incentivizing modularity and reusability of functions, while paving the way to infrastructural-specific run-time optimizations.

In this article, we presented DIFFUSE: a DIstributed and decentralized platForm enabling Function composition in Serverless Environments. The proposal relies on pluggable middleware solutions serving as conveyor of messages among the platform components. To this aim, different middleware configurations were presented and assessed under different scenarios, highlighting their strengths and weaknesses. Results show that networking techniques like RDMA may bring significant performance advantages and enhanced QoS guarantees. At the same time, systems based on standard

networking interfaces represent a valid alternative in environments with more conventional settings or specific constraints on development, deployment, or scale of the infrastructure.

DIFFUSE is under active development and in future work, we aim to introduce support for hybrid MoM deployments, exploiting also the in-host shared memory optimizations, spanning heterogeneous resources on the edge-to-cloud continuum. This feature allows exploiting the most convenient medium for function-to-function communication, depending on the local environment characteristics. Moreover, we are working on the introduction of a resource management mechanism able to handle the intelligent placement, scaling, replication and coordination of platform components and functions. At the core of this capability is the introduction of an observability layer providing real-time operational data on resource usage, data locations etc.

References

[1] J. Schleier-Smith *et al.*, "What Serverless Computing is and Should Become: The next Phase of Cloud Computing," *Commun. ACM*, vol. 64, no. 5, p. 76–84, apr 2021.

[2] A. Bujari, A. Corradi, L. Foschini, L. Patera, and A. Sabbioni, "Enhancing the Performance of Industry 4.0 Scenarios via Serverless Processing at the Edge," in *Proc. of IEEE International Conference on Communications (ICC)*, 2021, pp. 1–6.

[3] P. Patros *et al.*, "Toward Sustainable Serverless Computing," *IEEE Internet Computing*, vol. 25, no. 6, pp. 42–50, 2021.

[4] E. Jonas *et al.*, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, 01 2019.

[5] S. Shillaker *et al.*, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *Proc. of USENIX Annual Technical Conference (USENIX ATC)*, Jul. 2020, pp. 419–433.

[6] I. Baldini *et al.*, "The Serverless Trilemma: Function Composition for Serverless Computing," in *Proc. of ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, New York, NY, USA, 2017, p. 89–103.

[7] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *Proc. of USENIX Annual Technical Conference (USENIX ATC)*. Boston, MA: USENIX Association, Jul. 2018, pp. 133–146.

[8] S. S. Shinde, D. Marabissi, and D. Tarchi, "A Network Operator-biased approach for Multi-service Network Function Placement in a 5G Network Slicing Architecture," *Computer Networks*, vol. 201, p. 108598, 2021.

[9] F. A. Salaht, F. Desprez, and A. Lebre, "An Overview of Service Placement Problem in Fog and Edge Computing," *ACM Comput. Surv.*, vol. 53, no. 3, jun 2020.

[10] A. Sabbioni, L. Rosa, A. Bujari, L. Foschini, and A. Corradi, "A Shared Memory Approach for Function Chaining in Serverless Platforms," in *Proc. of IEEE Symposium on Computers and Communications (ISCC)*, 2021, pp. 1–6.

[11] Mellanox. (2014) RDMA Aware Networks Programming User Manual. [Online]. Available: https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf

[12] S. Jha *et al.*, "Derecho: Fast state machine replication for cloud services," *ACM Trans. Comput. Syst.*, vol. 36, no. 2, Apr. 2019.

[13] "OpenFaaS." [Online]. Available: https://www.openfaas.com

[14] E. van Eyk *et al.*, "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms," *IEEE Internet Computing*, vol. 23, no. 6, pp. 7–18, 2019.

[15] InfiniBand Trade Association. (2014, September) Supplement to InfiniBand Architecture Specification: RoCEv2. [Online]. Available: https://cw.infinibandta.org/document/dl/7781

[16] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design Guidelines for High Performance RDMA Systems," in *Proc. of USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 437–450.

[17] Linux posix message queue. [Online]. Available: https://man7.org/linux/man-pages/man7/mq_overview.7.html

[18] Apache kafka. [Online]. Available: https://kafka.apache.org/

[19] An introduction to redis streams. [Online]. Available: https://redis.io/topics/streams-intro

[20] Microsoft. Azure Functions Serverless Compute. [Online]. Available: https://azure.microsoft.com/services/functions

[21] Amazon. AWS Step Functions. [Online]. Available: https://aws.amazon.com/it/step-functions

[22] IBM. Apache OpenWhisk. [Online]. Available: https://openwhisk.apache.org

[23] I.E. Akkus *et al.*, "SAND: Towards high-performance serverless computing," in *Proc. of USENIX Annual Technical Conference (USENIX ATC)*, Boston, MA, Jul. 2018, pp. 923–935.

[24] K. Inc. Krustlet. [Online]. Available: https://krustlet.dev/

[25] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, p. 299–319, Dec. 1990.

[26] L. Rosa, W. Song, L. Foschini, A. Corradi, and K. Birman, "DerechoDDS: Strongly Consistent Data Distribution for Mission-Critical Applications," in *Proc. of IEEE Military Communication Conference (MILCOM)*, 2021, pp. 1–6.