FaaS Execution Models for Edge Applications

Claudio Cicconetti[a,*], Marco Conti[a], Andrea Passarella[a]

[a]*IIT, National Research Council, Pisa, Italy*

Abstract

In this paper, we address the problem of supporting stateful workflows following a Function-as-a-Service (FaaS) model in edge networks. In particular we focus on the problem of data transfer, which can be a performance bottleneck due to the limited speed of communication links in some edge scenarios and we propose three different schemes: a *pure FaaS* implementation, *StateProp*, i.e., propagation of the application state throughout the entire chain of functions, and *StateLocal*, i.e., a solution where the state is kept local to the workers that run functions and retrieved only as needed. We then extend the proposed schemes to the more general case of applications modeled as Directed Acyclic Graphs (DAGs), which cover a broad range of practical applications, e.g., in the Internet of Things (IoT) area. Our contribution is validated via a prototype implementation. Experiments in emulated conditions show that applying the data locality principle reduces significantly the volume of network traffic required and improves the end-to-end delay performance, especially with local caching on edge nodes and low link speeds.

Keywords: Edge computing, Serverless, Function-as-a-Service, Distributed computing, In-network intelligence

*Corresponding author

Email addresses: `c.cicconetti@iit.cnr.it` (Claudio Cicconetti), `m.conti@iit.cnr.it` (Marco Conti), `a.passarella@iit.cnr.it` (Andrea Passarella)

1. Introduction

Computation offloading has been a trending topic in the networking and cloud computing areas for some time now: it envisions that mobile or resource-constrained devices offload part of their processing activities to external entities with computation capabilities willing to undertake the effort. A few years ago, the interest has then shifted towards the edge of the network [1], as this enables latency-sensitive applications that cannot afford a trip to far-away data centers. However, recent cloud deployments already delocalize data centers so that they are closer to the users [2]: research activities should not rely only on a closer-is-better-for-latency motivation for edge computing, but rather look to the edge with a broader perspective and find what it can realistically provide in specialized use cases and applications.

One such opportunity that is emerging is to employ at the edge a Function as a Service (FaaS) model [3]: the application is decomposed into functions that are invoked individually or in a chain. FaaS is very well suited to many Internet of Things (IoT) applications of practical interest for what concerns the programming model (functional event-based), an efficient utilization of resources (both at device- and edge node-level) and the promises of high scalability. The latter stems from symbiosis with a *serverless computing* framework, where functions are invoked in containers that are orchestrated in a highly flexible virtualization infrastructure [4]. Results have demonstrated that serverless is more suitable than a microservice architecture for unpredictable requests accompanied by a large size of the response, due to the scaling agility [5]. Serverless/FaaS are major trends in cloud computing [6], thus edge deployments/applications could benefit from the ample availability of industry-grade commercial and open source solutions [7], even though some advances beyond the state of the art are required to make a good use of resources in this different environment [8].

In particular, serverless relies on the implicit assumption that the container location is irrelevant to performance, which has led to a *stateless* FaaS paradigm: applications that have a state rely on external services offered (and billed separately) by the cloud provider [9]. In [10] the authors have analyzed open-source and proprietary datasets and they have found that only 12% of the serverless applications in production are *truly* stateless, whereas the others rely on managed services such as storage (61%) and databases (48%). In the cloud, this may lead to a slightly sub-optimal utilization of the resources, e.g., due to the inability to keep hot content in caches [11], but at the edge the impact becomes much more ominous: here the cost of transferring data between function executors and external services is, in general, much higher than in a data center, hence the network may easily become a limiting factor [12]. Furthermore, one of the most appealing features of FaaS is the opportunity for the service provider to compose applications as complex workflows of invocations, so that the output of a function is not returned immediately to the user but delivered to one (or more) successors for further processing. This exacerbates the above implications of location dependency, which become crucial when migrating from a microservice to a serverless architecture [13].

In this paper, we address the problem of data transfer (including both arguments/return values and the application state) within a workflow of stateful functions, motivated by a practical use case illustrated in Sec. 2. Then, after reviewing the state of the art (Sec. 3), we summarize in Sec. 4 the findings in our previous work [14], where we have proposed three fully decentralized execution models for applications that can be modeled as chains of functions: PureFaaS, StateProp, and StateLocal. The execution models are extended to the more general case of applications modeled as Directed Acyclic Graphs (DAGs) in Sec. 5. As we will see later, PureFaaS, which is closest to state-of-the-art serverless platforms, is surpassed by StateProp and StateLocal, which achieve reduced traffic and smaller delays. However, they incur the cost of a slightly higher system complexity, because they require the ability to embed the application's state as function arguments (StateProp) or keep the state at edge nodes (StateLocal), in addition to a more profound knowledge of the application workflow (DAG of function invocations and state dependencies). We have implemented a prototype of the proposed solutions, which we use to compare their performance in

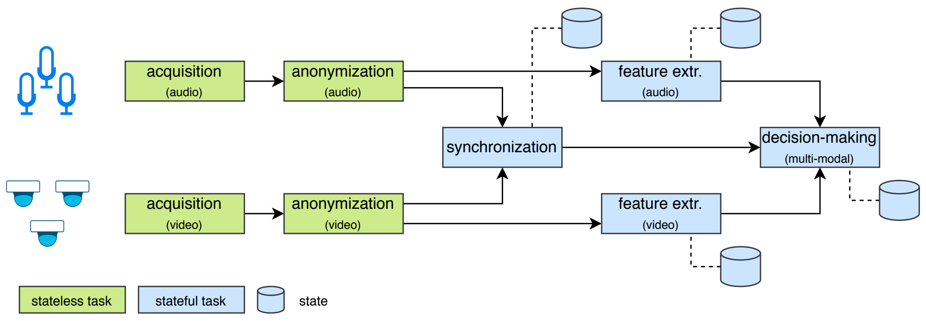

Figure 1: Archetypal smart city application for audio-video analytics in the MARVEL project.

emulated network experiments (Sec. 6). We draw the conclusions in Sec. 7.

2. Motivation

In this section we describe a motivating example, inspired from the activity ongoing in the H2020 MARVEL collaborative R&D project[1], co-funded by the European Commission, in which we participate. The project defines a framework for real-time analytics in smart cities, addressing several applications of high impact to citizens validated in two pilots, municipality of Trento and public streets in Malta: automated detection of anomalous traffic conditions, monitoring of crowded areas, protection of vulnerable users (pedestrians, cyclists) in street junctions, emotion recognition in public events, and many others.

All the applications have the same general structure illustrated in Fig. 1: starting from the acquisition of data from sensors, first there is an anonymization phase to remove personal data, then the relevant features are extracted and used to trigger a Machine Learning (ML)-driven decision-making process. Despite its potential benefits, *vanilla serverless computing cannot be adopted here* for two reasons. First, a stateless execution is not sufficient: (some of) the components require read/write access to a per-application state, e.g., with video streams to cross-correlate the current frame with previous ones in a window. Second, FaaS platforms support chain of functions but our applications have multiple sources

[1]`https://www.marvel-project.eu/`, last accessed Aug 12, 2022

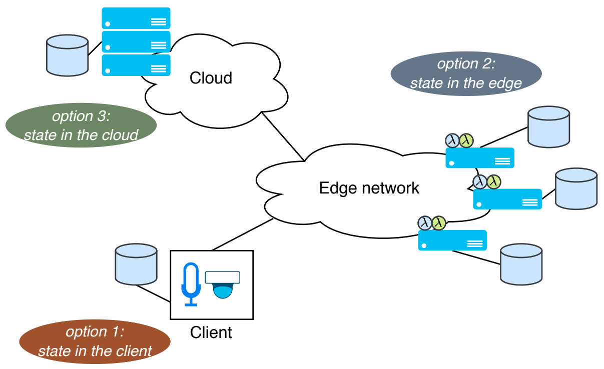

Figure 2: Architecture options on where to keep the application's state: option 1: the state remains in the client, which embeds it in the function arguments and return value so that the execution remain stateless; option 2: the state is kept by the edge nodes; option 3: the state is handled by a cloud service, which is the default today.

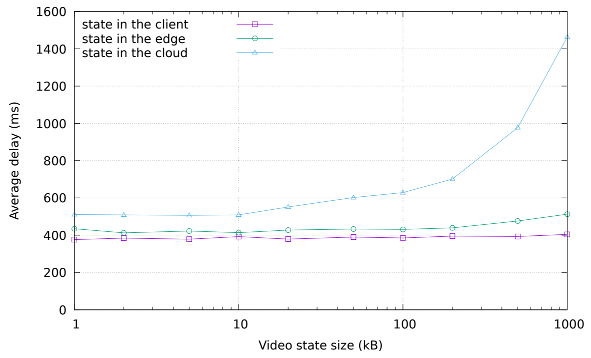

Figure 3: Average delay of the workflow in Fig. 1 with the three options in Fig. 2. The edge links have 1 Gb/s bandwidth with 1 ms latency, whereas the cloud node is connected through a 10 Mb/s link with 20 ms latency. The full details on the methodology and tools used are in Sec. 6.

per application (cameras + microphones), which requires the workflow to be designed as a DAG for synchronization and fusion purposes.

In this paper we address both these aspects: in Sec. 4 we discuss the possible options on where to maintain the application's state, summarized in Fig. 2, while the support of DAGs is illustrated in Sec. 5. In Sec. 6 we evaluate our proposed system with a prototype implementation in a mininet testbed, which also used for the motivating example in this section..

In Fig. 3 we show the average delay of the workflow when increasing the state size of the feature extraction video function from 1 kB to 1 MB; the state of the corresponding function for audio is 1/10 of the latter, those of the other functions 1/100. The state sizes used here are arbitrary but representative of real use cases. As can be seen, when the state is kept in the cloud the average delay is always higher than that in the other cases, and it grows significantly as the state size increases. On the other hand, there is no noticeable increase when the state is maintained at the edge or in the client, with the latter exhibiting the smallest delay (though at the cost of higher network traffic, not shown).

The results found in this motivating example show that the performance of data-intensive applications made of DAG stateful functions, such as real-time smart city analytics, significantly depends on where the state is kept, which is the subject of this work. It is worth mentioning that these applications are not a special case. For instance, also robotic applications are typically designed using a DAG model [15] and, in general, we have analyzed the traces of real-life cloud applications collected from a production system in an Alibaba data center[2] and we have found that a non-negligible fraction of applications consist of DAGs: 21.7%, with single tasks being 28.6% and chains 49.7%.

3. Related work

Serverless platforms in the cloud hinge on the underlying container orchestration systems, which handle autoscaling and are responsible for consistent performance. However, these orchestration tools are inefficient when used at the edge, where devices are heterogeneous and clustered, which causes sub-optimal performance [16]. For a comprehensive review on all the aspects of resource management in serverless systems we refer the interested reader to the survey in [17]. In the following, we focus on some works that are especially relevant to our contribution, that is the definition of suitable execution models to handle chain/DAG composition of stateful functions in serverless edge networks.

At the edge, the problem of data locality has been introduced neatly in [12], where the authors have proposed to influence resource scheduling in Kuber-

[2]`https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md`, last accessed Aug 12, 2022.

netes (K8s) by adjusting its internal weights based on metadata specified by the application developers. The more abstract problem of directly allocating functions to edge nodes for complex applications consisting of multiple interrelated functions has been studied in [18] with a mathematical formulation, which takes into account different categories of cost (i.e., activation, placement, proximity, sharing). The problem has been also studied in the context of Mobile Edge Computing (MEC) with DAG-modeled applications in [19], where the authors sought the goal of meeting as many application request deadlines as possible using an online algorithm approximating an NP-hard scheduling problem. Finally, in [20] the authors have proposed SEND, an in-network storage management system realized through edge-deployed data repositories, which places intelligently raw and processed data based on locality or popularity criteria. *Key difference:* *Our work is complementary to the above studies because we do not address the allocation of containers/tasks to edge nodes, but rather propose solutions to manage the applications' state in FaaS under a given allocation.*

Supporting stateful applications is one of the key research challenges identified in the position paper [4] for serverless computing in the cloud. A datastore for edge computing with consistent replicas has been also proposed in [21], which reconciles only the data that are relevant to a given session for performance reasons. In [22] the authors have proposed HydroCache, a distributed data caching system with multi-site causal consistency, which can be used as state management for serverless DAG functions and has been shown to outperform uncached platforms in the cloud. Fault tolerant function execution, with embedded garbage collection, has been addressed by the authors in [23] and found to be both effective and affordable on AWS Lambda. Finally, Boki [24] has been proposed as a serverless platform that enables stateful applications via shared logs, which have ordering, consistency and fault tolerance properties. *Key difference:* *Currently stateful FaaS in edge networks is largely unexplored, which is a motivation for our work to explore different approaches for argument and state distribution with chains and DAGs of function invocations. In a practical deployment, sophisticated state management systems can be used in* *combination with the execution models that we propose in this work.*

The need for efficient serverless platforms that can scale down to small edge devices is illustrated in [25], which proposes to integrate a computation model based on the actor pattern with content-based networking. To address also microcontrollers, the authors have used a pub-sub messaging system underneath. A similar approach has been followed in [26], which presents Faasm, where user-space isolation abstraction is provided via the use of the WebAssembly run-time environment and applications can share state using a hierarchical Key-Value Store (KVS).

Key difference: *The actor model has interesting similarities with serverless. As a matter of fact, the integration of WebAssembly platforms using the actor pattern with FaaS platforms is under way (e.g., wasmcloud*[3] *and OpenFaaS*[4]*). We believe our work makes a valid contribution across both domains, as we propose different execution models for stateful execution of chain/DAG workflows, which can be combined or tailored to the specific needs of the scenario and applications.*

Some recent works have focused on the specific issue of resource management for DAG workflows in serverless platforms. In [27] the authors have identified a set of techniques to make DAG schedulers aware of the serverless platform they are running on, tested on AWS Lambda. The opposite approach has been followed in [28], where the authors have defined an orchestration framework to match the application performance requirements via appropriate provisioning of containers in a K8s cluster.

Key difference: *These works contribute to the motivation of our research activity, as they deal with applications that can be modeled with DAGs, which however remains complementary: we focus on different schemes to pass on the arguments and states of the application, which is a different (though possibly related) problem to container resource management.*

[3]`https://wasmcloud.dev/`, last accessed Aug 12, 2022
[4]`https://www.openfaas.com/`, last accessed Aug 12, 2022

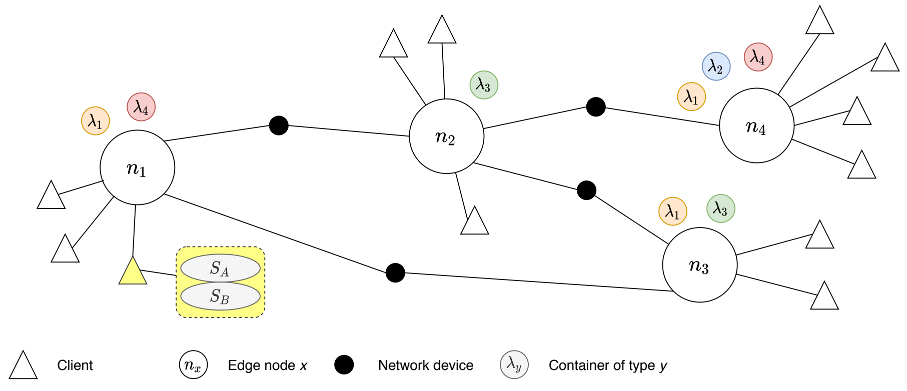

Figure 4: System model.

Finally, we mention that practical applications might also require mechanisms for the realization of patterns beyond the execution of stateful DAGs. Microsoft's commercial serverless platform, Azure Durable Functions (ADFs), also allows to define critical sections for the atomic execution of some functions in a workflow [29], whereas explicit parallelization of function execution has been investigated in [30].

Key difference: *We recognize that some applications have specific needs that cannot be addressed efficiently by a single solution. In this work we have focused on chain and DAG workflows, which are very common and already cover a broad range of applications of practical interest, and we leave for future work further specializations, including critical sections and explicit parallelism.*

4. Stateful function chains

In this section we introduce the system model and notation used in the paper and we summarize the findings in [14], which tackled the execution of chains of functions on serverless platforms deployed at the edge, which are extended to the more general case of DAG-modeled applications in the next section. We conclude the section with considerations about confidentiality in Sec. 4.1.

The system model is illustrated with the help of the example in Fig. 4. In the figure we have four edge nodes indicated as n_i, each hosting a serverless platform that can execute lambda functions of one or more types, indicated as

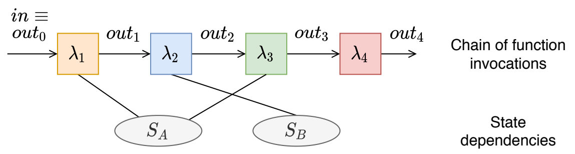

Figure 5: Example chain application.

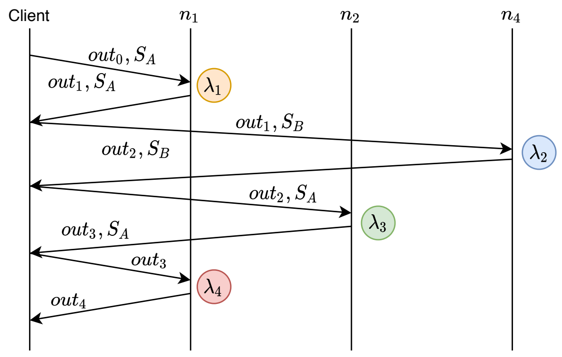

Figure 6: PureFaaS execution diagram of the application in Fig. 5 at the edge in Fig. 4.

λ_i, via a pool of *workers*. For instance, n_1 can execute lambda functions λ_1 and λ_4 but not λ_2 and λ_3.

Let us now consider the example user application depicted in Fig. 5: the client needs the input to be provided to λ_1, which also requires (and possibly modifies) the application state S_A, whose output out_1 needs to be provided to λ_2, also requiring access to S_B, which feeds λ_3 and so on, until the final output out_4 is returned to the client. The target application is colored in yellow and represented running on the client device, also showing its two states S_A and S_B. In the cloud, *stateful* functions are realized by means of *stateless* functions that access external services, such as in-memory databases or storage services. However, this approach is not efficient at the edge, as shown in Sec. 2. Our alternatives are to keep the state in the client vs. in the edge nodes. In [14] we have explored three different approaches, which are summarized below: PureFaaS and StateProp (the state remains in the client) and StateLocal (the state is kept by the edge nodes). In the following we assume that the allocation of functions to nodes is: $[\lambda_1, \lambda_2, \lambda_3, \lambda_4] \rightarrow [n_1, n_4, n_2, n_1]$.

With **PureFaaS**, the functions in the chain are executed one after another,

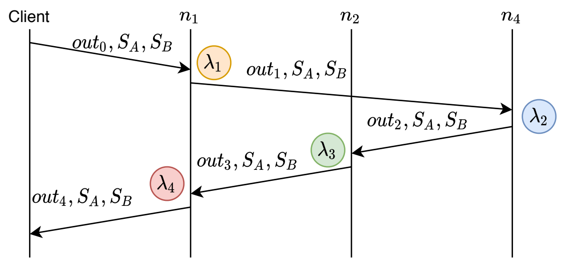

Figure 7: StateProp execution diagram of the application in Fig. 5 at the edge in Fig. 4.

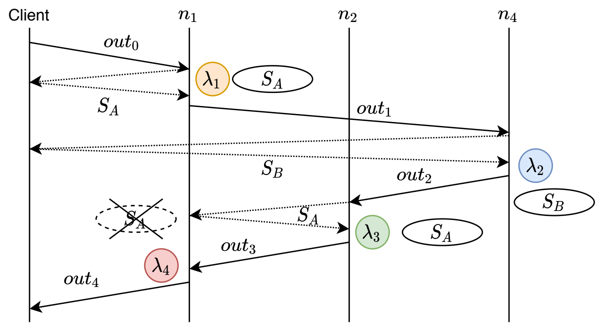

Figure 8: StateLocal execution diagram of the application in Fig. 5 at the edge in Fig. 4.

and the required state of each function is transferred back and forth with every invocation, as illustrated in Fig. 6. This strategy can be easily realized on commercial/open source serverless platforms provided that: (i) the signature of the function (both arguments and return value) supports the client embedding the required state[5]; (ii) the client is aware *a priori* of the state that will be needed by every next function invoked.

StateProp is similar but it makes use of the chaining capability made available by most serverless platforms. As shown in Fig. 7, the client embeds the full state of the application into the function arguments and return values: a function that does not use the embedded state will simply let it pass through, while the others will embed as function arguments the modified state received, which will eventually be returned to the client.

[5]Commercial platforms may limit the amount of data that can be embedded into function invocations [31]: for instance, with AWS this limit is a mere 32 KB, whereas with IBM it is 5 MB, but the overhead has been shown to increase non-linearly with the arguments' size. Only with Microsoft's ADFs it seems there is no theoretical limit, but a compression mechanism is triggered automatically above 60 KB.

Finally, **StateLocal** keeps the state in the edge nodes as illustrated in Fig. 8: rather than embedding the state in the function invocations, only *pointers* are passed. When a lambda function needs a state, it retrieves it via the pointer, and then it becomes its new owner, thus modifying the state's pointer in the subsequent function invocation along the chain. This way, the client will be eventually returned the list of updated pointers to all its states, to use them in subsequent application executions or to withdraw the states from the edge nodes, if ever needed.

4.1. Disclosure of proprietary information

Function composition in any serverless platform introduces the risk of disclosing proprietary information about the application's logic to the platform provider: even though the implementation of a single function can remain private, as only the end-points are needed for the sake of function invocation, some information about the algorithms being executed could be deduced by the way the functions are chained and their usage patterns. This risk becomes even greater with DAG applications made of elementary building blocks, as their richer expressivity could offer further insights about the overall service logic. While we recognize that there can be use cases where disclosing this minimal amount of information to a third party (and potentially a competitor) may not be deemed acceptable, we believe such a risk cannot be considered a show-stopper in the majority of practical scenarios. Therefore, we defer the investigation of the issue to future works in this area.

5. Extension to DAGs

In this section we extend the execution models in Sec. 4 to applications that can be modeled as DAGs. We introduce DAG-specific notation in Sec. 5.1, then address the extension of the stateful execution models in the previous section separately for PureFaaS (Sec. 5.2) and StateProp/StateLocal (Sec. 5.3).

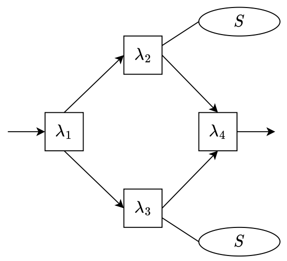

Figure 9: Task-state dependency graph of an example application.

5.1. State consistency

An application modeled as a DAG consists of a set of tasks (we use the terms *tasks* and *functions* interchangeably) with precedences: an edge $\lambda_i \rightarrow \lambda_j$ exists if task λ_i must be executed before task λ_j. The set of precedences define a directed graph, called **task dependency graph**, which cannot contain cycles by definition (recall the 'A' in DAG stands for Acyclic), otherwise the execution would never end. Dependencies can be of the input/output type, i.e., $\lambda_i \rightarrow \lambda_j$ means that the output of task λ_i is needed by task λ_j, or a means of synchronization like in a message-passing system, the distinction is irrelevant to our purposes. To keep the notation consistent with Sec. 4, we indicate with out_i the output of task i, even though we note that in a DAG there could be *multiple*, possibly different, outputs for each task. Support of different task outputs is a mere implementation detail, which does not affect the contribution illustrated in this section, except for a trivial generalization of the derivations.

In our work we address workflows make of *stateful functions*, thus any task may also depend on some states. As in [14] (summarized in Sec. 4), we capture the stateful nature of functions via the **state dependency graph**, which is an undirected graph where edge $\lambda_i \rightarrow S_x$ means that the task λ_j needs to access state S_x. The union of the task dependency graph and the state dependency graph produces the **task-state dependency graph**; an example is illustrated in Fig. 9 for an application made of four tasks, two of which (λ_2 and λ_3) use state S.

It can happen, like in the example in Fig. 9, that multiple tasks need to operate on the same state during a single execution of the application. De-

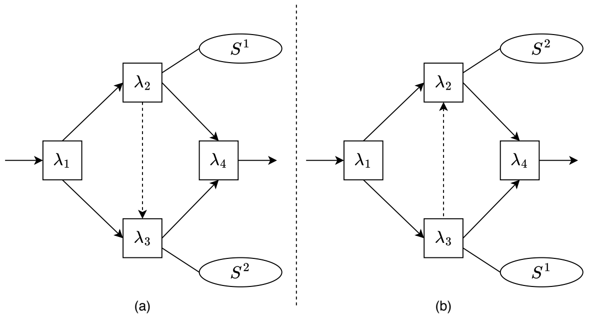

Figure 10: Virtual links added between tasks λ_2 and λ_3, both depending on state S.

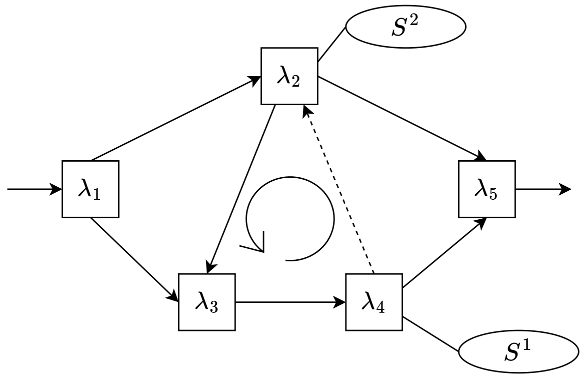

Figure 11: Causal consistency induces a cycle in the augmented task-state dependency graph.

pending on the internal logic of the application, it can happen that: (i) the order of execution does not matter; (ii) λ_2 must be executed before λ_3; (iii) λ_3 must be executed before λ_2. The serverless platform needs to know from the application the temporal order of execution of tasks that depend on a shared state to maintain the **causal consistency of the states**. We then capture such temporal order by augmenting the task-state dependency graph as follows: a virtual edge $\lambda_i \to \lambda_j$, represented as a dashed line in the figures below, is added if λ_i and λ_j both use the same state and λ_i must be executed before λ_j to guarantee causal consistency of the shared state. In the previous example, this means adding an edge $\lambda_2 \to \lambda_3$ if λ_2 has to be executed first (case 'a' in the figure) vs. $\lambda_3 \to \lambda_2$ if λ_3 has to be executed first (case 'b' in the figure). In this section we assign a superscript to states shared by multiple tasks to express the temporal dependency order: in Fig. 10 S^1 must be accessed first, S^2 second.

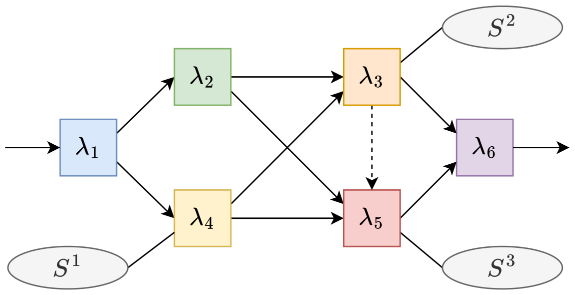

Figure 12: Augmented task-state dependency graph of an example application used to illustrate the extension of the execution models proposed in Sec. 4 to the case of DAG.

The addition of the virtual edges affects the **parallelism** that can be achieved: without state dependencies the DAG in Fig. 9 can executed as $\{\lambda_1, \lambda_2|\lambda_3, \lambda_4$ (where $|$ means that the two tasks can be executed in parallel), but with that in Fig. 10 this is not possible. Furthermore, in order for the augmented task-state dependency graph to remain well-formed, no cycles must exist, also including the virtual edges, i.e., the state-induced temporal order dependencies. Let us consider for instance the application in Fig. 11. The virtual edge $\lambda_4 \rightarrow \lambda_2$ would be needed, however this would create the cycle $\lambda_2 \rightarrow \lambda_3 \rightarrow \lambda_4$, which in turn leads to a *deadlock*: λ_4 cannot be executed until it receives the output of λ_3, which in turn cannot run until it receives the output of λ_2, which cannot access state S before the execution of λ_4 is complete. Since this kind of situations can be detected by the application, and they reflect a logic design issue, we assume hereafter that our applications of interest are only those with an acyclic augmented task-state dependency graph.

Key point: *With stateful DAG applications, the causal consistency of the execution must be guaranteed. We propose to do so by defining, for each state, the order in which the tasks depending on it will be executed, as reflected by virtual edges in the augmented task-state dependency graph.*

5.2. Pure Faas

The extension of PureFaaS approach is straightforward. At each step, there is a set of callable functions, which are all those whose task-state dependen-

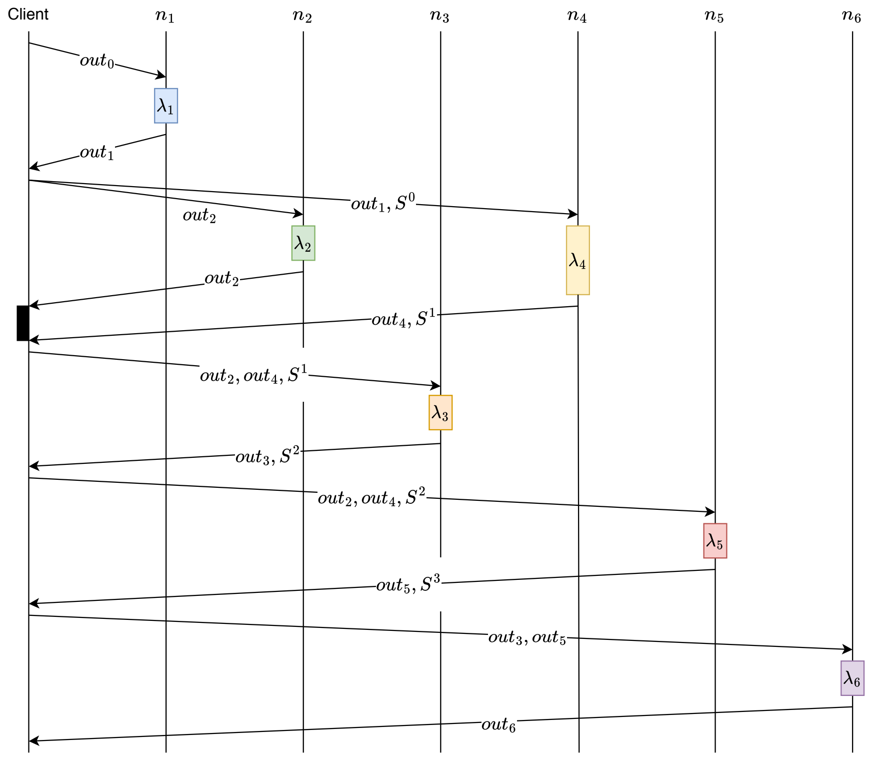

Figure 13: Execution of the example DAG application in Fig. 12 with PureFaaS.

cies are verified. The client can call them in parallel or in sequence (order is arbitrary) depending on its internal logic and capabilities.

To better illustrate our point, we make use of the example application in Fig. 12. The sequence diagram with PureFaaS is shown in Fig. 13, where we assume for better readability that function λ_i is always executed on edge node i. Moreover, with a slight overload of notation, with S^i we not only indicate the temporal order dependency of the state S in the augmented task-state dependency graph (as defined in Sec. 5.1), but we also refer to its subsequent modifications: S^0 is the initial state before the workflow invocation, S^1 its modified version after the execution of λ_4 (which connects to S^1 in the graph), and so on until the final version S^3 of the state at the end of the workflow. As

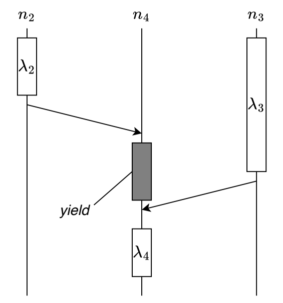

Figure 14: Example of execution of λ_4 with yield in the application in Fig. 9.

can be seen, the tasks λ_2 and λ_4 are executed in parallel, but the client has to wait for the slowest of the two (in the example: λ_4) before it can continue: this waiting time is represented in the diagram with a black rectangle. Apart from the opportunity to parallelize some tasks, there is no other change with respect to PureFaaS when used with chains of functions invocations.

Key point: PureFaaS remains the same with chains and DAGs.

5.3. StateProp/StateLocal

On the other hand, StateProp and StateLocal cannot be used with DAG applications without modifications like PureFaaS. There are different reasons for this, which we will explain hereafter. Briefly, we recall that StateProp/StateLocal both rely on the worker invoking the next function as the current one is complete; they differ on the way they manage the state: StateProp carries it along the function invocation chain, whereas StateLocal keeps it within the edge node that last used it.

First, it can happen that a task has more than one input, e.g., λ_4 in Fig. 9: in this case, both λ_2 and λ_3 want to execute λ_4 at the end of their respective tasks, so the whole notion of "every function executes the next one" is not as well-defined as with a chain of functions. We address this point by introducing the concept of *asynchronous calls*: when a function terminates, it always invokes the next function(s), i.e., its direct descendants according to the DAG, but this only triggers the execution of a task if all its inputs are available. If this is not the case, then the output of the predecessor is stored temporarily on the edge node and the function *yields* (the term is borrowed from asynchronous programming models and languages). A graphical illustration of the yield operation can be found in Fig. 14. Supporting this pattern increases the complexity of the serverless platform on the edge nodes, which have to maintain an ephemeral state for each incomplete operation. Such asynchronous calls, by themselves, do not solve the problem: in the example in Fig. 14 we have assumed that λ_2 and λ_3 both invoke λ_4 on the same edge node n_4, but in general this is not a piece of information that they have: the serverless platform treats every function call independently from others, which can result into the execution of the same function on different edge nodes. So, for instance, λ_2 may invoke λ_4 on edge node n_x ($x \neq 4$), which would result in a deadlock, since both the instances of λ_4 on n_x and n_4 will yield forever waiting for an input that will never come.

To support StateProp/StateLocal it is necessary that the mapping between functions and edge nodes is known to all the workers at least during a single execution of a DAG application. This way, we can make sure that all the workers will invoke the execution of descendants on the same edge nodes (i.e., n_4 from both λ_2 and λ_3 in the previous example). The main practical consequences are two: (i) the information on the mapping between functions and edge nodes has to be carried along the execution DAG, which slightly increases the protocol overhead; (ii) there must be a process that is able to "resolve" all the functions at the time the DAG is invoked (e.g., this can be done by the client), which can increase the start-up latency.

Frustratingly, all this is not sufficient to support StateProp/StateLocal. Consider again the trivial example in Fig. 9: both λ_2 and λ_3 depend on the same state S. Irrespective of the relative order, it will be necessary to transfer the updated state, modified by the first one to be executed (e.g., λ_2), to the other one (e.g., λ_3). But there is no invocation path between the two, i.e., λ_3 is not a descendant of λ_2 in the DAG, which makes it impossible to rely on the propagation of the state alone. Therefore, we propose a second modification: rather than embedding the state in the function arguments (or their references, for StateLocal), every function accessing a state will send it directly to the next

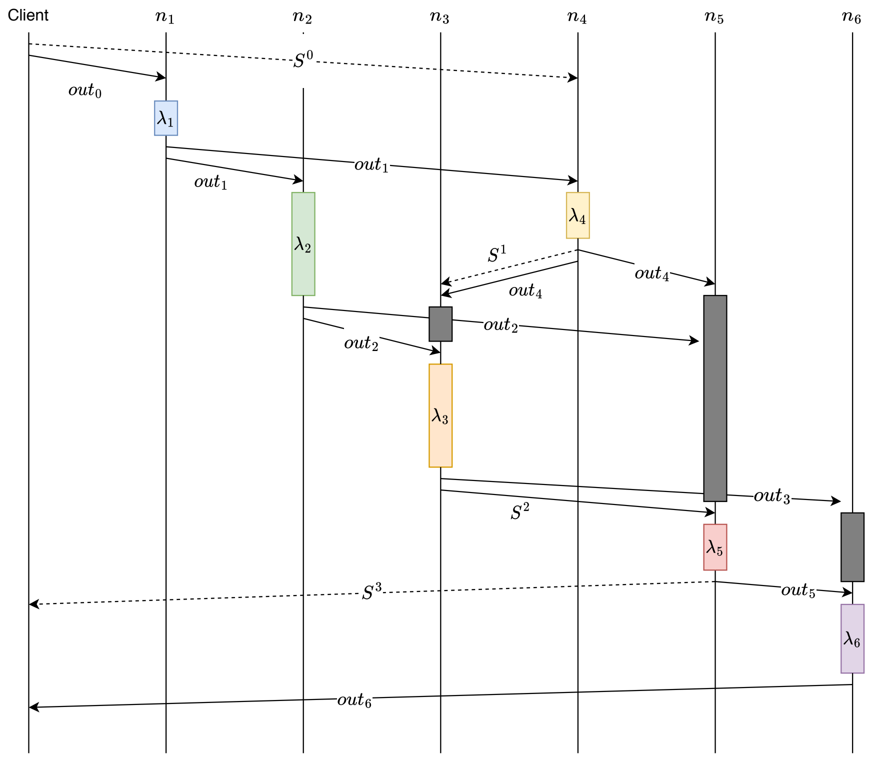

Figure 15: Sequence diagram of the execution of the example DAG application in Fig. 12 with StateProp/StateLocal.

worker that will use it, according to the state dependency graph and causal consistency constraints, both already known. Thus, the workers of a function must be ready to not only receive asynchronous calls, and temporarily store their arguments, but to also store updates states, also arriving asynchronously. We note that these modifications are not required with PureFaaS, as illustrated in Sec. 5.2 above, because the client provides implicit synchronization with that model.

In Fig. 15 we illustrate the sequence diagram with the application in Fig. 12 of StateProp/StateLocal, the only difference between the two being that with StateLocal the state has to be retrieved from the last owner (unless the worker executes on the same edge node). The amount of data transmitted with StateProp

Table 1: Traffic exchanged for the different policies (Chain results are from [14]).

Chain/DAG	Policy	Traffic exchanged
Both	Pure FaaS	$= \sum_{i=0}^{N} out_i + \sum_{i=1}^{N-1} out_i + 2\sum_{j=1}^{M} deg(S_j)S_j$
Chain	StateProp	$= \sum_{i=0}^{N} out_i + (N+1)\sum_{j=1}^{M} S_j$
Chain	StateLocal	$\leq \sum_{i=0}^{N} out_i + \sum_{j=1}^{M} deg(S_j)S_j$
DAG	StateProp	$= \sum_{i=0}^{N} deg^{+}(\lambda_i)out_i + \sum_{j=1}^{M} (1 + deg(S_j))\, S_j$
DAG	StateLocal	$\leq \sum_{i=0}^{N} deg^{+}(\lambda_i)out_i + \sum_{j=1}^{M} (1 + deg(S_j))\, S_j$

then becomes:

$$D_{sp}^{DAG} = \sum_{i=0}^{N} deg^{+}(\lambda_i)out_i + \sum_{j=1}^{M} (1 + deg(S_j))\, S_j, \tag{1}$$

where $deg^{+}(\lambda_i)$ is the out-degree of vertex λ_i, i.e., the number of its direct descendants. For StateLocal, D_{sp}^{DAG} is an upper bound.

Key point: StateProp and StateLocal can support DAG applications, but the following major modifications are needed: workers must support asynchronous function calls, the binding between functions and edge nodes must be known to all workers during a single DAG execution, and the states cannot be propagated along with the arguments. Collectively, these changes increase the complexity of the software to be run on edge nodes and the protocol overhead, as well as exacerbate possible concerns on disclosing proprietary information (see Sec. 4.1).

Table 1 summarizes the amount of traffic exchanged with all the schemes.

6. Performance evaluation

In this section we illustrate the prototype we have realized of PureFaaS, StateProp, and StateLocal (Sec. 6.1) and we report the results obtained in an emulated network (Sec. 6.2), which complement the simulations experiments in [14].

6.1. Implementation

We have implemented the execution models in ServerlessOnEdge, which is a decentralized framework to dispatch stateless FaaS functions at the edge, developed and maintained within our research group. The software is open source with a permissive MIT license and publicly available on GitHub[6] [32]. In ServerlessOnEdge the clients request the invocation of functions via *e-routers*, which play the role of intermediary with the serverless platforms by forwarding stateless requests to one of many destinations available depending on the load and network conditions. For the purpose of evaluating the performance of protocols and algorithms in controlled and repeatable conditions, we have also implemented so-called *e-computers*, which emulate serverless platforms with a given configuration, in terms of computation speed, memory, number of containers, etc. ServerlessOnEdge uses Google's gRPC[7] for communication among clients, e-routers, and e-computers.

PureFaaS was implemented as follows: (i) the client embeds the required states within the arguments at each function invocation; (ii) the e-computers return the embedded states as part of the function return value; (iii) multiple functions are invoked if the precedences are met (only in a DAG). On the other hand, implementing StateProp and StateLocal required more structural upgrades. We start with **StateProp**, which requires any intermediate e-computer to invoke the next function(s) in the chain/DAG and pass on all the application's states. First, we have implemented asynchronous function calls: they return immediately an empty acknowledgment, while the real output is provided to the client as an unsolicited response-only message by the last e-computer in the chain. Furthermore, an e-computer in our system does not know the destination of the next function in the chain: to solve this problem, we have installed on every e-computer a *companion e-router* that is used to dispatch the function calls generated by its e-computer as part of the function chain execution. To obtain consistent performance of StateProp for both chains and DAGs, we have implemented the same state propagation mechanism, even though this means that not all possible combinations of DAG and state dependencies are supported, see Sec. 5.3 (all those in the experiments are feasible).

[6]`https://github.com/ccicconetti/serverlessonedge/`, tag $\geq$ 1.2.1.
[7]`https://grpc.io/`, last accessed Aug 12, 2022

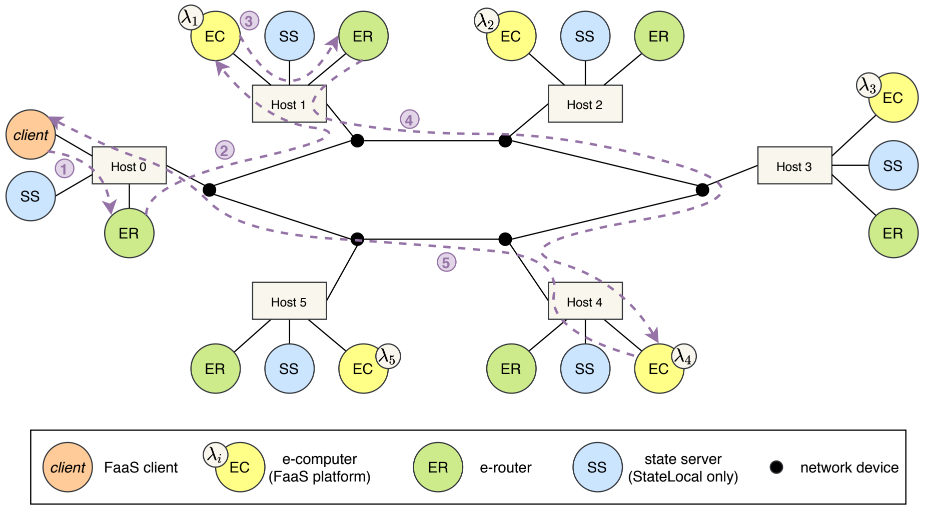

Figure 16: Scenario used for the validation with ServerlessOnEdge.

An example of invocation of the two-function chain $\{\lambda_1, \lambda_4\}$ is shown in Fig. 16 in a network with 5 edge nodes (from Host 1 to Host 5), while the client application is on Host 0. As can be seen, the e-router on Host 0 is used by the client for the invocation of the first function in the chain (λ_1, forwarded to Host 1), while the e-router on Host 1 receives the next invocation to λ_4 (forwarded to Host 4). The e-computer on Host 4 does not need to go through its companion e-router as it can send the final response to the client on Host 0. The system messages had to be modified so that the chain or DAG is embedded in every function request, along with the callback end-point for the final response.

Finally, **StateLocal** required the same upgrades as StateProp and a few others: (i) the system messages had to support remote states, i.e., states that are not embedded in the function call/response, but only referenced indirectly (in our case by their name and an end-point); (ii) the states are managed by new components called *state servers*, which are simple in-memory KVSs co-located with each e-computer and client, as shown in Fig. 16; (iii) the flow of messages is exactly the same as with StateProp, but at each function invocation the e-computer retrieves the remote states needed and then copies them into its local state server; to do this, the state dependencies were also embedded in the function invocation request messages.

6.2. Results

We have used the prototype implementation to carry out a campaign of experiments in an emulated environment, using mininet[8] to reproduce the topology illustrated in Fig. 16 above. The experiments are reproducible by means of the scripts published in the ServerlessOnEdge GitHub repository (experiment numbers: `400` and `401`), which also includes pointers to the raw results obtained in the research group servers. Since we are only interested in measuring the traffic, and its induced latency, for the different execution models proposed in Sec. 4, we use a single client, i.e., there is no contention on processing resources.

We have carried out two batches of experiments, respectively with function chains and DAGs. Let us start with **function chains**: the client executes back-to-back function chains of constant length L (3 or 6), in number of functions, where each function is drawn randomly from $\lambda_1, \ldots, \lambda_5$, possibly with repetitions. We assume that the application has S states (3 or 6), where state s_i has size $(1+i) \times 10\ kB$ (0-based indexing); each state depends on a randomly drawn set of functions, with random cardinality drawn from 0 (no dependencies) and L (all functions in the chain depend on the state). The size of the input argument and return value is assumed to be the same and equal to A (10 kB or 100 kB).

In Fig. 17 we compare the average end-to-end delay obtained with the execution models in all the scenarios separately, as the link rate between network devices increases from 1 Mb/s to 100 Mb/s. Note that the results are plotted in logarithmic scale in both axes. In the top left plot, PureFaaS and StateProp are almost overlapping: this is because the size of both states and argument is relatively small. Instead, until the link rate is below 20 Mb/s, StateLocal has a much lower average delay, thanks to its wiser only-as-needed transfer of states. However, with higher link rates, the advantage diminishes progressively until

[8]`http://mininet.org/`, last accessed Aug 12, 2022

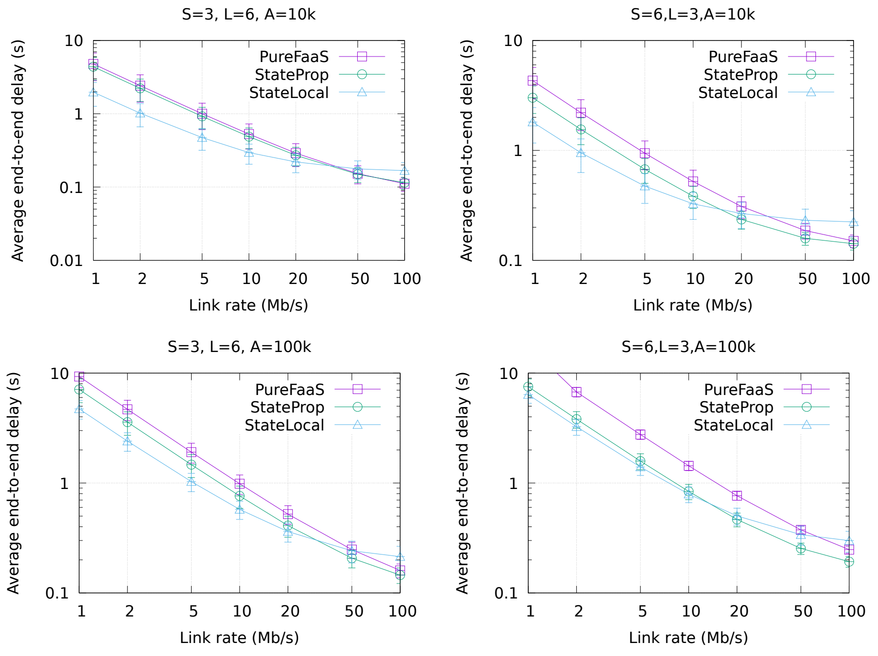

Figure 17: Function chains: Average end-to-end delay.

the delay becomes higher than that of PureFaaS/StateProp with 50 Mb/s and 100 Mb/s link rates: at such high connectivity rates, the data transfer becomes comparable with (or higher than) the time to establish the TCP connections to retrieve/update the states. This disadvantage of StateLocal could be reduced by employing persistent TCP connections towards the state servers or using a connection-less protocol, such as QUIC[9]. In the opposite scenario, i.e., bottom right plot in Fig. 17, the performance with StateProp and StateLocal are comparable, except for high link rates: this is because the chains are shorter than in the other scenario and the data transfer is dominated by the input/output argument, which is treated the same by the two schemes. The other cases, i.e., top right and bottom left in Fig. 17, are intermediate, with StateLocal achieving a better performance for all slow link rates, and PureFaaS always lying on top of StateProp.

[9]QUIC: A UDP-Based Multiplexed and Secure Transport – `https://datatracker.ietf.org/doc/html/rfc9000`, last accessed Aug 12, 2022.

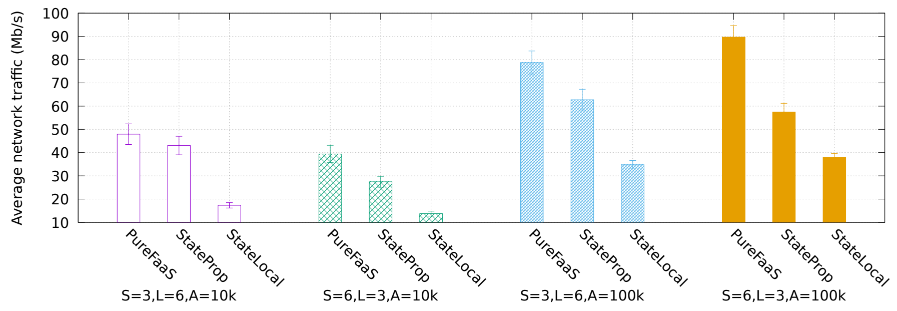

Figure 18: Function chains: Network traffic, with link rate 100 Mb/s.

Key point: *With function chains, when taking into account realistic protocol overheads, there is a trade-off between keeping the state local to edge nodes (StateLocal) and embedding it into function invocations, depending on the state size and network speed. StateProp always performs better than PureFaaS.*

We then provide a direct measure of the overhead in Fig. 18 by showing the average network traffic in all the scenarios, for the link rate 100 Mb/s, which is the one where StateLocal exhibits worst performance. As can be seen (in linear scale in this plot) the traffic generated with PureFaaS is always greater than that generated with StateProp, which in turn is always greater that that with StateLocal. We note that, unlike our previous results in [14], the data reported here include all protocol overhead, since the traffic is measured on the ports of the emulated network switches. The advantage of StateLocal is more prominent with a smaller argument size, i.e., with $A = 10\ kB$, but it is significant in all cases.

Key point: *With function chains, even with a high network speed, StateLocal has a significantly lower overhead than the other schemes, in terms of the traffic rate required, but this not always translates into a lower end-to-end latency.*

We now move to the **DAG** case, for which we considered applications made of a sequence of stages, each with a *branch* function that spawns multiple stateless calls followed by a stateful *collect* task. This structure is very typical of ML applications, which are today dominant in cloud and edge environments: this was confirmed by the study [33], where the authors have synthesized an

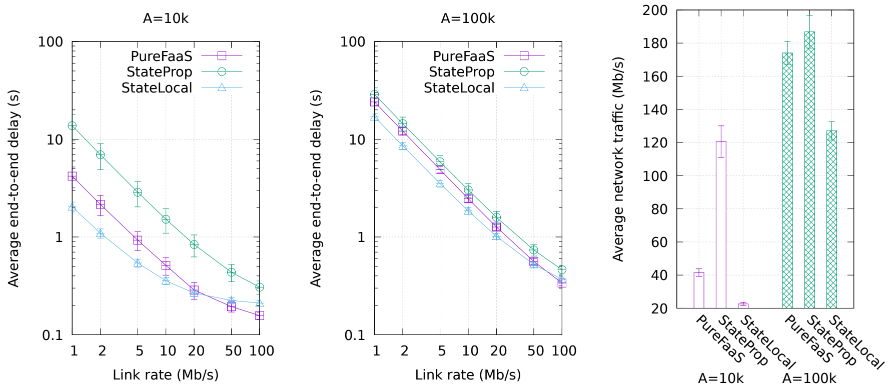

Figure 19: DAGs: Average delay, with variable link rates, and network traffic, with link rate 100 Mb/s.

artificial generator of workloads that captures accurately, in a statistical sense, the behavior of real-life applications in the wild. In the results below, we have used 3 stages with 5 branches per stage. The functions are selected randomly among those on the edge nodes, and the state dependencies are also random using the same approach as with chains. In Fig. 19 we show the average delay, with increasing link rate from 1 Mb/s to 100 Mb/s (with a log-log plot), as well as the network traffic only for the link rate 100 Mb/s. As above, we have used two argument sizes: $A = 10\ kB$ and $A = 100\ kB$. Unlike with function chains, in this scenario we find that PureFaaS outperforms StateProp in all conditions, with the advantage being more prominent with a smaller argument size. This is because the total number of functions called is much higher than in the chain scenario, which penalizes significantly the embedding of all the states in invocations and responses. Such a fee is not paid by StateLocal, which only transfers *references* to states and performs best in all conditions except with very high link rates, due to the overhead of state retrieve/update operations, as already discussed.

Key point: *With a high number of functions in DAGs, state propagation is only effective if references are carried within the function invocations and responses.*

7. Conclusions

In this paper we have explored the support of stateful applications on serverless platforms distributed on edge nodes. We have focused on the problem of transferring the state along an invocation of functions in chain and DAG workflows, and we have identified three alternative schemes, with different characteristics. We have developed a prototype implementation to prove the feasibility of our approaches and to measure performance with realistic protocol overheads. The results have shown that propagating the state along the chain of function invocations can reduce significantly the communication overhead. This leads to lower end-to-end application latency, especially with limited connectivity. However, with large DAG workflows, embedding the state for propagation is not effective anymore: in these cases it becomes mandatory to store the states locally on edge nodes and carry their references instead.

Acknowledgment

This work was partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 957337, project MARVEL.

[1] M. Campbell, Smart Edge : The Center of Data Gravity Out of the Cloud, Computer 52 (December) (2019) 99–102. doi:10.1109/MC.2019.2948248.

[2] N. Mohan, L. Corneo, A. Zavodovski, S. Bayhan, W. Wong, J. Kangasharju, Pruning Edge Research with Latency Shears, ACM HotNets 2020.

[3] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. K. Gaire, S. Dustdar, Serverless Edge Computing: Vision and Challenges, AusPDC 2021.

[4] A. Khandelwal, A. Kejariwal, K. Ramasamy, Le Taureau: Deconstructing the Serverless Landscape & A Look Forward, ACM SIGMOD 2020.

[5] C. F. Fan, A. Jindal, M. Gerndt, Microservices vs serverless: A performance comparison on a cloud-native web application, CLOSER 2020.

[6] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The Rise of Serverless Computing, Commun. ACM 62 (12) (2019) 44–54.

[7] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, F. Leymann, FaaSten your decisions: A classification framework and technology review of function-as-a-Service platforms, Journal of Systems and Software 175.

[8] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. Richard Yu, T. Huang, When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues, IEEE Wireless Communications (2021).

[9] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu, Serverless computing: One step forward, two steps back, CIDR 2019.

[10] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, A. Iosup, The State of Serverless Applications: Collection, Characterization, and Community Consensus, IEEE Transactions on Software Engineering 5589 (c) (2021).

[11] M. Shahrad, J. Balkind, D. Wentzlaff, Architectural implications of function-as-a-service computing, MICRO 2019.

[12] T. Rausch, A. Rashed, S. Dustdar, Optimized container scheduling for data-intensive serverless edge computing, Future Generation Computer Systems 114 (2021) 259–271.

[13] Z. Jin, Y. Zhu, J. Zhu, D. Yu, C. Li, R. Chen, I. E. Akkus, Y. Xu, Lessons learned from migrating complex stateful applications onto serverless platforms, ACM APSys 2021.

[14] C. Cicconetti, M. Conti, A. Passarella, On Realizing Stateful FaaS in Serverless Edge Networks: State Propagation, IEEE SMARTCOMP 2021.

[15] S. Alirezazadeh, L. Alexandre, Optimal Algorithm Allocation for Single Robot Cloud Systems, IEEE Transactions on Cloud Computing (to appear).

[16] F. Carpio, M. Delgado, A. Jukan, Engineering and Experimentally Benchmarking a Container-based Edge Computing System, IEEE ICC 2020.

[17] A. Mampage, S. Karunasekera, R. Buyya, A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions, ACM Computing Surveys (to appear).

[18] L. Wang, L. Jiao, T. He, J. Li, H. Bal, Service Placement for Collaborative Edge Applications, IEEE/ACM Transactions on Networking (2020).

[19] H. Liao, X. Li, D. Guo, W. Kang, J. Li, Dependency-aware Application Assigning and Scheduling in Edge Computing, IEEE Internet of Things Journal 4662 (c) (2021).

[20] A. C. Nicolaescu, S. Mastorakis, I. Psaras, Store edge networked data (SEND): A data and performance driven edge storage framework, IEEE INFOCOM 2021.

[21] S. H. Mortazavi, M. Salehe, B. Balasubramanian, E. De Lara, S. Puzhavakathnarayanan, SessionStore: A Session-Aware Datastore for the Edge, IEEE ICFEC 2020.

[22] C. Wu, V. Sreekanti, J. M. Hellerstein, Transactional Causal Consistency for Serverless Computing, ACM SIGMOD 2020.

[23] C. Zhang, H. Tan, H. Huang, Z. Han, S. H. Jiang, N. Freris, X. Y. Li, Online dispatching and scheduling of jobs with heterogeneous utilities in edge computing, ACM MobiHoc 2020.

[24] Z. Jia, E. Witchel, Boki: Stateful Serverless Computing with Shared Logs, ACM SIGOPS 2021.

[25] R. Hetzel, T. Kärkkäinen, J. Ott, μactor: Stateful Serverless at the Edge, MobileServerless 2021.

[26] S. Shillaker, P. Pietzuch, FAASM: Lightweight isolation for efficient stateful serverless computing, USENIX ATC 2020.

[27] B. Carver, J. Zhang, A. Wang, Y. Cheng, In search of a fast and efficient serverless DAG engine, IEEE/ACM PDSW 2019.

[28] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, C. Das, Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms, ACM SoCC 2021.

[29] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. Mcmahon, C. S. Meiklejohn, Durable Functions : Semantics for Stateful Serverless, PACMPL 2021.

[30] W. Zhang, V. Fang, A. Panda, S. Shenker, Kappa: A programming framework for serverless computing, ACM SoCC 2020.

[31] P. Garcia Lopez, M. Sanchez-Artigas, G. Paris, D. Barcelona Pons, A. Ruiz Ollobarren, D. Arroyo Pinto, Comparison of FaaS orchestration systems, IEEE/ACM UCC 2018.

[32] C. Cicconetti, M. Conti, A. Passarella, A Decentralized Framework for Serverless Edge Computing in the Internet of Things, IEEE Transactions on Network and Service Management 18 (2) (2020) 2166–2180.

[33] H. Tian, Y. Zheng, W. Wang, Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud, ACM SoCC 2019.