SDN-based fog and cloud interplay for stream processing

Michal Rzepka, Piotr Borylo, Marcos Assunção, Artur Lasoń, Laurent Lefèvre

Highlights

SDN-based Fog and Cloud Interplay for Stream Processing

Michał Rzepka, Piotr Boryło, Marcos D. Assunção, Artur Lasoń, Laurent Lefèvre

- Dynamic workload deployment is highly affected by resource allocation algorithms
- Transmission latency is a key factor of the operator allocation process
- Considering both resource utilization and request execution graph yields best results
- Two of the proposed algorithms provide both optimal resource usage and success rates

SDN-based Fog and Cloud Interplay for Stream Processing

Michał Rzepka[a,*], Piotr Boryło[a], Marcos D. Assunção[b], Artur Lasoń[a], Laurent Lefèvre[c]

[a]*AGH University of Science and Technology, Department of Telecommunications, Krakow, Poland*
[b]*Software Engineering and IT Department, École de Technologie Supérieure, Montreal, QC, Canada*
[c]*Inria Avalon, LIP Laboratory, École Normale Supérieure de Lyon, University of Lyon, France*

Abstract

This paper focuses on SDN-based approaches for deploying stream processing workloads on heterogeneous environments comprising wide-area networks, cloud and fog resources. Stream processing applications impose strict latency requirements to operate appropriately. Deploying workloads in the fog reduces unnecessary delays, but its computational resources may not handle all the tasks. On the other hand, offloading the tasks to the cloud is constrained by limited network resources and involves additional transmission delays that exceed latency thresholds. Adaptive workload deployment may solve these issues by ensuring that resource and latency requirements are satisfied for all the data streams processed by an application. This paper's main contribution consists of dynamic workload placement algorithms operating on stream processing requests with latency constraints. Provisioning of computing infrastructure exploits the interplay between fog and cloud under limited network capacity. The algorithms aim to maximize the ratio of successfully handled requests by effectively utilizing available resources while meeting application latency constraints. Experiments demonstrate that the goal can be achieved by detailed analysis of requests and ensuring balanced computing and network resources utilization. As a result, up to 30% improvement over the reference algorithms in success rate is observed.

Keywords: stream processing, fog computing, edge computing, SDN

1. Introduction

Fog computing, according to Cisco, is a highly virtualized platform that provides computing, storage, and networking services located at the Internet edge [1]. Other companies also support fog computing under various terms: Edge Computing (Akamai), Intelligent Edge (Intel), Cloudnet (Microsoft), or Multi-access Edge Computing (European Telecommunications Standards Institute). Fog infrastructure is often heterogeneous as various devices contribute to the overall computing power, including sensors, wearables, smartphones, base stations, vehicles, or network devices with extended functionality.[1]

1.1. Use cases

Deploying computing services in various locations between the data sources and the cloud is a popular concept. It allows for meeting the constraints imposed by modern applications, *e.g.* location awareness, low latency, mobility support, or geographical distribution of services. The most frequent use cases for fog computing include the Internet of Things (IoT), operational monitoring of large infrastructures, smart grids, smart cities, and intelligent buildings. The examples of stream processing applications are:

- intelligent traffic control [2–6] - processing of data collected from mobile sensors to provide optimal route selection, traffic monitoring and prediction, congestion avoidance, accident and anomaly detection or parking assistance;
- surveillance and event monitoring [2, 4, 7] - processing of data collected by geographically distributed sensors to perform object detection, face recognition or vandalism and accident detection;
- industrial process automation [2, 7, 8] - analysis of sensor data including product and equipment surface inspection, monitoring of production process parameters, or staff tracking.

1.2. Requirements and challenges

These emerging use cases require processing continuous data streams under very short delays - real-time data analysis

*Corresponding author
Email addresses: mrzepka@agh.edu.pl (Michał Rzepka), borylo@agh.edu.pl (Piotr Boryło), marcos.dias-de-assuncao@etsmtl.ca (Marcos D. Assunção), lason@kt.agh.edu.pl (Artur Lasoń), laurent.lefevre@ens-lyon.fr (Laurent Lefèvre)

[1] Cisco introduced an IOx platform that combines IOS and open-source Linux to support fog computing.

and feedback generation are essential for applications to operate correctly. For example, industrial automation systems impose strict requirements on delays to ensure that critical issues (*e.g.* machine overheating) are handled in a few milliseconds [7, 8]. In addition, an application may generate massive amounts of data, especially when video stream analysis is involved, as in the case of traffic monitoring based on CCTV cameras (surveillance and event monitoring) or on-board cameras (intelligent traffic control) [3]. As a result, enough network and computational resources are required to handle all the data streams successfully. The problem is aggravated by the growing number of Internet-connected devices, leading to the vast increase in the volume, variety, and velocity of data generated.

Several frameworks are available for conducting scalable and efficient stream processing, most of which follow a dataflow approach, *i.e.* data streams are processed as they traverse a graph of operators that perform algebra-like operations or user-defined functions. A data stream can consist of discrete signals, event logs, monitoring data, and time-series information and can present a high input data rate that stresses communication and computing infrastructure. Some of these stream processing frameworks can take advantage of distributed computing environments. For example, clouds can offer good performance, robustness, and elasticity - the ability to allocate or release resources to match the application workload. However, cloud Data Centers (DCs) are usually distant from the network edge, which can impact the performance when directing latency-sensitive data stream workloads to the cloud for processing.

Although many data stream applications can benefit from fog computing by using resources geographically close to data sources, fog computing is not simply a replacement for clouds and their practically unlimited computing resources and benefits from economies of scale. Fog computing is, therefore, a paradigm that extends the cloud to the network edge. As a result, using fog and cloud can simultaneously combine the cloud's scalability, global presence, elasticity, and sustainability with the latency reduction that fog computing offers. In an attempt to deploy fog computing widely, researchers and developers have customized mobile operating systems to offer fog features, provided middleware for deploying applications transparently across the cloud-thing continuum (local devices, fog, cloud) and designed multiple resource provisioning algorithms. This ecosystem reflects the process of *softwarization* followed by telecommunication infrastructure. A proper design of resource provisioning algorithms is, however, still a challenge. The workload deployment mechanisms should meet the requirements of the applications and combine advantages of both fog and cloud infrastructure to achieve maximum effectiveness while processing data streams. A proper resource allocation as presented in this paper can meet these requirements.

1.3. Our work

Our work focuses on resource provisioning and operator placement in a distributed environment comprising fog and cloud computing. Such a geographically distributed infrastructure requires efficient data transmission between fog and cloud resources. Wide Area Software Defined Networking (WA-SDN) is a promising solution due to its separation of data and control planes [9]. The control plane, denoted as an Software-Defined Network (SDN) controller, is a software entity, deployed separately from the network devices, that delivers forwarding rules to hardware and virtual devices via the OpenFlow protocol [10]. In addition, the SDN controller can communicate with the cloud and fog orchestration software via its northbound API.

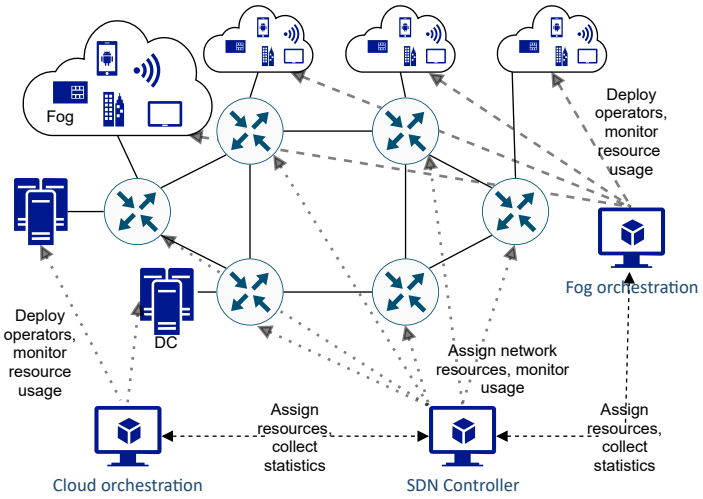


Figure 1: The concept of SDN-based cloud and fog interplay.

The previous work [11] presented a WA-SDN controller that communicates with fog and cloud orchestration software and coordinates the use of fog and cloud resources, as shown in Figure 1[2]. Since the SDN controller has complete knowledge of the network state, it could perform global traffic engineering to meet the latency requirements of stream processing applications.

The present work focuses on jointly optimizing the placement of stream processing operators across fog and cloud computing infrastructure and the routing of data streams over an SDN Wide Area Network (SD-WAN). The main contributions are the proposed heuristic algorithms that dynamically handle requests that specify data streams arriving at the infrastructure. In addition, the proposed mechanisms aim to meet latency requirements while respecting fog and network resource constraints and considering the delay resulting from data transmission from fog to data centers via the WAN. The proposed solution prevents wasting fog resources, retaining them to handle the requests most affected by the delays. Furthermore, the proper selection of connections between operators across the Wide Area Network (WAN) ensures an optimized use of network resources.

The rest of this paper is organized as follows. The next section discusses the related work. Section 3 provides the necessary background. It also contains a detailed description of an integrated architecture, application execution graph, request model, and formal problem statement. Section 4 presents existing and novel algorithms with an analysis of their properties.

[2]There are fog instances associated with every network node, not shown in the figure for simplification reasons.

Experimental setup and results are provided in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

Traditionally, stream processing applications are designed and optimized for deployment in cloud computing environments. For example, previous work investigated remote invocation techniques for stream processing applications to reduce response time by using parallel processing on a cluster of CPUs [12]. In addition, workflow-based mechanisms for running stream processing applications on a set of virtual machines in the cloud improved the overall application throughput [13]. Another framework solves the problem of allocating bandwidth in a data center internal network during runtime based on additional information obtained from the application layer [14].

The cloud computing settings considered in previous work are usually not suited to emerging stream processing applications due to the latency overhead introduced by communication between distant data centers. That is why fog computing has gained popularity. Tutorial work [7] proposes a general framework for data stream processing in the fog and indicates key differences with analogous cloud deployments. In [2], authors proposed a mature framework to deploy stream processing applications on a cluster of heterogeneous edge devices. Moreover, a distance-based approach to distributing stream processing tasks to nodes with limited computing capabilities showed similar performance [15].

Fog computing infrastructure, however, is limited by the power of its computing resources, their mobility, and their availability. There is, as a result, a trend towards considering deployments of infrastructure comprising cooperating fog and cloud resources, not only for stream processing applications. Several studies consider frameworks for running applications in cooperating cloud and fog infrastructure [16, 17]. Similarly, the work [4] investigates performance improvements achieved by moving an application closer to the edge.

SDN will increasingly enhance hybrid solutions comprising centralized clouds and fogs located at the network edge. In the work by Zamani *et al.* [18], the SDN controller establishes data routes to utilize not only edge and cloud resources but also network data centers adjacent to the transit network nodes. In [19], the SDN facilitates the distribution of computing tasks between fog and cloud in vehicular networks. SDN is also used to offload traffic between fog nodes ensuring end-to-end bandwidth guarantees [20].

The work by Yin *et al.* [21] and Hernandez *et al.* [6] focus on deploying stream processing applications in cooperating fog and cloud environments but ignoring the latency and time-efficiency issues. Instead, in Yin *et al.*'s work, fog nodes offer idle resources to preprocess data before sending them to the cloud with the assistance of the SDN controller. On the other hand, Hernandez *et al.*'s solution reduces the volume of data streams redundantly transmitted to the cloud.

Kallel *et al.* [22] and Li *et al.* [3] consider time-related issues but solely in the context of the processing time needed to handle data streams in the computing infrastructure. Kallel *et al.*'s introduces an approach to business process modeling for the IoT and non-IoT resources with different capacity and quality of service constraints. Li *et al.*, on the other hand, mainly focused on energy efficiency, proposing detailed end-to-end energy models for convergent fog-cloud infrastructures and applying them to the data stream analysis produced by cameras embedded in vehicles.

A few works examine the deployment of data stream processing applications in both fog and cloud infrastructure, considering the end-to-end latency resulting from computing and data transmission between fog and cloud via WAN. The communication with the cloud is modeled as an additional latency during the execution of various stream processing applications [5, 23–25]. Veith *et al.* [23] propose a static optimization model with operator topologies as series of parallel graphs and constrained by application requirements and network capabilities. Veith *et al.* extend their work by modeling the reconfiguration of data stream processing applications as a Markov Decision Process (MDP) and using reinforcement learning algorithms to solve it [24]. De Maio and Kimovski [25] also consider latency, proposing a model where the authors formulated an optimization problem and proposed a solving method considering three objectives, namely response time, reliability, and monetary cost. The smart parking scenario in [5] has numerous IoT sensors that generate a vast amount of data to be processed in the edge-fog-cloud continuum with respect to metrics specific to a particular smart parking scenario.

SpanEdge [26] reduces the latency incurred by WAN links by placing stream processing operators across an infrastructure composed of geo-distributed *near-the-edge* data centers. *SpanEdge* enables programmers to group stream processing graph operators either as a *local-task* or a *global-task*. Thus, the proposed algorithm is not responsible for determining the placement of each particular operator.

The works most similar to ours in terms of the system model, research problem, and motivation are those by Veith *et al.* [27] and Mehran *et al.* [28]. The authors of both works focus on end-to-end latency requirements of applications represented by graphs similar to those we consider. Veith *et al.*'s solution requires application profiling. To make the solution scalable and efficient, they divide operations into regions suitable for cloud or fog resources. Mehran *et al.*'s *many-to-one* matching algorithm assigns operators to resources based on two novel ranking models [28].

Table 1 summarizes how the related work handles similar problems. Some existing work aims at reducing latency during deployment of stream processing requests represented as graphs [2, 7, 12–15], but without considering cooperation between fog and clouds throughout WAN and without a realistic latency model. On the other hand, some approaches are focused on the fog and cloud interplay without (e.g. [4, 16, 17]) or with (e.g. [18–20]) a proper WAN latency model, but not handle stream processing requests represented as a graph. Some works consider stream processing in the fog-cloud infrastructure [6, 21], but disregard the nature of the request and issues related to implementation details. Therefore, the aspects of latency awareness and a detailed graph-based representation of

Table 1: Summary of related work.

Work	Fog and cloud interplay	Latency awareness		Stream processing request represented as a graph	Online optimization for concurrent requests	Transparency for (stream processing) application
		Realistic WAN network model	Aimed at minimizing latency			
[2, 7, 12–15]	✗	✗	✓	✓	✗	✓
[4, 16, 17]	✓	✗	✓	✗	not relevant	not relevant
[18–20]	✓	✓	✓	✗	not relevant	not relevant
[6, 21]	✓	✗	✗	✗	not relevant	not relevant
[3, 5, 22–25]	✓	✗	✓	✓	✗	✓
[26]	✓	✓	✓	✓	✗	✗
[27, 28]	✓	✓	✓	✓	✗	✓
This work	✓	✓	✓	✓	✓	✓

stream processing requests are ignored.

Another category of works aims at reducing latency while deploying stream processing requests represented as graphs [3, 5, 22–25]. However, these works lack a proper WAN latency model and do not solve the online optimization (on-arrival) problem for multiple requests arriving unpredictably and competing for available resources. More realistic network latency model is considered in work [26]. However, it is assumed that programmers of stream processing applications classify operators as global and local tasks indicating the placement in fog-cloud infrastructure. Therefore, it makes this solution non-transparent for stream processing applications. Furthermore, the solution proposed in [26] handles a single request to deploy an application graph neglecting competition for resources of dynamically arriving requests. Finally, our present work differs from approaches presented in [27, 28] primarily concerning the applicability of the proposed solutions. More precisely, Veith *et al.*'s and Mehran *et al.*'s solutions focus on static (offline optimization) cases where a given application runs for a long time. In [27], a deployment process can last up to five minutes, which is not applicable to online optimization. In addition, the experiments in [27] and [28] consider the deployment of a single stream processing application.

Our work proposes more general heuristics able to dynamically (on-arrival) handle stream processing requests represented as graphs that describe data streams. We conducted experiments emulating thousands of requests containing data streams competing for resources. The aim is to minimize latency in an infrastructure comprising fog and cloud using a realistic WAN network model. Furthermore, the proposed solutions are transparent for the developers of stream processing applications.

3. System architecture and problem formulation

This work considers an architecture composed of network and computing resources available in cloud and fog infrastructure. The cloud infrastructure comprises DCs associated with selected nodes in a WAN, while the fog consists of fog instances, one per network node. Cloud resources are considered unlimited given their high reliability, scaling, and abstraction capabilities. On the other hand, each fog instance is an abstract entity representing a set or a site of constrained resources. Furthermore, due to the physical proximity of their resources, latencies within fog instances are negligible compared to those imposed by the transmission via WAN to DC. Cloud and fog orchestration software, like OpenStack [29], are responsible for controlling computing resources and communicating with the SDN controller for resource provisioning. It also provides information about the available computing resources in each fog instance and the location of DCs. The SDN controller has complete knowledge of the network topology, the physical distances between nodes, and the network state, including information about installed flows.

Once a stream processing request arrives at the system, the SDN controller is responsible for provisioning network resources and requesting computing resources to the cloud and fog orchestration software. For this reason, we design the proposed algorithms as SDN applications on top of the SDN controller. Additionally, the SDN controller estimates the overall latency composed of the data transmission latency and the processing time associated with each operator, information provided by the orchestration software. These system capabilities enable introducing latency awareness to the proposed algorithms.

We consider the following metrics for assessing performance: the percentage of dataflows successfully handled, meaning that the request received sufficient network and computing resources to meet its latency requirements; and the utilization ratio of fog resources, network utilization, and statistics showing how many requests are handled by the fog, cloud, and both infrastructures simultaneously.

This section explains the application execution graph as a concept that specifies network and computing resources requirements of the particular stream processing request. Next, it proposes a complete request model to feed the algorithms with the necessary input data. Finally, the mathematical model is formulated, including all assumptions and constraints.

3.1. Application execution graph

Online processing frameworks usually structure applications as directed acyclical graphs with vertices that represent operators and edges that define how the data (*i.e.*, tuples) flows between the operators. In addition, a developer can provide parallelization hints or specify how many instances of each operator are needed. Then, the scheduler places the operator instances onto the available cluster resources. This work focuses on the placement of operators onto a distributed infrastructure composed of cloud and fog resources, considering an existing

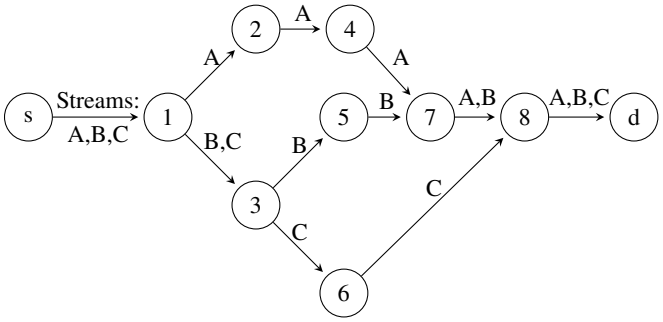


Figure 2: Example of application execution graph.

graph. The optimization and parallelization of the logical graph are out of the scope of this work.

A stream is considered a path of a graph from the source to the sink, and its computing requirements are equal to the sum of its operators' requirements. Moreover, the operators with lower indices send data to operators with higher indices for each graph and stream.

Fig. 2 presents an example of an application execution graph. Besides the source and destination (sink), seven other operators compose the graph. There are three distinct streams. Stream A consists of operators: s, 1, 2, 4, 7, 8, and d. Stream B consists of operators: s, 1, 3, 5, 7, 8, and d. Finally, stream C comprises operators: s, 1, 3, 6, 8, and d.

Please note that some operators belong to more than one stream, e.g., operator 1 is in all streams, operator 7 is in streams A and B. Analogously, the requirements on the network resources of each stream are equal to the sum of the requirements of its edges, where some edges belong to more than one stream. For clarity, a *downstream* operator processes a tuple after an *upstream* operator. For example, operator 4 is a downstream operator when compared to operator 2 within stream A.

The execution time-span of a path is the time needed to process a tuple, from source to sink, traversing all the operators along the path. The execution time-span of a graph is the largest time-span among the paths that compose the graph. In addition, the sum of execution time-span and latency resulting from data transmissions through the network is further compared against latency constraints. A request is successfully handled if all streams meet the latency constraints.

3.2. Request model

The need for an interface between the cloud and fog infrastructures was identified from the very beginning of the fog concept [1]. Furthermore, the standardization of this interface is mentioned as one of the critical issues in work [30] addressing the interplay between those entities. We propose the architecture that utilizes an SDN controller to ensure communication between fog and cloud orchestrators to perform global traffic engineering to meet the latency requirements of stream processing applications. Thus, a proper request model must be defined to ensure that all obligatory data is available for the controller.

A stream processing request (*req*) can be formally noted as $req = \{s, sink, G, C, T, l_{th}, n\}$ and, as presented in Listing 1, is

Listing 1: Request schema

```
"req": {
  "s": id,
  "sink": cloud/fog,
  "O": operatorsNo,
  "G": [[g(1,1), g(1,2), ... , g(1,O)]
        [g(2,1), g(2,2), ... , g(2,O)]
        .
        .
        [g(O,1), g(O,2), ... , g(O,O)]],
  "C": [c1, c2, ... ,cO],
  "T": [t1, t2, ... ,tO],
  "lth": latency,
  "n": tuplesNo
}
```

composed of the following elements:

1. Id of the source node (s).
2. Information on whether the request's sink is in the cloud or the fog (*sink*). If in the fog, it is always the fog instance associated with the source node.
3. The number of operators in the application execution graph (O).
4. Application execution graph represented by a matrix (G), in which indexes of columns and rows refer to graph operators. Value in i-th row and j-th column denotes the requirements on the network throughput between i-th and j-th operators. Zero value means that i-th and j-th operators are not connected.
5. Information about computing requirements of each operator represented by a vector (C) where the i-th element refers to i-th operator (indexing is consistent with the matrix representing the application execution graph). Single value for computing resources is an abstraction for heterogenous resources, e.g., CPU cycles or Random Access Memory. Such an approach is reasonable as heterogeneous computing resources scale well, and statistical multiplexing is applicable.
6. Information about tuple processing time by each of the operators, represented as a vector (T).
7. Latency threshold of the request (l_{th}). Request processing time-span must be below the threshold to successfully handle the request.
8. Total number of tuples to process in the deployed graph (n). This factor reflects that data sources are active only for some time and generate a limited amount of data.

3.3. Problem formal statement

In the formal problem definition, N and id indicate the number of network nodes and network node id, respectively. V is the set of nodes, and E is the set of links. The relation is as follows $\forall id \in 0...N$: $v_{id} \in V$, where v is a network node. V_{DC} describes the set of nodes associated with DCs. Nodes not associated with any DC are termed client nodes and denoted by V_C. The following relations are met:

$$V_{DC} \cap V_C = \emptyset \quad (1)$$

$$V_{DC} \cup V_C = V \tag{2}$$

V_F denotes the network nodes associated with fog instances. We consider that the fog infrastructure is present at every network edge; thus $V_F = V$. For the set $V_{DC} \cap V_F$ traffic is handled simply by the DC and is omitted from our study. The SDN controller has full knowledge about the network topology and its state, including resource utilization and the location of DCs (V_{DC}). The SDN controller executes each algorithm on a per-stream processing request basis. The request model is proposed in Section 3.2 and introduces some request-related symbols and constants ($req = \{s, sink, G, C, T, l_{th}, n\}$). Computing resources available in the node *id* are denoted as r_{id} and the corresponding set R is defined as $R = [r_1, r_2, ..., r_N]$. For nodes associated with DC:

$$v_{id} \in V_{DC} \colon r_{id} = \infty \tag{3}$$

Handling the stream processing request should be understood as the assignment of operators to the fog or cloud computing nodes and ensuring communication between the operators according to the application execution graph. Therefore, as each operator will consume computing resources, a sufficient amount of resources must be available. The i-th operator of a request can be deployed in the *id*-th node only if:

$$req.c_i \leq r_{id} \tag{4}$$

This constraint is especially significant in the case of fog computing instances having limited capacity. After assigning an operator to the node, available resources decrease: $r_{id} = r_{id} - req.c_i$, by analogy, resources are released when the request terminates. Total application requirements on computing resources are equal to:

$$\sum_{i=1}^{\{1,...,req.O\}} req.C[i], \tag{5}$$

while stream requirements on computing resources are limited only to those operators that belong to that particular stream:

$$\sum_{i=1}^{\{1,...,req.O\}} x_i \cdot req.C[i], \tag{6}$$

where $x_i = 1$ if i-th operator belongs to the stream being considered, otherwise $x_i = 0$.

Simultaneously, data sent between operators will consume network resources. Thus, congestion in terms of network traffic results in higher transmission times. This network delay is further added to the request execution time-span (*ets*) and may prevent it from meeting the latency threshold for a particular request. The *ets* is known in advance by the orchestration software and is the sum of event processing times for operators belonging to that particular stream:

$$ets = \sum_{i=1}^{\{1,...,req.O\}} x_i \cdot req.T[i]. \tag{7}$$

The network-related latency is applicable only if two adjacent operators are deployed in different infrastructure entities (fog and cloud), in which case the wide-area network is traversed. This network-related latency can be further divided into two components: transmission latency, constant for a particular WAN link ($trans_e, e \in E$) and buffering-related latency which depends on the load on that particular link ($buff_{ku} = f(l_{ku}), k \neq u \in V$, and l_{ku} is a load on link between nodes k and u, while f is a function translating load to the latency). Therefore, the overall time can be expressed as follows:

$$\sum_{i=1}^{\{1,...,req.O\}} x_i \cdot req.T[i] + trans_e + buff_{ku}, \forall e, k, u, \tag{8}$$

where e connects WAN nodes ($k \neq u$) hosting two adjacent operators deployed in nodes k and u. The request is considered as successfully handled only if this overall delay is lower than the latency threshold

Stream requirements on network resources are equal to:

$$\sum_{k=1}^{\{1,...,req.O\}} \sum_{u=1}^{\{1,...,req.O\}} y_{ku} \cdot req.G[k, u], \tag{9}$$

where $k \neq u \in V$ and $y_{ku} = 1$ if k-th and u-th are two adjacent operators belonging to the stream being considered, otherwise $y_{ku} = 0$. All constraints mentioned in the problem formulation are reflected in the algorithms proposed in the paper.

As mentioned previously, the primary performance metric is the percentage of successfully handled requests. *REQ* denotes the set of all requests, while REQ_h denotes requests successfully handled (enough network and computing resources were provisioned to meet latency requirements). The following objective function can reflect it:

$$\max |REQ|_h / |REQ|. \tag{10}$$

However, we also study the utilization of network and fog resources and whether requests are handled in the fog, cloud, or both types of infrastructure.

3.4. Architecture reliability

Although the research presented in the paper focuses on application-layer mechanisms built on top of the SDN stack, the resiliency of the key architecture components cannot be neglected in the problem formulation. As these building blocks of the environment are susceptible to a number of events such as hardware and software failures, power and network outages or resource starvation, some general ideas about possible failure scenarios and suggested countermeasures are to be pointed out. However, the detailed design of mechanisms that alleviate these issues is out of scope of this paper.

Fog instance A fog instance discussed in this paper is defined as a group of resources located close to one another and capable of communicating with other members of the group (Figure 1). There is no single point of failure in such an entity that could fully disrupt operations within a particular fog instance. Moreover, as we assume that stream processing requests and data to be processed originate in the fog instance, an

eventual failure of the fog data source implies the absence of requests and data to process. In case of failures that affect single fog resources, the allocation of the operators may be recalculated by the SDN controller. Similarly, connectivity issues that isolate a fog instance from the WAN could be addressed with fallback to a workload deployment algorithm that utilizes only local fog resources.

DC instance While properly designed data centers are generally considered reliable, the availability of certain DCs should be continuously monitored by the SDN controller to ensure that all data streams are processed as requested. In the event of DC outage, the SDN controller may perform rerouting of the traffic to other resources, maintaining continuity of the streams.

SDN controller The SDN controller is an essential part of the architecture. Its reliability should be ensured by maintaining redundant controller instances to reduce the risk of control plane outages. It is assumed that the controller has a full overview on the fog, networking and cloud components, as it continuously monitors resources and flows present in the environment. The data related to a single entity may be collected from various sources, e.g. neighboring nodes, network devices and the monitored entity itself. Therefore, the controller can detect any failures that may affect data streams processed in the network. On such occasions, it is possible to provide a proper response such as employing an alternative workload deployment algorithm, recalculating the execution graphs, rerouting the traffic, or reallocating the operators.

4. SDN-based Fog and Cloud Interplay for Stream Processing

This section details the dynamic algorithms operating on stream processing requests. The input requests for the algorithms are modeled as described in Section 3.2, Listing 1, while the procedure of generating the requests is described in Section 5. As a result of processing the request, an algorithm returns a decision on allocation of the operators in the fog or in the DC. As stated in Section 5.1, the SDN controller is responsible for the integration of fog, cloud, and network infrastructures. The algorithms can be easily implemented in any network controller as they use commonly available features of network controllers.

The first four algorithms are based on concepts presented in related works and considered state-of-the-art approaches. While scopes of these works vary and direct comparison is not possible, some general principles may be extracted and used to define algorithms that are possible to evaluate following the model described in Section 3. The resulting algorithms derived from related research are as follows:

- *Random* - random deployment of request operators both in the cloud and in the fog as in [31];
- *AllDC* - deployment of requests fully in the cloud as in [4, 25, 26, 28];
- *FogOnly* - deployment of requests fully in the fog as in [16];

Algorithm 1 Random

Input: *req*
1: **for** $i \in \{1, ..., req.O\}$ **do**
2: $target \leftarrow$ random from {*fog*,*cloud*}
3: **if** $target = cloud$ OR $req.C[i] > r_{req.s}$ **then**
4: Deploy operator i in the selected DC
5: **else**
6: Deploy operator i in the fog associated with $req.s$
7: **end if**
8: **end for**

- *FogGreedy* - deployment of request streams with an effort to saturate fog resources as in [28, 31].

Such defined algorithms are implemented in the testbed and used as reference approaches in the experiments.

On the other hand, *FogGreedy+*, *Network-Aware FogGreedy* and *Network-Aware Application Oriented* are novel algorithms, considered the main contribution of this paper. The algorithms are transparent for the application developer. They aim to handle requests by meeting the latency constraints defined separately for each request and preserving limited fog resources for requests with a strict latency threshold representing time-critical applications. Before presenting the algorithms and describing their properties, this section provides a few explanations.

Network awareness (referred to as *NetAwareness*) denotes that a particular algorithm tries to reduce network utilization. The proposed algorithms take advantage of SDN, supporting the interplay between cloud and fog orchestrators, and thus, implementing the *softwarization* concept.

As discussed in Section 3, the cloud infrastructure is a set of DCs associated with selected network nodes. Therefore, placing an operator in the cloud means selecting one DC for hosting it. In our previous work, we proposed anycast algorithms that can be directly applied here for DC selection in wide area networks [32] and [33]. The basic algorithm, referred to as *closest*, will be utilized. It selects the DC closest to the source of the request considering network hops. The main advantage is that the algorithm consumes as few network resources as possible and reduces transmission latency.

4.1. Random

Random is the reference algorithm that, for every single operator, randomly decides whether to deploy the operator in the cloud or in the fog. The only advantage of the *Random* algorithm is its simplicity. Excessive communication between operators deployed in the fog and in the cloud may impose increased WAN resource consumption. In the worst case, the communication between each pair of operators may overutilize WAN resources. Pseudocode 1 describes the *Random* algorithm.

4.2. AllDC

The *AllDC* reference algorithm is based on a single rule - all requests are fully deployed in the cloud, fog resources are not utilized at all. The algorithm generates a considerable amount of network traffic, as each of the streams has to reach the data

Algorithm 2 *FogOnly*

Input: *req*
1: *requirement* ← 0
2: **for** $i \in \{1, ..., req.O\}$ **do**
3: *requirement* ← *requirement* + *req.C*[*i*]
4: **end for**
5: **if** *requirement* > $r_{req.s}$ **then**
6: Reject the request
7: **else**
8: Deploy all operators in the fog associated with *req.s*
9: **end if**

center. The requests are also more likely to exceed their latency thresholds due to transmission delays in WAN.

4.3. FogOnly

The *FogOnly* reference algorithm allocates all requests fully in the fog and rejects the requests in case of insufficient fog resources (Pseudocode 2). Such an approach provides satisfactory performance in environments with abundant fog resources. However, the success rate is expected to suddenly drop as the fog instances reach their resource limit.

4.4. FogGreedy

For requests with sinks in the cloud, the *FogGreedy* reference algorithm will place all operators in the selected DC. However, if only the request's sink is in the fog, the algorithm tries to maximize the use of fog resources. More precisely, the algorithm sorts the streams according to their increasing requirements on computing resources and then handles the request in a per-stream manner. The algorithm will deploy the operators involved in a given stream in the fog if there are enough resources. Otherwise, it deploys the whole stream in the selected DC. Fig. 3 illustrates this *FogGreedy* property with a sink in the fog. The algorithm processes streams sorted by their processing requirements, and as a result, deploys streams *B* and *C* in the fog (Figs. 3b and 3c, respectively), and stream *A* in the cloud (Fig. 3d).

The *FogGreedy* algorithm focuses on utilizing fog resources. Its main drawback is that requests with loose latency requirements that could use the cloud may end up consuming fog resources. Such inefficiency may drain fog resources, and as a result, violate the constraints of more demanding requests. Compared to the *Random* algorithm, the number of unnecessary transmissions through the WAN is limited. However, additional WAN communication may occur when a fog instance has enough resources to deploy only a few streams of a given request. The communication happens because the operators deployed in the fog may also belong to streams partially placed in the cloud.

Pseudocode 3 describes the *FogGreedy* algorithm. Three auxiliary functions are also involved. *getStreams* (Pseudocode 4) processes the application execution graph and returns a matrix containing vectors describing streams in consecutive columns. Values in the vector's *i*-th cell are requirements on computing resources of the *i*-th operator, if only it is a part of that stream. The second algorithm, *getStreamRequirements* (Pseudocode 5),

Algorithm 3 *FogGreedy*

Input: *req*
1: **if** *req.sink* = cloud **then**
2: Deploy all operators in the selected DC
3: **else**
4: *row* ← 1
5: *stream* ← []
6: *streams* ← *getStreams*(*req*, *row*, *stream*, [])
7: *streams* ← *sortStreams*(*req*, *streams*)
8: **for all** *stream* ∈ *streams* **do**
9: **if** *getStreamRequirements*(*req*, *stream*) > $r_{req.s}$ **then**
10: Deploy *stream* in the selected DC skipping already deployed operators
11: **else**
12: Deploy *stream* in the fog associated with *req.s* skipping already deployed operators
13: **end if**
14: **end for**
15: **end if**

Algorithm 4 *getStreams*

Input: *req*, *row*, *stream*, *streams*
1: **for** $i \in \{1, ..., req.O\}$ **do**
2: **if** *req.G*[*row*, *i*] ≠ 0 **then**
3: *tmpStream* ← *stream*
4: *tmpStream*[*row*] ← *req.C*[*row*]
5: *tmpStream*[*i*] ← *req.C*[*i*]
6: **if** *i* = *req.O* **then**
7: *streams* ← [*streams*, *tmpStream*]
8: **else**
9: *streams* ← *getStreams*(*req*, *i*, *tmpStream*, *streams*)
10: **end if**
11: **end if**
12: **end for**
13: **return** *streams*

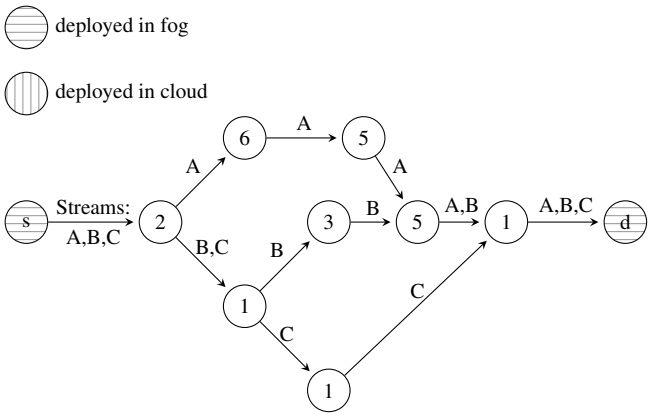
Algorithm 5 *getStreamRequirements*

Input: *req*, *stream*, *lastIndex*
1: *requirement* ← 0
2: **if** *lastIndex* = *null* **then**
3: *lastIndex* ← *req.O*
4: **end if**
5: **for** $i \in \{1, ..., lastIndex\}$ **do**
6: *requirement* ← *requirement* + *stream*[*i*]
7: **end for**
8: **return** *requirement*

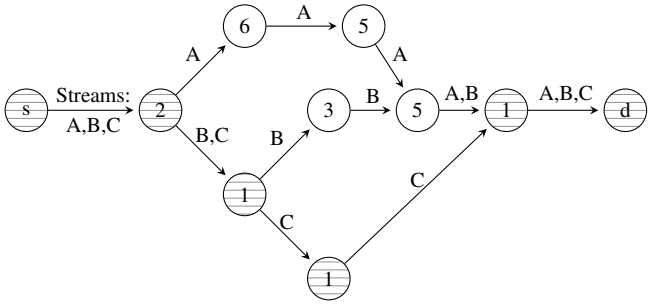
Algorithm 6 *sortStreams*

Input: *req*, *streams*
1: *sorted*
2: *streamsRequirements* ← []
3: **for all** *stream* ∈ *streams* **do**
4: *streamsRequirements* ← [*streamsRequirements*, *getStreamRequirements*(*req*, *stream*)]
5: **end for**
6: *sorted* ← sort *streams* by *streamsRequirements* (increasing order)
7: **return** *sorted*

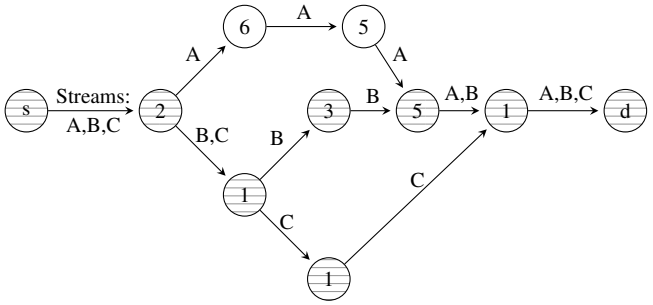
sums the requirements on computing resources for a stream represented as the mentioned vector. Finally, the *sortStreams* algorithm (Pseudocode 6) sorts the matrix returned by the *getStreams* algorithm by reordering columns describing streams according to the increasing requirements on computing resources.



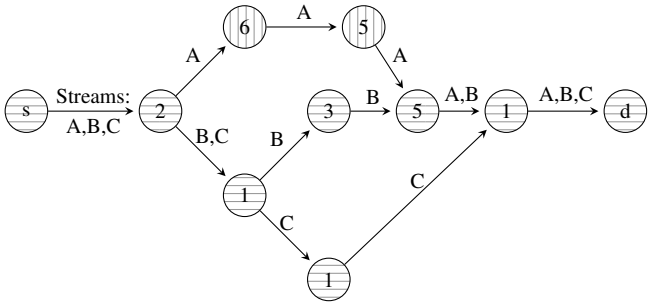
(a) Sink in the fog. Remaining resources in fog: 21 units.



(b) Processing third stream which has lowest computing requirements and can be fully deployed in the fog. Remaining resources in fog: 16 units.



(c) Processing second stream which has lowest computing requirements and can be fully deployed in the fog. Remaining resources in fog: 8 units.



(d) Processing first stream that cannot be fully deployed in the fog as remaining resources in the fog (8 units) are insufficient to handle requirements (11 units).

Figure 3: An example of *FogGreedy* properties.

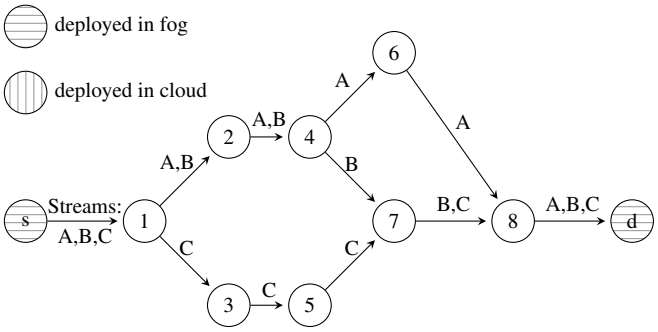
4.5. FogGreedy+

The proposed *FogGreedy+* algorithm extends *FogGreedy* by using fog resources more effectively. *FogGreedy+* places the whole application execution graph in the fog only if the sink is in the fog and there are enough fog resources to host all operators. A request with a sink in the cloud, on the other hand, is placed in the cloud.

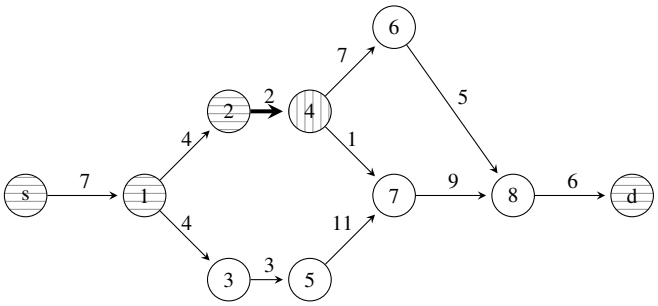
FogGreedy+ proceeds as follows for requests with sinks in the fog, whose operators cannot be fully deployed in the fog. First, it sorts the streams and then processes them one at a time. For each stream, *FogGreedy+* finds the graph edge with the lowest network requirements and for which it can place all upstream operators in the fog. This edge is then selected to traverse the WAN in the direction from the fog to the cloud. If the selected edge belongs to streams not yet processed by the algorithm, it is automatically selected to cross the WAN for those streams. Fig. 4 illustrates these *FogGreedy+* properties, where Fig. 4a shows the stream paths, whereas Figs. 4b and 4c present the network requirements. *FogGreedy+* first selects the edge that traverses the WAN for streams *A* and *B* simultaneously (Fig. 4b), and then in the following step picks the edge for stream *C* (Figs. 4c). It performs this operation until all streams have an edge selected for data transmission from the fog to the cloud or no fog resource is available. In the latter case, the algorithm places all remaining operators in the cloud.

FogGreedy+ must also find edges crossing the WAN in the opposite direction (from the cloud to the fog). For this purpose, the algorithm evaluates the application graph edges that meet two conditions: (1) they connect operators downstream to the first one deployed in the cloud, and (2) the algorithm can deploy these downstream operators in the fog. Then, the algorithm selects the edge with the lowest network requirements to cross the WAN from the cloud to the fog. Again, as the selected edge may be common to multiple streams, those streams are automatically selected with this edge crossing the WAN. This case is illustrated in Fig. 5 for the same streams presented in Fig. 4a. Initially, the algorithm selects the edge that traverses the WAN for stream *A* (Fig. 5a), and then in the following step, the algorithm picks the edge for streams *B* and *C* simultaneously (Figs. 5b). It repeats this operation for the remaining streams until they either carry data to the sink in the fog or all fog resources are used. In the latter case, it deploys all remaining operators in the cloud, and the streams carry data back to the fog directly before the sink. Pseudocode 7 describes the auxiliary function *getSortedEdges* which, for a particular stream, returns directly connected operators in the form of the list ordered according to the increasing requirements on network resources. The auxiliary function *getSortedEdges* is used by the *FogGreedy+* algorithm presented in Pseudocode 8. We omit the description of finding edges feasible to cross the WAN from cloud to fog for brevity.

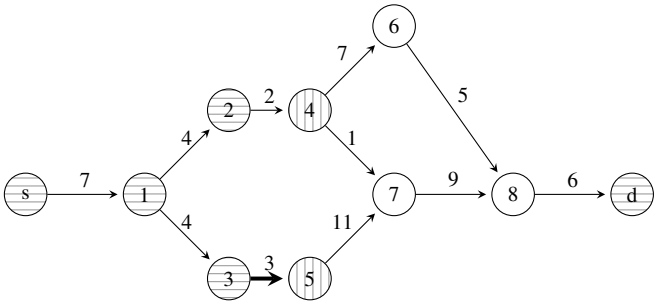
Intuitively, the proposed *FogGreedy+* utilizes fog resources more effectively than *FogGreedy* as it can deploy particular streams simultaneously in fog and cloud. However, such use of fog resources comes at the expense of additional data transmission between fog and cloud. That is why *FogGreedy+* selects the graph edges with the lowest network requirements. However, similarly to *FogGreedy*, *FogGreedy+* is susceptible to draining fog resources by using them to serve requests that the cloud could handle.



(a) Sink in the fog and fog resources are insufficient to handle whole request in the fog.



(b) Edges show requirements on network resources. When processing the first stream, the edge with the lowest network requirements is selected (bold), and the selected operators are deployed in the fog and in the cloud. This selection affects the second stream as it shares the selected edge with the other stream.



(c) Processing third stream: edge with the lowest requirements on network resources selected (bold).

Figure 4: An example of *FogGreedy+* properties.

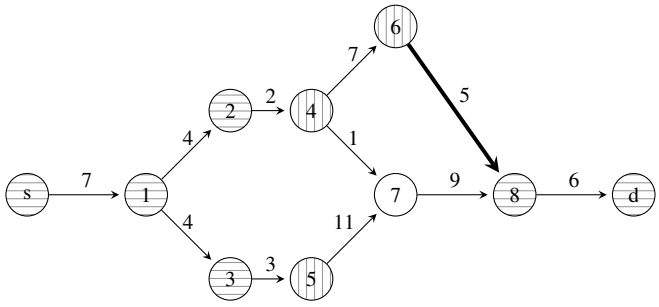
Algorithm 7 *getSortedEdges*

Input: *req*, *tmpG*, *stream*
1: *sortedEdges* ← []
2: **for** $i \in \{1, ..., req.O\}$ **do**
3: **for** $j \in \{1, ..., req.O\}$ **do**
4: **if** $stream[i] > 0$ AND $stream[j] > 0$ AND $tmpG[i,j] \neq 0$ **then**
5: add (i,j) pair to *sortedEdges* based on $tmpG[i,j]$ value in increasing order
6: **end if**
7: **end for**
8: **end for**
9: **return** *sortedEdges*

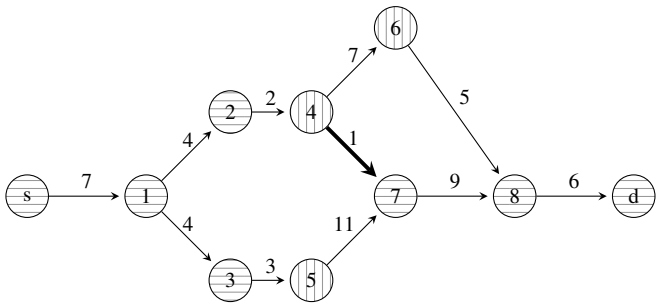
4.6. Network Aware FogGreedy - NAFogGreedy

The proposed *NAFogGreedy* algorithm introduces *NetAwareness* by trying to mitigate the overhead caused by excessive data transfer between fog and cloud. Instead of evaluating each stream individually, it considers a request as a set of operators. Thus, the algorithm deploys the entire application execution graph in the fog only if the sink is also in the fog and there are enough fog resources to host all the operators. Otherwise, *NAFogGreedy* places the whole request in the cloud. The algorithm also deploys in the cloud all requests whose sinks are in the cloud.

The algorithm reduces the communication between operators deployed in the cloud and the fog at the expense of less efficient fog utilization. However, it remains susceptible to the same issue as *FogGreedy* and *FogGreedy+* algorithms: requests with loose latency requirements may use fog resources even if deploying them in the cloud successfully meets their requirements. Such inefficient resource usage may over-utilize fog resources, and as a result, exceed the latency threshold of more stringent requests. Pseudocode 9 describes the *NAFogGreedy* algorithm.



(a) Processing first stream: edge with the lowest requirements on network resources selected (bold).



(b) Processing second stream: edge with the lowest requirements on network resources selected (bold). After deployment third stream is also fully deployed.

Figure 5: An example of *FogGreedy+* properties regarding the transmission from cloud to fog.

4.7. Network-Aware Application Oriented - NAAO

The proposed *Network-Aware Application Oriented* algorithm deploys in the cloud all requests with sinks in the cloud and requests with a sink in the fog that exceed available fog resources. In addition, it considers the latency constraint for requests that cannot be fully handled in the fog. First, the algorithm calculates the execution time-span (*ets*) of each stream in the request. If the execution time-span of any stream exceeds the latency constraint, then the algorithm places the request in the cloud to avoid wasting fog resources. After that, it computes the cloud execution time-span (*cets*) for each stream by

Algorithm 8 *FogGreedy+*

Input: *req*
if *req.sink* = cloud **then**
  Deploy all operators in the selected DC
else
  $requirement \leftarrow 0$
  for $i \in \{1, ..., req.O\}$ **do**
    $requirement \leftarrow requirement + req.C[i]$
  end for
  if $requirement \leq r_{req.s}$ **then**
    Deploy all operators in the fog associated with *req.s*
  else ▷ Sink in the fog but request cannot be fully deployed in the fog
    $streams \leftarrow getStreams(req, 1, [], [])$
    $streams \leftarrow sortStreams(req, streams)$
    for all $stream \in streams$ **do**
      if any operator in *stream* has been already deployed in fog **then**
        $j \leftarrow$ index of first operator of *stream* deployed in fog
        for $i \in \{1, ..., j\}$ **do**
          if $req.C[i] \leq r_{req.s}$ **then**
            Deploy operator i in the fog associated with *req.s*
          end if
        end for
      else
        $sortedEdges \leftarrow getSortedEdges(req, req.G, stream)$
        $edgeFound \leftarrow false$
        for all $edge \in sortedEdges$ **do** ▷ Consecutive edges with the lowest requirements
          $requirement \leftarrow getStreamRequirements(req, stream, edge.i)$
          if $requirement \leq r_{req.s}$ **then** ▷ Operators before edge can be deployed in fog
            $edgeFound \leftarrow true$
            for $k \in \{1, ..., edge.i\}$ **do**
              if $stream[k] > 0$ **then**
                Deploy operator k in the fog associated with *req.s*
                $stream[k] \leftarrow 0$ ▷ Not to reassign operators
              end if
            end for
          end if
          if *edgeFound* **then**
            break ▷ If edge found move to another stream
          end if
        end for
      end if
    end for
    for all $stream \in streams$ **do**
      By analogy find the least demanding edge to span WAN from cloud to fog. All operators after that edge must be able to be deployed in fog.
    end for
    Deploy all not deployed operators in the selected DC
  end if
end if

Algorithm 9 *NAFogGreedy*

Input: *req*
if *req.sink* = cloud **then**
  Deploy all operators in the selected DC
else
  $requirement \leftarrow 0$
  for $i \in \{1, ..., req.O\}$ **do**
    $requirement \leftarrow requirement + req.C[i]$
  end for
  if $requirement > r_{req.s}$ **then**
    Deploy all operators in the selected DC
  else
    Deploy all operators in the fog associated with *req.s*
  end if
end if

adding the estimated delay of data transfer to the selected data center considering its physical distance. *cets* is an estimate of the actual time needed to handle the request. Thus, *NAAO* deploys in the fog, streams whose latency constraint is lower than the cloud execution time-span. On the other hand, it deploys in the cloud streams whose latency constraint is higher than *cets*

Algorithm 10 *Network Aware Application Oriented*

Input: *req*
if *req.sink* = cloud **then**
  Deploy all operators in the selected DC
else
  $requirement \leftarrow 0$
  for $i \in \{1, ..., req.O\}$ **do**
    $requirement \leftarrow requirement + req.C[i]$
  end for
  if $requirement > r_{req.s}$ **then**
    Deploy all operators in the selected DC
  else
    $streams \leftarrow getStreams(req, 1, [], [])$
    for all $stream \in streams$ **do**
      $ets \leftarrow$ retrieve for *stream*
      if $ets > req.l_{th}$ **then**
        Deploy all operators in the selected DC canceling any previous deployments
        break
      end if
      Temporarily deploy *stream* in the selected DC skipping already deployed operators
      $cets \leftarrow$ retrieve for temporarily deployed *stream*
      if $cets > req.l_{th}$ **then**
        Deploy *stream* in the fog associated with *req.s* skipping already deployed operators
      else
        Deploy *stream* in the selected DC skipping already deployed operators
      end if
    end for
  end if
end if

since the cloud is sufficient to meet their requirements.

The *NAAO* algorithm requires additional information about the requests latency threshold and processing time of each operator in the application execution graph. The algorithm then provides the SDN controller with this information along with the requests (see 3.2). Hence, the *NAAO* algorithm mitigates the inefficient use of fog resources, which is the main drawback of the *FogGreedy* approaches. Operators consume fog resources only if they can meet the latency constraints of each application stream. The *NAAO* algorithm also introduces *NetAwareness* by limiting unnecessary data transmission between cloud and fog based on estimation of *cets* metric.

To sum up, the *Random* algorithm handles requests equally regardless of whether the sink is in the cloud or in the fog. *AllDC* and *FogOnly* deploy requests fully in the cloud or in the fog, respectively. The general idea of *FogGreedy* is to deploy requests in the cloud and in the fog considering the desired sink. The algorithm processes requests examining one stream at a time and considering the limited fog resources when deploying operators in the fog. *FogGreedy+* operates similarly to *FogGreedy* when handling streams with a sink in the cloud or in the fog, and that could be deployed in the fog. It, however, applies more sophisticated rules to requests with a sink in the fog that cannot be entirely deployed in the fog. The *NAFogGreedy* algorithm introduces *NetAwareness* by considering requests as an integral set of operators instead of processing each stream individually. The algorithm deploys in the cloud requests with a sink in the cloud and requests that it cannot fully deploy in

the fog. The *NAAO* algorithm combines *NetAwareness* with application properties. *NAAO* compares stream cloud execution time-spans with request latency constraints to decide whether to deploy a stream in the cloud or in the fog.

Table 2 summarizes the differences between the algorithms regarding their approach to request processing, behavior in case of insufficient fog resources, and *NetAwareness*. More sophisticated algorithms integrate cloud, fog, and network infrastructures using an SDN controller, which handles the application execution graph, collects information about computing resource utilization, and estimates execution times, steering the traffic according to the implemented algorithms.

5. Performance Evaluation

This section describes the experimental setup and analysis of obtained results. The following paragraphs describe the setup used to carry out the experiments, emulation parameters, final results, and conclusions drawn from the results.

5.1. Experimental Setup

The algorithms were evaluated in an emulated SDN environment. The testbed comprises several SDN switches (network nodes), an SDN controller, virtual hosts representing fog instances and DCs, a request generator, request handlers, traffic generators, and a database. Fig. 6 depicts the essential system components and interfaces, whereas Fig. 7 shows the network topology. Both figures share some building blocks:

- network nodes (Fig. 6a, 7a);
- virtual hosts with fog instances (Fig. 6b, 7b) and DC instances (Fig. 6c, 7c);
- data plane connections (Fig. 6d, 7d) and control plane connections (Fig. 6e, 7e);
- the SDN controller (Fig. 6f, 7f).

Topology The network topology was deployed using the Mininet emulator and consisted of 14 network nodes identified by numbers from 0 to 13. All networks nodes were running a virtual switch (Open vSwitch) managed by the SDN controller Ryu. Additional hosts for fog instances were created and directly connected to all nodes, while hosts for DC instances were created and directly connected only to nodes 2, 4, 7, 9, 11.

SDN controller An external application implemented within the SDN controller provides all the features required in the experiments. The communication between the switches and the controller follows the OpenFlow protocol version 1.3 (Fig. 6e). The controller application is responsible for topology discovery, route calculation, path setup, and gathering network statistics. Upon successful network provisioning, the controller establishes separate OpenFlow sessions with each switch. The topology discovery function generates outbound LLDP packets on each switches' interface using *OFPT_PACKET_OUT* messages and tracks their reception reported by *OFPT_PACKET_IN* messages. As a result, the controller can create an entire network graph. Based on the collected information, the SPF algorithm computes the shortest paths between each pair of hosts connected to the network nodes. The controller then sets up paths by sending a series of *OFPT_FLOW_MOD* messages that create appropriate flow table entries on each network node, enabling layer-3 connectivity between all provisioned virtual hosts. In the meantime, the controller triggers periodical polling of information from network nodes regarding their network interface statistics using *OFPMP_PORT_STATS* messages (Fig. 6g) and saves them to the database (Fig. 6h).

Request generator The request generator component (Fig. 6i) generates both background traffic and fog requests. During the warmup phase, the request generator creates complete sets of both request types for the whole emulation run. It then places the requests on a timeline object (Fig. 6j) and sequentially dispatches them to the request handler via a REST API (Fig. 6c) following their timestamps.

Background requests, simple data transmissions between two random hosts, are described by 3 parameters: inter-arrival time, bandwidth, and duration. Each parameter is drawn randomly from an exponential distribution. These requests emulated network traffic unrelated to stream processing and provided an additional means of adjusting resource consumption.

Fog requests, the primary concern of the experiment, are generated to meet the requirements stated in the paper. The initial parameters of each fog request are: inter-arrival time generated randomly using an exponential distribution and bandwidth constraints for traffic processed by the first operator. As a first step, a uniform distribution provides a random number of operators between 5 and 10. Then, a random number of stages was selected from a uniform distribution, considering that the total number of stages could not exceed the total number of operators. Note that the request stage is an abstract term denoting operators that belong to different streams and can process data in parallel (each operator does not have any downstream or upstream operators in the same stage). Therefore, stages were introduced solely for the request generation procedure and did not affect request execution.

Next, the operators are assigned to stages, assuming the following:

- at least one operator in each stage;
- the first and the last stages contain only one operator;
- the operators in a stage have lower indices than operators in all following stages;
- each operator has at least one connection with the previous stage and with the following stage; and
- the operators communicate only between adjacent stages.

Additional connections are randomly added between stages, as operators can fork execution and aggregate tuples from two or more upstream operators. The next step randomly selects the operator computing requirements and its tuple processing time,

Table 2: Summary of the algorithms.

Algorithm	Processing manner	Insufficient fog resources	NetAwareness
Random	operator by operator	all other operators to cloud	no
AllDC	request as unity	does not use fog resources	no
FogOnly	request as unity	reject the request	no
FogGreedy	stream by stream	whole stream to cloud	no
FogGreedy+	stream by stream	analyze application graph	minor
NAFogGreedy	request as unity	whole request to cloud	yes
NAAO	stream by stream	whole request to cloud	yes

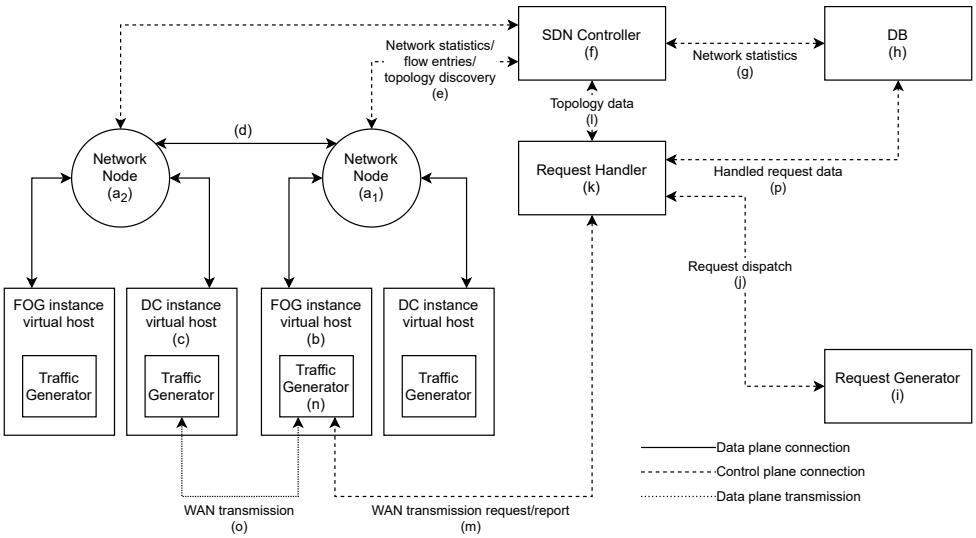


Figure 6: Testbed architecture.

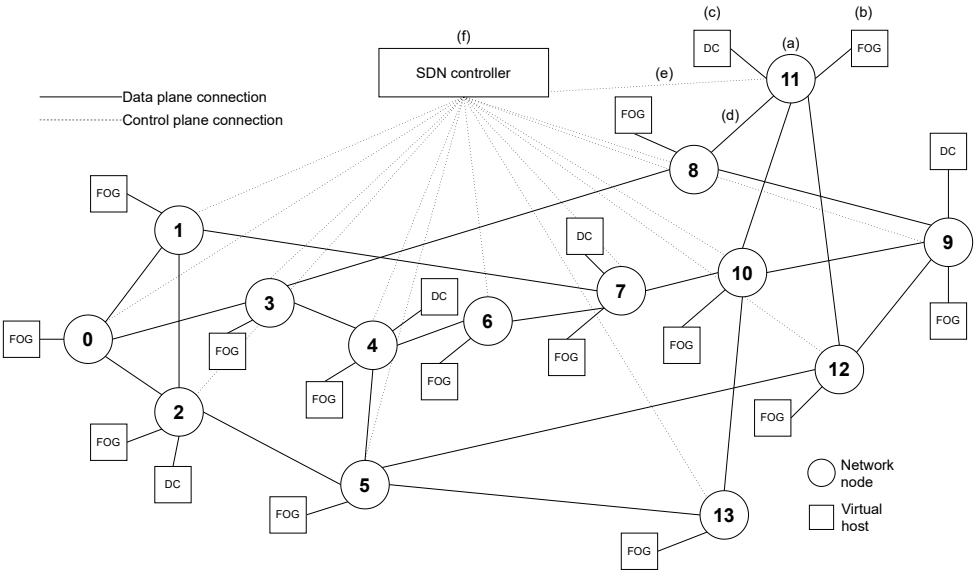


Figure 7: Experimental topology.

the bandwidth requirements of each edge between operators, and the latency threshold.

Regarding tuple processing time, we consider two types of operators: short (average tuple processing time equals the average latency of a single link in a network) and long (average tuple processing time equals 10 times the average latency of a single link in a network). The request generation selects operator types randomly with equal probability, drawing the actual

tuple processing time from an exponential distribution.

Specific operators may perform data aggregation and compression, and network requirements could randomly vary between stages. Once we have all operators, connections, and the required bandwidth values, we can create the G matrix of the request. Note that the matrix does not contain the sink.

We consider the longest stream, time-wise, to compute the latency threshold for a request. Not meeting the threshold is acceptable but considered an anomaly. Therefore, on average, 10% of the requests have the latency threshold lower than the sum of operators tuple processing times of the longest stream. Another 40% of the requests have the threshold high enough to be met even if numerous transmissions between fog and cloud occur. The threshold for all other requests considers a successful execution optimizing the number of transmissions through the WAN. We decided that the mean value of the latency threshold should be equal to the sum of the tuple processing times of the longest stream, plus $2 * 3 * MTT$, where MTT is the mean transmission time through the link in the network. The rationale is to assume 2 transmissions through the WAN, on average three hops between fog and cloud.

Request handler The request handler (Fig. 6k) provides a simple REST API that receives requests dispatched by the request generator. First, each incoming request is deserialized and validated. Then an appropriate handling procedure is applied, either for background or fog requests.

In the case of a fog request, the process involves selecting an appropriate handling algorithm concerning the initially applied testbed configuration and running the algorithm to determine the deployment of particular operators and streams within the request. Then the request handler reserves the required fog resources and creates a list of WAN transmissions that traffic generators running on virtual host instances can perform. A similar list is created when handling a background request.

Using the topology data retrieved from the SDN controller (Fig. 6l), the request handler determines the exact information about the endpoints of the planned transmission, including the relevant network node (Fig. $6a_1$,a_2) numbers and virtual host IPs (Fig. 6b,c). The handler than hands the WAN transmission requests (Fig. 6m) to the traffic generator (Fig. 6n) running on the source host. The handler tracks the execution of all transmissions within the request and saves the collected statistics on the database (Fig. 6o).

Traffic generator Each virtual host in the topology runs a separate traffic generator (Fig. 6n) based on the *nuttcp* tool. The generators maintain TCP data streams (Fig. 6o) following the parameters dictated by the request handler: bandwidth, duration, source, and destination host. Upon termination of each transmission, the generator reports its detailed statistics back to the request handler (Fig. 6m).

5.2. Results

The experiments were carried out in two stages. Initially, multiple emulation batches were run with different values of available fog resources (100-30000) and network link capacities (50 Mbps-1 Gbps). Each of the batches consisted of separate 3-hour runs of all 7 algorithms. The statistics collected from these initial batches included sampled values of resource utilization collected in 15 s intervals. These data sets were used to gain a preliminary insight into resource consumption and network performance caused by the input parameters.

Based on the data, it was decided that 3 scenarios should be considered: *Low* (network utilization within 10-30%), *Medium* (network utilization within 30-50%) and *High* (network utilization over 50%). The scenarios varied by the computational resources available in each fog instance and WAN link capacity to examine the performance and effectiveness of the algorithms under different loads. The parameters applicable for each scenario are summarized in Table 3. Note that the request generation parameters explained in Section 5.1 remained constant between all scenarios.

Final results were collected analogously by running several batches for each scenario. Different random number generator seed values were used for batches within a single scenario, while the same seeds were applied to all scenarios. Summary network and fog resources utilization levels were assessed by calculating the 80-percentile value of all samples collected during a single emulation run. This value expressed as a fraction of the maximum available fog or network resources constituted a metric that determined the prevalent environment state throughout the single emulation run. The accuracy of the results was ensured by calculating 95% confidence intervals of values observed in the emulation batches. The confidence intervals are represented by error bars included in the figures.

The results achieved in *Low*, *Medium* and *High* scenarios are presented in Fig. 8, 9 and 10, respectively. The first graph in each figure illustrates the success rates, considered the key indicator of each algorithm's performance. The two following graphs present the measured network and fog resource utilization (evaluated as described in the paragraph above) used to assess the impact of the algorithms on network resource usage and the effectiveness of the resource allocation. The last graph presents a distribution of requests with regards to the applied deployment type:

- *All-Fog* (all operators within a request were deployed in the fog);
- *All-Cloud* (all operators within a request were deployed in the cloud);
- *Mixed* (request operators were deployed both in the fog and in the cloud);
- *Rejected* (the request was rejected and no operators were deployed at all - exclusive to the FogOnly algorithm).

The analysis of these deployment type ratios is used to verify the design assumptions and differences between the evaluated algorithms.

The success rates measured in all scenarios illustrate the difference between the reference and the proposed algorithms. In the *Low* scenario, Random successfully handled only ca. 60% of the requests, while FogOnly and AllDC achieved only slightly better rates (between 70-80%). At the same time, the other algorithms succeeded with more than 80% of the requests. Note

Table 3: Scenarios and traffic generator parameters.

Parameter	Scenario		
	Low	**Medium**	**High**
Fog resources	1000	500	500
Link capacity	100 Mbps	100 Mbps	50 Mbps
Expected network utilization	10-30%	30-50%	50-100%
Batch count	15		
Background requests: mean interarrival time	60 s		
Background requests: mean bandwidth	50 Mbps		
Background requests: mean duration	300 s		
Fog requests: mean interarrival time	60 s		
Fog requests: length	100000 tuples		
Fog requests: initial stage bandwidth	$[25\ Mbps, 50\ Mbps]$		

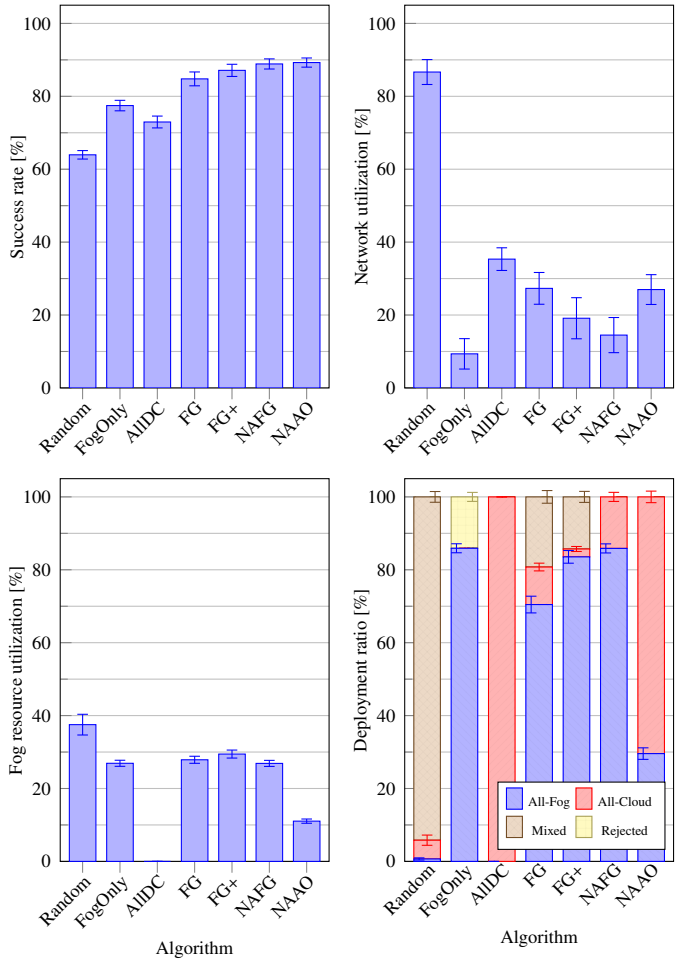


Figure 8: "Low" scenario results.

that FogOnly performs better than Random and AllDC only in the *Low* scenario, when most requests may be redirected to the fog instances. In the *Medium* and *High* scenarios, fog instances become saturated, and we observe a significant drop of success rate (below 50%). Similarly, the AllDC algorithm is affected by the availability of network resources and notes a reduction of success rate in the *High* scenario. While FG, FG+ and NAFG each exhibit a progressive improvement over their predecessors, the best success rates were always observed in the case of the NAAO algorithm and, only slightly worse, NAFG. These two algorithms were the most effective regardless of the resource consumption and network load throughout all the scenarios.

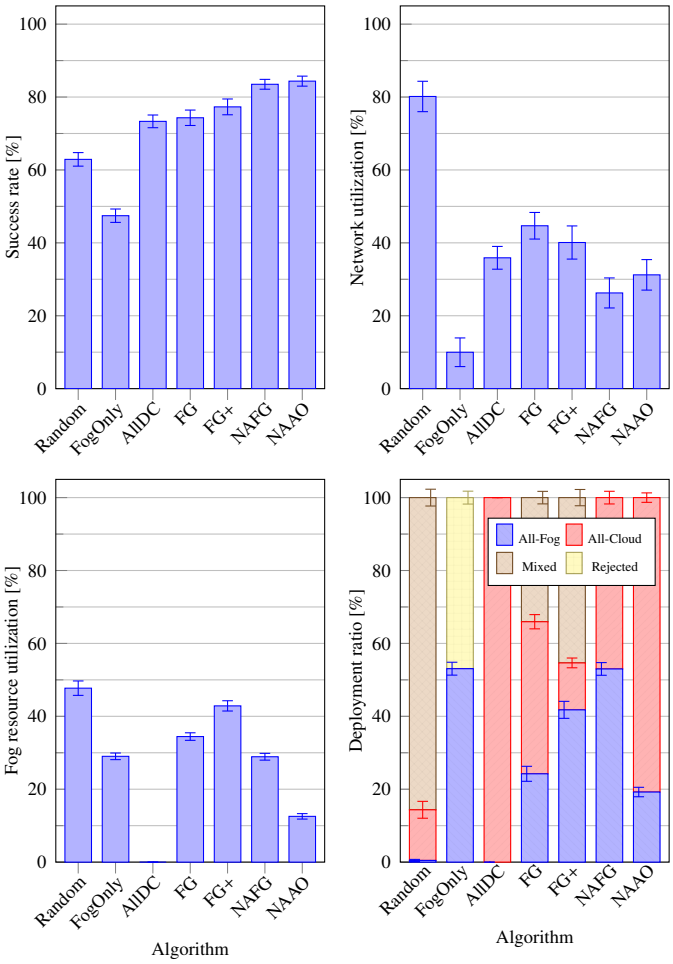


Figure 9: "Medium" scenario results.

The differences between specific algorithms are explained more comprehensively by the other metrics evaluated in the experiments. Based on the network utilization measurements, the Random algorithm contributes to substantial traffic related to WAN transmissions (75-95% of link capacity). Considerably high network utilization was also caused by the AllDC algorithm, as it required a WAN transmission for each of the requests. The FG and FG+ algorithms achieved ca. 20-30% utilization levels in the *Low* scenario, but led to relatively high network consumption in the *Medium* and *High* scenarios, exceeding 80% in the latter case. On the other hand, the lowest network utilization of ca. 10-20% was observed for the FogOnly algorithm that did not generate any traffic (only background traffic streams were present in the network). The second-lowest values were always observed with the NAFG algorithm that succeeded in keeping the utilization below 60% in the *High* scenario and reduced it even more (below 30%) in the other scenarios. Similar and only slightly higher utilization levels were observed with the NAAO algorithm in the *Medium* and *High* scenario.

The comparison of fog resources utilization speaks in favor of AllDC and NAAO regardless of the scenario. As expected, the AllDC algorithm used no fog resources due to handling all the requests in the cloud. The NAAO algorithm steadily maintained resource utilization at ca. 10%, which is 4-6 times lower than in case of the Random algorithm that was the least effective in this scope. The FogOnly and NAFG algorithms provided the third best results (ca. 30%), with a significant improvement over the FG and FG+ algorithms in the *Medium* and *High* scenarios. It is also observed that fine-grained resource allocation based on operators (as in Random) or streams (as in FG) allows to utilize more fog resources than in case of deploying the request as a whole (as in FogOnly).

Regarding the ratio of observed deployment types, the algorithms vary by preference of all-fog, all-cloud or mixed deployments. As expected, the Random algorithm resulted in the highest number of mixed deployments (with operators present both in the fog and in the cloud) that were observed in case of ca. 80% of the requests. Because the decisions made by the algorithm are based on a purely random choice, and no performance constraints are considered, the algorithm leads to a sparse allocation of the operators. This contributes to a significant number of WAN transmissions and increases the network load. Furthermore, requests are affected by additional transmission latencies and repeatedly exceed their latency thresholds, leading to one of the worst results observed in all scenarios. Alternatively, the FogOnly algorithm deployed as many (ca. 50 - 85%) requests as possible fully in the fog. In case of insufficient resources, the requests were rejected. On the other hand, AllDC deployed all requests fully in the cloud. However, these approaches resulted in unsatisfactory success rates, as using a single deployment cannot guarantee efficient handling of requests in a heavily loaded environment.

The other algorithms address these issues by considering the expected impact of excessive network transmissions and resource limitations. The FG algorithm that attempts to deploy streams fully in the fog significantly reduces the number of mixed deployments in favor of the other deployment types. More balanced use of different deployment types contributes to improvement in terms of success rates. The concept further evolves in the FG+ algorithm that prefers deploying the operators in the fog and determines the least demanding WAN transmissions to be executed in case the request cannot fully fit in the fog. Eventually, more all-fog deployments are observed than the previous algorithms, and the negative effects of necessary data transmissions are reduced. As a result, significantly better success rates are provided than the Random, FogOnly and AllDC algorithms.

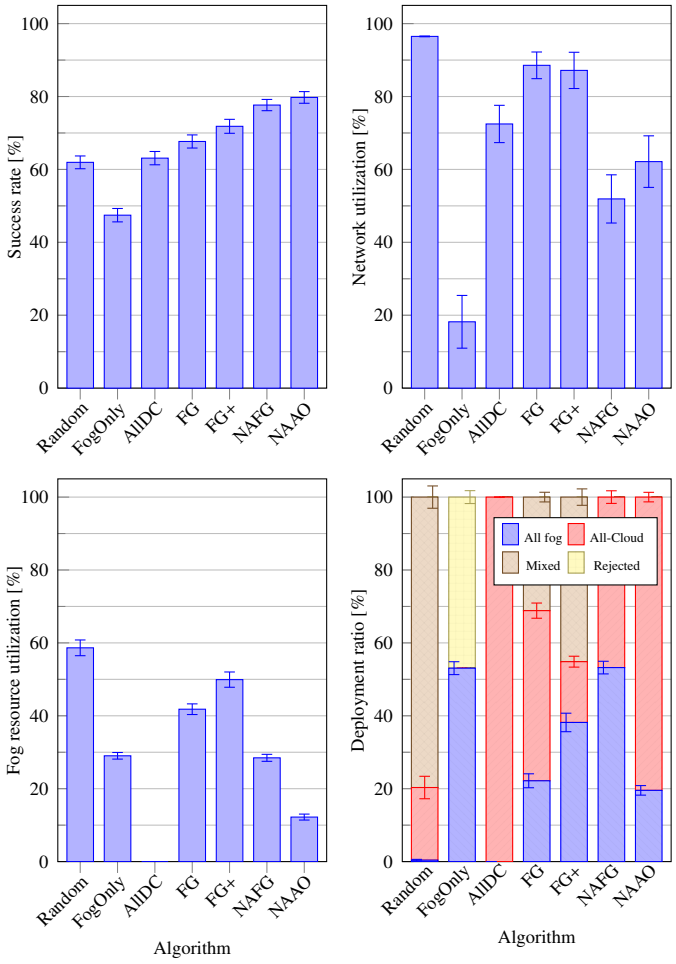


Figure 10: "High" scenario results.

NAFG and NAAO further mitigate the negative impact of network transmissions. For example, the NAFG implements a mechanism that deploys all the operators either in the fog or in the cloud, eliminating mixed deployments and effectively reducing network utilization. In the case of abundant fog resources, as in the *Low* scenario, all-fog deployments are highly preferred. Such distribution of deployment types is reflected in higher fog utilization in certain cases. On the other hand, the NAAO algorithm implements a more complex mechanism that introduces an additional metric based on the estimated execution time span. Due to more accurate estimation of possible latencies related to additional WAN transmissions, efficient op-

erator deployment is possible. This results in more requests deployed fully in the cloud, preserving fog resources for time-critical requests. Although the algorithm considers mixed deployments to be acceptable, the latency constraints eventually lead to only all-fog and all-cloud deployments, similarly as in the case of NAFG. Still, in case of exceptionally demanding requests and insufficient fog resources, applying mixed deployments may improve the performance. With this approach, a superior overall success rate and satisfactory resource utilization are achieved. It should be noted that despite disparate approaches to operator deployment, both NAFG and NAAO provide the best success rates and differ only by preferred deployment location (fog or cloud) and resource utilization.

Based on these observations, it can be concluded that the results are highly affected by the number of WAN transmissions triggered by the algorithms and delays introduced by these transmissions. The transmission latency is a key factor for accurate estimation of request execution timespan and should be considered when determining the optimal allocation of the operators. Algorithms that do not consider it are less effective and may lead to saturation of either fog or network resources. This is proven by the inferior performance of the Random algorithm that provides worse results than its alternatives. Moreover, the issues cannot be solved by deploying all the requests exclusively in the fog (FogOnly) or the cloud (AllDC). Both fog resources and network resources (required for WAN transmissions to the cloud) are limited and significantly affect the performance when overutilized. Therefore, the desired solution should be designed on the basis of fog and cloud interplay. As a first step towards the desired solution, the latency factor is indirectly referred to in the FG algorithm, where the preference of fog deployment helps to reduce the negative impact of network transmissions. The additional estimation of transmission bandwidth in FG+ is introduced as a further improvement. However, even reasonable (considering resource utilization) partitioning of the operators between the fog and the cloud may introduce unacceptable delays that inevitably affect performance. Eventually, FG+ provides a few percent worse success rates than its successors. The issue is best solved by the NAFG and NAAO algorithms. Their results may be connected to the absence of mixed deployments. NAFG is the least complex algorithm that at the same time, follows reasonable assumptions. Deploying all operators either in the fog or in the cloud yields good results in both success rate and fog resource utilization. In addition, the algorithm reduces network load the most. As a result, NAFG is the second-best algorithm in the success rate comparison, being only slightly worse than NAAO in the *High* scenario. The last of the algorithms, NAAO, is not only effective but also intuitive. Although it considers both compute and network resources, the algorithm is not overly complex and directly uses network latency metrics that can be easily estimated. In addition, NAAO allocates fog resources only when fog usage is profitable, thus enabling an approach that guarantees the efficient use of limited fog resources.

Considering the overall results, both NAFG and NAAO are suitable for use in stream processing environments. Aside from the best success rates, they achieve satisfactory resource utilization levels. A choice for the specific environment may be based on its characteristics, namely available resources and network policies. While NAFG minimizes network utilization and leads to a slightly higher fog utilization, NAAO does the opposite and prefers higher network utilization over-allocating fog resources. Therefore, either of the two algorithms is capable of satisfying the requirements of different environments.

5.3. Practical implications

The practical implications of using these algorithms in real-world scenarios should be considered regarding the example applications mentioned in Section 1.1. In all 3 cases (intelligent traffic control, surveillance and event monitoring, and industrial automation), the applications can process data within limited time and resources. As observed, choosing a workload deployment algorithm significantly affects resource consumption and the percentage of successfully handled requests. The number of successful requests determines, for instance, how many critical events an algorithm can handle in time in production or how many mobile video sources it can analyze at once. Therefore, both the reliability and usability of the application highly depend on the underlying algorithm. They may be entirely different for any two of the presented algorithms, *e.g.* Random and NAAO.

6. Conclusions

The research presented in this paper involved the design and thorough evaluation of 7 algorithms responsible for dynamic workload deployment in stream processing environments. The algorithms varied by implemented approach and considered different decision factors such as the number or volume of WAN transmissions, fog resource occupancy, or effective transmission latency caused by the designated workload deployment. The experiments were carried out in an emulated virtual network environment and involved the evaluation of 4 metrics in 3 different load scenarios. Achieved results demonstrated that a proper choice of the algorithm has a critical impact on the overall performance of stream processing applications.

While the reference algorithms could not provide satisfactory results, all 3 proposed approaches (FG+, NAFG, NAAO) introduced significant success rate and resource consumption improvements. Moreover, the last two proposed algorithms outperform their predecessors with ca. 80% and higher success rates, depending on the scenario, and differ by preferred deployment type. Similar results are possible regardless of choice between network and fog resource optimization.

The evaluation of the algorithms confirms that the success rates may be maximized by considering request execution graphs and ensuring balanced utilization of both computing and network resources. Moreover, data transmission latency is a crucial factor in the operator allocation process. However, the stream processing environment is affected by several other factors such as costs of cloud and fog computational resources, costs and availability of network resources, and characteristics of stream processing requests, namely resource require-

ments and latency thresholds. Future research can further explore these to provide even more efficient workload deployment mechanisms in specific use cases.

Acknowledgments

Part of this work conducted by the AGH was funded by the Dean under Grant No. 15.11.230.292 and the work conducted by INRIA was funded by the Chist-ERA STAR project.

References

[1] F. Bonomi, et al., Fog Computing and Its Role in the Internet of Things, in: Proc. the First Edition of the MCC Workshop on Mobile Cloud Computing, Helsinki, Finland, 2012, pp. 13–16.

[2] R. Dautov, S. Distefano, Stream Processing on Clustered Edge Devices, IEEE Transactions on Cloud Computing (2020) 1–1. doi:10.1109/TCC.2020.2983402.

[3] Y. Li, A.-C. Orgerie, I. Rodero, B. L. Amersho, M. Parashar, J.-M. Menaud, End-to-end energy models for Edge Cloud-based IoT platforms: Application to data stream analysis in IoT, Future Generation Computer Systems 87 (2018) 667–678.

[4] P. Silva, A. Costan, G. Antoniu, Investigating Edge vs. Cloud Computing Trade-offs for Stream Processing, in: 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 469–474. doi:10.1109/BigData47090.2019.9006139.

[5] H. Cao, M. Wachowicz, An Edge-Fog-Cloud Architecture of Streaming Analytics for Internet of Things Applications, Sensors 19 (2019) 3594. doi:10.3390/s19163594.

[6] L. Hernandez, H. Cao, M. Wachowicz, Implementing an edge-fog-cloud architecture for stream data management, in: 2017 IEEE Fog World Congress (FWC), 2017, pp. 1–6. doi:10.1109/FWC.2017.8368538.

[7] S. Yang, IoT Stream Processing and Analytics in the Fog, IEEE Communications Magazine 55 (2017) 21–27. doi:10.1109/MCOM.2017.1600840.

[8] M. Santos De Brito, S. Hoque, R. Steinke, A. Willner, Towards programmable fog nodes in smart factories, in: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), 2016, pp. 236–241. doi:10.1109/FAS-W.2016.57.

[9] P. Borylo, A. Lason, J. Rzasa, A. Szymanski, A. Jajszczyk, Green Cloud Provisioning Throughout Cooperation of a WDM Wide Area Network and a Hybrid Power IT Infrastructure, Journal of Grid Computing 14 (2016) 127–151.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: Enabling Innovation in Campus Networks, ACM SIGCOMM Computer Communication Review 38 (2008).

[11] P. Borylo, A. Lason, J. Rzasa, A. Szymanski, A. Jajszczyk, Energy-Aware Fog and Cloud Interplay Supported by Wide Area Software Defined Networking, in: IEEE Int. Conf. on Communications (ICC 2016), IEEE, Kuala Lumpur, Malaysia, 2016.

[12] P. Basanta-Val, N. Fernández-García, L. Sánchez-Fernández, Predictable remote invocations for distributed stream processing, Future Generation Computer Systems 107 (2020) 716–729.

[13] H. Cao, C. Q. Wu, L. Bao, A. Hou, W. Shen, Throughput optimization for storm-based processing of stream data on clouds, Future Generation Computer Systems 112 (2020) 567–579.

[14] W. Aljoby, X. Wang, T. Fu, R. Ma, On SDN-Enabled Online and Dynamic Bandwidth Allocation for Stream Analytics, in: 2018 IEEE 26th International Conference on Network Protocols (ICNP), 2018, pp. 209–219.

[15] R. Bharath Das, G. Di Bernardo, H. Bal, Large Scale Stream Analytics Using a Resource-Constrained Edge, in: 2018 IEEE International Conference on Edge Computing (EDGE), 2018, pp. 135–139. doi:10.1109/EDGE.2018.00027.

[16] M. Mukherjee, S. Kumar, M. Shojafar, Q. Zhang, C. X. Mavromoustakis, Joint Task Offloading and Resource Allocation for Delay-Sensitive Fog Networks, in: ICC 2019 - 2019 IEEE International Conference on Communications (ICC), 2019, pp. 1–7.

[17] Z. Hong, W. Chen, H. Huang, S. Guo, Z. Zheng, Multi-Hop Cooperative Computation Offloading for Industrial IoT–Edge–Cloud Computing Environments, IEEE Transactions on Parallel and Distributed Systems 30 (2019) 2759–2774. doi:10.1109/TPDS.2019.2926979.

[18] A. R. Zamani, M. Zou, J. Diaz-Montes, I. Petri, O. Rana, A. Anjum, M. Parashar, Deadline Constrained Video Analysis via In-Transit Computational Environments, IEEE Transactions on Services Computing 13 (2020) 59–72. doi:10.1109/TSC.2017.2653116.

[19] A. A. Khadir, S. A. H. Seno, SDN-based offloading policy to reduce the delay in fog-vehicular networks, Peer-to-Peer Networking and Applications 14 (2021) 1261–1275.

[20] L.-A. Phan, D.-T. Nguyen, M. Lee, D.-H. Park, T. Kim, Dynamic fog-to-fog offloading in SDN-based fog computing systems, Future Generation Computer Systems 117 (2021) 486–497.

[21] B. Yin, W. Shen, Y. Cheng, L. X. Cai, Q. Li, Distributed resource sharing in fog-assisted big data streaming, in: 2017 IEEE International Conference on Communications (ICC), 2017, pp. 1–6. doi:10.1109/ICC.2017.7996724.

[22] A. Kallel, M. Rekik, M. Khemakhem, Iot-fog-cloud based architecture for smart systems: Prototypes of autism and COVID-19 monitoring systems, Software Practice and Experience 51 (2020). doi:10.1002/spe.2924.

[23] A. da Silva Veith, M. Dias de Assuncao, L. Lefèvre, Latency-Aware Placement of Data Stream Analytics on Edge Computing, in: Service-Oriented Computing, Springer International Publishing, Hangzhou, Zhejiang, China, 2018, pp. 215–229.

[24] A. da Silva Veith, F. R. de Souza, M. D. de Assunção, L. Lefèvre, J. C. S. dos Anjos, Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing, in: Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Association for Computing Machinery, New York, NY, USA, 2019.

[25] V. De Maio, D. Kimovski, Multi-objective scheduling of extreme data scientific workflows in fog, Future Generation Computer Systems 106 (2020) 171–184.

[26] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, V. Vlassov, SpanEdge: Towards Unifying Stream Processing over Central and Near-the-Edge Data Centers, in: 2016 IEEE/ACM Symposium on Edge Computing (SEC), 2016, pp. 168–178. doi:10.1109/SEC.2016.17.

[27] A. da Silva Veith, M. Dias de Assuncao, L. Lefevre, Latency-Aware Strategies for Deploying Data Stream Processing Applications on Large Cloud-Edge Infrastructure, IEEE Transactions on Cloud Computing (2021) 1–1. doi:10.1109/TCC.2021.3097879.

[28] N. Mehran, D. Kimovski, R. Prodan, A Two-Sided Matching Model for Data Stream Processing in the Cloud - Fog Continuum, in: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Los Alamitos, CA, USA, 2021, pp. 514–524.

[29] OpenStack, Open Source Cloud Computing Software, `https://www.openstack.org/`, 2015. Accessed 03 Jul. 2021.

[30] A. Botta, W. de Donato, V. Persico, A. Pescapé, Integration of Cloud computing and Internet of Things: A survey, Future Generation Computer Systems 56 (2016) 684–700. doi:https://doi.org/10.1016/j.future.2015.09.021.

[31] I. Sarkar, M. Adhikari, N. Kumar, S. Kumar, A collaborative computational offloading strategy for latency-sensitive applications in fog networks, IEEE Internet of Things Journal (2021) 1–1. doi:10.1109/JIOT.2021.3104324.

[32] P. Borylo, A. Lason, J. Rzasa, A. Szymanski, A. Jajszczyk, Anycast Routing for Carbon Footprint Reduction in WDM Hybrid Power Networks with Data Centers, in: IEEE Int. Conf. on Communications (ICC 2014), IEEE, 2014, pp. 3714–3720.

[33] P. Borylo, A. Lason, J. Rzasa, A. Szymanski, A. Jajszczyk, Fitting Green Anycast Strategies to Cloud Services in WDM Hybrid Power Networks, in: IEEE Global Communications Conference (GLOBECOM 2014), IEEE, 2014, pp. 2592–2598.