Latency-aware Placement of Stream Processing Operators

Raphael Ecker[1], Vasileios Karagiannis[2], Michael Sober[3], Elmira Ebrahimi[3], and Stefan Schulte[3]

[1] TU Wien, Vienna, Austria
[2] Center for Digital Safety & Security, Austrian Institute of Technology, Vienna, Austria
vasileios.karagiannis@ait.ac.at
[3] Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, TU Hamburg, Hamburg, Germany
{name.lastname}@tuhh.de

Abstract. The rise of the Internet of Things and Fog computing has increased substantially the number of interconnected devices at the edge of the network. As a result, a large amount of computations is now performed in the fog generating vast amounts of data. To process this data in near real time, stream processing is typically employed due to its efficiency in handling continuous streams of information in a scalable manner. However, most stream processing approaches do not consider the underlying network devices as candidate resources for processing data. Moreover, many existing works do not take into account the incurred network latency of performing computations on multiple devices in a distributed way. Consequently, the fog computing resources may not be fully exploited by existing stream processing approaches. To avoid this, we formulate an optimization problem for utilizing the existing fog resources, and we design heuristics for solving this problem efficiently. Furthermore, we integrate our heuristics into Apache Storm, and we perform experiments that show latency-related benefits compared to alternatives.

Keywords: Stream Processing · Fog Computing · Edge Computing · Internet of Things · Apache Storm.

1 Introduction

Data stream processing is widely used for processing data in near real time, e.g., in factory automation or banking scenarios [5]. While a stream processing application could in theory be executed on a single computational resource, the scale and scope of many applications require distributing the stream processing operators on different computational resources. Distributing the operators also allows to parallelize tasks, which aids significantly in scaling the operations. Originally, most approaches to enable distributed data stream processing relied on cloud-based resources [4]. However, more recently, the utilization of fog resources in addition to the cloud has gained a lot of attention [12].

The fog can be seen as a continuum stretching from the edge of the network to the cloud, allowing to make use of computational resources anywhere in between [13]. These resources can be, e.g., single-board computers, sensor nodes, cloudlets at the edge of the network, routers and switches, or virtual machines in the cloud [26]. Notably, the fog does not replace the cloud; instead, cloud-based resources can be included in the fog [14]. If compared to the centralized data centers of the cloud, fog computing provides geographically distributed resources closer to the data sources [13]. This is especially helpful in scenarios where the data sources are also geographically distributed, e.g., in the Internet of Things (IoT), since it allows to decrease the communication overhead both regarding network latency and the amount of data to be transferred to the cloud.

When applying fog computing principles in data stream processing, the overall processing response time depends on how the operators are arranged based on the location of the data sources. With the geographical distribution of computational resources in the fog, the distances and therefore also the network latency between the resources become larger. This leads to the question of how operators should be distributed on available resources in order to meet the near real-time requirements of many stream processing applications. This is also known as the stream operator placement problem [25].

While many approaches to optimize the placement of stream processing operators have been introduced, most of them do not consider the network latency between the operators. Consequently, existing approaches may not be taking full advantage of fog environments which can include a variety of distributed computational resources. To address this concern, this work aims at designing a latency-aware stream operator placement approach for the fog. To this end, we formulate the placement of the operators on fog resources as an optimization problem. Since similar problems are typically NP-hard [17], we focus on designing heuristics that approximate the optimal solution efficiently. Furthermore, we build a prototype based on Apache Strom, and we perform experiments that show great potential due to reducing the stream processing latency, compared to alternatives.

The remainder of this paper is structured as follows: In Section 2, we provide an overview of related work. Afterward, Section 3 offers information regarding the utilized system model, and Section 4 presents the design of our latency-aware placement heuristics. Then, we experiment with the performance of the heuristics in Section 5, and we conclude this paper in Section 6.

2 Related Work

So far, optimizing the placement of stream processing operators has been mostly investigated with a focus on cloud-based resources. Nevertheless, fog-based approaches also exist [12]. Since placing operators on computational resources is usually considered an NP-hard problem (as mentioned in Section 1), aiming for an optimal solution may be too computationally intensive. To avoid this, efficient heuristics can be used. Some existing works apply a greedy heuristic

that finds a fitting resource allocation [25]. However, greedy solutions may suffer from finding local optimums which may differ significantly from the global optimum. Another approach is to decompose the problem into multiple partitions, which can be solved more efficiently. In the case of fog computing, the usage of location for the partitioning has been discussed, while networking aspects have been mostly neglected [21]. For instance, Eskandari et al. [9] present *P-Scheduler* which allows the partitioning of nodes and edges in a hierarchical way. Instead of geographical separation, other approaches organize the fog in a logical hierarchy. For example, Nardelli et al. [17] present a solution whereby fog-based resources are arranged into a hierarchical tree based on network capabilities. Notably, the aforementioned approaches do not focus on the latency of the underlying network links which is within the scope of our work.

Approaches that consider the underlying network also exist. For example, Pietzuch et al. [19] apply spring relaxation to estimate latencies in a three-dimensional Euclidean space. However, this paper presents early work without details regarding integration into cloud and fog computing environments. Interestingly, subsequent work by Cardellini et al. [6,7] integrates similar concepts into Apache Storm for finding fog resources with low utilization and high availability. Prosperi et al. [20] present *Planner* which identifies subgraphs of a topology to be deployed in the cloud or at the edge. Thus, several approaches have been proposed for taking into account some networking aspects of placing operators in the fog. However, most of them do not focus on the latency of the underlying network links that connect the operators. In this work, we formulate an optimization problem that considers these links to reduce the latency of the processing.

3 System Model

Previous approaches define a strict separation of resources into a multi- or often three-layer fog model including edge, fog and cloud. In the work at hand, the system is viewed as a computing continuum in which the type of resource is not considered relevant in the context of the operator placement decisions. Instead, a resource is characterized by the computational capabilities it can provide. This way, all available resources are taken into account whether they are network devices, single-board computers at the edge, or powerful servers. In addition, all the network links between the resources of the system are taken into account, which is essential to achieve an efficient distribution of operators, e.g., with reduced latency and increased throughput [20].

Resource-constrained IoT devices such as wireless sensor nodes are not considered full-fledged resources in our system. The reason for this decision is that such resources may be highly unreliable due to lossy wireless transmissions or limited battery lifespans. Nevertheless, since such devices are omnipresent in IoT and fog environments, we take them into account through their connection to a supervisor. A *supervisor* is a component that is integrated into a resource to be responsible for task allocation locally. If resource-constrained devices that

Algorithm 1: Process to update a supervisor's position in the latency cost space.

```
   Input: estimatedPosition=current coordinate, minEpsilon=minimum required
          movement to continue iterating
   Output: updated estimated position
 1 peers=peerSelection()
 2 getUpdatedPeerPositions(peers)
 3 measurePeerLatencies(peers)
 4 maxMovement=MAXFLOAT
 5 iteration=0
 6 while maxMovement>minEpsilon and iteration<100 do
 7 |   movement=Vector(0,0,0)
 8 |   foreach peer in peers do
 9 |   |   d=calculateForce(peer,estimatedPosition)
10 |   |   movement=movement+d
11 |   end
12 |   estimatedPosition+=movement
13 |   maxMovement=movement.length
14 |   iteration++
15 end
16 return estimatedPosition;
```

can offer computational capabilities are connected to a resource, the supervisor is responsible for assigning tasks to them. We also model the way whereby the performance of network links between supervisors is measured. For the estimation of link latencies, an approach using spring relaxation is used due to its decentralized design that does not require coordination [19]. Each supervisor independently estimates its coordinates in a three-dimensional cost space. In this space, the distance between two coordinates correlates to the estimated latency between the supervisors. Initially, each supervisor's coordinates are randomized and are then adjusted based on periodic latency measurements [19].

Algorithm 1 shows the process of executing a periodic latency measurement. When a supervisor can communicate with other supervisors, these are referred to as peers. Few peers are assumed to be known in the beginning to bootstrap the algorithm. First, a supervisor adjusts its position by assuming the coordinates of other peers as fixed (Lines 1-3). Spring relaxation is then applied between the supervisor and its peers to pull the supervisor into a position closer to the measured latencies (Lines 8-12). This is repeated iteratively until the total movement across all peers becomes insignificant or a maximum number of iterations is reached (Line 6) [19]. The new position is then returned (Line 16). After this process, the supervisor saves the new position in a key-value store and waits until the periodic latency estimation starts again to refine the coordinates based on new measurements or updated positions of peers. The interval of the periodic estimation is slightly randomized to prevent peers from potentially being synchronized which can lead to cyclic updating based on outdated measurements.

Algorithm 2: Process to calculate the force based on spring relaxation.

Input: peer, estimated Position
Output: d=movement update vector
1 d=peer.position-estimatedPosition
2 force=multiplierConstant*log(d.length/peer.latency)
3 d=d/d.length
4 d=d*force
5 **return** d;

To calculate the spring force in Line 9 of Algorithm 1, we present Algorithm 2. First, a unit vector of the distance between the supervisors is calculated (Lines 1, 3). The force is defined as $c_1 * log(d/c_2)$, with c_1 being a constant, c_2 the distance (e.g., using network ping), and d the distance in the latency space (Line 2). This force is then applied in the correct direction by multiplying it with the unit vector (Line 4). The logarithmic scale is used for preventing the forces between very distant positions from becoming too large in comparison to the smaller forces [15]. After the position of a supervisor in the latency cost space is updated, only a few other supervisors are considered as peers [23]. These are selected based on latency so that the peers of a supervisor can be used for operator placement with low communication latency. This way, a heuristic for operator placement can use these peers for achieving low-latency stream processing.

4 Optimization Problem

In this section, we present the proposed approach. First, Section 4.1 formulates an optimization problem tailored to the system model discussed in Section 3. Then, Section 4.2 presents heuristics for solving this problem efficiently.

4.1 Formulation

The placement problem is formulated as a cost-minimization problem. The cost function is shown in Equation 1. This equation consists of four weighted parameters with weights w_1 to w_4 which regulate the amount of influence of each parameter.

$$s(x) = w_1 * s_{lat}(x) + w_2 * s_{sup}(x) + w_3 * s_{co}(x) + w_4 * s_{event}(x) \quad (1)$$

$s_{lat}(x)$ is the highest estimated network latency that is accumulated during the processing of an event in the topology T. For any operator, $s \in T_{source}$ we define that $s_{lat}(s) = 0$. For the other operators in the topology, applies that $\forall o \in T : s_{lat}(o) = max(s_{lat}(p) + l_{p,o} : p \in o_{predeccesors})$ with $l_{a,b}$ being the estimated network link latency from the operator a to b. The latency score of the topology is the maximum among all sinks: $s_{lat}(T) = max(s_{lat}(s) : s \in T_{sinks})$. $s_{sup}(x)$ is used to condense the placement of operators, and is defined as

$s_{sup}(x) = \frac{|supervisors \in x|}{|supervisors|}$. This parameter aims at reducing the total number of employed supervisors allowing to temporarily shut down redundant supervisors. This reduces unused resources. $s_{co}(x)$ is a measure of the co-location of operators. If $p(o)$ is the placement of an operator o then $s_{co}(x) = \frac{|e_{o_1,o_2} \in T : p(o_1) = p(o_2)|}{|e_{o_1,o_2} \in T|}$. Finally, $s_{event}(x)$ counts the events emitted over not co-located edges. With $t(e)$ being the events emitted on an edge e, $s_{event}(x)$ can be defined as $s_{event}(x) = \frac{\sum_{e_{o_1,o_2} \in T : p(o_1) \neq p(o_2)} t(e_{o_1,o_2})}{\sum_{e_{o_1,o_2} \in T} t(e_{o_1,o_2})}$. In practice, $t(e)$ might not be measured exactly because most stream processing frameworks only have a metric about the emitted events for each operator. Nevertheless, $t(e)$ can be approximated by dividing the operator-based statistic by the count of outgoing edges. Thus, $s_{co}(x)$ relates to the general performance while $s_{event}(x)$ focuses on important operators being co-located. All parameters have a range of $[0, 1]$ except for the latency $s_{lat}(x)$ that has a range of $[0, \infty]$. To account for this range, w_1 is set to $\frac{1}{1000000}$ so that it becomes insignificant and acts mostly as a deciding factor between similarly scored solutions. All the other weights w_2 to w_4 are set to 1.

Regarding constraints, every operator is assigned to one supervisor, while each supervisor can have operators of one topology. Furthermore, the memory and CPU usage of the supervisors cannot exceed the integrated capacities. Overall, the cost function favors co-located placements, while the constraints ensure the efficient use of the available computational resources.

4.2 Heuristics

To solve the aforementioned optimization problem, we design three heuristics: hill-climbing, ant-system, and hybrid. All heuristics aim at using computational resources sparingly so that they can be executed iteratively for online scheduling. Thus, instead of striving to find an optimal solution (that may be too computationally intensive), our heuristics aim at approximating the optimal solution, which is preferred for real-time stream processing. To further reduce the computational overhead of the heuristics, a topology is rescheduled periodically, e.g., every 30 seconds, and only if there have been system updates.

Hill-climbing: This heuristic tries to find a modification of a placement that reduces the cost while also reducing the number of constraint violations, or maintaining it the same [24]. This means that a suboptimal placement will never be selected. Consequently, this heuristic might lead to getting stuck in local optimums. Hill-climbing uses the following three operations to modify a placement:

1. Moving one operator to a different supervisor.
2. Swapping the placement of two operators which are not co-located.
3. Moving all operators of a supervisor to a different supervisor.

Ant System: This heuristic is based on the real-world collaborative pathfinding of ants [8]. A placement is represented by a path (in a graph) that starts at an

Algorithm 3: Process of finding a placement based on the ant system.

```
   Input: currentPlacement
   Output: new optimised placement
1  bestAnt=currentPlacement
2  pheromone=initialisePheromone(bestAnt)
3  while no exit condition fulfilled do
4  |  ants=generatePopulationOfAntsWithPaths()
5  |  bestAnt=selectBestAnt(ants,bestAnt)
6  |  pheromone=placeAndDecayPheromone(pheromone,ants)
7  |  updateExitConditions()
8  end
9  return bestAnt;
```

operator, ends at a supervisor, and alternates between supervisors and operators until all operators are included in the path exactly once. Edges from the operators to the supervisors represent individual operator assignments. When the same edges are frequently chosen, they are given priority in future assignments based on a calculated pheromone. Notably, the order of the operators can affect the placement since the first operators are preferred. This can lead to local optimums. To avoid local optimums, the operator order is randomized for every placement.

Algorithm 3 shows the process of the ant system. First, the pheromone of each edge (i, j) is initialized with a configurable parameter p_i. A previous placement can also affect the initial pheromones by executing the pheromone placement p_c times for a path that is equivalent to the current placement (Lines 1-2). Then, m ants are generated each one having a random path (Line 4). The best ant is updated (Line 5) and the pheromones on the graph edges are modified accordingly (Line 6). The iteration count and the number of iterations since the last score are used as exit conditions (Lines 3,7). When the algorithm ends, the best placement is returned (Line 9). The mathematic formulations to execute the path generation (Line 4), scoring (Line 5), and pheromone update (Line 6) are explained below.

The probability of an ant k at time t to move from a node i in the graph to j is shown in Equation 2. It consists of an a priori heuristic, for which a custom one is defined in Equation 3, and an a posteriori heuristic, i.e., the pheromone. α and β are used as parameters to weigh the importance of both elements [8].

$$p_{ij}^{k} = \begin{cases} \frac{|\mathcal{T}_{ij}(t)^{\alpha}| \cdot |\eta_{ij}(t)^{\beta}|}{\sum_{k \in \text{allowed movements}} |\mathcal{T}_{ik}(t)^{\alpha}| \cdot |\eta_{ik}(t)^{\beta}|} & j \in \text{allowed movements} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The a priori heuristic in Equation 3 is used for finding solutions with more co-locations. The output value of this heuristic is multiplied by 2 if another operator of the topology is placed on the supervisor, or if a predecessor or successor operator is placed on the supervisor (to achieve co-location). If the placement overloads the resources, the value is divided by 2. This way, the pheromone encourages or discourages certain decisions, thereby guiding the exploration.

$$\eta_{ij} = 1 \cdot 2^{\text{pre or suc on } j} \cdot 2^{j \text{ already used}} \cdot \frac{1}{2^{\text{overloads } j\text{'s CPU}}} \cdot \frac{1}{2^{\text{overloads } j\text{'s mem}}} \quad (3)$$

Equation 4 shows the decay of the pheromone which is based on the parameter ρ and the newly placed pheromone of all ants for an edge (i, j) [8]. In this ant system variant, a minimum pheromone amount p_m on each edge is enforced to ensure that the random exploration of alternate paths never stops [22].

$$\mathcal{T}_{ij}(t) = \max(\rho \cdot \mathcal{T}_{ij}(t-1) + \sum_{k=1}^{m} \Delta\mathcal{T}_{ij}^{k}, p_m) \quad (4)$$

The pheromone placed on an edge (i, j) by the ant k is defined in Equation 5. This equation uses a constant Q to scale the placed pheromone and S_k as the score of the ant k's solution. This ensures that a lower score results in more pheromones placed, thereby attracting more ants to better solutions [8].

$$\Delta\mathcal{T}_{ij}^{k} = \begin{cases} \frac{Q}{S_k} & k\text{th ant uses edge (i,j) in path} \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

To ensure that constraint violations are reduced, S_k is defined using a combined score, i.e., $S_k = score(k) + 10000 * constraintViolations(k)$.

Hybrid: This heuristic combines the benefits of hill-climbing and the ant system [24]. The placement selection of the ant system is very effective at exploring the solution space by avoiding local optimums. At the same time, hill-climbing is more effective in performing smaller adjustments (with the risk to get stuck in local optimums). The hybrid approach first runs the ant system for effective exploration and then executes hill-climbing for making small adjustments.

5 Evaluation

To evaluate our approach, we build a system for stream operator placement, and we run preliminary experiments to compare our heuristics with existing alternatives. To make this evaluation more comprehensible, Section 5.1 describes the evaluation environment, and Section 5.2 presents the produced results.

5.1 Environment

In this evaluation, we experiment using a custom implementation of our heuristics, which we integrate into Apache Storm. In addition, we run experiments using two other approaches: the default scheduler of Apache Storm, and the Resource-Aware Scheduler [2, 3, 18]. The former assigns operators in a round-robin way, while the latter aims at high resource utilization with reduced latency.

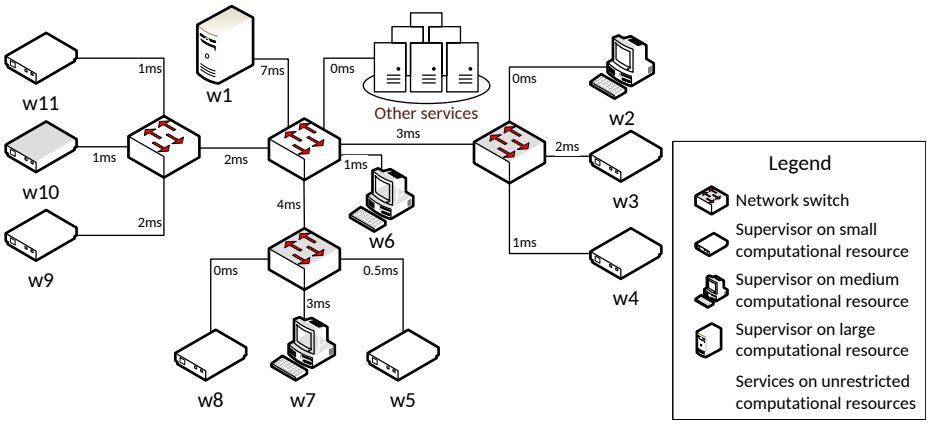

Fig. 1: Emulated network topology for the evaluation.

Fig. 1 shows the emulated network topology we use. This topology includes $w1$ to $w11$ which are the Apache Storm supervisors running in Docker containers. The symbol *other services* in Fig. 1 represents ancillary services required for running the experiments (such as Nimbus, Zookeeper and Redis). The resource capacities that are available for each supervisor are: $w1$ has 1.5 logical cores of CPU and 2048 Megabytes (MB) of memory, $w2, w6$ and $w7$ have 1 logical core of CPU and 1024 MB of memory, and $w3$ to $w5$, as well as $w8$ to $w11$, have 0.3 logical cores of CPU and 768 MB of memory. Notably, the resource capacities of the supervisors vary to represent different devices used in fog computing. The link latencies also vary (as shown in Fig. 1) to represent the distance of distributed computational resources found in fog computing environments [10, 11, 16]. The network emulation is executed on an Intel i7-4770k with 3.5Ghz, four CPU cores and eight threads using memory of 16GB DDR3 with 1600MHz. Ubuntu 20.04.4 LTS was used as the operating system with version 2.4.0 of Apache Storm. Finally, Containernet is used for running the containers as hosts in the topology, while Quagga handles the routing [1].

5.2 Results

To produce preliminary results which show how each examined approach performs, we run experiments using a randomly generated topology. Specifically, to generate this topology, we start from a single source and add either one (with 60% probability) or two (with 40% probability) operations. The resulting topology has 12 operations and 22 operators. The CPU and memory requirements of the operations are also randomized. As input data for the topology, we implement a data generator that always sends the current timestamp. This timestamp is propagated to the sink, which helps us measure the latency of the processing.

Fig. 2 shows a representative case of the minimum latency measured during the experiments, for each examined approach. The Y axis represents the latency

value in milliseconds (ms), while the X axis represents the corresponding CPU cycle of the runtime. The runtime of this experiment is executed for about 700 cycles. Each point in Fig. 2 represents a deployment, i.e., an execution of a stream operator placement heuristic that deploys the operators. For example, at CPU cycle (around) 90, the default scheduler is executed to perform a deployment which results in (around) 110 ms of latency for the stream processing.

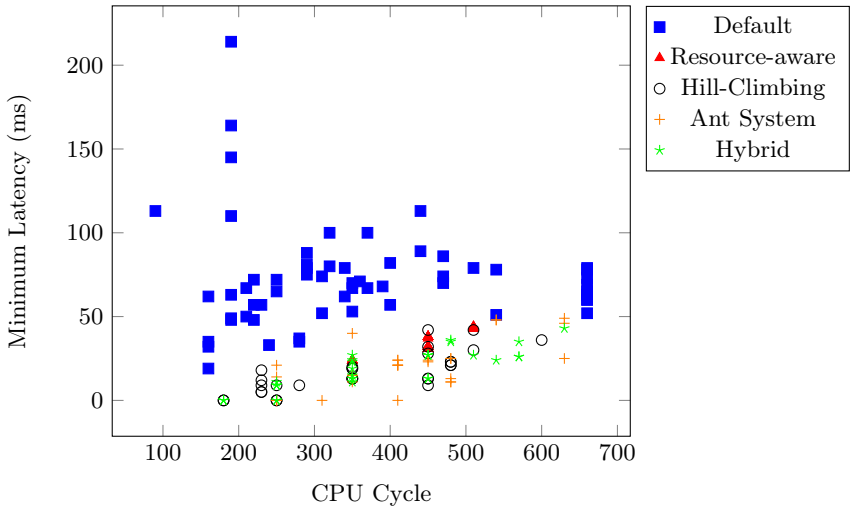

Fig. 2: Minimum latency of the placements for all the examined heuristics.

The results of Fig. 2 show clearly that the default scheduler, which does not aim for co-location of operators, incurs the highest latency. The resource-aware scheduler, which aims for lowering latency, produces results with much lower values which are, however, relatively high compared to the other heuristics. Hill-climbing performs similarly or better than the resource-aware scheduler due to performing frequent adjustments. The ant system exhibits a wide variance of results with latency that is close to the minimum and the maximum, compared to the other heuristics (apart from the default scheduler). Finally, the hybrid heuristic performs similarly or slightly better than hill-climbing without much variance in the results.

Overall, we note that our preliminary results show that the three heuristics we design for solving the stream operator placement problem, i.e., hill-climbing, ant system, and hybrid, tend to outperform the baselines when it comes to processing latency. This can be attributed, to a large degree, to the formulation of the optimization problem presented in Section 4.1. Specifically, the presented formulation in combination with the hybrid heuristic exhibit a lot of potential due to combining the powerful exploration phase of the ant system with the smaller adjustments of hill-climbing.

6 Conclusion

In this paper, we formulate an optimization problem for the placement of stream operators that is designed for processing IoT data in fog computing environments. This optimization problem takes into account the potentially distributed computational resources of the fog and favors the co-location of operators, which can lead to lower processing latency. Furthermore, we design heuristics for solving this optimization problem efficiently. Since the presented preliminary results show great potential, we plan to refine our Apache Storm-based prototype to record various metrics, e.g., regarding throughput, and resource utilization, so that we can take additional measurements. In addition, we plan to perform an exhaustive evaluation considering various diverse topologies in order to acquire a comprehensive view of the heuristics' performance.

Acknowledgements The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development as well as the Christian Doppler Research Association is gratefully acknowledged.

References

1. Apache Software Foundation: Quagga routing suite. `https://www.quagga.net/` (2018), accessed: 2022-31-03
2. Apache Software Foundation: Apache storm documentation: Resource aware scheduler. `https://storm.apache.org/releases/2.4.0/Resource_Aware_Scheduler_overview.html#Enhancements-on-original-DefaultResourceAwareStrategy` (2022), accessed: 2022-31-03
3. Apache Software Foundation: Apache storm documentation: Scheduler. `https://storm.apache.org/releases/2.4.0/Storm-Scheduler.html` (2022), accessed: 2022-31-03
4. de Assunção, M.D., Veith, A.D.S., Buyya, R.: Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. Network and Computer Applications **103**, 1–17 (2018)
5. Axenie, C., Tudoran, R., Bortoli, S., Hassan, M.A.H., Sánchez, C.S., Brasche, G.: Dimensionality reduction for low-latency high-throughput fraud detection on datastreams. In: 18th IEEE International Conference On Machine Learning And Applications. pp. 1170–1177. IEEE (2019)
6. Cardellini, V., Grassi, V., Presti, F.L., Nardelli, M.: Distributed qos-aware scheduling in storm. In: 9th ACM International Conference on Distributed Event-Based Systems. pp. 344–347. ACM (2015)
7. Cardellini, V., Grassi, V., Presti, F.L., Nardelli, M.: On qos-aware scheduling of data stream applications over fog computing infrastructures. In: 2015 IEEE Symposium on Computers and Communication. pp. 271–276. IEEE (2015)
8. Dorigo, M., Maniezzo, V., Colorni, A.: Ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics, Part B **26**(1), 29–41 (1996)

9. Eskandari, L., Huang, Z., Eyers, D.M.: P-Scheduler: Adaptive Hierarchical Scheduling in Apache Storm. In: Australasian Computer Science Week Multiconference. p. 26. ACM (2016)
10. Hasenburg, J., Grambow, M., Bermbach, D.: Mockfog 2.0: Automated execution of fog application experiments in the cloud. IEEE Transactions on Cloud Computing **11**(01), 58–70 (2021)
11. Hasenburg, J., Grambow, M., Grünewald, E., Huk, S., Bermbach, D.: Mockfog: Emulating fog computing infrastructure in the cloud. In: IEEE International Conference on Fog Computing. pp. 144–152. IEEE (2019)
12. Hiessl, T., Karagiannis, V., Hochreiner, C., Schulte, S., Nardelli, M.: Optimal Placement of Stream Processing Operators in the Fog. In: 3rd IEEE International Conference on Fog and Edge Computing. pp. 1–10. IEEE (2019)
13. IEEE: IEEE Standard 1934-2018 for Adoption of OpenFog Reference Architecture for Fog Computing (Aug 2018)
14. Karagiannis, V., Frangoudis, P.A., Dustdar, S., Schulte, S.: Context-aware routing in fog computing systems. IEEE Transactions on Cloud Computing **11**(01), 532–549 (2021)
15. Kobourov, S.G.: Spring embedders and force directed graph drawing algorithms. CoRR **abs/1201.3011** (2012)
16. Mayer, R., Graser, L., Gupta, H., Saurez, E., Ramachandran, U.: Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures. In: IEEE Fog World Congress. pp. 1–6. IEEE (2017)
17. Nardelli, M., Cardellini, V., Grassi, V., Presti, F.L.: Efficient operator placement for distributed data stream processing applications. IEEE Transactions on Parallel and Distributed Systems **30**(8), 1753–1767 (2019)
18. Peng, B., Hosseini, M., Hong, Z., Farivar, R., Campbell, R.H.: R-storm: Resource-aware scheduling in storm. In: 16th Annual Middleware Conference. pp. 149–161. ACM (2015)
19. Pietzuch, P.R., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.I.: Network-aware operator placement for stream-processing systems. In: 22nd International Conference on Data Engineering. p. 49. IEEE (2006)
20. Prosperi, L., Costan, A., Silva, P., Antoniu, G.: Planner: Cost-efficient execution plans placement for uniform stream analytics on edge and cloud. In: 2nd IEEE/ACM Workflows in Support of Large-Scale Science. pp. 42–51. IEEE (2018)
21. Skarlat, O., Nardelli, M., Schulte, S., Dustdar, S.: Towards QoS-aware fog service placement. In: IEEE International Conference on Fog and Edge Computing. pp. 89–96. IEEE (2017)
22. Stützle, T., Hoos, H.H.: MAX-MIN ant system. Future Generation Computer Systems **16**(8), 889–914 (2000)
23. Szymaniak, M., Presotto, D.L., Pierre, G., van Steen, M.: Practical large-scale latency estimation. Computer Networks **52**(7), 1343–1364 (2008)
24. Tsai, C., Rodrigues, J.J.P.C.: Metaheuristic scheduling for cloud: A survey. IEEE Systems Journal **8**(1), 279–291 (2014)
25. Varshney, P., Simmhan, Y.: Characterizing Application Scheduling on Edge, Fog and Cloud Computing Resources. Software: Practice and Experience **50**(5), 558–595 (2020)
26. Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., Kong, J., Jue, J.P.: All one needs to know about fog computing and related edge computing paradigms: A complete survey. Journal of Systems Architecture **98**, 289–330 (2019)