Adaptive Fog Service Placement for Real-time Topology Changes in Kubernetes Clusters

Tom Goethals[a], Bruno Volckaert[b] and Filip de Turck[c]
Department of Information Technology, Ghent University - imec, IDLab, Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium

Keywords: Fog Computing, Fog Networks, Edge Networks, Service Mesh, Service Scheduling, Edge Computing.

Abstract: Recent trends have caused a shift from services deployed solely in monolithic data centers in the cloud to services deployed in the fog (e.g. roadside units for smart highways, support services for IoT devices). Simultaneously, the variety and number of IoT devices has grown rapidly, along with their reliance on cloud services. Additionally, many of these devices are now themselves capable of running containers, allowing them to execute some services previously deployed in the fog. The combination of IoT devices and fog computing has many advantages in terms of efficiency and user experience, but the scale, volatile topology and heterogeneous network conditions of the fog and the edge also present problems for service deployment scheduling. Cloud service scheduling often takes a wide array of parameters into account to calculate optimal solutions. However, the algorithms used are not generally capable of handling the scale and volatility of the fog. This paper presents a scheduling algorithm, named "Swirly", for large scale fog and edge networks, which is capable of adapting to changes in network conditions and connected devices. The algorithm details are presented and implemented as a service using the Kubernetes API. This implementation is validated and benchmarked, showing that a single threaded Swirly service is easily capable of managing service meshes for at least 300.000 devices in soft real-time.

1 INTRODUCTION

Recent years have seen the rise of technologies such as containers, and more recently unikernels (Madhavapeddy et al., 2013), triggering a move from purely cloud-centered service deployments to fog computing and edge computing (Bonomi et al., 2012), in which services are deployed close to their consumers instead of in monolithic data centers.

Simultaneously, the number of devices in the edge dependent on cloud services, sometimes capable of running containers themselves, has grown rapidly. Between several initiatives for smart cities (Latre et al., 2016; Spicer et al., 2019) and an ever increasing variety of IoT devices, this ensures a continuing growth of cloud-connected devices. Fig. 1 shows the relation of the edge to the fog and the cloud, and the large amount and variety of devices within it.

The combination of IoT and fog computing provides a wide range of improvements, for example in efficiency and user experience. However, scheduling services in the fog requires a different approach than scheduling in the cloud.

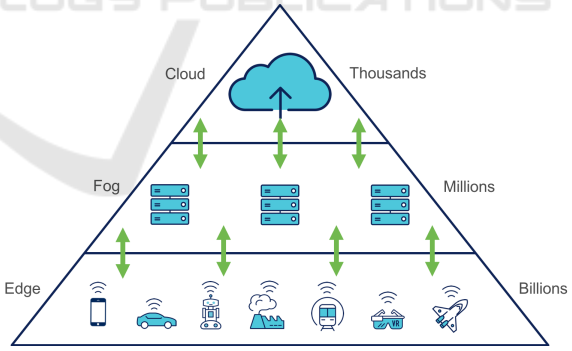


Figure 1: The conceptual difference between the cloud, the fog and the edge. The cloud is centralized and consists of few data centers, while the edge is everywhere, containing a large amount and variety of devices.

The main difference is that instead of being located in centralized data centers, the fog and edge are spread homogeneously over a large physical area, possibly containing hundreds of thousands of devices. Network grade and quality can vary by orders of mag-

[a] https://orcid.org/0000-0002-1332-2290
[b] https://orcid.org/0000-0003-0575-5894
[c] https://orcid.org/0000-0003-4824-1199

Goethals, T., Volckaert, B. and de Turck, F.
Adaptive Fog Service Placement for Real-time Topology Changes in Kubernetes Clusters.
DOI: 10.5220/0009517401610170
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 161-170
ISBN: 978-989-758-424-4

nitude over the entire fog, while latencies are much higher than in the cloud itself.

These properties result in a larger variety of network conditions and problems. Any scheduling solution should be able to work around broken lines of communication and changing network conditions.

Because of its decentralized nature, it is harder to scale services in the fog than in the cloud. It is not always possible or useful to scale services in place when demand from edge devices spikes, and deploying services closer to end users is more complex because of the size of the fog. Additionally, the topology of the fog and edge are constantly in flux, requiring real-time service migrations and scaling.

On the other hand, there are some challenges that remain mostly unchanged from cloud deployments. Node resource limitations need to be taken into account by the service scheduler, whether those are hardware resources or calculated metrics. Furthermore, because underloaded nodes have a relatively high resource overhead (e.g. operating system, idle services), the solution should strive for a minimal number of fog service deployments while placing them in optimal locations. This approach will attempt to minimize both total resource use, and access times for consumers in the edge. Finally, thresholds can be defined on certain metrics in order to guarantee certain levels of responsiveness or free resources.

To summarize, a performant fog service scheduler should:

- ***Req. 1*** work on the scale of hundreds of thousands of edge devices
- ***Req. 2*** be able to handle changing network conditions and topologies
- ***Req. 3*** take fog node resource limits and distance metrics between nodes into account
- ***Req. 4*** minimize the number of instances required for any fog service deployment

This article proposes Swirly, and its implementation for use with Kubernetes, to meet these requirements. Swirly is a scheduler that runs in the cloud or fog, which plans fog service deployments with a minimal number of service instances. It does so while optimizing the distance to edge consumers according to a chosen measurable metric. Furthermore, it can incorporate changes to the network and topology in real-time.

Section 2 presents existing research related to optimizing service deployments. Section 3 explains how the proposed algorithm works, while section 4 explains how it is implemented in Kubernetes. In section 5, an evaluation setup and methodology are presented to verify the performance of Swirly. The results of the evaluations are presented and discussed in section 6, with suggestions for future work in section 7. Finally, the conclusions are presented in section 8.

2 RELATED WORK

Shifting workloads between the cloud and edge hardware has been extensively researched, with studies on edge offloading (Mach and Becvar, 2017), cloud offloading (Kumar and Lu, 2010), and osmotic computing (Villari et al., 2016).

Many strategies exist for fog container deployment scheduling, ranging from simple but effective resource requests and grants (Santoro et al., 2017), to using deep learning for allocation and real-time adjustments (Morshed et al., 2017).

Initial research into fog computing and service scheduling dates from before the concept of the fog, for example Oppenheimer et al. (Oppenheimer et al., 2005), who studied migrating services in federated networks over large physical areas. This work takes into account available resources, network conditions, and the cost of migrating services between locations in terms of resources and latency.

Zhang et al. (Zhang et al., 2012) present an algorithm for service placement in geographically distributed clouds. Rather than focusing on resources as such, their algorithm makes placement decisions based on changing resource pricing of cloud providers.

Aazam et al. provide a solution for fog data center resource allocation based on customer type, service properties and pricing (Aazam and Huh, 2015a), which is also extended to a complete framework for fog resource management (Aazam and Huh, 2015b).

In more recent research, Santos et al. (Santos et al., 2019) present a Kubernetes-oriented approach for container deployments in the fog in the context of Smart Cities. Their solution is implemented as an extension to the Kubernetes scheduler and takes network properties of the fog into account.

Artificial intelligence is also making headway into fog scheduling research. For example, Canali et al. (Canali and Lancellotti, 2019) tackle fog data preprocessing with a solution based on genetic algorithms. Their solution distributes data sources in the fog, while minimizing communication latency and considering fog node resources.

Zaker et al. (Farzin Zaker and Shtern, 2019) propose a distributed look ahead mechanism for cloud resource planning. Rather than provisioning more resources to counter network load, they attempt to optimize bandwidth use through the configuration of

overlay networks. The predictive look ahead part is implemented by using the IBK2 algorithm.

Finally, Bourhim et al. (El Houssine Bourhim and Dieye, 2019) propose a method of fog deployment planning that takes into account inter-container communication. Their goal is to optimize communication latencies between fog-deployed containers, which is obtained through a genetic algorithm.

In summary, recent research focuses on artificial intelligence to find near-optimal solutions for a given network topology, and an implementation of such an algorithm in the Kubernetes scheduler has been shown to work. However, genetic algorithms are unfit to quickly react to constant changes in large network topologies. Moreover, the Kubernetes implementation in this paper is designed so it can be run anywhere, not only on a Kubernetes master node.

3 ALGORITHM

This section explains the concepts on which Swirly is built, and its operations. Swirly is designed around the assumption that some fog services are used by most, if not all, edge devices. This allows for a simple and flexible approach to building large service topologies. Examples of this can be found in IoT, where every device (e.g. smart bulb, climate control) connects to the same cloud service (e.g. device registration, status information, control webhooks).

For the rest of this paper, a network with fog and edge devices with frequent changes to its topology will be referred to as a swirl. The algorithm, which is designed to build optimal service topologies in a swirl, is named Swirly after this concept. There are two types of nodes widely used throughout the paper. Edge nodes are devices at the network edge, which act as consumers of fog services. Fog nodes, which are located in the fog, provide services for edge nodes as determined by Swirly. While a service topology refers to the output of the algorithm, in which fog nodes are assigned to provide services for each edge node, the physical layout of the swirl is referred to as the node topology.

Fig. 2 illustrates how Swirly builds a service topology from a small collection of edge nodes and fog nodes. In the first step (Fig. 2a), all fog nodes are idle and six edge nodes are in need of service providers. Assuming a maximum distance of 100 units between edge nodes and fog nodes, Fig. 2b shows that the algorithm determines that two fog nodes should be activated to accommodate all edge nodes. Most of the edge nodes are serviced by the fog node closest to them, but for the edge nodes closest to the inactive fog node, the numbers show that they can be sufficiently serviced by a non-optimal fog node. However, these fog nodes do not have infinite capacity, and at some point the third fog node will be activated as in Fig. 2c.

Fig. 3 shows the result of Swirly on a large scale. Edge nodes have been colored according to the fog node which acts as their service provider, while fog nodes themselves are shown as red dots (inactive) or green dots (active).

When Swirly is started, it has a collection of fog nodes and their available resources. No further information is needed, apart from an IP address or another effective method of reaching them.

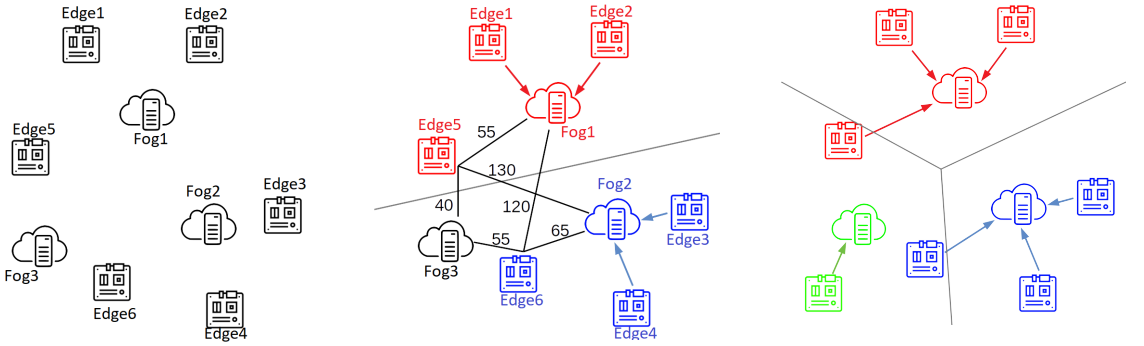
3.1 Adding Edge Nodes

When adding an edge node, the algorithm first examines the closest active fog node to the edge node. If that fog node has any spare capacity, it is assigned as service provider for that edge node. However, if there is no active fog node yet, or there is no fog node with spare capacity, or all active fog nodes are beyond the maximum distance, then the fog node closest to the edge node is activated and assigned as service provider for the edge node. In the case that even the closest fog node is more than the maximum distance away from the edge node, it is still assigned as service provider. Using this simple approach, ***Req. 3*** and ***Req. 4*** are satisfied because new service instances are deployed only if there is no other fog node acceptably close or available.

3.2 Edge Node Updates

To fulfill ***Req. 2***, Swirly must support changing topologies, so edge nodes will have to periodically report their metric distances to fog nodes to Swirly. Some suggestions for distance metrics are discussed in subsection 3.5. If Swirly receives an update from an unknown edge node, it will add it as discussed in the previous subsection. For other updates, update and remove operations are required. For all operations, it is important to note that the distances reported by edge nodes are pre-sorted by increasing distance, so the closest fog node is found in constant time.

The remove operation starts by unassigning an edge node from its fog node and removing it from the swirl. It then checks if the fog node is underutilized. If so, it attempts to evacuate all remaining edge nodes serviced by that fog node to other nearby fog nodes. This process will fail if any edge node is assigned to a fog node beyond the maximum distance, in which case the evacuation is reverted. If successful, the orig-



(a) Uninitialized service topology.

(b) All edge nodes assigned to service providing fog nodes. Numbers represent ping times from specific edge nodes to fog nodes.

(c) An additional fog node is activated because of resource limitations.

Figure 2: Different stages of building a service topology with Swirly, assuming a maximum distance of 100 to service providers.

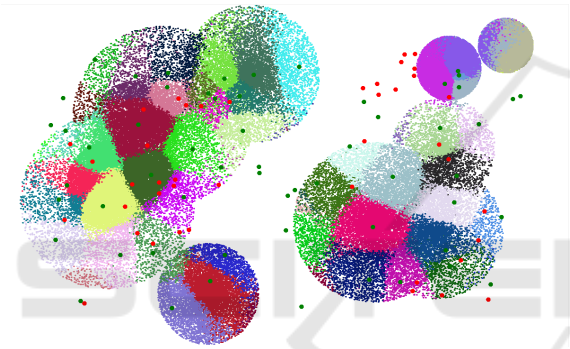


Figure 3: Visualization of a service topology generated by Swirly. Big red dots are inactive fog nodes, big green dots are active fog nodes servicing nearby edge nodes.

inal fog node is removed from the service topology. While this process may increase overall distance between fog nodes and edge nodes, it is assumed that any distance below the maximum distance is equally acceptable.

The evacuation process ensures that remove operations produce the same service topologies as if the exact subset of edge nodes that absolutely require a specific fog node had never been in the topology, keeping it consistent with the add operation. Because the minimum and maximum resource limits are configurable, this allows Swirly to optimize the number of active fog nodes, and thus minimize total resources, while keeping it from overloading any single node.

The update operation updates the fog node distances for a specific fog node. Should the distance from an edge node to its current service provider suddenly increase beyond the maximum distance, the algorithm will remove it from the service topology and add it again in an attempt to assign a closer fog node.

Note that a good distance metric combined with swirl updates not only enables Swirly to act on topological changes, but also to avoid or partially evacuate fog nodes experiencing load spikes and network problems.

3.3 Fog Node Updates

Fog nodes should also send Swirly periodic updates containing their free resources, but this only changes their availability for further edge node assignments. Reassigning edge nodes to another service provider when resources on a fog node run out is not currently implemented. While periodic resource updates enable Swirly to detect new fog nodes in the swirl, fog node dropout is not automatically detected and the remove operation for fog nodes has to be explicitly called. This could be further extended with a heartbeat mechanism or by having fog nodes themselves call the remove operation.

3.4 Request Redirection

Redirecting service requests from edge nodes to the correct service providers is outside the scope of this paper. However, because the entire service topology is known to Swirly, it should not be overly difficult to propagate changes to a (distributed) DNS system, or any other facility that handles request redirection.

3.5 Distance Metric

Swirly is not meant to directly tackle QoS and load balancing issues. Instead, it relies on a generic metric which indicates the ability of a fog node to support service requests from a specific edge node. A good

metric can improve the efficiency of Swirly and its responsiveness to changes by incorporating various factors that indicate network and node performance, but this must be balanced against computational and networking overhead. This section discusses some simple metrics that can be used to evaluate Swirly.

Calculating distances between nodes using geometric coordinates is reliable, accurate and generally requires little processing time. Additionally, geographical coordinates can track moving objects effectively and the distances can be calculated by Swirly itself, with a minimal network overhead for coordinate updates. However, this metric does not allow for any significant changes based on node or network status.

Using ping times has some notable advantages in that they implicitly contain an indication of network quality and physical distance. The value of this metric can be measured by using the ping command, although it is often blocked by routers and firewalls. Additionally, the overhead associated with this metric increases linearly with both the number edge nodes and fog nodes.

Because ping is a useful metric, these issues can be fixed by using a very lightweight web service on both edge nodes and fog nodes to determine the latency between software service endpoints. The disadvantages of this approach are that the packet sizes are larger than those of a simple ping, and that it requires slightly more processing time. However, it is unlikely to be blocked.

For the implementation in this paper, the last approach will be used. Every edge node will periodically determine its distance to each known fog node. The rest of this subsection aims to show that this does not result in an unacceptably high network overhead.

- The example assumes 200000 edge nodes, using 200 fog nodes as service providers
- Each edge node will attempt to determine its distance to fog nodes once every minute
- The size of a ping packet is 56 bytes on Unix
- wget shows that a suitable web service request is 159 bytes and a response is 202 bytes

Using these numbers, each fog node has to process about 3333 requests per second for a total of 4Mbps incoming and 5Mbps outgoing.

To avoid overloading nodes that are under heavy load and frequent pinging of distant nodes, the frequency can be reduced by an order of magnitude for fog nodes more than two times the maximum distance away. For large networks, this should reduce total traffic considerably. However, no concrete numbers for this can be determined since they are fully dependent on the network topology.

Using P as the measuring period in seconds and S as the message size in bytes (15 for IP address + 4 for an integer number), equation 1 gives an overhead of 98Mbps for the server hosting Swirly, which is significant but not insurmountable.

$$T = 8S \cdot \frac{|E| \cdot |F|}{P} \tag{1}$$

To reduce this overhead, a configuration option is included that keeps edge nodes from reporting fog node distances unless they have changed significantly or cross the maximum distance. For geographically widespread swirls, this measure is likely to reduce traffic by an order of magnitude, but concrete numbers can not be determined since they depend on the specific node topology of the swirl. Section 7 discusses further options to reduce network overhead.

Finally, it can be argued that this approach is acceptably resilient. The main risk is that the webservice for the ping mechanism stops working, but because of its simplicity this is very unlikely to happen unless the node it is running on goes down. As mentioned before, the ping frequency can be reduced for distant nodes to avoid flooding the network. Simple timeouts can be used to detect unavailable nodes, whether because they are offline, unreachable or overloaded.

3.6 Performance

Table 1 shows the computational complexity of the operations discussed in this section. In the case of the remove operation, most of its complexity comes from its reliance on the add operation. Similarly, the complexity of the update operation is a result of its use of the remove and add operations. However, the most common cases for all operations are $O(1)$, as will be shown in section 6.

The memory requirements of Swirly are easier to model. Because it has a list of edge nodes, each of which has a list of fog nodes sorted by distance, the total memory required is $O(EF)$.

4 KUBERNETES IMPLEMENTATION

While section 3 described how Swirly works, this section explains the specifics of implementing it in Kubernetes. The solution consists of three services; one in the cloud, one deployed on each fog node and one

deployed on each edge node. All services are configurable in terms of service locations, endpoint names, thresholds and polling times.

4.1 Swirl Service

The Swirl service keeps track of the node topology of the swirl and runs Swirly to determine a suitable service topology at any given time. Generally, this service will run in a pod in the cloud, but technically it can be run on any node, including in the fog. This service will not take any active steps to discover the node topology of the swirl. Instead, fog and edge nodes that call its service methods will be implicitly added to the swirl. To support removing nodes from the swirl, it subscribes to the Kubernetes API for node changes. This implementation does not assume that each edge node always requires services in the fog, because edge services could be deployed or removed on a specific edge node at any time. To determine which edge nodes require support services in the fog, the Kubernetes API is monitored for deployments of a specific pod. Only edge nodes with such a deployment are taken into account when generating the service topology. Using this approach, managing the node topology of the swirl is separated from support service monitoring and deployment. With minor changes, this enables the implementation to generate service topologies for multiple types of support services while using a single node topology, thereby using a minimal amount of memory. However, for the rest of this paper Swirly will be used to deploy a single type of support service on fog nodes.

By using the Kubernetes API to track node changes, the size of the swirl is subjected to the maximum limit of 5000 nodes (Kubernetes, 2019) in Kubernetes. Using a different approach to track node statuses and deployments would allow for swirls with up to 5000 fog nodes and hundreds of thousands of edge nodes. As in section 3.3, this could be achieved by using heartbeat mechanisms and having nodes explicitly call add and remove methods.

This service exposes the following methods:

- getFogNodeIPs: called by edge nodes when they are initialized, it returns the list of known fog node IP addresses
- updateFogNodePings: periodically called by edge nodes when they measure new or significantly different distances to fog nodes
- updateFogNodeResources: periodically called by fog nodes to update their free resources in the Swirly algorithm

Table 1: Summary of algorithm operation complexity. Most common cases are marked in bold.

	Best	Worst
Add	**O(1)**	$O(1/(1-\|E\|/\|F\|))$
Remove	**O(1)**	$O(\|F\|/(1-\|E\|/\|F\|))$
Update	**O(1)**	$O(\|F\|/(1-\|E\|/\|F\|))$

4.2 FogNode Service

This service runs on each fog node and is deployed as a daemonset in Kubernetes. It periodically measures the free and total resources of its node and reports them to the Swirl service. It also exposes the ping method, which is used by the EdgeNode service to determine the distance between edge and fog nodes.

4.3 EdgeNode Service

The EdgeNode service runs on each edge node, and for the purposes in this paper is also deployed as a daemonset in Kubernetes. On initialization, it fetches the known list of fog nodes from the Swirl service. It then periodically measures its distance to all fog nodes in the list by calling their Ping method and sends the results to the Swirl service if required. Finally, it exposes the setFogNodes method, which allows the Swirl service to update the list of fog nodes when changed.

5 METHODOLOGY

To verify that Swirly fulfills ***Req. 1***, its processing speed and its memory requirements are evaluated. Additionally, to show that it generates suitable service topologies, the output service topology for a small scale swirl is examined. In this chapter, the hardware and setup for the evaluations are discussed.

5.1 Node Processing and Memory

For the node processing and memory requirements evaluations, Swirly is isolated from the Kubernetes API so its stand-alone performance can be measured. It is run on a single server on the IDlab Virtual Wall (imec, 2019), which has 48GiB RAM and a Xeon E5-2650 CPU at 2.6GHz. In these evaluations, the algorithm is run on swirls ranging from 50.000 to 400.000 edge nodes, in steps of 50.000, while the number of fog nodes varies from 50 to 550 in steps of 50. In some cases, the results will start at a higher number of fog nodes due to resource constraints. For example, resource limits are configured so that 300.000 edge

nodes require 400 fog nodes. For each combination of edge nodes and fog nodes, 20 random swirls are generated for Swirly to process. This ensures that a good variety of swirls is generated so the entire performance range of the algorithm can be evaluated.

To generate the large scale swirls required by the evaluations, a topology generator is added to the solution which generates edge nodes and fog nodes randomly within an area of 1200 by 800 units. In order to simulate populated areas, edge nodes are generated in circles of various sizes which can overlap and whose density is highest in their centers. Fog nodes, on the other hand, are generated randomly over the entire area. Fig. 3 shows a visualization of a topology generated by a .NET implementation of Swirly equivalent to the Kubernetes implementation.

Latency is chosen as a distance metric, and it is defined so that one unit equals 1ms. However, to simulate the fuzziness of latency, it is randomized between 80% and 120% of its distance value. The maximum distance between edge nodes and their service providing fog nodes is set at 100ms. Because it is possible that edge nodes are generated which do not have a fog node within maximum distance, the evaluation results focus on average distance.

The implementation of Swirly and the evaluation code are made available on Github[1].

To measure how long it takes Swirly to add edge nodes, the evaluations measures the time it takes to build an entire service topology from scratch. This number is then normalized to the time it would take to add 10.000 edge nodes to a topology of that size. For the delete operation, it is measured how long it takes to delete 10.000 edge nodes from a finished service topology. The performance of the update method is not measured, because is entirely dependent on how latencies fluctuate in a given swirl, which are unlikely to be simulated realistically.

Memory consumption is read from /proc/<pid>/statm every time Swirly finishes building a service topology from a swirl. It is then printed to stdout, where it is collected by a batch script for further processing.

5.2 Generated Service Topology

To show that the Kubernetes implementation of Swirly generates appropriate service topologies, a small-scale swirl is manually set up on the IDlab Virtual Wall with a Kubernetes master, 3 fog nodes and 6 worker nodes as shown in Fig. 2a. The services described in section 4 are deployed on these nodes and Swirly builds a service topology from the information reported by the fog and edge nodes. For the purposes of the test, the code is slightly modified so that the fog node ping method sleeps for a predetermined amount of time depending on the edge node calling it, to simulate various hardcoded distances.

[1]The code will be made available upon acceptance and once an appropriate open source license model is selected. For review purposes code can be requested from the main contact author.

6 EVALUATION

6.1 Edge Node Processing

Fig. 4 shows the time required to add 10000 edge nodes to service topologies of various sizes. As predicted, the time required to add edge nodes increases as the number of edge nodes increases, only to fall again as more fog nodes are made available. Eventually it levels off at a constant value for each series, which increases sublinearly with the amount of edge nodes in the service topology.

While these results mostly agree with the computation complexity in table 1, the last effect merits some explanation. The predicted performance does not take into account some properties of the swirl such as edge and fog node densities. The observed effect can be explained through edge node density. Since the physical size of the swirl stays the same, edge node density increases along with edge node count, and fog nodes will eventually run out of servicing capacity. When that happens, they can not service all edge nodes within their maximum distance, so some edge nodes need to be assigned to suboptimal fog nodes. In worst cases some edge nodes can not be serviced at all, although this is avoided by the conditions of the evaluation. On the other hand, this means that performance would likely be constant if the physical size of the swirl expands along with the number of nodes. To avoid the effect in areas with dense populations of edge devices, it suffices to add more fog nodes and lower the maximum distance slightly.

Finally, the whiskers indicate that depending on the swirl, the time required to build a service topology can vary from 50% to 300% of the average, but it stabilises as the number of fog nodes increases.

The time required to remove 10.000 edge nodes from a service topology is shown in Fig. 5. In this case, performance is almost ideal, increasing slowly with the number of edge nodes and decreasing slightly with the number of fog nodes. The results indicate that the worst case performance of the delete operation is rarely triggered and does not overly affect performance.

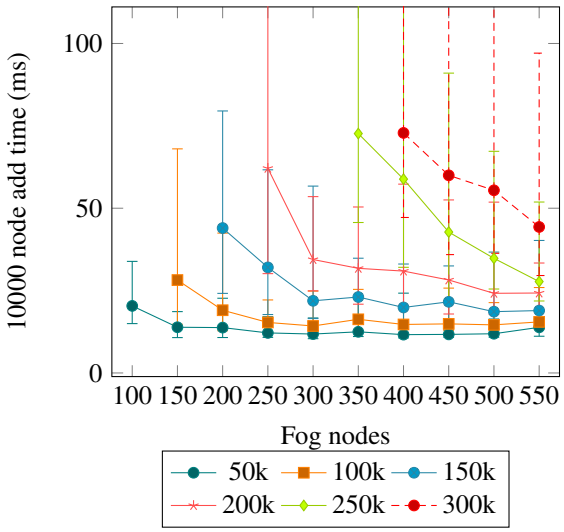


Figure 4: Time required to add 10000 edge nodes to service topologies of varying sizes.

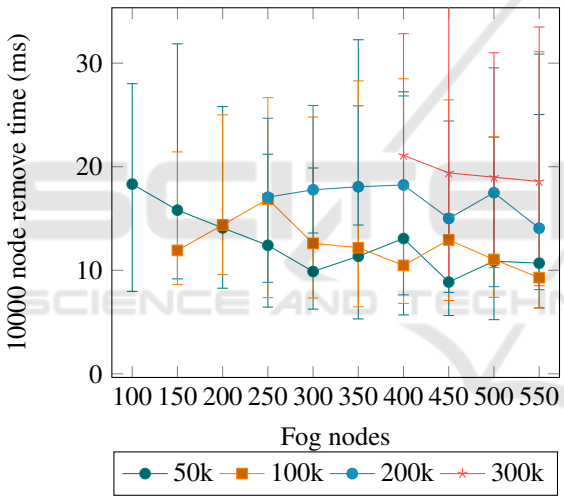


Figure 5: Time required to remove 10000 edge nodes from service topologies of varying sizes.

However, as with the add operation, the whiskers show that performance varies significantly, from 50% to around 200% of the average.

6.2 Memory

The memory requirements of Swirly are shown in Fig. 6. An important observation is that memory use appears to jump in distinct increments, always doubling at the same number of fog nodes independent of edge nodes. However, this is specific to the Golang implementation. Because Swirly keeps a sorted list of fog node pings for each edge node, these lists all double in size at the exact same moment, causing the jumps in the chart. Other than the observed jumps, memory

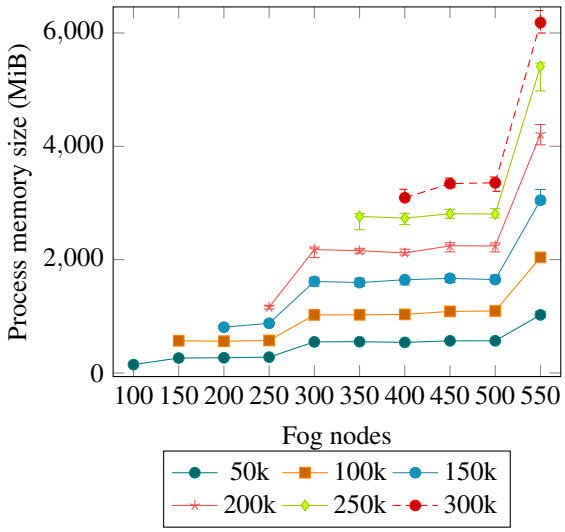


Figure 6: Memory required for swirls of varying sizes.

use correlates perfectly with the predicted $O(EF)$ requirement, unaffected by the randomness of the generated swirls.

6.3 Generated Service Topology

Table 2 shows the distances between fog nodes and edge nodes for the small-scale evaluation topology from Fig. 2a. Swirly has two choices for using only two out of three fog nodes for this swirl; either Fog1 and Fog2, or Fog2 and Fog3, with the first combination having slightly lower overall distances. Any solution which activates all three fog nodes is unacceptable for this node topology.

Fig. 2b shows the actual service topology generated by Swirly in Kubernetes, which is the most efficient one where Fog1 and Fog2 are activated. The numbers in this figure indicate distances between fog and edge nodes. These distances indicate that rather than activating Fog3, Swirly assigns the remaining edge nodes to already active nearby fog nodes, balancing a slight advantage in average distance against number of service instances as per ***Req. 3*** and ***Req. 4***. It is unknown if random factors could activate Fog3 rather than Fog1, but repeated iterations seem to indicate not. While generating the service topology, Swirly deploys a fog service to the correct fog nodes through the Kubernetes API, while Fog3 was left inactive.

Table 2: Distances between fog nodes and edge nodes for the small-scale evaluation topology.

	Edge1	Edge2	Edge3	Edge4	Edge5	Edge6
Fog1	30	50	110	130	55	120
Fog2	140	120	20	60	130	65
Fog3	95	85	120	110	40	55

7 DISCUSSION AND FUTURE WORK

As shown in the results, the algorithm scales very well in terms of processing time, but its memory requirements can quickly grow beyond a single server. There is no easy solution to further reduce memory use, other than partitioning fog and edge clouds over several machines. Despite this, the trends in the results suggest that a single Swirly service should be able to organize fog support services for up to 300.000 edge nodes and 550 fog nodes. For contemporary servers with 64GiB RAM, the maximum number of nodes can be extrapolated to about 1.000.000 edge nodes and 1.500 fog nodes. Because the algorithm is not multi-threaded, it may be useful to run several instances on a single machine, each of which organizes a specific region in the fog and edge. Dividing into n regions would also alleviate memory pressure by a factor of n, since

$$nO(\frac{E}{n}\frac{F}{n}) = \frac{1}{n}O(EF) \quad (2)$$

where the left side represents memory pressure with n regions, which is $1/n$ the original requirement. This would allow a further increase in fog and edge nodes by $\sqrt{n}$ each.

In section 3.5, the bandwidth requirements are calculated of the simple distance metric used in this paper. Despite suggested mitigating actions, the required bandwidth could grow to unsustainable levels for extremely large swirls with millions of devices. As with memory use, there is no easy solution to this other than to partition the swirl by region, which may result in worse performance at partition borders.

The current implementation of Swirly deploys services to the fog using Kubernetes, but in order to redirect service requests to the correct fog nodes, it should interact with distributed DNS plugins deployed on the cluster, override them, or deploy a separate system. Additionally, the implementation currently only supports one fog service, but could easily be modified to support any number of services to monitor and deploy.

While fog node updates are fully supported in Swirly, their impact is minimal. Fog nodes can be added at any time and their free resources can change, these events do not directly influence the service topology. Rather, Swirly only takes them into account when processing the next edge node. Ideally, the algorithm should examine if any edge nodes should be reassigned if a fog node is changed.

In terms of memory use and bandwidth requirements, it is better to switch to a fully distributed approach, in which the cloud algorithm is eliminated and each edge node becomes responsible for finding its own optimal service provider.

8 CONCLUSIONS

The introduction presents four requirements for a useful large-scale fog service scheduler. It should work with fog networks containing hundreds of thousands of devices, while being able to handle changing network topologies. It should also take node resource limits and distance metrics between nodes into account. Finally, it must minimize the number of fog service deployments required to service a set of edge nodes.

Swirly is proposed as a service deployment scheduler, and section 3 shows how it fulfills the requirements by design. Several node distance metrics are discussed, and a simple but reliable metric is chosen for the Kubernetes implementation. To verify the Kubernetes implementation of Swirly, it is evaluated in terms of memory use and node processing speed, and its output validated using a small, purpose-built topology.

The results mostly adhere to the computational complexity, but the algorithm slows down sublinearly as the density of edge nodes increases. This leads to the prediction that for service topologies that grow in physical size rather than density, Swirly will require constant processing time. When edge node density increases, fog node density and algorithm parameters will also need to change.

Solutions based on heuristics (e.g. genetic algorithms) will likely generate better solutions, but they are not suitable for real-time updates in large topologies, and they will require more time to generate ideal solutions.

Finally, some topics for future work are discussed, including DNS support, reducing network overhead, better metrics and governing multiple types of services simultaneously. However, a distributed approach is likely to solve the most important problems concerning Swirly.

ACKNOWLEDGEMENTS

The research in this paper has been funded by Vlaio by means of the FLEXNET research project.

REFERENCES

Aazam, M. and Huh, E.-N. (2015a). Dynamic resource provisioning through fog micro datacenter. In *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. IEEE.

Aazam, M. and Huh, E.-N. (2015b). Fog computing micro datacenter based dynamic resource estimation and pricing model for IoT. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE.

Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12*. ACM Press.

Canali, C. and Lancellotti, R. (2019). A fog computing service placement for smart cities based on genetic algorithms. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications.

El Houssine Bourhim, H. E. and Dieye, M. (2019). Intercontainer communication aware containerplacement in fog computing. In *15th International Conference on Network and Service Management, CNSM 2019, IFIP Open Digital Library, IEEE Xplore, ISBN: 978-3-903176-24-9*.

Farzin Zaker, M. L. and Shtern, M. (2019). Look ahead distributed planning for application management in cloud. In *15th International Conference on Network and Service Management, CNSM 2019, IFIP Open Digital Library, IEEE Xplore, ISBN: 978-3-903176-24-9*.

imec (2019). imec virtual wall: https://www.ugent.be/ea/idlab/en/research/research-infrastructure/virtual-wall.htm.

Kubernetes (2019). Kubernetes - building large clusters: https://kubernetes.io/docs/setup/cluster-large/.

Kumar, K. and Lu, Y.-H. (2010). Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56.

Latre, S., Leroux, P., Coenen, T., Braem, B., Ballon, P., and Demeester, P. (2016). City of things: An integrated and multi-technology testbed for IoT smart city experiments. In *2016 IEEE International Smart Cities Conference (ISC2)*. IEEE.

Mach, P. and Becvar, Z. (2017). Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656.

Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. (2013). Unikernels. *ACM SIGPLAN Notices*, 48(4):461.

Morshed, A., Jayaraman, P. P., Sellis, T., Georgakopoulos, D., Villari, M., and Ranjan, R. (2017). Deep osmosis: Holistic distributed deep learning in osmotic computing. *IEEE Cloud Computing*, 4(6):22–32.

Oppenheimer, D., Chun, B., Patterson, D., Snoeren, A. C., and Vahdat, A. (2005). Service placement in shared wide-area platforms. In *Proceedings of the twentieth ACM symposium on Operating systems principles - SOSP '05*. ACM Press.

Santoro, D., Zozin, D., Pizzolli, D., Pellegrini, F. D., and Cretti, S. (2017). Foggy: A platform for workload orchestration in a fog computing environment. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE.

Santos, J., Wauters, T., Volckaert, B., and Turck, F. D. (2019). Towards network-aware resource provisioning in kubernetes for fog computing applications. In *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE.

Spicer, Z., Goodman, N., and Olmstead, N. (2019). The frontier of digital opportunity: Smart city implementation in small, rural and remote communities in canada. *Urban Studies*, page 004209801986366.

Villari, M., Fazio, M., Dustdar, S., Rana, O., and Ranjan, R. (2016). Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83.

Zhang, Q., Zhu, Q., Zhani, M. F., and Boutaba, R. (2012). Dynamic service placement in geographically distributed clouds. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*. IEEE.