Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

Near real-time optimization of fog service placement for responsive edge computing

Tom Goethals*, Filip De Turck and Bruno Volckaert

Abstract

In recent years, computing workloads have shifted from the cloud to the fog, and IoT devices are becoming powerful enough to run containerized services. While the combination of IoT devices and fog computing has many advantages, such as increased efficiency, reduced network traffic and better end user experience, the scale and volatility of the fog and edge also present new problems for service deployment scheduling.

Fog and edge networks contain orders of magnitude more devices than cloud data centers, and they are often less stable and slower. Additionally, frequent changes in network topology and the number of connected devices are the norm in edge networks, rather than the exception as in cloud data centers.

This article presents a service scheduling algorithm, labeled "Swirly", for fog and edge networks containing hundreds of thousands of devices, which is capable of incorporating changes in network conditions and connected devices. The theoretical performance is explored, and a model of the behaviour and limits of fog nodes is constructed. An evaluation of Swirly is performed, showing that it is capable of managing service meshes for at least 300.000 devices in near real-time.

Keywords: Fog computing, Fog networks, Edge networks, Service mesh, Service scheduling, Edge computing

Introduction

In recent years, the rise of technologies such as containers [1] and more recently unikernels [2] has triggered a wave of research into edge and cloud offloading. This has resulted in a move from purely cloud-centered service deployments to fog computing and edge computing [3, 4], in which services are deployed close to their consumers instead of in monolithic data centers.

Simultaneously, there has been a move to deploy containerized services, which in turn often rely on cloud services, ever closer to consumers such a IoT devices. Between several initiatives for smart cities [5, 6] and a rapidly increasing variety of IoT devices, this ensures a continuing growth of cloud-connected devices.

*Correspondence: togoetha.goethals@ugent.be
Ghent University - imec, IDLab, Department of Information Technology, Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium

While the combination of IoT and fog computing offers a wide array of advantages, such as improvements in efficiency and user experience, it also exacerbates some of the service deployment scheduling challenges already present in the cloud, such as taking network bandwidth, network reliability and distances between nodes into account.

Instead of being located in centralized data centers, the fog and edge are spread over a large physical area, containing hundreds of thousands of devices. This means that network grade and quality can vary by orders of magnitude, from DSL lines to fiber optics, while the distances involved result in much higher latencies between nodes than in cloud data centers. A widespread and heterogeneous network also results in a larger variety of network conditions and problems. Therefore, a scheduling solution should not only be able to handle changing network conditions, but also slow or broken lines of communication. The decentralized nature of the fog and the edge is also an important factor. In the cloud, a service can simply be scaled up if demand suddenly spikes. In the fog however, it is not always possible or useful to simply scale in place. In order to minimize access times for the edge and provide the right amount of capacity for each service, the entire fog topology must be taken into account. Because of this, any change in the fog topology can trigger migrations of or extra service instances, as can edge nodes coming online, going offline, or moving to a different location.

On the other hand, there are also some challenges that remain mostly unchanged from cloud deployments. A deployment scheduler still has to take into account the limited resources of the nodes it can deploy services on, whether those are hardware resources (CPU, memory, network saturation) or calculated load metrics. Furthermore, although the solution should strive for an optimal placement of service instances in the fog to minimize access times for consumers, it should do so efficiently by using a minimal number of service instances. To guarantee a certain level of responsiveness to consumers, one or more metrics and thresholds can be defined on fog nodes[1], for example latency, uptime, etc.

To summarize, the requirements for a good algorithm for fog service scheduling are:

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

This article proposes Swirly as a solution to these requirements. Swirly is an algorithm that runs in the cloud or fog, which plans fog service deployment with a minimal number of instances, while optimizing the distance to edge consumers according to any measurable metric. Furthermore, it can incorporate changes to the network and topology almost in real-time.

"Related work" section presents existing research related to optimizing service deployments. "Swirly" section explains how the proposed algorithm works and how to choose a good metric, while "Theoretical properties" section analyzes its theoretical performance and the shape of the resulting service topologies. In "Evaluation methodology" section, an evaluation setup and methodology are presented to verify various performance aspects of the algorithm. The results of the evaluations are presented and discussed in "Results" section, with suggestions for future work in "Future work" section. Finally, "Conclusion" section gives a short overview of the goals stated in this introduction, and how the algorithm and its properties meet them.

Related work

Shifting workloads between the cloud and edge hardware has been extensively researched, with studies on edge offloading [7], cloud offloading [8], and osmotic computing [9].

Many strategies exist for fog container deployment scheduling, ranging from simple but effective resource requests and grants [10], to using deep learning for allocation and real-time adjustments [11].

Initial research into fog computing and service scheduling dates from before the concept of the fog, for example Oppenheimer et al. [12], 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. [13] 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 [14], which is also extended to a complete framework for fog resource management [15].

In more recent research, Santos et al. [16] 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. [17] 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. [18] 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. This is different from the approach in this article, which does not consider network load by itself, and attempts to migrate service deployments to manage resources.

Finally, Bourhim et al. [19] propose a method of fog deployment planning that takes into account inter-container communication. Their goal is to optimize com-

[1]A fog node can be a single device or server, or a decentralized micro data center. Anything that allows the deployment of services outside the cloud proper.

munication latencies between fog-deployed containers, which is obtained through a genetic algorithm.

Most of these approaches are centered on the cloud or small scale fog networks, and use a large number of parameters to construct an optimal, but static solution. In some cases, they may also rely on historical data for training. The algorithm discussed in this article however aims to quickly construct solutions using an edge-centered approach, taking into account node resources and generic heuristic. The speed of the algorithm allows it to process node updates in near-realtime for fog and edge networks orders of magnitude larger than those commonly found in proofs of concept in related work. An additional benefit is that the heavy lifting of calculating the heuristic value is offloaded to edge devices, where it has far less impact due to being spread out. Finally, the solution is meant for dynamically evolving networks for which historical training data may be hard or impossible to acquire.

Swirly

Contrary to the fog deployment solutions discussed in the previous sections, Swirly works under the assumption that some fog services will be used by most, if not all, devices in the edge. This allows for a simple but flexible approach which is very suited to building large service topologies.

Throughout the remainder of this article, a fog network (including the edge) with frequent changes to its network and nodes will be referred to as a swirl. This term refers to the swirly motion which fog makes when it stirs and moves. Hence the name of the algorithm, Swirly, which attempts to build an optimal service topology in a swirl. Additionally, edge nodes are devices at the network edge which act as consumers of fog services, while fog nodes are service providers hosting fog services. Therefore, edge nodes are assigned fog nodes as service providers, or are serviced by fog nodes. Finally, a (service) topology refers to the result of the algorithm, in which all edge nodes are assigned a fog node. When referring to the physical layout of the input nodes, the term node topology is used. Table 1 defines all the symbols used in algorithms in this section.

Figure 1 illustrates how Swirly forms a service topology from a collection of edge nodes E and fog nodes F. The algorithm starts with a number of unassigned edge nodes. It then determines that these nodes are all within an acceptable distance of two fog nodes, which are initialized and used as service providers. The line indicates how the service topology is divided between these two nodes. As more edge nodes join the service topology, it becomes necessary to initialize the third fog node, further dividing the service topology. Finally, an edge node pops up which should be serviced by the green fog node, which is already full. Therefore, this last edge node is serviced by the blue fog node.

Table 1 Definitions of symbols used in algorithms 1, 2 and 3

Symbol	Definition
E	all edge nodes in the service topology
F	all fog nodes in the swirl
E_x	an edge node in the service topology
$\|E\|$	the number of items in E
F_x	a fog node in the service topology
$\|F\|$	the number of items in F
e	an edge node without a service provider
$F(e)$	fog nodes ordered by distance from e
$E(F_x)$	edge nodes serviced by F_x
A	set of active fog nodes
$D_{e,f}$	distance between edge node e and fog node f
D_e	distances of edge node e to all fog nodes
d	a distance according to the chosen metric
d_{max}	maximum distance between an edge node and fog node
$R_{x,f}$	current level of resource x on fog node f
$RL_{x,f}$	limit for resource x on fog node f
$LL_{x,f}$	lower limit for resource x on fog node f
RD_x	default resource x increase for a service client

Figure 2 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.

The rest of this section will describe how the described functionality is implemented by specific functions of Swirly. The add operation is meant only to assign a service provider to newly detected edge nodes, while the update method is used to receive updates from edge nodes and potentially assign them a different service provider. Finally, the delete method removes nodes from the topology altogether. Fog node add, update and delete operations are also discussed in the relevant sections.

Throughout all these operations a distance metric is required to determine which fog node is the best service provider for an edge node. The effects of the choice of a distance metric are discussed in "Impact of distance metric" section.

Adding new nodes

In order to build a service topology, all edge nodes E that require a certain service are added to the topology one by one as per Algorithm 1. Generally, the algorithm attempts to find the active fog node F_a closest to the given edge node e using the list of distances D_e. If successful, the fog

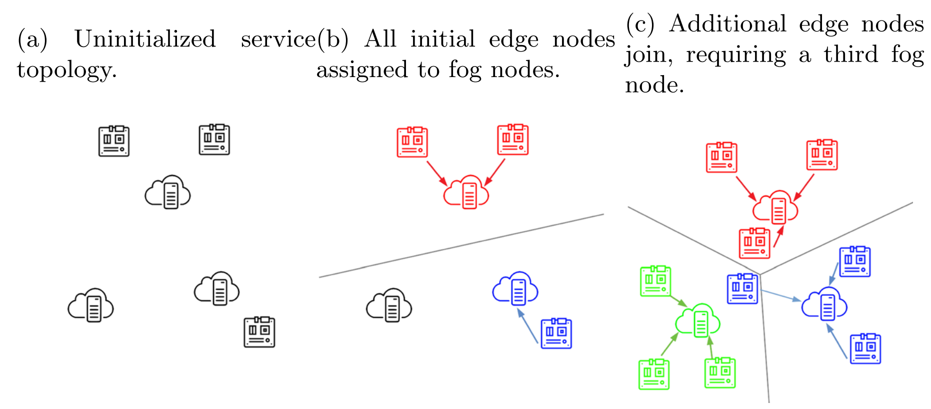
(a) Uninitialized service topology.
(b) All initial edge nodes assigned to fog nodes.
(c) Additional edge nodes join, requiring a third fog node.

Fig. 1 Different stages of building a service topology with Swirly

node F_a is assigned to the edge node as service provider. If there is no active fog node yet ($A = \emptyset$), or there is no fog node with free resources ($F_a = \emptyset$), Swirly finds the closest inactive fog node F_i instead. That fog node then gets activated and assigned as a service provider to the edge node *e*. There is a single caveat here; if the closest available fog node is beyond the maximum distance ($D_{e,F_a} > d_{max}$ and $F_i = F_a$), the algorithm has no choice but to assign it as service provider for an edge node. The support function ClosestFogNode returns the fog node F_c with free resources closest to an edge node *e*. A parameter *active* can be supplied to indicate if active only active fog nodes should be considered.

This operation ensures that ***Req. 3*** and ***Req. 4*** for a useful deployment scheduler are met within a reasonable amount of processing time.

Algorithm 1: Adding a single edge node to the service topology

```
function Add(e, distances, $d_{max}$) is
    if $A = \emptyset$ then
        $D_e$ = distances
        $F_i$ = ClosestFogNode($e$, false)
        initialize $F_i$
        assign $F_i$ to $e$
    else
        $F_a$ = ClosestFogNode($e$, true)
        if $F_a = null$ then
            $F_i$ = ClosestFogNode($e$, false)
            add $F_i$ to $A$
            initialize $F_i$
            add $e$ to $E(F_i)$
        else if $D_{e,F_a} > d_{max}$ then
            $F_i$ = ClosestFogNode($e$, false)
            if $F_i = F_a$ then
                add $e$ to $E(F_a)$
                $R_{x,a}$ += $RD_x$
            else
                initialize $F_i$
                add $e$ to $E(F_i)$
        else
            add $e$ to $E(F_a)$
            $R_{x,a}$ += $RD_x$

function ClosestFogNode(e, active) is
    $F_c$ = null
    for $F_x \in F(e)$ do
        if (!active or (active and $F_x \in A$)) and $R_{x,i} < RL_{x,i}, \forall i$ then
            $F_c = F_x$
    return $F_c$
```

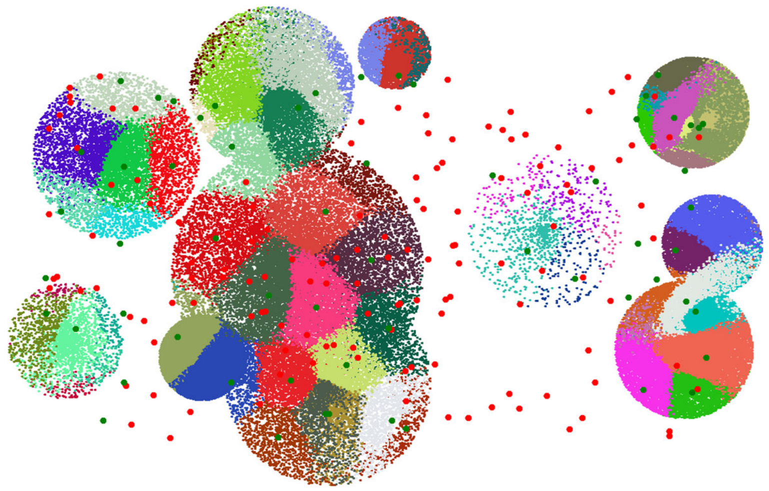

Fig. 2 Visualization of a service topology generated by Swirly. Big red dots are inactive fog nodes, big green dots are active fog nodes servicing nearby edge nodes

Note that during an add operation, the resource use $R_{x,i}$ of the selected fog node is increased by a configurable amount. This is to avoid assigning it to too many edge nodes simultaneously, and is periodically corrected by updates from the fog nodes containing their actual free resources.

Adding additional fog nodes to the topology is also supported, but this does not trigger a reorganization of currently assigned edge nodes. Rather, newly added fog nodes will only provide services to edges nodes that are added or updated after they were initialized.

Node updates

In order to fulfill ***Req. 2***, Swirly must support topology updates. As a requirement for these updates, all edge nodes must periodically report the distances between them and every fog node to Swirly. Exactly how they do this is left to implementation, although some suggestions and their impacts are given in the next subsection. For this subsection, it is important to note that these lists of distances to fog nodes are pre-sorted by increasing distance, so Swirly can always find the closest fog node in constant time. This is also the case for the ClosestFogNode function in Algorithm 1.

To keep a service topology up to date in a swirl, the algorithm needs operations to update edge nodes (Algorithm 2) and remove them from the service topology (Algorithm 3).

The Remove operation starts by removing the edge node e from its fog node F_e. After that, it checks if any of the resources of the fog node are below the lower limit, in which case it attempts to move all remaining edge nodes $E(F_e)$ it services to other nearby fog nodes by removing them from F_e and calling the Add operation. This process fails if any edge node E_x would be assigned a new fog node F_{alt} which is more than the maximum distance d_{max} away, unless that is already the case for the current fog node ($D_{E_x,F_{alt}} > d_{max}$). To support reverting this operation in case of failure, each reassignment is kept in the map *altNodes*. Upon failure, the algorithm iterates over each pair E_i, F_i in *altNodes*, removing E_i from F_i and reassigning it to F_e. On success, the fog node F_e is torn down and removed from the topology.

The Remove operation makes use of a support method ClosestFogNodeExcept, which is essentially the same as the ClosestFogNode method, but the fog node F_{except} can not be returned as a result.

Note that this process produces the same result as if the exact subset of nodes that absolutely required a specific fog node as service provider had never been in the original set of edge nodes, so it remains consistent with the Add operation.

The Update operation updates the set of *distances* of an edge node e to each fog node in F. In case the new distance d_{new} from the edge node to its service provider F_e increases beyond the maximum distance d_{max}, the algorithm calls the Remove and Add operations for e, in an attempt to assign it a better service provider.

Note that the total performance of the update method is dependent on how efficiently the set of distances can be updated. However, this can happen in constant time, which is implicitly assumed in "Processing" section.

The distance metric, combined with the Update and Remove operations not only enables Swirly to act on topological changes, but also to implicitly avoid fog nodes which are experiencing load spikes and network issues.

As with adding fog nodes, fog node updates only change the available resources for further edge node assignments. Removing a fog node will attempt to assign new fog nodes to the edge nodes that depend on it. Reassigning edge nodes if their service provider suddenly runs out of resources is not currently implemented. Instead, it can be argued that it is optimal to rebuild the entire service topology when fog nodes are added or their available resources change drastically, since these cases require examining every edge node to find its optimal service provider considering the new information. Therefore, no further implementation is needed.

Algorithm 2: Updating the status and distance metrics of an edge node

```
function Update(e, distances, $d_{max}$) is
    $d_{old} = D_{e,F_e}$
    $D_e$ = distances
    $d_{new} = D_{e,F_e}$
    if $d_{old} < d_{new}$ and $d_{new} > d_{max}$ then
        Remove(e, $d_{max}$)
        Add(e, $d_{max}$)
```

Algorithm 3: Removing a single edge node from the service topology

```
function Remove(e, $d_{max}$) is
    remove $e$ from $E(F_e)$
    if $R_{e,i} < LL_{e,i}, \exists i$ then
        altNodes = $\emptyset$
        revert = false
        while !revert and $E(F_e) != \emptyset$ do
            $E_x = E(F_e)[0]$
            $F_{alt}$ = ClosestFogNodeExcept($E_x$, $F_e$)
            if $F_{alt} = null$ then
                revert = true
            else
                if $D_{E_x,F_{alt}} > d_{max}$ then
                    revert = true
                else
                    remove $E_x$ from $E(F_e)$
                    Add($E_x$, $d_{max}$)
                    altNodes[$E_x$] = $F_{alt}$
            if revert then
                for $(E_i, F_i) \in altNodes$ do
                    remove $E_i$ from $E(F_i)$
                    add $E_i$ to $E(F_e)$
            else
                teardown $F_e$

function ClosestFogNodeExcept(e, active, $F_{except}$) is
    $F_c$ = null
    for $F_x \in F(e)$ do
        if $F_x \neq F_{except}$ and (!active or (active and $F_x \in A$)) and $R_{x,i} < RL_{x,i}, \forall i$ then
            $F_c = F_x$
    return $F_c$
```

Swirly can not directly detect fog node failures, so it can not actively react to service availability issues. However, its design allows for two methods to make it more resilient to hardware failures. The first is choosing a distance metric that can reflect imminent node failures, which passively forces the algorithm to choose more suitable fog nodes to host services on, as shown in "Impact of distance metric" section. The second option is to actively remove a fog node from the algorithm when its failure is detected by external components. While this method requires some extra computation, it can react to hardware failures in less than a second.

So far, this article has not touched on the actions required to redirect service requests from edge nodes to the correct fog nodes. While such functionality is beyond the scope of this article, the network addresses of all nodes are known, along with the topology generated by Swirly. Therefore, it should not be overly difficult to propagate changes to a DNS server, a distributed DNS plugin (e.g. for Kubernetes), or a webservice on edge nodes that redirects requests at the source.

Impact of distance metric

While the performance and inner logic of the algorithm are unaffected by the choice of distance metric between edge nodes and fog nodes, a good metric can improve efficiency and responsiveness to changes. On the other hand, some of the more useful metrics may cause a lot of processing and network overhead between the nodes and the algorithm. In this section, some ideas are discussed for useful metrics.

The simplest metric that can be used depends only on geographical coordinates. While it requires a reasonably accurate location for each node, it does not usually change unpredictably or rapidly. The downside of this metric is that any changes in network or fog node performance can not be detected in order to avoid availability problems. The advantage is that the algorithm can keep track of all node locations by receiving regular updates from edge nodes, and calculate distances as required. Thus, the network overhead will be minimal using this approach.

Another possible metric is the latency between edge nodes and fog nodes. The values of this metric can be easily measured using the ping command, but this results in an overhead that grows linearly with both edge nodes and fog nodes. Additionally, the ping command is often blocked on servers or routers, in which case it is useless. The biggest advantage of this metric is that it can detect network and fog node issues in real-time, so edge nodes can be assigned a different fog node as service provider.

A third metric, which also aims to determine network latency, uses 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, [20] shows that even a reasonably simple server can easily handle millions such requests per minute.

The last two metrics require that all edge nodes periodically determine their distance to each fog node, and report the results to the algorithm so it can adjust the service topology. In order to show that this does not result in an unacceptably high network overhead, the following numbers have been determined:

- 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 ping requests per second for a total of 1.5Mbps, both incoming and outgoing. In the case of a webservice, the traffic increases to 4Mbps incoming and 5Mbps outgoing.

Additionally, to avoid overloading nodes that are already under heavy load and to avoid frequent distance measuring to nodes that are too far away, the frequency can be reduced by an order of magnitude for fog nodes more than two or three times the maximum distance away. For larger 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.

Finally, a quick calculation can determine the network overhead for the server hosting Swirly using:

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

Where P is the measuring period in seconds and S is the message size in bytes (15 for IP address + 4 for an integer number). The result is 98Mbps, which is significant but not insurmountable. Some actions can be taken to reduce this number significantly, such as not reporting distances that have not changed by more than 30%, unless they cross the maximum distance. For geographically widespread topologies, this could likely reduce traffic by an order of magnitude or more, but again concrete numbers can not be determined as they rely on the specific network topology.

Theoretical properties

To fulfill part of ***Req. 1*** put forth in the introduction, this section discusses the theoretical properties of the algorithm. The processing power and memory requirements are analyzed in-depth, and for a full understanding of the output of the algorithm, a theoretical model for the resulting service topologies is constructed.

Processing

Adding an edge node to the topology can result in several cases. In all cases, the sorted list of fog node distances for the edge node is consulted to determine its optimal service provider at that time.

In the most common case, the selected fog node has enough free resources for additional service clients, and the edge node is simply directed to that fog node. The resulting performance is $O(1)$.

In the worst case, there is a chance that the optimal fog node is already full and the algorithm has to search for the next best fog node with free resources. Let there be a chance p that any node is already full, such that

$$p = c\frac{|E|}{|F|} = cL_F \tag{2}$$

where L_F is the average load of edge nodes assigned to fog nodes and c is a constant that normalizes L_F. If there are enough edge nodes E and fog nodes F, then the expected number of iterations to find a suitable fog node is

$$r = \frac{1}{1-p} = \frac{1}{1-c\frac{|E|}{|F|}} \tag{3}$$

and worst case performance is $O(1/(1-|E|/|F|))$, with a maximum of $O(|F|)$ because there can never be more iterations than there are fog nodes.

Note that both cases depend on the node topology of the swirl. When there are not enough fog nodes in areas with a high edge node density, the worst case performance will occur more often. The influence of the node topology on performance is further examined under "Generated topology" section.

Like the Add operation, deleting an edge node has several possible cases. In the best case, it is removed from its fog node for $O(1)$.

In some cases, the fog node will be underutilized after a remove, so the algorithm attempts to migrate the remaining edge nodes of that fog node to other fog nodes. Since the utilization threshold is independent of E and F, this operation is $O(1)$, albeit with an unusually large impact. This heavy operation is amortized over $O(|E|/|F|)$ removes and uses the add operation when moving edge nodes, resulting in a worst case performance of $O(F/(1-|E|/|F|))$.

Finally, the update operation can cause a different fog node being assigned to an edge node. When this happens, both a remove and add operation are executed. This can lead to the worst case scenario for both, in which case performance is $O(|F|/(1-|E|/|F|))$. Most updates however will only be a simple status update for $O(1)$.

Table 2 summarizes the performance for all operations.

Table 2 Summary of algorithm operation complexity. Most common cases are marked in bold

	Best	Worst
Add	$\boldsymbol{O(1)}$	$O(1/(1-\lvert E\rvert/\lvert F\rvert))$
Remove	$\boldsymbol{O(1)}$	$O(\lvert F\rvert/(1-\lvert E\rvert/\lvert F\rvert))$
Update	$\boldsymbol{O(1)}$	$O(\lvert F\rvert/(1-\lvert E\rvert/\lvert F\rvert))$

Memory

Swirly requires a number of maps and lists to support the processing performance described above, but the greatest impact on memory is that for each edge node, the algorithm must keep a sorted list of distances to each fog node in the topology. Therefore, the predicted memory requirement for the node hosting the algorithm is $O(|E| \cdot |F|)$. For edge nodes, the memory requirement is $O(F)$.

Generated topology

To verify that the algorithm can satisfy the requirements for a good fog service topology and to identify edge cases and possible problems, it is important to first construct a theoretical model of the expected output given the node topology of the swirl.

While fog and edge networks are intrinsically discrete, they can be described analytically if they are large and dense enough. As a first step, the densities of edge nodes and fog nodes in any network are

$$\rho_F = f(x,y), \rho_E = g(x,y) \tag{4}$$

where both f and g are functions that give the amount of nodes per surface area at x, y. While Cartesian coordinates are used here, it would be better to use densities in the coordinate system that describes distances in the chosen metric. However, the coordinate transformation may be unknown and impossible to construct, so Cartesian is used for illustrative purposes.

For the next step, the servicing area of every active fog node can be modeled using three different parameters:

- a_E, circumscribed by r_E, is the capacity area of a fog node, which determines how many edge nodes it can service based on its capacity C_e
- a_F, bounded by r_F, is the responsibility area of a fog node. All edge nodes within this area should be serviced by the fog node, since it is either the closest fog node or no other fog node can be used.
- a_p, bounded by r_p, is the proximity area of a fog node. All edge nodes within this area are close enough that they can be serviced by the fog node without going over the maximum metric value.

In some cases r_E, r_F and r_p will have only a single value, such as in Fig. 3a. In others, only their maximum value is relevant, such as in Fig. 3b where $r_F > r_E$. In these cases

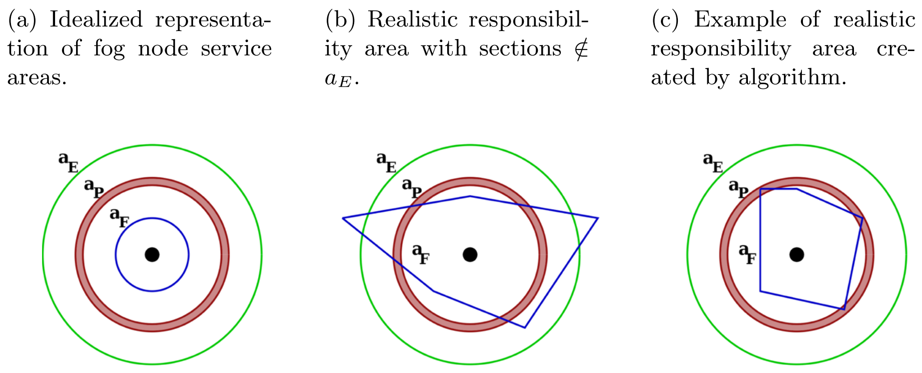
(a) Idealized representation of fog node service areas.
(b) Realistic responsibility area with sections $\notin a_E$.
(c) Example of realistic responsibility area created by algorithm.
a_E
a_P
a_F

Fig. 3 Graphic representation of idealized and realistic fog node service areas

they will be treated as scalars to simplify notation. Other cases will specifically show them as functions with their required parameters.

r_p can be determined using the following equation, which in most cases will resemble a circular shape:

$$r_p(x,y) = \sqrt{(x^2+y^2)}, \forall x,y : h(x,y) = M_m \tag{5}$$

or, for a more intuitive approach using polar coordinates, where r can be substituted for metric distances

$$r_p(\theta) = r, \forall r : h(r,\theta) = M_m \tag{6}$$

Where $h(x,y)$ gives the distance metric value at any given location x, y and M_m is the maximum distance value. Note that in both cases, points at various distances from the origin can be mapped onto the same metric distance, making a transform back to the original coordinate system impossible. In the case of certain basic distance metrics, in which metric distance between points divided by their geographical distance is more or less constant, the equation reduces to

$$r_p = C_p M_m \tag{7}$$

r_F and r_E can be naively described as circle radii using

$$r_F = \frac{1}{\sqrt{\pi\rho_F}}, r_E = \sqrt{\frac{C_e}{\pi\rho_E}} \tag{8}$$

However, the equation for r_E is not accurate when ρ_E varies a lot over its entire area. A more accurate solution would be solving the following equation for r to find r_E:

$$\int 2\pi r\rho_E(r)\cdot \mathrm{d}r = C_e \tag{9}$$

This formula takes into account that only r is known by measuring the metric distance between nodes and not θ, so only the integration over r is performed. It is worth noting that in many cases the density of edge nodes is relatively constant over most geographical areas that can be covered by a single (group of) fog node(s), although it may vary over time. Therefore Eq. 8 can be used if it is re-evaluated periodically.

Finally, r_F is unlikely to vary significantly at the scales used in this section. When plotted, these three radii resemble Fig. 3a. Note that r_p is quite fuzzy because as mentioned, $h(x,y)$ can map physically different points on significantly different values of the distance metric. Figure 3a represents the ideal case for the relative sizes of the radii. There are 5 other permutations, all of which can be problematic for the following reasons:

- If $r_F > r_E$, the fog node will not have enough capacity to handle its entire responsibility area. This means there are not enough active fog nodes, or they are not in the right places.
- If $r_F > r_P$, the fog node will have to support some nodes that fall outside its proximity area, so that some edge nodes will have a greater metric value than technically allowed. Again, this points to not enough active fog nodes or erroneously placed fog nodes.
- If $r_p > r_E$, the fog node has sufficient capacity to handle its responsibility area, but can not handle a changing topology where it has to start servicing extra edge nodes that fall between r_E and r_P. This means the fog nodes are not sufficiently powerful to support the maximum metric value.

Assuming a basic metric, two requirements for the proper formation of a fog service network can be constructed from these cases by substituting r_p, r_F and r_E with their definitions from Eqs. 7 and 8:

$$M_m^2\rho_F(x,y) \geq \frac{1}{\pi C_p^2} \tag{10}$$

$$C_e\rho_F(x,y) \geq \rho_E(x,y) \tag{11}$$

The left sides of these requirements represent the factors that can be easily controlled or tuned, while the right sides represent factors that are unpredictable or unavoidable. Since everything in Eq. 10 except $\rho_F(x, y)$ is constant under the assumptions, it can be used to determine a proper minimum value for ρ_F, with Eq. 11 indicating where more fog nodes or better hardware should be provided.

A final topic of discussion concerns the shape of a_F. In reality, r_F will not be a single number describing the radius of a circle. Because the algorithm attempts to assign edge nodes their closest fog node, the responsibility area of fog nodes will look like pieces of a Voronoi diagram. This effect is also visible in Figs. 1c and 2. As Fig. 3b shows, it is possible that the responsibility area a_F has sections that lie outside a_p or a_E. The algorithm only allows sections outside a_p when there is no closer fog node, and it never allows sections outside a_E, even if there is no alternative. In general, it will attempt to create responsibility areas as shown in Fig. 3c.

Note that the shapes of a_F depend entirely on the node topology of the swirl and the amount of required fog nodes to service all edge nodes, so any further discussion of the service topology is reserved for the results section.

Evaluation methodology

To fully verify that Swirly fulfills the ***Req. 1***, its performance must be evaluated. This chapter describes the physical hardware setup used to evaluate Swirly, as well as implementation details for the evaluated version. It also details how the theoretical model constructed in the previous section can be evaluated and validated, and how to determine the practical performance of the algorithm.

Swirly is evaluated on a single machine with 48GiB RAM and a Xeon E5-2650 CPU at 2.6GHz. In all cases, the algorithm is the only process running apart from the operating system. Each evaluation was run with 50000 to 400000 edge nodes in steps of 50000, and 50 to 550 fog nodes in steps of 50. However, because the algorithm considers resource limitations, some results series start at a higher number of fog nodes. For example, it is not possible to service 300000 edge nodes with less than 400 fog nodes. For every parameter set, 20 iterations of Swirly are run on a uniquely generated swirl. The maximum distance between edge nodes and fog nodes is set to 100. It is entirely possible that edge nodes are generated which do not have a fog node within the maximum distance, so the evaluations and results are entirely focused on average distance as an indicator.

Algorithm implementation

The topology visualization in Fig. 2 was made with a .NET implementation of Swirly. For the evaluations in this section, Swirly is implemented in Golang. Because the goal is to measure the impact of the algorithm itself, there are no integrations with any sort of DNS or service/container scheduling software.

Edge nodes and fog nodes are generated randomly over an area of 1200 by 800 "units". In order to simulate urbanized areas, edge nodes are generated in circles of varying sizes, which are slightly denser in the center. These circles may overlap and often form more complex shapes, as in Fig. 2. Fog nodes are generated without regard for edge node density, to evaluate the ability of Swirly to pick exactly the right nodes to service any area. The chosen distance metric is latency. For simplicity, latency is defined so that one unit generally equals 1ms. However, because latency is inherently fuzzy, this distance is randomized between 80% and 120% of the unit distance.

Because the sets of generated nodes are completely randomized, some cases will work well with Swirly and others will be adversarial. A range of possibilities is explored and discussed in "Results" section.

The Golang implementation of Swirly and the evaluation code are made available on Github[2].

Processing time

To accurately measure the exact time it takes Swirly to perform operations, a swirl is generated up front and the Golang time library is used to determine how long an operation using that swirl takes. For the add operation, the evaluation measures how long it takes to build an entire service topology from scratch, which is then averaged to the time it would take to add 10000 nodes. For the remove operation, it is simply measured how long it takes to remove 10000 nodes from a completed service topology. The performance of the update operation is measured similarly to the delete operation. 10000 nodes are physically moved far away from their original fog node and it is measured how long it takes the algorithm to assign them another one.

Memory requirement

The evaluation of memory requirement starts with having Swirly build a service topology from a swirl with a certain amount of nodes. Memory consumption is then read from /proc/< *pid* >/statm and printed to stdout, where it is collected by a batch script.

Topology efficiency

Throughout the processing evaluations, some statistics are kept on how many fog nodes are required to build any service topology and what the minimum, average and maximum distances between edge nodes and fog nodes are. These statistics are used to attempt to determine how close Swirly comes to constructing a perfect service topology. Using edge node to fog node distances, Swirly

[2] https://github.com/togoetha/swirly

is compared to a random selection of fog nodes (e.g. the default Kubernetes scheduler [21, 22]), and to the best possible theoretical solution if fog node resources are infinite. Additionally, the amount of fog nodes used by Swirly is compared to an optimal solution where each fog node is exactly at maximum capacity, although admittedly this optimal solution would not have any space left for extra nodes or unexpected load. Finally, two extreme topology types are compared to see how Swirly reacts to certain geographical features. The first is a perfectly equal distribution of edge nodes, while in the second the edge nodes are split into 4 circular, non-overlapping clusters, one in each corner of the topology.

Results

This section contains the results for the evaluations described in "Evaluation methodology" section, along with a discussion of the results. Most charts have whiskers to indicate extreme values, but in some cases they have been cut off to keep the charts readable.

Processing

Figure 4 shows the average time required to add 10000 nodes of swirls of various sizes to a service topology. The results mostly adhere to the computational complexity calculated in "Processing" section. As the number of edge nodes increases the operation gets slower, and it gets faster again with more fog nodes available, eventually leveling out at a constant time per edge node. However, for higher numbers of edge nodes the constant time never quite reaches that of smaller topologies, putting real performance somewhere between the best and worst cases, increasing sublinearly with the amount of edge nodes.

It should be noted that ρ_E is an important factor here; if the physical size of the swirl were to expand with the number of edge nodes, the time required would remain almost constant. In terms of the model constructed in "Generated topology" section, this is because r_E shrinks as edge node density increases but r_p and a_F remain the same, so eventually r_E will become smaller than r_p and a_F. This means the fog nodes no longer have enough capacity to handle their service areas properly, so the algorithm requires ever more time to find a suitable fog node.

The result of this is that Swirly is suitable for constructing large service topologies, but it requires a large amount of high capacity fog nodes in densely populated areas to keep performance up.

Finally, the whiskers indicate that depending on the node topology of the swirl, the time required to add edge nodes can vary from 50% to 300% of the average with a high $|E|/|F|$ ratio, but it stabilizes as the number of fog nodes increases.

Figure 5 shows the time required to remove 10000 edge nodes from service topologies of various sizes. The performance of this operation is almost ideal, showing a slight decrease with the number of fog nodes and only

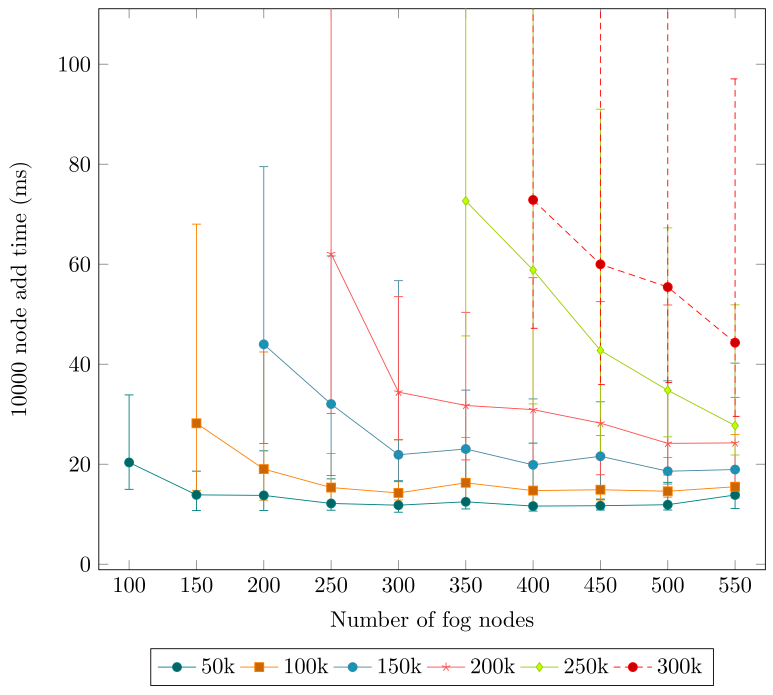

Fig. 4 Time required to add 10000 edge nodes to service topologies of varying sizes. Legend numbers represent thousands of edge nodes

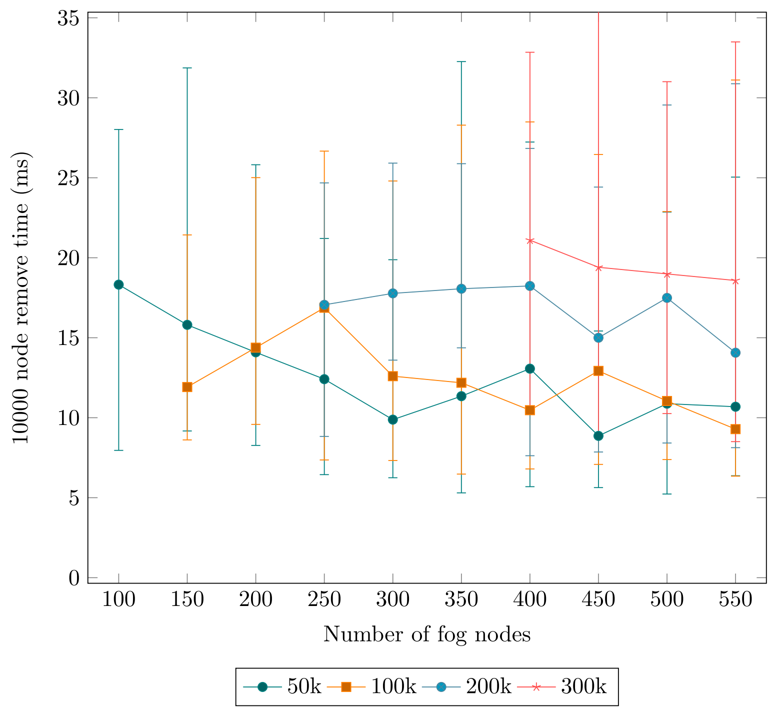

Fig. 5 Time required to remove 10000 edge nodes from service topologies of varying sizes. Legend numbers represent thousands of edge nodes

marginally increasing with the number of edge nodes. Since the edge node density effect is also present here, the results imply that individual remove operations only rarely trigger a worst case performance scenario.

As with the add operation, performance can vary wildly from around 50% to 200% of the average.

The performance of the update operation is shown in Fig. 6. Again, this mostly adheres with the theoretical performance, which is the sum of the delete and add operations. The relatively constant performance of the delete operation reduces the curves of the add operation, resulting in a mostly constant time to update edge nodes. However, the performance features of the add operation still apply, as performance decreases slightly with edge node density, and adding more fog nodes improves performance up to a certain point. Note that the update method is forced into worst-case behaviour for this evaluation; every edge node is moved to another fog node, whereas in reality this will not always be the case and performance will be more constant. Finally, the numbers of Fig. 6 are not exactly the sum of the add and delete operations, but this is due to certain effects of the evaluation code which can not be subtracted from the measured time.

Predicting a maximum number of devices from these numbers is difficult, since the amount of required node updates in a real-life topology depends on many factors, for example the volatility of the clients, choice of distance metric and update period. Under the extreme conditions that an update is sent by every node every second, and extrapolating from the results, a maximum number of nodes around 200.000 to 300.000 edge nodes can be supported when running single-threaded on the test hardware.

Memory

The memory requirements of Swirly are shown in Fig. 7 for swirls of varying sizes. The first observation here is that memory use jumps up in distinct steps here. This is caused by the specific implementation of the algorithm in Golang. Golang arrays and maps double in size each time they reach their full capacity, and for each edge node the algorithm keeps a list of distances to fog nodes. Therefore, all these maps double in size at the exact same time, causing the jumps in the chart.

Apart from this peculiar effect, memory use adheres perfectly to the theoretical predictions, and despite the randomly generated swirls there is almost no difference in memory use between iterations.

Considering these results, the product of the number of edge and fog nodes should remain below 1.500.000.000 on a common cloud server with 64GiB memory. For example, 1.000.000 edge nodes can be assigned to 1.500 fog nodes, or 3.000.000 edge nodes to 500 fog nodes.

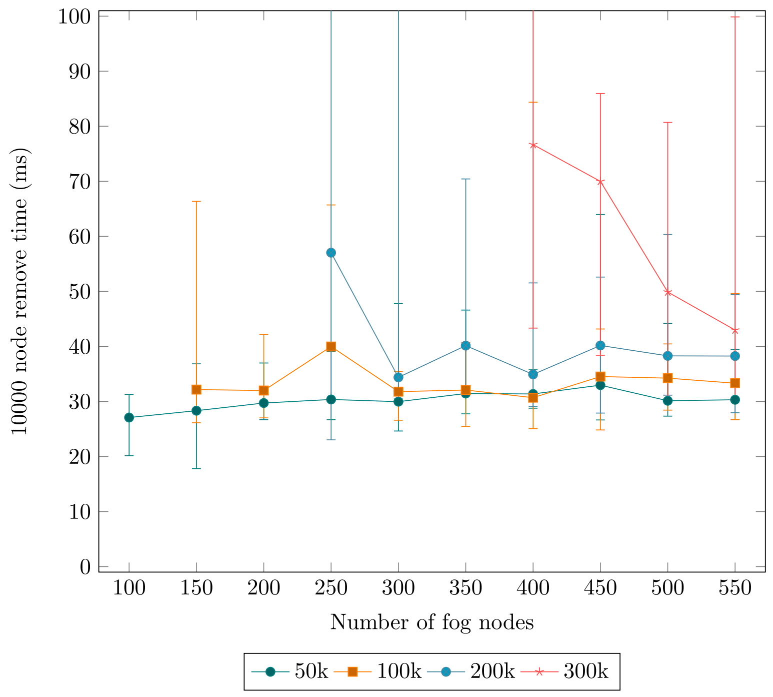

Fig. 6 Time required to move 10000 edge nodes to another fog node in service topologies of varying sizes. Legend numbers represent thousands of edge nodes

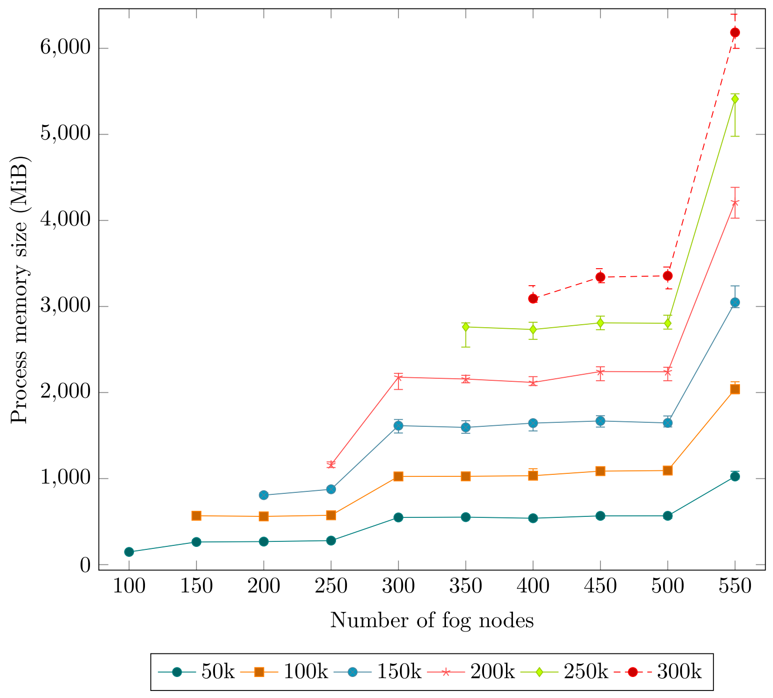

Fig. 7 Memory required for service topologies of varying sizes. Legend numbers represent thousands of edge nodes

Topology

Figure 8 compares the average edge to fog distances of topologies generated by Swirly (100k, 150k) to theoretically ideal service topologies (100kmin, 150kmin) and service topologies achieved by choosing fog nodes at random (e.g. Kubernetes scheduler, 100krnd, 150krnd). The ideal topologies do not consider resource use on fog nodes, therefore they can never really be achieved. However, since calculating actual ideal solutions is NP-hard, this is the only comparison which can be practically achieved. The random topologies are achieved by selecting the same number of fog nodes required by each iteration of Swirly, but at random.

For low numbers of fog nodes, and thus a high $|E|/|F|$ ratio, the output of Swirly is very close to randomized topologies, but still below it. This is because the algorithm has very little choice of fog nodes; most of them have to be used due to resource constraints and only about 10% are free to optimize the topology. This is further elaborated by Fig. 9. As the number of available fog nodes increases, Swirly starts to generate topologies that come closer to the ideal cases, but it eventually levels out at about twice the average distance of the theoretical ideal topologies.

Note that the 150k series topologies, despite a 50% increase in edge nodes over the 100k series, eventually have average distances which are only about 1-2% higher, which is neglegible considering the range of the whiskers. The average distances of Swirly topologies are twice as high as those of the ideal cases, apart from the case of 150 fog nodes, likely indicating that they cannot be reduced much further without removing fog node resource requirements.

A final observation concerns the maximum distance, which was set at 100. Swirly clearly struggles to stay below it when the $|E|/|F|$ ratio is high, and even fails at times. However, as the number of edge nodes increases, even the worst topologies generated by the algorithm have average distances well below the maximum distance. On average, the random topologies have average distances slightly above the maximum distance, and even the best random topologies barely outperform the worst Swirly topologies.

The observed results can be explained through the model of servicing areas. In cases where there are few fog nodes, a_F will have sections beyond r_P and possibly r_E for most fog nodes, which means they have to service a lot of edge nodes that are technically too far away, as in Fig. 3b. As the number of fog nodes increases, a_F will grow smaller since the physical area remains constant, reducing average distance by removing the extremes. Figure 3c is an example of this.

Finally, Fig. 9 shows the number of fog nodes used by Swirly topologies compared to the absolute minimum of fog nodes that could be used. It is important to note that these numbers are averaged over all topologies with

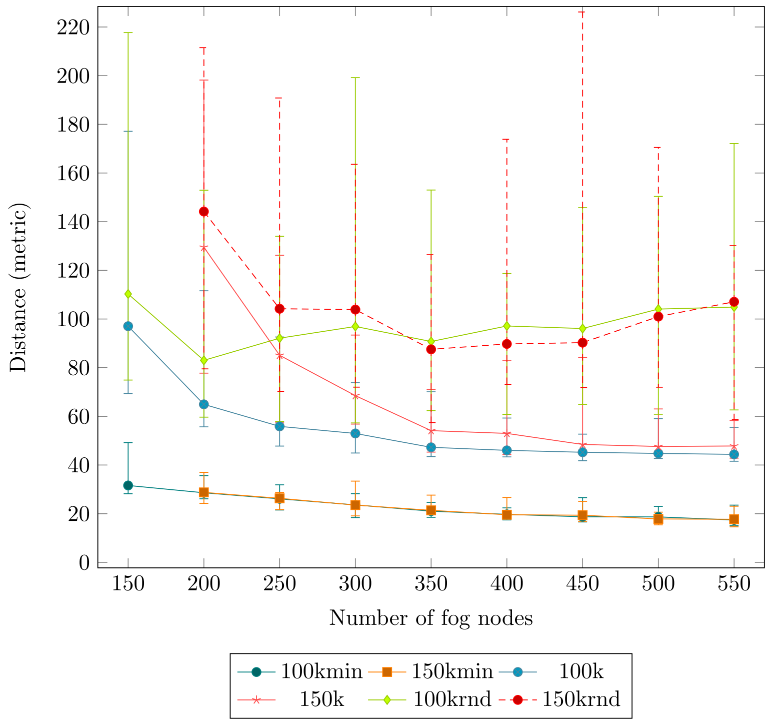

Fig. 8 Average distance for different types of service topologies. Legend numbers represent thousands of edge nodes, min denotes theoretical ideal service topologies, while rnd indicates service topologies where the fog nodes are randomly chosen

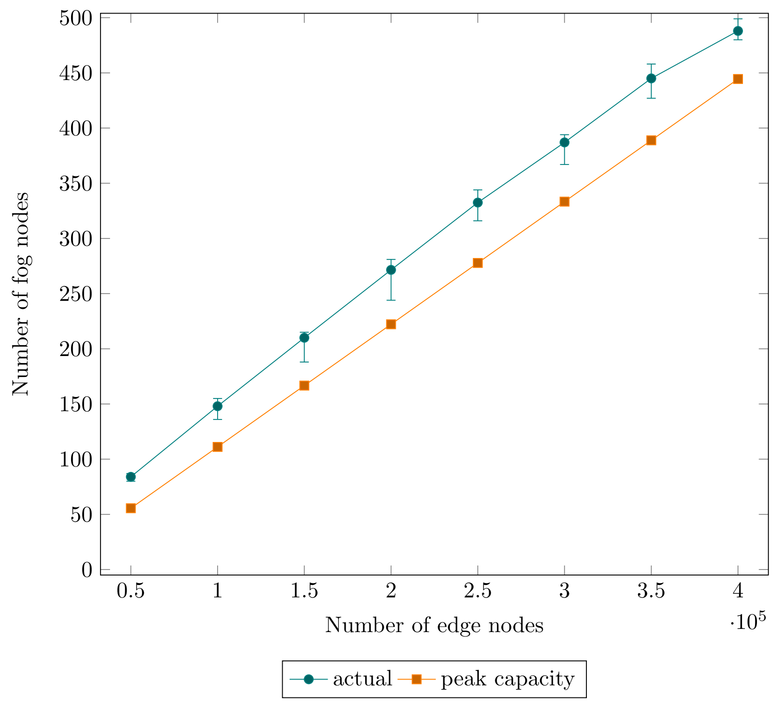

Fig. 9 Number of fog nodes activated by varying number of edge nodes

the same number of edge nodes, but with varying numbers of fog nodes. This indicates that even if Swirly has more choice of fog nodes, it primarily tries to fill activated fog nodes to capacity first, and uses very few to optimize edge to fog distances when needed. On average, Swirly uses about 30% to 10% more nodes than strictly required, leaving some fog nodes with free capacity in case the topology expands. This confirms that Swirly fulfills the fourth requirement for a useful service scheduler.

Again, this can be explained in terms of the service area model. Swirly will always attempt to generate a topology in which fog nodes have responsibility areas as shown in Fig. 3c. Once a_F fits inside r_p and r_E is unchanged, Swirly will not activate any more fog nodes, no matter how many are available. However, if the algorithm has more fog nodes available from the start, it will construct slightly better service topologies by activating the ones closest to larger clusters of edge nodes. The rise in fog nodes seen in Fig. 9 is the result of r_E reducing as the number of edge nodes increases, and Swirly reacting by activating fog nodes to shrink a_F.

Figure 10 shows how Swirly performs with physical topologies with different organizations. The tendency of the add operation to be slower when $|E|/|F|$ is higher is exaggerated when edge nodes are clustered together in remote groups, whereas the effect is all but gone in a perfectly equal distribution. With few available fog nodes, it takes 2.5 times as long to set up a service topology when edge nodes are clustered as it does with an equal distribution of edge nodes. This can be mostly attributed to the fact that the fog nodes distribution was not modified and thus Swirly has to search longer to find suitable fog nodes once the nearby ones are full. However, when both types of physical topology are given an unnecessarily high number of fog nodes, Swirly still needs about 40% more time to set up a service topology with clustered edge nodes. This confirms, as indicated by Eqs. 10 and 11, that overall performance is significantly impacted not only by global node density, but also by local node density.

Future work

The presented Swirly algorithm could easily be adapted to work with the Kubernetes scheduler by managing fog service deployments through the Kubernetes API. In order to redirect service requests to the correct fog nodes, it could interact with distributed DNS plugins deployed on the cluster, override them, or deploy a separate system.

When implementing Swirly for a specific orchestrator, it may be advantageous to split the data structures so that topologies for several services can be generated from the same nodes and distance data. With minimal changes, it is possible to keep track of which services should be deployed to any fog node, based on edge node requirements.

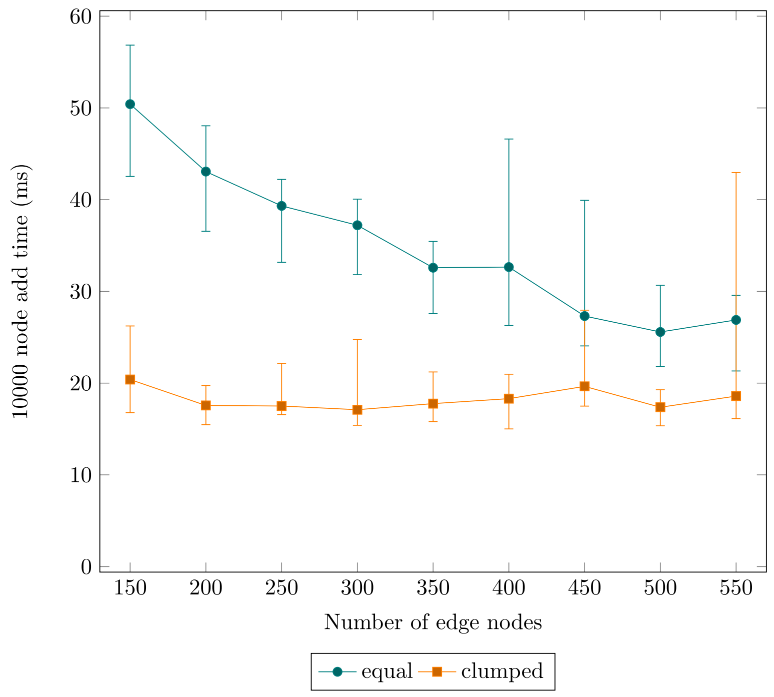

Fig. 10 Effect of equal distribution of edge nodes versus clustered edge nodes in a topology with 100.000 edge nodes. In the clustered series, edge nodes are distributed over 4 equally sized circles at the corners of the topology

Swirly does not yet fully support dynamic fog node updates. When fog nodes send updates to Swirly with their free resources, the algorithm only uses this to determine if edge nodes can be placed on those fog nodes in the future. Ideally, the algorithm would act on the resource updates by detecting critically low levels of free resources on certain fog nodes and reassigning edge nodes.

The results section shows that while Swirly scales very well in terms of processing time, its memory requirements will quickly grow beyond the reach of all but the most powerful servers. Since the algorithm relies on having a distance from each edge node to each fog node, there is no easy solution to this. However, geofencing or some type of partitioning may be able to help. In "Swirly" section some options were discussed to reduce the required bandwidth of periodic node updates to Swirly. If this mechanism is changed to cut off fog nodes altogether based on metric distance or geographical distance, memory requirements should go down drastically. Another option is to simply split the topology into parts based on logical or geographical regions, but this may result in a significantly worse result at the borders between partitions.

For these reasons, it may be 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.

Conclusion

In the introduction, a number of requirements are presented for a useful large-scale fog service scheduler. It should work on a scale of hundreds of thousands of edge devices while being able to handle changing network conditions on topologies. Simultaneously, it should take resource limits of fog nodes and distance metrics between fog nodes and edge nodes into account. Finally, it must also minimize the number of fog nodes required for any fog service deployment required by a set of edge nodes.

Swirly is proposed as a solution to these requirements, and "Swirly" and "Theoretical properties" sections show that it fulfills these requirements in theory. Different metrics are discussed, along with their advantages and disadvantages in terms of network overhead and reliability. The theoretical performance of Swirly is explored, and the fog servicing area model is constructed to explain the behaviour and capacity of fog nodes in a service topology.

To verify its performance, Swirly is evaluated in terms of memory use and processing power. While the results mostly confirm the theoretical performance, they showed that the algorithm tends to slow down sublinearly as the density of edge nodes increases. This effect is explained through the service area model. An important prediction is that for service topologies that grow in physical size rather than density, Swirly will require constant processing time. If edge node density does increase, fog node density and algorithm parameters need to change as well.

As discussed, the upper limit for the amount of nodes will most likely be dictated by memory consumption.

Further evaluations show that service topologies generated by Swirly converge towards a minimal average distance between edge nodes and fog nodes, which is well below the defined maximum value the algorithm needs to consider. Furthermore, the results show that average distances under Swirly are 30% to 55% lower than for randomly selected fog nodes (e.g. the default Kubernetes scheduler). While solutions based on heuristics (e.g. genetic algorithms) are likely to generate better solutions, they will also require more time to do so, and they do not allow for real-time updates.

Some topics for future work are discussed, for example better metrics, reducing metric network overhead, and reducing memory use. It is likely that a distributed approach to Swirly would be the best solution to tackle the last two challenges.

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

Authors' contributions
Tom Goethals conceived of the initial idea, wrote the algorithm and carried out the experiments as a PhD student under the supervision of Filip De Turck and Bruno Volckaert. Tom Goethals wrote the manuscript, with support and revision from Filip De Turck and Bruno Volckaert.

Authors' information
Tom Goethals received the master's degree in Information Engineering Technology from University College Ghent, Belgium in 2013. After several years as a software engineer, he joined the Internet Technology and Data Science Laboratory (IDLab) at Ghent University-imec in 2018 to pursue a Ph.D, during which he has received multiple best paper awards. His current research deals with scalable and reliable software systems for Smart Cities, working on various projects in cooperation with industry partners.
Filip De Turck leads the network and service management research group at the Department of Information Technology of the Ghent University, Belgium and imec. He (co-) authored over 500 peer reviewed papers and his research interests include telecommunication network and service management, and design of efficient virtualized network and cloud systems. In this research area, he is involved in several research projects with industry and academia, serves as Chair of the IEEE Technical Committee on Network Operations and Management (CNOM), and is on the TPC of many network and service management conferences and workshops. Prof. Filip De Turck serves as Editor in Chief of IEEE Transactions of Network and Service Management (TNSM), and steering committee member of the IEEE Conference on Network Softwarization (IEEE NetSoft).
Bruno Volckaert is professor advanced programming and software engineering in the Department of Information Technology (INTEC) at Ghent University and senior researcher at imec. He obtained his Master of Computer Science degree in 2001 from Ghent University, after which he worked on his PhD on resource management for Grid computing, obtained in 2006. His current research deals with reliable and high performance distributed software systems for a.o. Smart Cities, distributed decision support systems, scalable cybersecurity detection and mitigation architectures and autonomous optimization of cloud-based applications. He has worked on over 45 national and international research projects and is author or co-author of more than 125 peer-reviewed papers published in international journals and conference proceedings.

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

Availability of data and materials
The test cases are generated by an algorithm, which will be made available under an appropriate open source license upon acceptance of the article, as described in the "Evaluation methodology" section.

Competing interests
The authors declare that they have no competing interests.

Received: 9 January 2020 Accepted: 2 June 2020
Published online: 19 June 2020

References

1. Pahl C, Brogi A, Soldani J, Jamshidi P (2019) Cloud Container Technologies: A State-of-the-Art Review. IEEE Trans Cloud Comput 7(3). https://doi.org/10.1109/TCC.2017.2702586
2. Madhavapeddy A, Mortier R, Rotsos C, Scott D, Singh B, Gazagnaire T, Smith S, Hand S, Crowcroft J Unikernels: library operating systems for the cloud. In: ASPLOS '13 Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems. https://doi.org/10.1145/2499368.2451167
3. Bonomi F, Milito R, Zhu J, Addepalli S Fog computing and its role in the internet of things. In: MCC '12 Proceedings of the first edition of the MCC workshop on Mobile cloud computing. pp 13–16. https://doi.org/10.1145/2342509.2342513
4. Hu P, Dhelim S, Ning H, Qiu T (2017) Survey on fog computing: architecture, key technologies, applications and open issues. J Netw Comput Appl 98(15):27–42. https://doi.org/10.1016/j.jnca.2017.09.002
5. Latre S, Leroux P, Coenen T, Braem B, Ballon P, Demeester P City of things: An integrated and multi-technology testbed for, IoT smart city experiments. In: 2016 IEEE International Smart Cities Conference (ISC2). https://doi.org/10.1109/ISC2.2016.7580875
6. Spicer Z, Goodman N, Olmstead N The frontier of digital opportunity: Smart city implementation in small, rural and remote communities in Canada. https://journals.sagepub.com/doi/10.1177/0042098019863666 https://doi.org/10.1177/0042098019863666
7. Mach P, Becvar Z (2017) Mobile Edge Computing: A Survey on Architecture and Computation Offloading. IEEE Commun Surv Tutor 19(3). https://doi.org/10.1109/COMST.2017.2682318
8. Kumar K, Lu Y-H (2010) Cloud Computing for Mobile Users: Can Offloading Computation Save Energy? Computer 43:51–56. https://doi.org/10.1109/MC.2010.98
9. Villari M, Fazio M, Dustdar S, Rana O, Ranjan R (2016) Osmotic Computing: A New Paradigm for Edge/Cloud Integration. IEEE Cloud Comput 3(6). https://doi.org/10.1109/MCC.2016.124
10. Santoro D, Zozin D, Pizzolli D, De Pellegrini F, Cretti S Foggy: a platform for workload orchestration in a Fog Computing environment. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). https://doi.org/10.1109/CloudCom.2017.62
11. Morshed A, Jayaraman PP, Sellis T, Georgakopoulos D, Villari M, Ranjan R (2017) Deep Osmosis: Holistic Distributed Deep Learning in Osmotic Computing. 4 6. https://doi.org/10.1109/MCC.2018.1081070
12. Oppenheimer D, Chun BN, Snoeren AC, Patterson DA, Vahdat A Service Placement in a Shared Wide-Area Platform. In: Proceedings of the 2006 USENIX Annual Technical Conference. https://doi.org/10.1145/1095810.1118581
13. Zhang Q, Zhu Q, Zhani MF, Boutaba R Dynamic Service Placement in Geographically Distributed Clouds. In: 2012 IEEE 32nd International Conference on Distributed Computing Systems. https://doi.org/10.1109/ICDCS.2012.74
14. Aazam M, Huh E-N Dynamic resource provisioning through Fog micro datacenter. In: 2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops). https://doi.org/10.1109/PERCOMW.2015.7134002
15. Aazam M, Huh E-N 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. https://doi.org/10.1109/AINA.2015.254
16. Santos J, Wauters T, Volckaert B, De Turck F Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications. In: IEEE Conference on Network Softwarization (NETSOFT). https://doi.org/10.1109/NETSOFT.2019.8806671

17. Canali C, Lancellotti R A Fog Computing Service Placement for Smart Cities based on Genetic Algorithms. In: 9th International Conference on Cloud Computing and Services Science. https://doi.org/10.5220/0007699400810089
18. Zaker F, Litoiu M, Shtern M Look Ahead Distributed Planning For Application Management In Cloud. In: 2019 15th International Conference on Network and Service Management (CNSM). IEEE. https://doi.org/10.23919/cnsm46954.2019.9012693
19. Bourhim EH, Elbiaze H, Dieye M Inter-container Communication Aware Container Placement in Fog Computing. In: 2019 15th International Conference on Network and Service Management (CNSM). IEEE. https://doi.org/10.23919/cnsm46954.2019.9012671
20. Goethals T, Sebrechts M, Atrey A, Volckaert B, De Turck F Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications. In: 2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2). https://doi.org/10.1109/SC2.2018.00008
21. Kubernetes Production-Grade Container Orchestration. https://kubernetes.io/. Accessed 6 Dec 2019
22. Scheduler K. https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/. Accessed 6 Dec 2019

Publisher's Note