Journal of Network and Systems Management (2021) 29:16
https://doi.org/10.1007/s10922-020-09581-6

Self-organizing fog support services for responsive edge computing*

Tom Goethals · Filip De Turck · Bruno Volckaert

Received: 23 June 2020 / Accepted: 4 December 2020

Abstract Recent years have seen fog and edge computing emerge as new paradigms to provide more responsive software services. While both these concepts have numerous advantages in terms of efficiency and user experience by moving computational tasks closer to where they are needed, effective service scheduling requires a different approach in the geographically widespread fog than it does in the cloud. Additionally, fog and edge networks are volatile, and of such a scale that gathering all the required data for a centralized scheduler results in prohibitively high memory use and network traffic. Since the fog is a geographically distributed computational substrate, a suitable solution is to use a decentralized service scheduler, deployed on all nodes, which can monitor and deploy services in its neighbourhood without having to know the entire service topology.

This article presents a fully decentralized service scheduler, labeled "SoSwirly", for fog and edge networks containing hundreds of thousands of devices. It scales service instances as required by the edge, based on available resources

* This is a post-peer-review, pre-copyedit version of an article published in Journal of Network and Systems Management. The final authenticated version is available online at: https://doi.org/10.1007/s10922-020-09581-6

Tom Goethals (corresponding author)
Ghent University - imec, IDLab, Department of Information Technology
Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium
Tel.: +32 486 289096
E-mail: togoetha.goethals@ugent.be

Filip De Turck
Ghent University - imec, IDLab, Department of Information Technology
Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium
E-mail: filip.deturck@ugent.be

Bruno Volckaert
Ghent University - imec, IDLab, Department of Information Technology
Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium
E-mail: bruno.volckaert@ugent.be

and flexibly defined distance metrics. A mathematical model of fog networks is presented, along with a theoretical analysis and an empirical evaluation which indicate that under the right conditions, SoSwirly is highly scalable. It is also explained how to achieve these conditions by carefully selecting configuration parameters. Concretely, only 15MiB of memory is required on each node, and network traffic in the evaluations is less than 4Kbps on edge nodes, while 4-6% more service instances are created than by a centralized algorithm.

Keywords fog computing · fog networks · service scheduling · service orchestration · edge computing · service provisioning · swarm intelligence

1 Introduction

In recent years, fog computing and edge computing have emerged as paradigms to improve the QoS, efficiency and responsiveness of software services by offloading certain computational tasks from the cloud to the geographically widespread fog[1]. Simultaneously, various IoT applications and Smart City initiatives[2] have increased the importance and the physical size of the fog and edge, and the diversity of functionality deployed in them.

While both fog and edge computing can be advantageous by moving software services closer to end users and data in the network edge[3], the service scheduling algorithms used in the cloud are not designed for the conditions in the heterogeneous and geographically spread out fog and edge.

Unlike cloud networks, fog and edge networks are very volatile, with constantly changing network conditions and topologies. Not only can devices suddenly appear or disappear in the network, but the physical location of edge nodes such as vehicles and mobile phones can also change rapidly.

Clouds generally contain powerful servers that can run many service instances, and services can be migrated between machines with little penalty. However, in the fog hardware is often less powerful and it is harder to migrate services due to the distances and resource limits involved. Therefore, it is important to take these constraints into account. As a corollary, scaling in the cloud is very geographically constrained, while the nature of the fog is more suited to scaling across many devices over a wide geographical area, placing service instances close to where they are required to minimize latency.

Finally, because cloud servers are clustered in data centers, service scheduling can usually be handled by a single, centralized scheduler instance. Even distributed cloud data centers are usually few in number and connected by high-bandwidth connections, so that a single scheduler instance in any of them can gather all the required information to make scheduling decisions. However, the fog and edge contain orders of magnitude more devices than clouds, connected at lower bandwidths. Combined with the volatility of the fog, this makes gathering all the required data for scheduling decisions on a single node unrealistic; a fully decentralized solution is more suitable.

While a decentralized solution can address the issues listed so far, it can also pose new problems. Most importantly, the resource use on fog nodes and

edge nodes will be higher than with a centralized algorithm, because each node needs to perform part of the network communication and calculation that would otherwise be done in the cloud. Additionally, since there is no longer any static, single controlling entity (e.g. the cloud), global actions such as creating an overview of all running nodes or forcefully pushing updates will be more difficult.

This article presents a decentralized approach to serverless fog and edge service scheduling, named SoSwirly (Self organizing Swirly), which is based on five requirements taken from the challenges presented in this section:

- ***Req. 1*** Handle changing topologies and moving nodes in near real-time
- ***Req. 2*** Take fog node locations and resource limits into account
- ***Req. 3*** Balance the number of service instances versus QoS requirements, such as minimal overall latency
- ***Req. 4*** Scale to hundreds of thousands of edge devices through a self-organizing, decentralized approach
- ***Req. 5*** Work on a wide range of fog and edge devices by minimizing resource requirements

This article shows how the proposed solution meets the requirements by building a theoretical model of fog nodes as service providers, leading to an implementation of SoSwirly and a theoretical and empirical analysis of its performance. The conclusions show that SoSwirly is highly scalable under the right conditions, and that these conditions can be achieved in a variety of situations by tuning configuration parameters depending on fog node density.

For 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. A fog node is synonymous with a service provider and similarly, a service client or edge node client indicate an edge node. Finally, an edge service refers to a process of any kind that is dependent on the fog and runs on an edge device.

The rest of the article is organized as follows: Section 3 presents existing research related to the topic, while in Section 4 a mathematical model of fog nodes and fog networks is constructed as a basis for the implementation and evaluation of the solution. The solution itself and its implementation details are presented in Section 5, while Section 6 explores the theoretical properties of the implementation. The evaluation methodology for the solution in a number of scenarios is described in Section 7, the results of which are presented in Section 8. Finally, Section 9 discusses a number of topics for future work, and Section 10 summarizes how the solution and the evaluation results meet the proposed requirements.

2 Motivation

Previous work related to fog service deployment resulted in Swirly[4], which is aimed at large-scale deployments of services in fog networks and can handle real-time topology updates. Swirly allows the combined optimization of a

single generic distance metric and number of service instances in a fog network. However, due to its centralized approach, the memory requirement and network traffic at the node running the algorithm are limiting factors for its scalability. To solve these problems, and to eliminate the need for a cloud node, this work aims to create a distributed approach for the same optimization problem, while fulfilling the requirements put forth in Section 1. The core tenet of this approach is that each node is responsible for mapping only a small nearby part of the global topology, its neighbourhood. This concept of neighbourhoods can, with the right configuration parameters, ensure that nodes only have to communicate with a constant number of neighbours, no matter how large the node topology grows. Additionally, the traditional planning hierarchy is inverted by making edge nodes responsible for deciding which fog nodes they want to address for a specific service.

As an example application, when using roadside units (fog nodes) near highways the algorithm makes it possible for services to "follow" clusters of cars (edge devices) by allowing edge nodes to constantly switch to the nearest units for service calls, and increasing the number of services in busy areas. Because of the decentralized approach, this can be done locally and on a large scale, without the need for cloud processing and the accompanying additional latency.

3 Related Work

A literature review by Maenhaut, Volckaert, Ongenae and De Turck[5] discusses challenges related to service orchestration, resource management and pricing in (distributed) clouds and the fog. An overview of the challenges in fog and edge computing is presented by Avasalcai, Murturi and Dustdar[6]. Their study focuses specifically on challenges in resource management, security and network management, with regards to the volatile network conditions and low-power devices often found at the network edge.

To support the scaling of services in the fog and edge, several studies have focused on extending the concept of serverless computing to these areas. For example, the Fog Functions presented by Cheng, Fuerst, Solmaz and Sanada[7] are compact and scalable pieces of functionality, created through a custom programming model which ensures independency of the device and platform they are run on. Gadepalli, Peach, Cherkasova, Aitken and Parmer[8] argue that current virtualization technologies are too demanding and slow for responsive services on extremely low-power devices. They present aWsm, a serverless framework based on WebAssembly with startup times measured in microseconds. Although the implementation of SoSwirly presented in this article is focused on containers, the design allows for the adoption of these or similar virtualization technologies.

A recent literature review of container orchestration in fog networks[9] shows that the most popular research directions in this area involve Docker containers, scaling, QoS, and resource management. The approach presented

in this article is in line with these facets, although the design of SoSwirly allows for flexible QoS requirements and the use of other forms of software deployment (e.g. Virtual Machines, unikernels[10]).

A study by Guerrero, Lera and Juiz[11] compares several multi-objective evolutionary algorithms in the context of fog service placement. Although these algorithms can take a large number of parameters into account and produce near-optimal results, they are by necessity used in centralized schedulers and for small-scale scheduling due to their speed. In contrast, SoSwirly aims to optimize only distance and number of services, but at a much larger scale. Similarly, a study by Hosseinzadeh et al.[12] provides an overview of various multi-objective optimization algorithms in service networks, but in the context of selecting optimal services rather than deploying them in optimal locations. Stévant, Pazat and Blanc[13] propose a framework which monitors and optimizes service placement to minimize QoS requirements, represented by response time in their evaluation. The difference with the approach in this article is that SoSwirly attempts to balance QoS requirements versus total used resources. Although the default Kubernetes[14] scheduler is centralized, an alternative scheduler by Casquero et al.[15] allows for distributed fog scheduling in Kubernetes. However, this approach is aimed at industrial automation, and the work in this article aims to go beyond the scale limits of Kubernetes[16]. Considering the cloud-oriented nature of Kubernetes, and its limits in terms of deployable pods, this work is not aimed at Kubernetes clusters, but instead uses small parts of its API where useful in communication between nodes. While the proposed solution in this manuscript primarily relies on decentralization to avoid network traffic bottlenecks caused by monitoring nodes from a central location, the work by Vaton et al.[17] models network delay with hidden Markov models to greatly reduce the amount of monitoring traffic at the cost of processing power. However, the decentralized nature of SoSwirly allows tuning each node separately should they encounter network bottlenecks, and since it is aimed at low-resource devices with little processing power to spare, advanced traffic reduction methods are not integrated. A closely related framework proposed by Santos, Wauters, Volckaert and De Turck[18] is aimed at optimizing latency in 5G networks and takes an application-oriented approach, placing the constituent microservices of an application on specific fog nodes to minimize latency and traffic overhead. Unlike their work, the solution in this article is agnostic of application architecture and transitive dependencies; a similar effect emerges from placing each service as close as possible to its direct dependencies, which can be achieved by carefully designing the distance metric for SoSwirly.

Compared to the aforementioned studies, the novel aspect of SoSwirly is that it is fully decentralized and works in near real-time, reacting to topology changes in milliseconds to seconds, depending on network latencies. It combines most of the discussed features by taking into account node resources, node mobility, service latency, and the number of service instances, while allowing for specific implementations using additional parameters. Meanwhile, the framework itself uses minimal resources and is highly scalable, because

it only considers the status of nearby nodes. To the best of our knowledge, there are no decentralized frameworks or solutions with the same goals and properties.

Table 1: Definitions of symbols used in Section 4.

Symbol	Definition
A_E	the service area of a fog node based on its resources, see C_e
A_F	area closer to a fog node than to any other fog node
A_P	area within D_m metric distance
A	generic area, usually the entire service topology
r_E	radius of A_E, or distance to its border in a specific direction
r_F	distance(s) to the border of A_F
r_P	radius of A_P, or distance to its border in a specific direction
$\rho_E(x, y)$	edge node density at position x, y
$\rho_F(x, y)$	fog node density at position x, y
C_e	number of service clients supported by a fog node, based on its resources
C_p	scale factor between physical distance and metric distance
C_f	expected fog node density, used in areas with low ρ_F
D_m	maximum metric distance value
D(p1,p2)	metric distance between p1 and p2
A_{px}	the geographical area represented by a single pixel

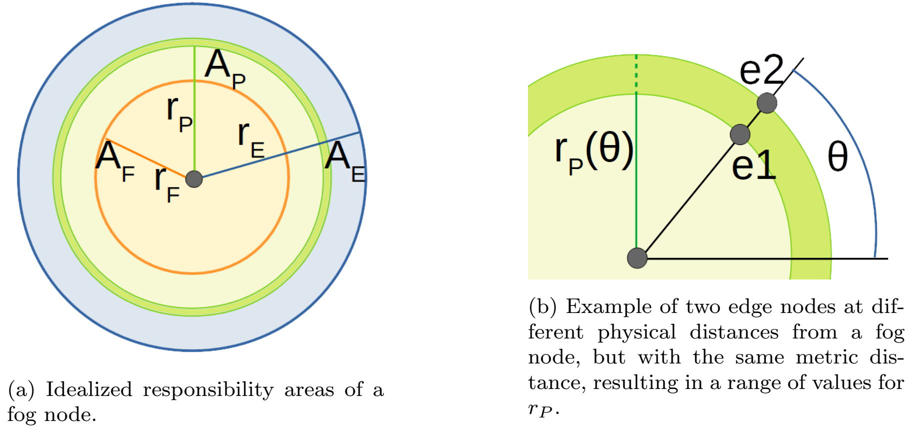
4 Fog Node Model

In this section, a theoretical model of fog nodes and fog networks is constructed, which helps to meet the requirements put forward in Section 1 that are related to the fog network topology, specifically ***Req. 1*** through ***Req. 4***. Table 1 summarizes the symbols used in this section that are not explicitly defined through equations.

As per ***Req. 3***, SoSwirly should support QoS requirements. This is achieved by defining a distance metric from fog nodes to edge nodes, and optimizing it so that edge nodes are always serviced by the "closest" available fog node in terms of the distance metric. Furthermore, a maximum distance value can be defined to put a limit on the search area for fog nodes and to serve as a hard cap for acceptable service quality (e.g. the fog node is too distant, is planned to go down, has limited bandwidth).

As shown in Fig. 1, each fog node has a number of *service areas*, which can be idealized as circles, based on its properties and the fog nodes in its neighbourhood:

- A_E, defined by radius r_E, is the capacity area of a fog node, representing the physical area it can service based on the capacity C_e imposed by its resources.
- A_F, with radius r_F, is the responsibility area of a fog node. All edge nodes within this area should be serviced by a fog node because they are closest to that fog node than any other.



(a) Idealized responsibility areas of a fog node.

(b) Example of two edge nodes at different physical distances from a fog node, but with the same metric distance, resulting in a range of values for r_P.

Fig. 1: The different responsibility areas of a fog node.

– A_P, with radius 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 distance metric value.

The radii of these areas can be naively calculated using

$$r_F = \frac{1}{\sqrt{\pi \rho_F}} \tag{1}$$

$$r_E = \sqrt{\frac{C_e}{\pi \rho_E}} \tag{2}$$

$$r_P(\theta) = r, \forall r, \theta : D(r, \theta) = D_m \tag{3}$$

Where ρ_F and ρ_E represent the density of fog and edge nodes respectively, and D_m is the maximum distance value between a fog node and any edge node it services. $D(r, \theta)$ is a function that returns the metric distance of the point r, θ. Note that the distance function can return multiple values for r_P for any θ, as shown in Fig. 1b. In this example, e1 and e2 lie in the same direction θ, but with a different r. However, the metric distance for both is equal, so $r_P(\theta)$ returns a range of values. Therefore, for A_P to be an actual circle, its radius must be the maximum value of $r_P(\theta)$. In cases where the distance metric preserves the relative ordering between all points, r_P can be reduced to

$$r_P = C_p D_m \tag{4}$$

In which C_p is a constant that describes how the distance metric scales with the physical distance between points. The definition of r_E naively assumes a constant edge node density. Assuming the distance metric uses a coordinate

system (x', y'), a more accurate value for r_E is found by solving the following equation[19] for the area A, which can be transformed back to Cartesian coordinates:

$$\int\int_A J(x', y')\rho_E(x', y') \cdot \mathrm{d}x' \cdot \mathrm{d}y' \ = C_e \tag{5}$$

While the Jacobian[20] $J(x', y')$ makes this equation cumbersome, the coordinate system can be treated as polar coordinates where r is the metric distance and θ is the direction of the node:

$$\int\int_0^{2\pi} r\rho_E(r, \theta) \cdot \mathrm{d}r \cdot \mathrm{d}\theta = C_e \tag{6}$$

Since the distance function can not calculate θ, the following version can be used in cases where θ is not known without losing accuracy:

$$\int 2\pi r\rho_E(r) \cdot \mathrm{d}r = C_e \tag{7}$$

In all cases, for a uniform density ρ_E these equations reduce to Eq. 2

$$\int\int_0^{2\pi} r\rho_E \cdot \mathrm{d}r \cdot \mathrm{d}\theta = C_e \tag{8}$$

$$\left[\frac{r^2\theta}{2\rho_E}\right]_0^{\theta=2\pi} = C_e \tag{9}$$

$$r = \sqrt{\frac{C_e}{\pi\rho_E}} \tag{10}$$

Note that while Eq. 2 assumed the use of the Euclidean metric, this version is based on the distance metric expressed as polar coordinates. Therefore, the resulting r must be converted from metric distance to geographical distance before using it with Cartesian coordinates (e.g. a geographical map). Using these definitions, some conditions can be stated for a well-organized service topology:

- 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 distance.

These requirements can be expressed in a single equation as

$$r_F \leq r_P \leq r_E \tag{11}$$

While this equation has its own merit in predicting efficient fog node operation given sufficient information, these relations can be used to determine requirements for the fog node density ρ_F. For the moment, r_P is assumed to be calculated as in Eq. 4. Then, since

$$\frac{1}{\sqrt{\pi\rho_F}} \leq min\left(C_p D_m, \sqrt{\frac{C_e}{\pi\rho_E}}\right) \tag{12}$$

there are two conditions for ρ_F, namely

$$\frac{1}{\sqrt{\pi\rho_F}} \leq C_p D_m \Rightarrow \rho_F \geq \frac{1}{\pi C_p^2 D_m^2} \tag{13}$$

$$\frac{1}{\sqrt{\pi\rho_F}} \leq \sqrt{\frac{C_e}{\pi\rho_E}} \Rightarrow \rho_F \geq \frac{\rho_E}{C_e} \tag{14}$$

As a result, the following equation shows how to calculate minimal fog node density at any location based on edge node density and the maximum distance:

$$\rho_F \geq max\left(\frac{1}{\pi C_p^2 D_m^2}, \frac{\rho_E}{C_e}\right) \tag{15}$$

At this point C_p can be substituted with a more advanced relation between geographic distance and metric distance. The *metric pressure* P represents the inverse concept of C_p, indicating the inflation of metric distance with respect to geographical distance at any point, calculated over the shape of the topology. Assuming a point $p1$ with Cartesian coordinates, and that $D(p1, p2)$ gives the metric distance between $p1$ and any other point $p2$, the metric pressure P can be defined at $p1$ as

$$P(p1) = \frac{\int_A \frac{D(p1,p2)}{||p1,p2||} \cdot \mathrm{d}A}{A} \tag{16}$$

Where $p2$ represents each point in the area A of the Swirl and $||p1, p2||$ is the Euclidean distance between $p1$ and $p2$. In practice, P will either be known from a few measurements or will not need to be calculated, reducing the computational load this would represent.

Using this definition, it follows that P can replace $1/C_p$ in Eq. 15, since a smaller C_p inflates $D(p1, p2)$ and thus P. As a result, equations can be constructed to calculate the minimal fog node density $\rho_F(x, y)$ at any location in a Swirl, and the minimum number of fog nodes N_{fog} required to service a given area of a Swirl, using any distance metric:

$$\rho_F(x,y) \geq max\left(\frac{P(x,y)^2}{\pi D_m^2}, \frac{\rho_E(x,y)}{C_e}\right) \tag{17}$$

$$N_{fog} \geq \int_A max\left(\frac{P(x,y)^2}{\pi D_m^2}, \frac{\rho_E(x,y)}{C_e}\right) \cdot \mathrm{d}A \tag{18}$$

4.1 Fog Neighbourhood Discovery

The maximum distance value D_m in Eq. 4 represents the maximum distance between edge nodes and their service providers. However, fog nodes also interact with each other in the form of a neighbourhood discovery process, which is explained in Section 5.1. Ideally, the maximum distance used in this discovery process would be a single value, for example D_m used by edge nodes. However, Fig. 2 illustrates how this can lead to problems. While $f3$ is in the green circle, and thus should clearly be known to $f1$, $f2$ is too far from $f1$ for $f3$ to be discovered through it.

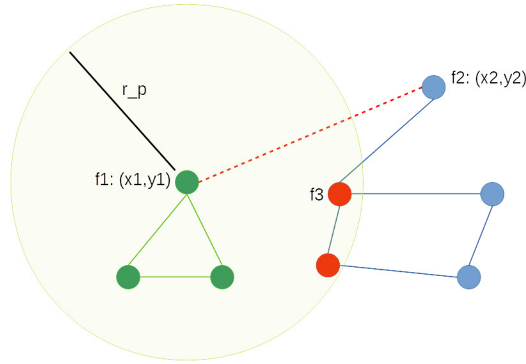


Fig. 2: Potential problem with fog discovery when using a constant maximum distance for discovery. Fog node f3, despite being in the green circle representing the neighbourhood of f1, will not be discovered because the connecting node f2 is too far away from f1. A_P, with radius r_P, is drawn as a circle for illustrative purposes.

This problem can be solved if, similar to how minimum fog node density is calculated, the maximum discovery distance D_{mf} depends on both maximum metric distance and local fog node density, so that

$$D_{mf}(x,y) = max\left(D_m, \frac{C_f}{\sqrt{\pi\rho_F(x,y)}}\right) \tag{19}$$

Where C_f is a constant representing the expected fog node density. The higher this number is, the more connected the fog network will be, but traffic and resource use will increase with it. Note that this does not solve the problem of $f1$ being unable to discover $f2$, but it will enable $f2$ to discover $f1$. Eventually, $f3$ will find $f1$ through $f2$, at which point $f1$ will discover its remaining neighbours.

There is a possible solution for D_{mf} which would allow the use of a single value while discovering all required nodes, if instead of $\rho_F(x, y)$ the minimum value of ρ_F over the entire topology is used. However, this would result in an unnecessarily large neighbourhood for most fog nodes, and a lot of traffic overhead.

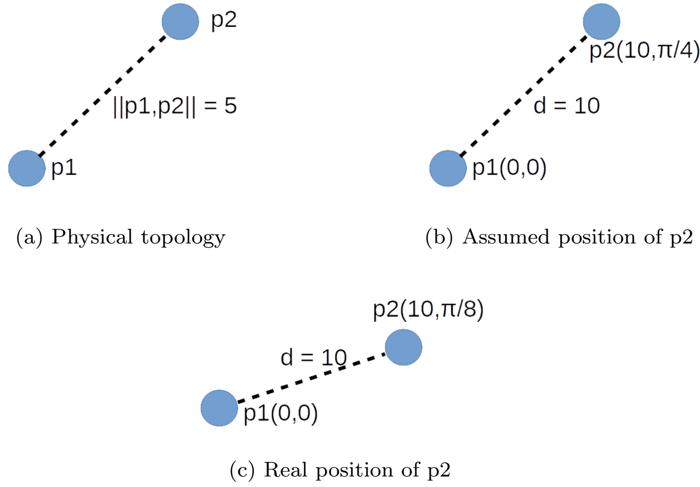
4.2 Distance Metric Coordinates

The native coordinate system of the distance metric may contain any number of dimensions with discontinuous values (e.g. binary flags or enum values). However, an explicit coordinate transformation with respect to a fully differentiable manifold[21] is required to properly calculate positions and relative distances between fog nodes and edge nodes in terms of the distance metric. Additionally, while the Euclidean metric is (implicitly) used in all equations so far, with the distance metric used as a function, the distance metric should instead be treated as a metric, but one which is only defined on its native coordinate system. This subsection shows how the distance metric values determined by nodes can be used to construct surrogate distance metric coordinates for nodes in Cartesian and polar coordinates.

In terms of polar coordinates, the metric distance d is assumed to scale with r, which is also the assumption in C_p in Eq. 4. Examples of metrics with this property are geographical distance between nodes and latency under normal circumstances, within a reasonable margin of error. This type of distance metric transforms the topology shown in Fig. 3a into distances and node positions similar to those shown in Fig. 3b. However, in other cases the distance metric will be more complex and it will not scale equally with Cartesian or polar coordinates in every dimension, nor at every location, as shown in Fig. 3c. Moreover, two physically very different nodes may map onto the same location in terms of distance metric coordinates.

This presents problems for advanced functionality where the coordinates of the nodes are required, e.g. predicting node movement and trajectory in terms of the distance metric. However, if the distance metric preserves relative Euclidean distances, accurate surrogate coordinates can be constructed for each node, similar to Fig. 3b, by assuming that r is the measured distance d and θ is the same as the angle between both nodes in polar coordinates.

When constructing surrogate coordinates from the point of view of a point $p1$, it can be placed at the origin with coordinates $(0, 0)$. The surrogate coordinates of a point $p2 : (r, \theta)$ relative to $p1$ are thus defined in polar coordinates as



(a) Physical topology

(b) Assumed position of p2

(c) Real position of p2

Fig. 3: Example of erroneous assumption of distance metric coordinates from known positions and measured metric values between $p1$ and $p2$. θ is not known from measurements, but has to be assumed.

$$p2' = (D(p1, p2), \theta) \tag{20}$$

In Cartesian coordinates, the metric ratio s can be used to scale the relative distance between $p1$ and $p2$:

$$s = \frac{D(p1, p2)}{||p1, p2||} \tag{21}$$

The entire issue is circumvented in all equations so far by using the geographical (Cartesian) coordinates of nodes, either because it makes no difference or because the potential error is very small. For example in Eq. 17 $\rho_E(x, y)$ is used instead of $\rho(r, \theta)$, but this makes no difference since it merely describes the local node density of an infinitesimal region, without respect to another point. Other concepts, such as the metric pressure P, were specifically designed to avoid having to use the metric coordinate system. However, these surrogate coordinates may be useful in future work.

4.3 Fog Hardware Placement

While Eq. 17 allows for the calculation of the minimum required fog node density at any location, it is very general and difficult to implement. The evaluations in this article use a more straightforward distance metric so that $1/C_p$ can be used rather than $P(x, y)$. Furthermore, it is more useful to have a version of the equations that can handle discrete regions with piecewise constant densities (e.g. pixels).

The discrete version of Eq. 17 using a straightforward distance metric is

$$\rho_F(x,y) \geq max\left(\frac{1}{\pi C_p^2 D_m^2}, \frac{\rho_E(x,y)A_{px}}{C_e}\right) \tag{22}$$

in which A_{px} represents the surface area of a single region. In the case of pixels, this is a constant and the following can be substituted

$$C_a = \frac{A_{px}}{C_e}, \rho_p = \frac{1}{\pi C_p^2 D_m^2} \tag{23}$$

which are both constant, resulting in a short version of Eq. 22

$$\rho_F(x,y) \geq max(\rho_p, \rho_E(x,y)C_a) \tag{24}$$

This equation can be used to determine the minimum total number of fog nodes over the entire topology, giving the discrete version of Eq. 18

$$N_{fog} \geq \sum^{X}\sum^{Y} max(\rho_p, \rho_E(x,y)C_a) \tag{25}$$

Note that both Eq. 24 and 25 are only valid if the distance metric preserves relative distances so that the quotient of the metric distance and physical distance can be represented by C_p, and furthermore that all discrete regions are the same size, in which case A_{px} is constant. In other cases, the equations will be similar, but more terms will have to be calculated for each region.

5 Fog Service Provisioning

SoSwirly, the solution presented in this article, requires a number of logical components to fulfill the requirements listed in Section 1. A decentralized node discovery algorithm is required to keep track of changing topologies on a large scale, as per ***Req. 1*** and ***Req. 4***. Similarly, ***Req. 2*** and ***Req. 3*** must both be taken into account in a scheduler component, which finds the best fog node to use as a service provider. Upon finding a suitable node, or when the service provider changes, ***Req. 1*** requires a component that can forward any requests to the correct node. This section describes how these logical components are implemented by software components, with ***Req. 5*** implicitly present in the design through the choice of Golang and lightweight dependencies, e.g. using a straightforward DNS solution, using FLEDGE agents instead of Kubelets. FLEDGE[4] is a lightweight container orchestrator for edge nodes, similar to K3s[22], but fully compatible with Kubernetes master nodes.

The chosen distance metric is implemented as a custom ping web service that determines latency. This choice of metric, which is implicitly assumed to scale with geographical distance, allows the basic properties of SoSwirly to be explained intuitively, by relying on the reduced forms of the equations presented in Section 4.

The logical components are divided among two separate services, which are implemented in Golang for the evaluations in Section 7. The *fog node service*, deployed on all fog nodes, is responsible for managing fog services based on edge node requests and for discovering other fog nodes in its neighbourhood. The *edge node service* is deployed on all edge nodes and monitors which services on an edge node require fog support services. When required, these support services are requested from the nearest (active) fog node, which is found by traversing fog node neighbourhoods discovered by the *fog node service*.

Central to both services is the algorithm for node discovery, which fog nodes use to discover their neighbourhoods and edge nodes use to find an optimal service provider.

In principle, the edge node service is not limited to running on edge nodes. It can also be deployed on fog nodes, allowing a tiered fog architecture in which each layer detects the services running on its devices and attempts to deploy the required support services in a higher layer.

5.1 Neighbourhood Discovery

Algorithm 1 presents the general outline of the algorithm used to discover nearby fog nodes from any node.

At the start of each discovery round, a queue N_{check} is created from the elements of the list of known nodes N_{known}, and nodes discovered through other means N_{new} which do not yet have a distance assigned to them. The list of nodes to ignore N_{ignore} is initialized, to which nodes are added that are beyond the maximum distance and should not be contacted again this round. As long as there is a node left in N_{check}, the first element n_c is taken from the queue. The algorithm first checks if the distance to n_c should be updated this round by calling *ShouldReping*. If so, the distance $d[n_c]$ is determined and updated and d_{max} is calculated by the *AdjustedDistance* function according to Eq. 19. If the new distance to n_c is smaller than d_{max}, the node is added to (or updated in) the list of known nodes, and its list of known nodes is fetched into N_{new}. These nodes are in turn added to the list of nodes to check through the *MergeNodes* function, which only adds those nodes of N_{new} to N_{check} which are not in N_{check}, N_{known} or N_{ignore}. Additionally, the flag *inReach* is set to true, and the minimal distance d_{min} for this round is updated if necessary. If the new distance to n_c is larger than d_{max}, the node is added to the list of nodes to ignore. However, if the minimum distance in this discovery round is also larger than d_{max}, the algorithm has not yet found a way to nodes within its neighbourhood. Therefore, the list of nodes known to n_c is fetched and merged with the list of nodes to check, in an attempt to find closer nodes. Additionally, if there is only one known node, or the distance to n_c is smaller than the smallest distance so far, d_{min} is also updated to narrow the search field. If neither of these conditions are true, c_n is removed from the list of known nodes if it is in there. When the queue N_{known} is empty, the *inReach*

```
function Discover(selfNodeType, N_known, N_new) is
    N_check = N_known ∪ N_new
    N_ignore = ∅
    if N_check ≠ ∅ then
        d_min = D(selfNodeType, N_check[0])
    inReach = false
    while N_check ≠ ∅ do
        n_c = N_check[0]
        Remove(N_check, n_c)
        if ShouldReping(n_c) then
            d[n_c] = D(selfNodeType, n_c)
            d_max = AdjustedDistance(Len(N_known))
            if d[n_c] ≤ d_max) then
                add(N_known, n_c)
                N_new = GetKnownNodes(n_c)
                MergeNodes(N_check, N_new, N_ignore)
                inReach = true
                d_min = min(d_min, d[n_c])
            else
                Add(N_ignore, n_c)
                if d_min ≥ d_max then
                    N_new = GetKnownNodes(n_c)
                    MergeNodes(N_check, N_new, N_ignore)
                if Len(N_known) ≤ 1 or d[c_n] ≤ d_min then
                    d_min = d[c_n]
                else
                    Remove(N_known, c_n)
                end
            end
    end
    if inReach then
        N_remove = ∅ for n ∈ N_known do
            if d[n] >AdjustedDistance(Len(N_known)) then
                Add(N_remove,n)
        end
        N_known = N_known ∩ N_remove
end
function ShouldReping(c_n) is
    customizable implementation, default true
end
function GetKnownNodes(c_n) is
    webservice call to n_c
end
function AdjustedDistance(n) is
    ρ_local = ρ_ass/n
    return d_max·max(1, √(ρ_local)
end
function MergeNodes(N_check, N_new, N_ignore) is
    N_check = N_check ∪ (N_new ∩ N_ignore ∩ N_known)
end
```

Algorithm 1: Node discovery mechanism used in both the fog node service and the edge node service.

flag is checked to remove any nodes beyond the maximum distance from the list of known nodes, because such nodes may have been added while d_{min} was larger than d_{max}

When contacting another node, a node will always pass its node type (e.g. edge or fog) in the request. When a fog node is contacted by an edge node, it will simply respond and the distance between the two can be determined by the edge node. When a fog node is contacted by another fog node, it will check if it already discovered that node; if not it will add the node to a queue of nodes to contact on the next discovery round.

As explained in Section 4.1, the maximum distance for fog node neighbourhood discovery should not be a constant value, but should depend on local fog node density as well. The local fog node density can be estimated by considering how many fog nodes have been discovered by using the default maximum distance; the new maximum distance can be calculated from it according to Eq. 19. However, edge nodes still use a constant maximum distance to discover optimal service providers in accordance with Eq. 4.

Ideally, all discovered fog nodes should be contacted during each discovery round to determine their new distance. However, to reduce neighbourhood size and limit edge node traffic, the *ShouldReping* function can be implemented differently for fog and edge nodes, enabling different timeouts on nodes that are far away. For example, while some fog nodes beyond the maximum distance are kept in the list of known nodes, this feature allows the algorithm to only contact them every 2 or 3 rounds.

5.2 Fog Node Service

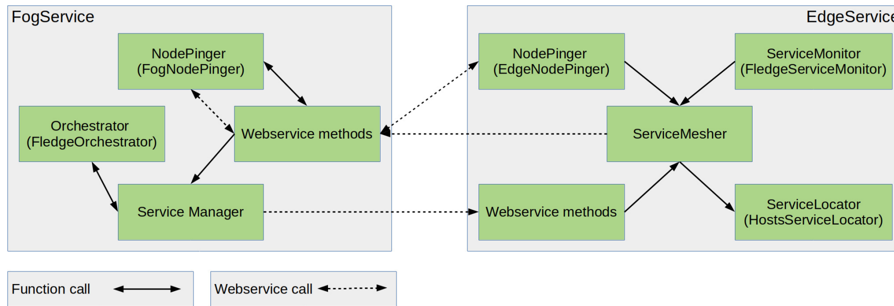


Fig. 4: Components of the fog node service and the edge node service and their interactions.

The left side of Fig. 4 shows the components of the fog node service and their interactions.

The *FogNodePinger*, an implementation of *NodePinger*, periodically attempts to discover fog nodes in its neighbourhood according to Algorithm 1.

It is used by the webservice methods to return a list of known fog nodes, and in turn calls on the webservice methods of other fog nodes for neighbourhood discovery.

The *Service Manager* handles requests from edge nodes to add or remove them as service clients. When adding a service client, it checks if there are sufficient resources to handle extra service clients or services, and starts any services needed by the edge node client if they are not yet running. When a service is started, it is loaded from a configuration file containing a Kubernetes PodSpec[23] which is sent to the *Orchestrator* for deployment. When a service client is removed, the Service Manager checks each service used by that client to see if there are still enough other clients using them. If the number of clients for any service is below a configurable minimum, all remaining edge node clients are notified to try and find a different fog node for that service. If successful, the migration is confirmed to each client, they are then removed from that service and the *Service Manager* instructs the *Orchestrator* to stop and remove the service itself. If a single edge node fails to find another suitable service provider, the edge node clients are notified that the migration should be reverted.

The *FledgeOrchestrator* is an implementation of *Orchestrator* and deploys Kubernetes pods through a FLEDGE agent. It supports only the two methods described above; one to deploy a pod and one to stop a pod.

5.3 Edge Node Service

The components of the edge node service are shown on the right side of Fig. 4, along with their interactions and web service calls to and from the fog node service.

As in the fog node service, the *EdgeNodePinger* is responsible for discovering nearby fog nodes as per Algorithm 1. The difference is that the results of each discovery round are actively used, by sending them to the *ServiceMesher*, which takes further action based on the support services required by the edge node.

To determine which support services are required, the *FledgeServiceMonitor* periodically checks all running pods managed by the local FLEDGE agent and compares them to a configurable map of edge service names and the fog services they depend on. When a new pod is detected, the map is consulted and any required fog services are forwarded to the *ServiceMesher*.

The *ServiceMesher* uses the fog nodes discovered by the *EdgeNodePinger* to find a suitable service provider for the required services passed by the *FledgeServiceMonitor*. For each service S it first attempts to find a fog node which already has S deployed, in order of increasing distance, up to the maximum distance. If this fails, the closest node with enough available resources to deploy S is selected as service provider, no matter the distance. The selected node is then notified that the edge node requires S, and registers itself as a service client for S at that fog node. Note that this can result in multiple ser-

vice providers for a single edge service, depending on pre-existing deployments in the fog. If for any reason a fog service deployment fails, or the edge node can not be registered as a client, the search for a service provider continues. This approach is in line with the original Swirly algorithm[24], which prefers nearby fog nodes with active service deployments and free resources over empty fog nodes when possible. Any connections from the edge service are redirected to the correct fog node by the *HostsServiceLocator*. For the purposes of this article, this naive implementation of *ServiceLocator* assumes that the name of the fog service is used as a DNS name when it is called by the edge service. This assumed DNS name is linked to the IP address of the fog node by adding a line in the hosts file [25] of the edge node.

As described in Section 5.2, a fog node can ask edge node clients to migrate to a different service provider for a specific service. When this happens, the *ServiceMesher* attempts to find a service provider as described in the previous paragraph, with the exception that the resulting fog node must already be running the required fog service, and is different from the current service provider. If a suitable fog node is found, the edge node assigns it as backup service provider and adds itself as a client for the required service to ensure that it can be serviced. If any of these steps fail, the edge node reports to the original service provider that the migration has failed, otherwise it reports the migration can proceed. In the final step, the edge node client is instructed by the fog node to either confirm or revert the migration. When confirming, the backup service provider is given active status and the *HostsServiceLocator* is updated accordingly. When reverting, the edge node removes itself from the backup service provider as client.

6 Theoretical Performance

In terms of processing and memory, the average computational complexity of neighbourhood discovery using Algorithm 1 is $O(r_P^2\rho_F)$. r_P^2 describes the area with which a fog node interacts, while the amount of interaction increases with ρ_F. Extending this to interaction with edge nodes, the complexity is $O(r_P^2\rho_F)$ on edge nodes and $O(r_P^2\rho_E)$ on fog nodes. However, in the worst case a fog node has to search a significant fraction of all fog nodes to find any that belong to its neighbourhood. These adverse situations may be caused by topological features as shown in Fig. 2, or because the (randomly) assigned start node for the discovery process is several times the maximum distance away. In both cases performance will converge to the average situation within a few discovery rounds, but worst case complexity is therefore $O(F)$ in terms of processing.

Detecting and deploying support services has a best case complexity of $O(1)$ in terms of processing, since the closest fog node is known from the discovery process and can be found in constant time. In situations with very few fog resources left, worst case performance is $O(r_P^2\rho_F)$, since all neighbouring fog nodes may have to be consulted to find one with free resources. Average complexity is between these extremes, depending on the amount of nearby

Table 2: Summary of processing complexity.

	Best	Average	Worst
Neighbourhood discovery	-	$O(r_P^2 \rho_F)$	$O(F)$
Fog discovery (edge node)	-	$O(r_P^2 \rho_F)$	$O(F)$
Fog discovery (fog node)	-	$O(r_P^2 \rho_E)$	$O(r_P^2 \rho_E)$
Service deployment	$O(1)$	$O(1/(1-\frac{\rho_E}{\rho_F}))$	$O(r_P^2 \rho_F)$

edge nodes and fog nodes, resulting in $O(1/(1-\rho_E/\rho_F))$. In all cases, memory complexity is $O(r_P^2 \rho_F)$. Table 2 summarizes the computational complexities for all cases.

In all cases, the complexity of network throughput will be the same as processing complexity, since every action in neighbourhood discovery and service deployment requires a webservice call.

Because the best and average case complexities are based only on local node densities and maximum discovery distance, ***Req. 4*** is met by the theoretical performance of SoSwirly.

7 Evaluation Methodology

This section describes the evaluation setup, the scenarios involved in the evaluation and their limitations, in which terms SoSwirly is evaluated and how the results are measured. The code of SoSwirly, including the tools used for the evaluations, is made available on Github[1].

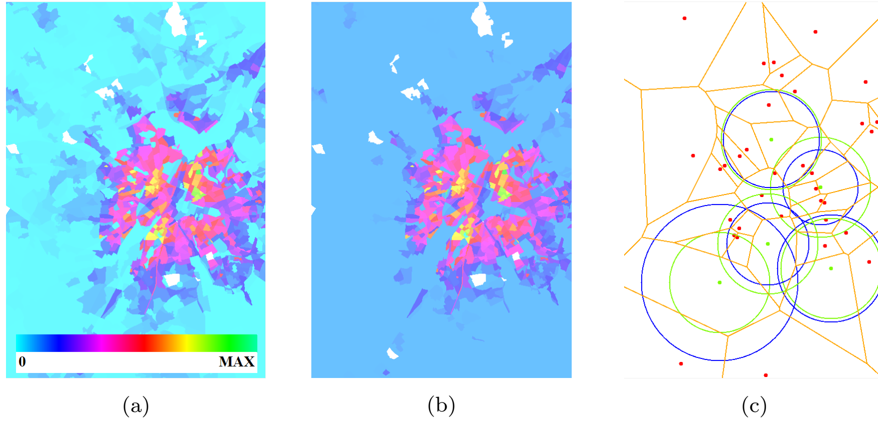
7.1 Evaluation Setup

All evaluations are run on the IDlab Virtual Wall [26] using two servers, each with 48GiB RAM, two Intel E5-2650v2 CPUs and a Gigabit Ethernet connection. One server is used to host a number of fog node services, while the other hosts a number of edge node services.

To support running multiple fog and edge nodes on a single machine, and for other nodes to be able to access them, a number of changes are made to the code that are enabled through the configuration flag *testMode*. *testMode* is designed to use incremental node names and port numbers so that each node knows where to reach another node by name alone. Additionally, it disables the actual deployment of a pod in FLEDGE so as to not overload the server with pod deployments. Finally, only one FLEDGE instance is started, running a single instance of the service which edge nodes are configured to detect, and each edge node service monitors this single instance.

While the services can thus be run on a single server, they create a large number of network connections between them. This was found to put a practical limit on how many nodes can be emulated simultaneously, so the evalua-

[1] https://github.com/togoetha/soswirly



(a) (b) (c)

Fig. 5: Part of the physical area of Belgium used to generate node topologies and evaluate SoSwirly. The scale in (a) represents edge node density, where teal is low density, and red through green represent high densities. (b) shows the required fog node density as calculated using Eq. 24, using the same density scale. The darker blue shows that in the surrounding towns, ρ_E is overridden by the required maximum distance. (c) shows the service areas of selected fog nodes (background omitted). Orange is A_F, blue is A_E and green is A_P.

tions are limited to a maximum of 250 fog nodes and 150 edge nodes. Although this does not represent the topological scale SoSwirly is meant to operate at, it is impossible to simulate thousands of nodes on the Virtual Wall, and conclusions can still be drawn from evaluations within these limits.

7.2 Swirl Generator

The evaluations use randomized Swirls created by the Swirl generator. This generator accepts a number of parameters (e.g. number of fog nodes, maximum distance) and a population density bitmap as input, and uses the equations from Sections 4 and 4.3 to generate edge and fog nodes at suitable, but random, positions. The output of this tool is a set of configuration files that are used to start all the generated nodes on the evaluation servers. An example input density bitmap is shown in Fig. 5a, with the results of Eq. 24 in Fig. 5b. A small-scale visualization of the output of the generator is shown in Fig. 5c, overlaid with calculated A_P, A_F and A_E of a few selected fog nodes. Note that the responsibility areas take the shape of a Voronoi diagram and A_F is not defined by a single value r_F. This is because the equations in Section 4 assume an ideal shape for A_F, while the actual shape is dependent on random factors (e.g. placement of fog nodes, choice of active fog nodes, stability of distance metric over time). However, SoSwirly will attempt to produce A_F as close to ideal as possible, and the equations still hold when r_F is defined as the maximum distance between a fog node and the boundary of A_F.

The density bitmaps are generated by another tool, which combines official census data and GEOJSON[27] data into population density bitmaps. For the evaluations, data from StatBel[28] on the statistical sectors of Belgium[29] and population per statistical sector[30] is used. The statistical sectors are on average $1km^2$, although they are smaller in cities, and larger in sparsely populated areas, making them sufficiently fine-grained.

7.3 Scenarios

The base density bitmap used to generate Swirls is similar to that shown in Fig. 5a, but encompasses a larger area of around $660km^2$. Its right side is focused on Brussels, while the middle and left include the surrounding countryside. It contains population densities from $0/km^2$ to around $40000/km^2$. Considering the artificial nature of the evaluation, the parameters needed for the equations in Section 4.3 are approximated. Population density is used as a substitute for $\rho_E(x, y)$, while fog node capacity C_e is set at 50.000, making C_a $1,25{\cdot}10^{-8}km^2$. For the chosen metric and the given scenario, C_p can be set to 1 pixel per metric distance unit. These parameters are used to calculate the minimal fog density for each scenario. No cases are examined that contain only heavily populated areas or countryside, since in cities r_P can simply be lowered to suit the higher ρ_E and ρ_F, stabilizing performance, and in the countryside r_P can be increased to suit the discovery process. A mixed case exposes the stability and performance of SoSwirly when using a single set of parameters over a wide range of node densities.

As explained in Section 3, there are fundamental differences between SoSwirly and recent related work. The results of solutions based on evolutionary algorithms can be superior, but these solutions are centralized and can not run in near real-time. Others run in real-time, but require a central component or have different use cases (e.g. optimizing all the components of an IoT application in the fog). Keeping these nuances in mind, the efficiency of SoSwirly is compared to the conceptually comparable but centralized Swirly, and NSGA-II, a generic multiobjective genetic algorithm[31]. For NSGA-II, the optimization parameters are average distance between fog and edge nodes, and the number of service instances. The same parameters are used as by Guerrero, Lera and Juiz[11], except the evaluation is performed using both 400 and 5000 generations. Additionally, since the maximum distance is a soft restriction in SoSwirly, any distance above 100 is penalized by a factor of 5 in NSGA-II to discourage it.

7.3.1 Neighbourhood Discovery

This scenario determines the impact of the neighbourhood discovery algorithm on fog nodes. Using only a set of fog nodes, the neighbourhood discovery algorithm is evaluated in terms of CPU use, memory use and network traffic. The number of fog nodes is varied from 50 to 250 in steps of 50, while the

maximum distance is varied from 50 to 150 in steps of 50. Due to practical limits in the evaluation setup, the combination of 150 maximum distance and 250 fog nodes is not tested. For each combination of parameters, SoSwirly is run for 10 generated swirls. Additionally, the accuracy of neighbourhood discovery is measured to determine the optimal maximum distance. For this scenario, discovery rounds are run on each node every 5 seconds. While this reaction time may be too high for certain scenarios with stringent real-time requirements, it is configurable, and can be lowered significantly depending on the requirements.

Using Eq. 24, the minimum number of fog nodes N_{fog} for the evaluation is 119 for a maximum distance of 50, 40 for a maximum distance of 100 and 27 for a maximum distance of 150. Although the test parameters go below the minimum number of nodes for a maximum distance of 50, these cases will show how SoSwirly reacts when not provided with enough fog nodes.

7.3.2 Service Deployment

In this scenario, the combined impact of fog node discovery, service detection and fog service deployment on edge nodes is measured in terms of CPU use, memory use and network traffic. The maximum discovery distance is set at 100, and 100 fog nodes are used for this scenario. Due to hardware limitations, this scenario was only evaluated for 100 and 150 edge nodes.

7.3.3 Topology Efficiency

Additional information extracted during the evaluation of the *Service Deployment* scenario is used to determine how effectively SoSwirly minimizes both the number of active fog nodes and the average distance between edge and fog nodes. Both of these optimization parameters will be compared between SoSwirly and the centralized algorithms Swirly and NSGA-II, in addition to the time taken for each algorithm to determine a solution. For both Swirly and NSGA-II, the scenario is first prepared and only the timing of the actual algorithm is measured. For SoSwirly, the measured time includes some web service calls due to its decentralized nature and the necessity to gather node information ad-hoc.

7.4 Resource Measurement

Each evaluation is run for 5 to 10 minutes, depending on the required time to start all nodes, with resource levels logged every 10 seconds. The full range of data over time is not presented in this article because most of it represents the evaluation script starting up the required nodes. The results presented focus on the last measurement, which is the most resource intensive period representing a fully initialized, stable cluster in which neighbourhood discovery is mostly completed, but is still periodically checking for new nodes.

Considering the number of processes involved in the evaluations, CPU use per process is approximated by measuring global CPU use and dividing it by the number of nodes. The disadvantage of this approach is that it is impossible to measure the specific CPU use of nodes in extreme situations, such as the densely populated city center. However, this approach was chosen because measuring CPU use for each process separately may have an undue influence on CPU use itself. Despite this approach, the results allow for an analysis of CPU use in different situations in Section 8.

Unlike CPU use, memory use is measured for each process separately, taken from /proc/<pid>/stat. Because of this, memory use is examined for only one topology per set of evaluation parameters rather than 10, allowing for an analysis of extreme cases. This approach is representative of all cases; while the topologies are generated randomly, they are still governed by the rules of the model in Section. 4. This is confirmed by comparing memory use per node across all the topologies; if the memory use of the most demanding nodes of each topology is compared, the standard deviation is 1.9%. For the least demanding nodes, the standard deviation is 2.1%.

Network traffic is measured globally from /proc/net/dev and divided by the number of nodes, for the same reasons CPU use is measured globally. In the *Neighbourhood Discovery* scenario, only the lo adapter is examined because all traffic is local, in the *Service Deployment* scenario only the traffic on the *eth0* adapter is measured, which indicates traffic between fog nodes and edge nodes.

8 Results

This section contains the results for the evaluations described in Section 7, along with a discussion of the results. The values in the presented charts are median values, with error bars representing minimum and maximum values.

8.1 Fog Discovery

Fig. 6 shows the memory required by the fog node service for different maximum distances and varying numbers of fog nodes. In the median cases, memory use increases linearly with the number of fog nodes in the topology, which is in line with the expected theoretical performance. However, the maximum distance has less of an effect than expected. This is due to the dynamic adjustment of the maximum distance for nodes in low density areas per Eq. 19. Remembering that for a maximum distance of 50, 118 fog nodes would be required, many fog nodes can not find any neighbours unless they use Eq. 19 to dynamically increase their maximum discovery distance. The results show that this mechanism works properly, increasing the maximum discovery distance significantly when required, to the point that fog nodes have a similar memory requirement for maximum distances of 50 and 100. This also explains

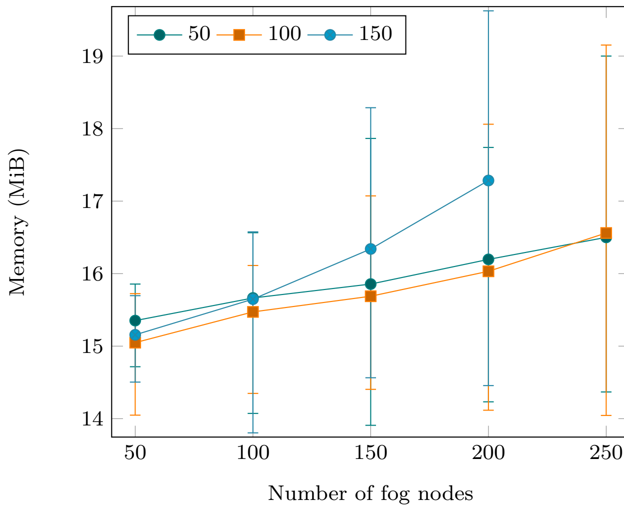


Fig. 6: Memory requirements of the fog service for different amounts of fog nodes, and maximum discovery distances of 50, 100 and 150 units.

why median memory use at these maximum distances barely rises with the number of fog nodes, since for a lot of nodes the dynamic increase in maximum distance will be lower, resulting in only slightly larger neighbourhoods on average. A maximum distance of 150 removes the effect of this mechanism entirely, causing a behavior similar to theoretical complexity.

The minimum cases all require around 14MiB memory, independent of the evaluation parameters. These are nodes in low density areas which have few neighbours even when the maximum distance is stretched. The maximum cases represent the nodes in the city center that have large amounts of neighbours regardless of maximum distance. Their memory use increases as expected with both maximum distance and number of fog nodes.

Overall, memory use is between 14MiB and 20Mib, which is low enough to deploy the fog service on low-power hardware with minimal resources.

The CPU use per node is shown in Fig. 7. Note that the results are represented in % of a single core. In the median case, CPU use is low, barely increasing with fog node density and increasing linearly with maximum distance. These results are better than expected from the theoretical performance, even in cases that are unaffected by dynamically adjusted maximum distance. The minimum and maximum cases indicate that overall performance can vary from about 50% to 500% of median performance. However, the extreme cases may be influenced by the evaluation itself, if the CPU use sample happens to coincide with a lot of process monitoring activity.

To give an indication of the spread of CPU use for nodes in different situations, the number of neighbours discovered by each node are monitored during the evaluation. For example, with 50 fog nodes and a maximum distance of

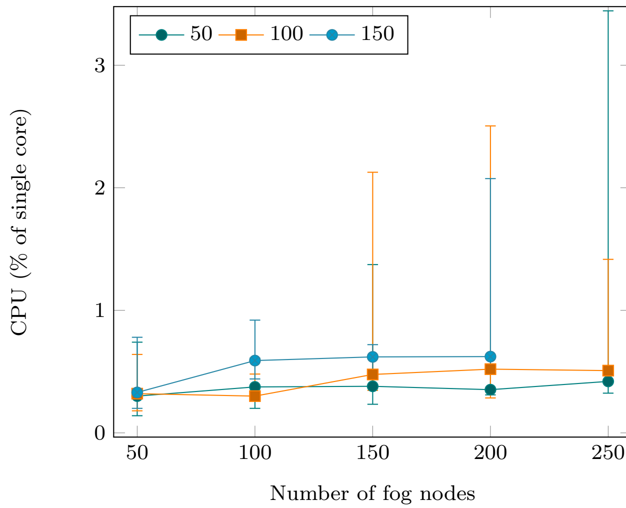


Fig. 7: CPU load of the fog service for different amounts of fog nodes, and maximum discovery distances of 50, 100 and 150 units.

50, each node has 1 to 4 neighbours, with a median of 2. In the case of 200 fog nodes and a maximum distance of 150, nodes have 1 to 84 neighbours, with a median of 38. Therefore, it is likely that the median cases in the chart also represent the median CPU use of nodes in each topology, and the most demanding nodes need about 200% of the CPU use of those cases.

In absolute numbers, the fog node service requires 0.3-0.6% of one CPU core. Although the evaluations are run on a relatively powerful CPU, the service should be able to run on low-power hardware with no performance problems.

Fig. 8 shows the network throughput per fog node related to neighbourhood discovery. These results are in line with the theoretical performance, although less pronounced than expected, with relatively small error margins caused by randomly generated topologies. Network throughput is less affected by dynamically adjusted maximum distance because the nodes with an increased maximum distance contact fewer other nodes overall. In turn, they have smaller lists of fog nodes known to them, causing both less and smaller responses during the discovery process. This means that while the memory use results show that the adjustment mechanism results in finding more neighbouring nodes, this does not necessarily result in higher network traffic for those nodes affected by it. In absolute numbers, network throughput is 30Kbps to 110Kbps.

In real life scenarios, the parameters of SoSwirly (e.g. maximum discovery distance, expected fog node density) can be tuned so that the number of discovered neighbours is similar to that in the evaluations. Certainly, no more than 100 immediate neighbours should be tracked by any fog node. In combi-

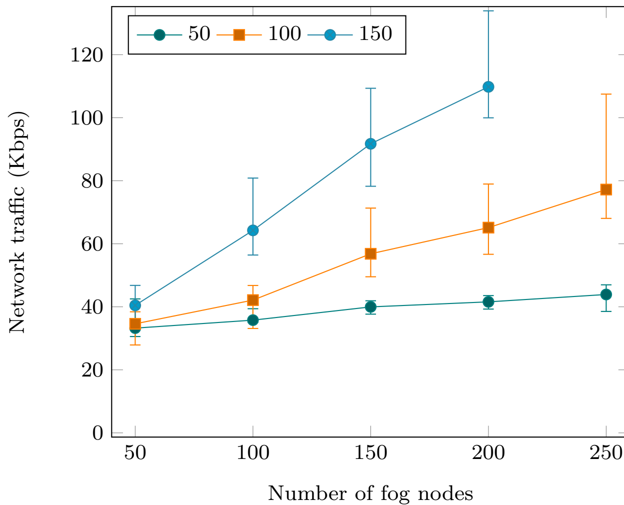


Fig. 8: Network traffic at a single fog node for different amounts of fog nodes, and maximum discovery distances of 50, 100 and 150 units.

nation with the results, this means that the fog service is capable of running on low-power devices with limited resources, and that it is highly scalable as long as node densities remain low enough.

For example, in Internet of Vehicles (IoV) applications using roadside units, the distance metric will likely involve vehicle velocity, and discovery rounds will be frequent due to rapidly changing relative distances. Because these roadside units are usually at the same distance from each other along well-defined trajectories and can be discovered in sequence, the maximum discovery distance for fog nodes (roadside units) can be lowered so each unit only discovers a few others in its neighbourhood. For fast moving edge nodes (cars), the maximum distance at which they can discover fog nodes can be increased to ensure they can reach sufficiently distant fog nodes in case a switch is required. This does not lead to performance problems, since the fog nodes themselves are sparsely connected.

The results indicate that, with proper tuning of parameters for a given node topology, the discovery algorithm behaves according to its theoretical performance, that it scales according to ***Req. 4***, and that the resource requirements are sufficiently low to satisfy ***Req. 5***.

Fig. 9 shows the accuracy of the neighbourhood discovery algorithm. This number represents how many nodes in its neighbourhood a fog node has actually discovered. The result represents a global number, indicating how many neighbours in total have been discovered compared to a perfect neighbour graph. A high accuracy is important, since it determines how well edge nodes can find an optimal service provider by traversing fog node neighbourhoods. Paradoxically, accuracy goes down as the number of fog nodes increases, espe-

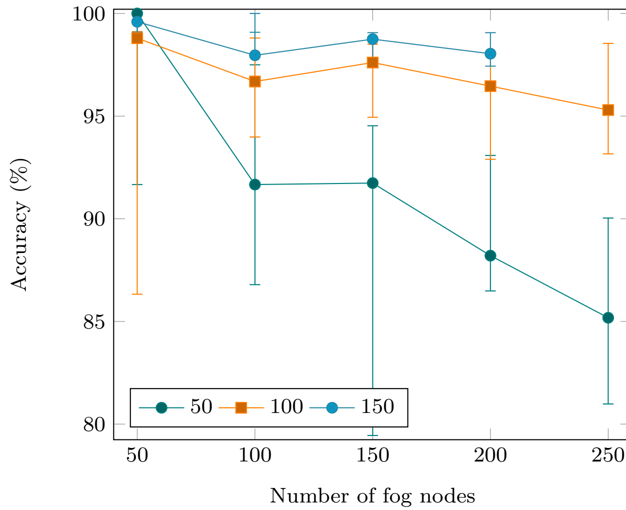


Fig. 9: Accuracy of neighbourhood discovery for different amounts of fog nodes and maximum discovery distances of 50, 100 and 150 units. 100% means a fog node discovered all other fog nodes within the maximum distance.

cially with a maximum distance of 50 where not even the dynamic adjustment mechanism can compensate. However, this is largely up to the random generation of swirls, since fog nodes are sometimes placed too far away from others, start their discovery process with nodes on the other side of the topology, or may encounter adverse features as shown in Fig. 2. For minimum distances over 100, accuracy is high enough that edge nodes can find an optimal service provider in over 95% of the cases, even under these adverse conditions.

8.2 Service Deployment

Fig. 10 shows the network traffic and CPU use observed during the service deployment evaluation. Although both network traffic and CPU use rise about 10% as 50% more edge nodes are added, this is likely the result of more fog nodes being activated to service the additional edge nodes, which in turn leads to more active fog nodes being discovered and tracked by each edge node. In absolute terms, the edge node service causes only 3Kbps network traffic and requires 0.2% of a single CPU core. Even considering the powerful processors in the evaluation setup, this shows that it can run on low-power, low-resource edge hardware. Memory requirements are similar to those of the fog node service, between 14 and 16.5MiB depending on local fog node density. In swirls with 150 edge nodes, only 0.5% more memory is required in the median case than in those with 100 edge nodes, showing that the number of edge nodes in a swirl has no significant effect on the memory use of the edge node service.

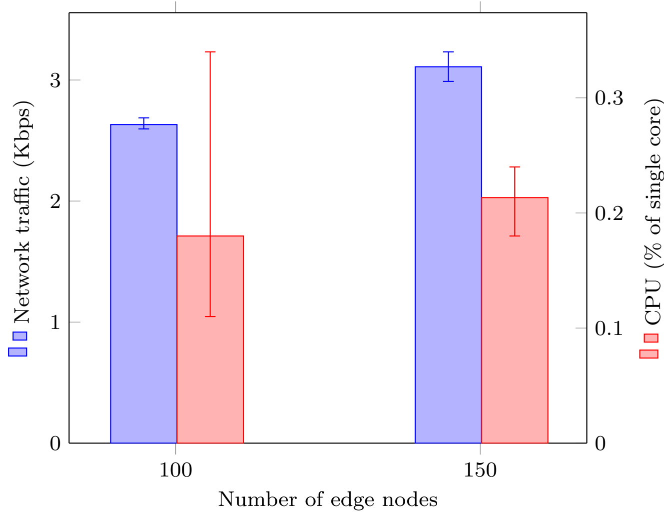


Fig. 10: Average network traffic at edge nodes for service topologies with 100 fog nodes, and 100 or 150 edge nodes.

These results indicate that service discovery and deployment by edge nodes scales according to ***Req. 4***, and that the resource requirements are low enough to meet ***Req. 5***.

8.3 Topology Efficiency

Fig. 11 shows how many fog nodes are activated by the evaluated algorithms in service topologies with either 100 or 150 edge nodes. SoSwirly is almost as efficient as Swirly, requiring 4% to 6% more nodes in the median cases. Considering that Swirly is a centralized approach, while in SoSwirly each node acts in its own best interests, a small efficiency penalty is to be expected. The results also show both Swirly and SoSwirly outperforming NSGA-II. Even after 5000 generations, NSGA-II activates about 10% more nodes than SoSwirly, showing that SoSwirly can generate good service topologies with only its limited knowledge about neighborhoods.

The distance between edge nodes and their service providing fog nodes is shown in Fig. 12. In the solutions generated by SoSwirly, median distance is only 1-2% higher than in the solutions of the centralized Swirly, and both are well below the maximum metric distance of 100. However, the maximum distances in the solutions generated by SoSwirly are almost 300% higher than those in topologies generated by Swirly. Note that this mostly concerns edge nodes that are beyond the maximum distance of any fog node to begin with. These cases are few and far between, and the enormous distances between them and their assigned fog nodes are always caused by adverse features in

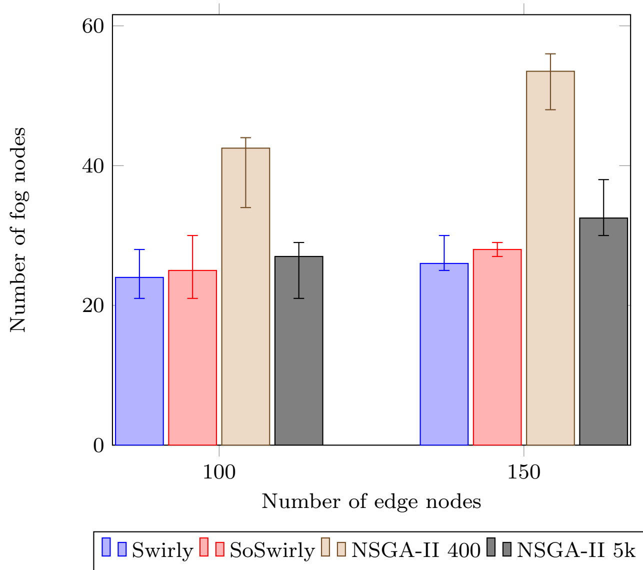


Fig. 11: Number of activated fog nodes in service topologies with 100 or 150 edge nodes, comparing SoSwirly to centralized algorithms. NSGA-II is evaluted at 400 and 5000 (5k) generations.

the node topology, combined with worst-case starting nodes for neighbourhood discovery, both issues of randomly generated topologies that can be fixed in real-life scenarios. Compared to NSGA-II after 5000 generations, the median distance in topologies generated by Swirly is about 30% lower, showing an overall better optimization when it comes to latency. However, the maximum distances are about 43% higher than those generated by NSGA-II (546 vs 382), owing to a lack of global knowledge about the topology.

Finally, Fig. 13 shows the times required for each algorithm to determine their solutions. Note that this chart uses a logarithmic scale to accommodate NSGA-II, and that the time measured for Swirly is somewhat unreliable because it is very close to 0ms. Compared to Swirly, SoSwirly requires about 100 times more processing time to optimize the service topology. However, in the case of SoSwirly, the measured time necessarily includes gathering fog node information via web service calls, whereas for Swirly and NSGA-II the information is already present. Nevertheless, all nodes of SoSwirly require between 20ms and 180ms to find an optimal service provider. Depending on the number of generations used by NSGA-II, it is around 100 to 1000 times slower than SoSwirly. Considering the efficiency of NSGA-II, this shows that SoSwirly can work significantly faster and more efficient than NSGA-II in the evaluated scenario.

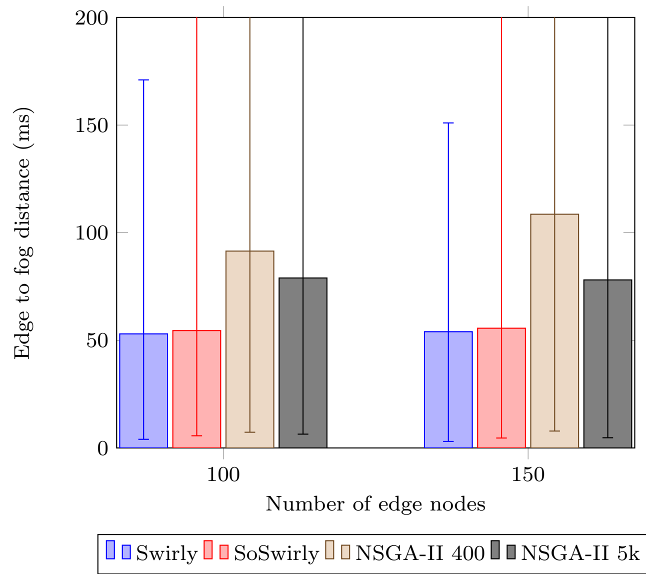


Fig. 12: Distance of edge nodes to fog nodes in service topologies with 100 or 150 edge nodes, comparing SoSwirly to centralized algorithms. NSGA-II is evaluted at 400 and 5000 (5k) generations.

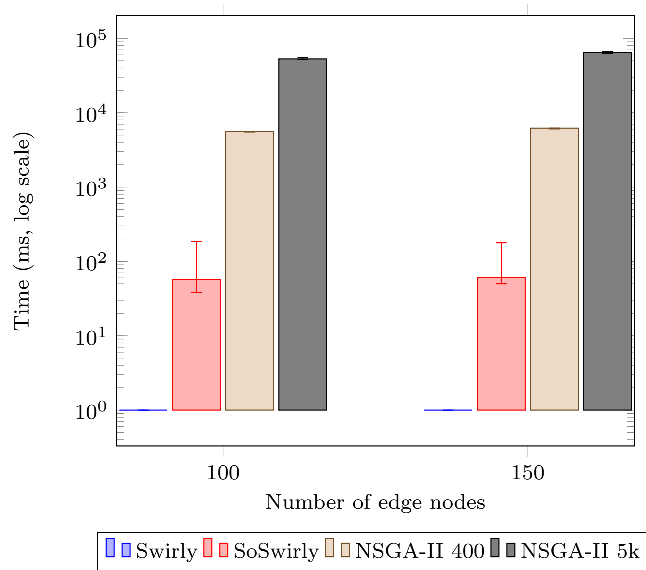


Fig. 13: Processing time required to organize service topologies with 100 or 150 edge nodes, comparing SoSwirly to centralized algorithms. NSGA-II is evaluted at 400 and 5000 (5k) generations.

9 Future Work

While this article presents a fully operational, self-contained solution for self-organizing fog service scheduling, there are several possible improvements and additions to both the concept and implementation.

SoSwirly relies heavily on passive monitoring of nodes to solve problems. For example, if a fog node becomes unavailable, this will be detected during the next discovery round. However, until that happens any requests to services on that node may fail. Active monitoring of the services used by an edge node can reduce the time required to find a new service provider, improving QoS.

In areas with high edge node densities, it can be useful to scale services instances within small but powerful fog nodes, rather than scaling the number of fog nodes. This is not possible with the implementation of SoSwirly presented in this article, but alternative implementations of the *ServiceLocator* can provide this functionality in the edge node service without modifying the core algorithms and components. However, the fog node service would need additional per-service parameters and resource monitoring to automatically scale them as their load changes. Alternatively, a container orchestrator that supports service scaling by default can be supported by implementing *Orchestrator*, to be used on fog nodes where it is applicable.

In general, edge nodes calculate the distance to many fog nodes in their neighbourhood during each discovery round, but most of this information is discarded after the closest nodes are found. If the discovery rounds happen sufficiently frequently, the historical distance data may be used to determine if the edge node, or any fog node, is moving, and at which time it would be advantageous to proactively switch to another fog node. However, as seen in Section 4, the exact coordinates of a fog node can not be known, so any solution to this must rely on reported distances alone.

Edge nodes determine their optimal service provider through discovery rounds, during which the distance to each eligible fog node is updated. To support true real-time updates, fog nodes could send distance updates to the edge nodes they service whenever a significant change in distances occurs. This would result in more CPU use on fog nodes, but would also allow edge nodes to instantly switch to another service provider if their current one is experiencing technical issues or is moving too far away.

10 Conclusion

In the introduction, the challenges of scheduling services with regards to the scope and hardware properties of the fog and edge are discussed, and a number of requirements for the presented solution are proposed.

SoSwirly, a fully decentralized fog service scheduler, is presented as a solution. A mathematical model of fog nodes and fog networks is presented, which forms the basis for an implementation of SoSwirly in Golang and an analysis of the fog nodes required for a specific swirl. It is explained how the

proposed requirements are met by the design of SoSwirly, and its theoretical performance is explored.

To verify its performance, SoSwirly is experimentally evaluated in terms of CPU use, memory use and network traffic. The results show that in all cases, performance is equal to or better than the theoretical performance, confirming that SoSwirly is highly scalable in geographical terms, as long as the maximum discovery distance and node densities are balanced at every location. It is explained that in some cases the dynamic scaling of maximum discovery distance inflates the memory use and CPU use of nodes in topologies with a small default discovery distance, and that the overall resource requirements are low enough to be deployed on a wide range of devices.

Further evaluations show that the node discovery algorithm is accurate enough for edge nodes to find their optimal service provider by traversing fog node neighbourhoods. Additionally, the service topologies generated by SoSwirly are generally as efficient as those generated by the centralized Swirly. Additionally, it is shown that SoSwirly can generate service topologies much faster than a generic algorithm such as NSGA-II, and that the topologies are more efficient than those generated by NSGA-II.

Some topics for future work are discussed, for example more extensive monitoring of node availability, scaling to multiple service instances on fog nodes and allowing edge nodes to change service providers proactively.

Acknowledgements 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 is made available on Github as described in the Evaluation Methodology section.

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.

Funding

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

Code Availability

All code related to the article is made available on Github as described in the Evaluation Methodology section.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, J.P. Jue, Journal of Systems Architecture **98**, 289 (2019). DOI 10.1016/j.sysarc.2019.02.009
2. E. Ismagilova, L. Hughes, Y.K. Dwivedi, K.R. Raman, International Journal of Information Management **47**, 88 (2019). DOI 10.1016/j.ijinfomgt.2019.01.004
3. F. Karatas, I. Korpeoglu, Future Generation Computer Systems **93**, 156 (2019). DOI 10.1016/j.future.2018.10.039
4. T. Goethals, F.D. Turck, B. Volckaert, in *Internet of Vehicles. Technologies and Services Toward Smart Cities* (Springer International Publishing, 2020), pp. 174–189. DOI 10.1007/978-3-030-38651-1_16
5. P.J. Maenhaut, B. Volckaert, V. Ongenae, F.D. Turck, Journal of Network and Systems Management **28**(2), 197 (2019). DOI 10.1007/s10922-019-09504-0
6. C. Avasalcai, I. Murturi, S. Dustdar. Edge and fog: A survey, use cases, and future challenges (2020). DOI 10.1002/9781119551713.ch2
7. B. Cheng, J. Fuerst, G. Solmaz, T. Sanada, in *2019 IEEE International Conference on Services Computing (SCC)* (IEEE, 2019). DOI 10.1109/scc.2019.00018
8. P.K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, G. Parmer, in *2019 38th Symposium on Reliable Distributed Systems (SRDS)* (IEEE, 2019). DOI 10.1109/srds47363.2019.00036
9. W. do Espirito Santo, R. de Souza Matos Junior, A. de Ribamar Lima Ribeiro, D.S. Silva, R. Santos, in *2019 15th International Conference on Network and Service Management (CNSM)* (IEEE, 2019). DOI 10.23919/cnsm46954.2019.9012677
10. A. Madhavapeddy, D.J. Scott, Communications of the ACM **57**(1), 61 (2014). DOI 10.1145/2541883.2541895
11. C. Guerrero, I. Lera, C. Juiz, Future Generation Computer Systems **97**, 131 (2019). DOI 10.1016/j.future.2019.02.056
12. M. Hosseinzadeh, H.K. Hama, M.Y. Ghafour, M. Masdari, O.H. Ahmed, H. Khezri, Journal of Network and Systems Management **28**(4), 1639 (2020). DOI 10.1007/s10922-020-09553-w
13. B. Stévant, J.L. Pazat, A. Blanc, in *Proceedings of the 10th International Conference on Cloud Computing and Services Science* (SCITEPRESS - Science and Technology Publications, 2020). DOI 10.5220/0009319902370244
14. Kubernetes. Production-grade container orchestration (2020). URL https://kubernetes.io/
15. O. Casquero, A. Armentia, I. Sarachaga, F. Perez, D. Orive, M. Marcos, in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (IEEE, 2019). DOI 10.1109/etfa.2019.8869219
16. Kubernetes. Building large clusters (2020). URL https://kubernetes.io/docs/setup/best-practices/cluster-large/
17. S. Vaton, O. Brun, M. Mouchet, P. Belzarena, I. Amigo, B.J. Prabhu, T. Chonavel, Journal of Network and Systems Management **27**(1), 188 (2018). DOI 10.1007/s10922-018-9464-1
18. J. Santos, T. Wauters, B. Volckaert, F.D. Turck, Entropy **20**(1), 4 (2017). DOI 10.3390/e20010004

19. P. Dawkins. Paul's online notes - section 4-8 : Change of variables (2020). URL https://tutorial.math.lamar.edu/Classes/CalcIII/ ChangeOfVariables.aspx
20. S. Waldman. Jacobian matrix (2020). URL https://math.wikia.org/wiki/Jacobian_matrix
21. B.F. Schutz, *Geometrical Methods of Mathematical Physics* (Cambridge University Press, 1980). DOI 10.1017/cbo9781139171540
22. Rancher. K3s - lightweight kubernetes (2020). URL https://k3s.io/
23. Kubernetes. Kubernetes api - podspec (2020). URL https://godoc.org/k8s.io/api/core/v1#PodSpec
24. T. Goethals, F.D. Turck, B. Volckaert, Journal of Cloud Computing **9**(1) (2020). DOI 10.1186/s13677-020-00180-z
25. Linux. Hosts - static table lookup for hostnames (2020). URL https://www.man7.org/linux/man-pages/man5/hosts.5.html
26. IDlab. Virtual wall (2020). URL https://www.ugent.be/ea/idlab/en/research/research-infrastructure/virtual-wall.htm
27. IETF. The geojson format (2020). URL https://tools.ietf.org/html/rfc7946
28. StatBel. The belgian statistical office (2020). URL https://statbel.fgov.be/en
29. StatBel. Statistical sectors (2020). URL https://statbel.fgov.be/en/open-data/statistical-sectors
30. StatBel. Population by statistical sector (2020). URL https://statbel.fgov.be/en/open-data?category=209
31. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, IEEE Transactions on Evolutionary Computation **6**(2), 182 (2002). DOI 10.1109/4235.996017

Author biographies

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 IDLab at Ghent University 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 Ghent University, Belgium and imec. He (co-) authored over 700 peer reviewed papers and his research interests include design of efficient softwarized network and cloud systems. 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 steering committee member of the IM, NOMS, CNSM and NetSoft conferences. Prof. Filip De Turck serves as Editor-in-Chief of IEEE Transactions on Network and Service Management (TNSM), and was recently elevated as an IEEE Fellow.

Bruno Volckaert is professor advanced programming and software engineering at IDLab (Ghent University) and senior researcher at imec. In 2006 he obtained a PhD on Grid resource management. Current research deals with reliable high performance distributed software systems for Smart Cities, scalable cybersecurity detection and autonomous optimization of cloud-based applications.