Pervasive and Mobile Computing 71 (2021) 101316

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Distributed algorithms based on proximity for self-organizing fog computing systems

Vasileios Karagiannis *, Stefan Schulte

Distributed Systems Group, TU Wien, Vienna, Austria

ARTICLE INFO

Article history:
Received 26 May 2020
Received in revised form 11 December 2020
Accepted 12 December 2020
Available online 23 December 2020

MSC:
00-01
99-00

Keywords:
Fog computing
Edge computing
Hierarchical structures
Flat structures
Self-organization
Internet of Things
IoT

ABSTRACT

Various performance benefits such as low latency and high bandwidth have turned fog computing into a well-accepted extension of the cloud computing paradigm. Many fog computing systems have been proposed so far, consisting of distributed compute nodes which are often organized hierarchically in layers. To achieve low latency, these systems commonly rely on the assumption that the nodes of adjacent layers reside close to each other. However, this assumption may not hold in fog computing systems that span over large geographical areas, due to the wide distribution of the nodes.

To avoid relying on this assumption, in this paper we design distributed algorithms whereby the compute nodes measure the network proximity to each other, and self-organize into a hierarchical or a flat structure accordingly. Moreover, we implement these algorithms on geographically distributed compute nodes, and we experiment with image processing and smart city use cases. Our results show that compared to alternative methods, the proposed algorithms decrease the communication latency of latency-sensitive processes by 27%–43%, and increase the available network bandwidth by 36%–86%. Furthermore, we analyze the scalability of our algorithms, and we show that a flat structure (i.e., without layers) scales better than the commonly used layered hierarchy due to generating less overhead when the size of the system grows.

1. Introduction

End-to-end communication between devices at the edge of the network and the cloud has been observed to create conditions with high round-trip times (i.e., high communication latency) and bottlenecks in the available network bandwidth [1]. Fog computing emerged to cope with such issues by exploiting the computational resources of the various compute nodes that span from the cloud to the edge of the network [2]. These nodes may belong to cloud operators (e.g., Amazon and Microsoft data centers) [3], network operators (e.g., networking devices with spare or added computational capabilities such as base stations and access points) [4,5], or privately-owned dedicated compute nodes at the edge of the network (e.g., cloudlets and fog nodes) [6]. By exploiting such nodes, fog computing aims at processing data, e.g., from Internet of Things (IoT) devices, close to the data sources [7]. This relieves the Internet backbone of network traffic and hinders the formation of network bottlenecks [8]. Moreover, since the data is processed close to the data sources, fog computing is able to provide low latency and high bandwidth [9,10].

Notable advances in creating fog computing systems propose a hierarchical structure with the cloud at the top, and various compute nodes organized in layers below, i.e., between the cloud and the IoT devices at the edge of the

* Corresponding author.
E-mail addresses: v.karagiannis@dsg.tuwien.ac.at (V. Karagiannis), s.schulte@dsg.tuwien.ac.at (S. Schulte).

https://doi.org/10.1016/j.pmcj.2020.101316

network [11–13]. In such structures, reduced communication latency is achieved by processing the data on the nodes of the lower layers. Notably, this latency reduction depends on the network proximity among the nodes of the adjacent layers, and does not come automatically due to the hierarchical organization of the nodes [11]. Thus, in distributed systems with nodes that may span over large geographical areas (such as in fog computing), measuring the network proximity and organizing the nodes accordingly, is necessary to ensure the efficiency of the hierarchical structure. However, most current approaches for creating hierarchical fog computing systems rely on hierarchical structures but do not take actual proximity measurements, and do not discuss how the nodes of such a structure are discovered and organized in a way that fulfills latency- and bandwidth-related goals (e.g., [14–16]). This might not pose a concern for statically configured small-scale systems, but may become a problem when the task of creating the system structure needs to be automated, e.g., for large-scale systems. A further repercussion of not considering how an efficient hierarchical structure can be created, is that the overhead associated with organizing the nodes into a hierarchy, i.e., the control messages required for creating appropriate logical links among the nodes of a fog computing system, is not analyzed. This repercussion may lead to scalability issues in case the overhead grows significantly (and causes network bottlenecks) when many nodes join the system (as shown in Section 4.3.5).

While the overhead of organizing the nodes of a fog computing system is usually not discussed in state-of-the-art research, there are approaches that try to circumvent the assumption that the nodes of adjacent layers reside close to each other. Prominent approaches employ location coordinates and communication latency to estimate the network proximity between compute nodes [17,18]. By using the communication latency as an indicator of network proximity (e.g., using round-trip times), the nodes measure the values of network latency to reach each other, and form logical links with the nodes of the lowest latency values [17]. However, latency measurements may not be a reliable indicator of network proximity since it can fluctuate, e.g., due to network bottlenecks [19]. When using location coordinates as an indicator of network proximity, the compute nodes are assumed to be preconfigured with coordinates that can be used for estimating the proximity between nodes, e.g., according to Euclidean distance [18]. Nevertheless, there is no guarantee that the distance from location coordinates corresponds to network proximity. In addition, preconfiguring each node with coordinates that actually correspond to network proximity, may become cumbersome in large-scale systems since the number of nodes might be too large to allow individual configuration for each node.

To tackle these problems, we propose self-organizing fog computing systems. A self-organizing fog computing system can be created using a distributed algorithm that runs on all the compute nodes of the system in order to execute the required control operations for achieving the desired system structure [20]. This counters the aforementioned problems in the following ways: *i*) The process of adding new nodes is defined (by the algorithm) which makes it possible to model the resulting overhead in order to examine scalability aspects. *ii*) Integrating network proximity measurement in the self-organization enables the nodes to self-organize based on their proximity to each other, and repeat this self-organization with each new node that joins the system. This ensures that even when the compute nodes are geographically distributed, the network proximity to each other is taken into account while creating the system structure.

Thus, in this paper, we design algorithms so that each time a new compute node joins the system, this new node and the existing nodes take network proximity measurements to each other, and based on these measurements they self-organize into a predefined structure (either hierarchical or flat) in a plug-and-play manner. The proposed algorithms are designed to be agnostic of the utilized network proximity measure. We do this to make our algorithms more flexible since in some fog computing systems, one proximity measure may be more appropriate (or available) than others. Nevertheless, we make sure that our algorithms can theoretically work with latency measurements and number of hops, although practically (i.e., in the evaluation in Section 4), we use only the number of hops as an indicator of network proximity. The reason we focus on these two measures is that they are typically associated with network proximity in end-to-end communication [1].

We position this work in the area of self-organizing algorithms for the coordination of distributed compute nodes [21]. Such algorithms aim at organizing a number of nodes that may become too large to be centrally coordinated, which finds use in pervasive computing, fog and edge computing, and IoT scenarios [21]. This is an important aspect in today's networks because more and more computations are performed on a potentially large set of compute nodes which require self-organization to achieve the desired system structure [22]. Potential use cases include, e.g., multimedia applications. For example, an image processing application may require computations for user-generated data with low communication latency, while utilizing a network of (non-mobile) compute nodes [21]. In case such applications are deployed on large-scale networks (such as the Internet), the compute nodes may be agnostic of the underlying physical network topology [23,24]. When two compute nodes communicate with each other, all the physical links on the path between these two nodes serve as a single logical link [25]. This logical link however, hides the information of network proximity. Consequently, a compute node may not know which other compute nodes reside in close network proximity, i.e., which other compute nodes can be reached with low communication latency. Therefore, in a network of compute nodes as shown in Fig. 1, assuming that the spatial distance between nodes in the figure corresponds to network proximity, the compute nodes can form various logical links to each other. Through these links, the nodes exchange data (initially acquired by a data source, e.g., images from a smartphone as shown in the figure) in order to perform the required computations in a distributed manner. However, some of these links may connect nodes in low network proximity (straight lines in Fig. 1), and bear lower communication latency than others that connect nodes in high network proximity (dashed lines in Fig. 1). Even though the network proximity between the compute nodes is abstracted by the logical links,

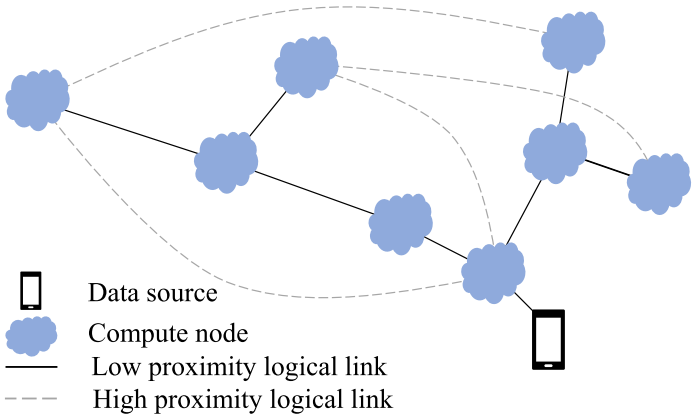


Fig. 1. Example of a system structure with low and high proximity logical links.

it is still possible for the nodes to take measurements, e.g., using round-trip times, or hop count. This way, each node can acquire information about the network proximity, and consequently, the communication latency to reach other nodes. Thus, while a simple approach may organize the participating compute nodes disregarding the network proximity between the nodes, a more advanced approach can take measurements and organize the compute nodes accordingly. This reduces the communication latency significantly (as shown in the evaluation in Section 4).

For this reason, in this paper we aim at enabling the compute nodes of fog computing systems to self-organize based on network proximity. The proposed algorithms enable the compute nodes to take network proximity measurements, and select logical links between nodes in close network proximity automatically. Our objective is to reduce the communication latency, to improve the quality of service of the applications, and to allow the system to scale to a large degree while incorporating various geographically distributed compute nodes, which can be particularly useful for systems that span over large geographical areas [26–29].

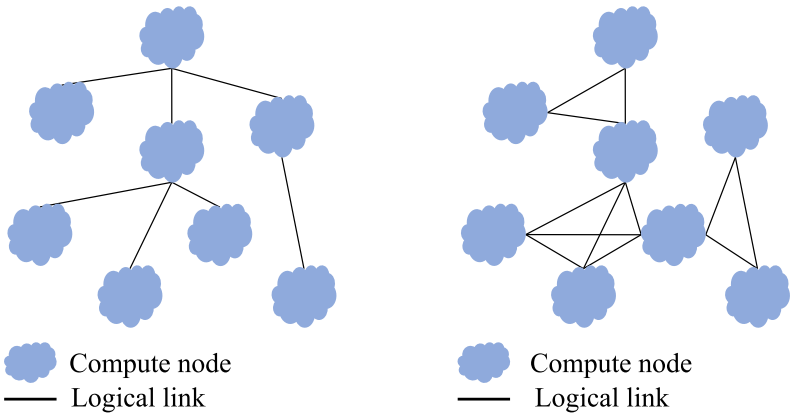
Notably, our approach can be used either as the basis for a novel fog computing system, or as a supplement to already proposed systems which rely on a hierarchical or flat system structure, but do not describe how to create it. Our contributions include the design and implementation of two distributed algorithms that enable the self-organization of the nodes of a fog computing system based on network proximity. Moreover, we implement the proposed algorithms, and we experiment with image processing and smart city use cases. Our results show that in fog computing systems with geographically distributed compute nodes, self-organization reduces the communication latency of latency-sensitive processes in both hierarchical and flat structures by 27%–43%, and increases the available network bandwidth by 36%–86% (compared to alternative state-of-the-art methods). Moreover, based on empirical results and by using predictive methods, we show that a flat structure is able to scale better than the commonly used layered hierarchy due to generating less overhead when the size of the system grows.

The rest of the paper is organized as follows: Section 2 provides a discussion of related work. Then, Section 3 presents the utilized system model along with the design and functionality of two distributed algorithms that enable the self-organization of the nodes of a fog computing system. Afterwards in Section 4, we deploy an implementation of these algorithms on geographically distributed compute nodes, and we discuss the differences between our approach and alternative methods. Finally, Section 5 concludes this work, and provides future research directions.

2. Related work

The work at hand investigates self-organizing fog computing systems that consist of geographically distributed compute nodes. Accordingly, related work is primarily from the field of fog computing system organization and establishment. Most related approaches in this field utilize either a hierarchical or a flat structure for organizing the various compute nodes of a fog computing system [16,30], as shown in Fig. 2. Notably, both Fig. 2(a) and Fig. 2(b) depict the exact same network of geographically distributed compute nodes, which is why the disposition of the nodes is identical. However, each approach may change the logical layout of the system (i.e., hierarchical or flat), and consequently, the incurred communication latency when executing applications (as shown in Section 4).

To present existing approaches in a comprehensive manner, we divide them into these two categories based on the utilized system structure. Specifically, fog computing systems that form hierarchical structures are presented in Section 2.1, and fog computing systems that form flat structures are discussed in Section 2.2. Afterwards, in Section 2.3, we discuss work from the field of self-organization and self-organizing clustering, which is also important related work for our approach.



(a) Hierarchical structure. (b) Flat structure.

Fig. 2. Examples of typical fog computing structures from the literature.

2.1. Hierarchical structures

Most current state-of-the-art research approaches for creating fog computing systems utilize a hierarchical structure. For instance, Kiani et al. [16] propose a hierarchical structure with geographically distributed compute nodes, and investigate related performance benefits based on simulations that consider either negligible or significant delay between compute nodes. Gao et al. [15] present a hierarchical fog computing system with dynamic resource allocation based on traffic prediction, and show latency reductions using extensive simulations. Adhikari et al. [14] propose a hierarchical fog computing system for executing applications based on priorities. In this system, it is assumed that there are gateways which receive tasks from IoT devices, and offload these tasks to the compute nodes of the hierarchy, while no algorithm to achieve this hierarchy is presented.

Even though such approaches can be used for creating hierarchical fog computing systems, they do not provide algorithms that take into account actual network proximity measurements. This may compromise the efficiency of the system, as discussed in Section 1. Moreover, none of these works provides results from real-world implementations with distributed compute nodes. In our work, we design algorithms for creating hierarchical fog computing systems based on network proximity, and we also implement them considering actual use cases. Furthermore, we use logic similar to the aforementioned approaches as a baseline (in Section 4.2), in order to show the differences of considering network proximity measurements in a real-world setting.

Regarding further related work, Skarlat et al. [18] propose the FogFrame framework for building a hierarchical structure with the cloud at the top, and various compute nodes below, which are organized in layers. To create this structure, FogFrame relies on a fog controller which decides where to place each new node that joins the system, based on the Euclidean distance from preconfigured location coordinates. Thus, this work relies on two assumptions: (*i*) that each node is preconfigured with location coordinates, and (*ii*) that the distance from these coordinates actually corresponds to network proximity. Instead of relying on preconfigured information, in our approach the nodes self-organize based on the proposed distributed algorithms. Moreover, the self-organization takes into account actual network proximity measurements (i.e., network hops, or latency measurements) in order to organize the geographically distributed compute nodes more efficiently.

Saurez et al. [31] present a fog computing system which consists of geographically distributed fog and cloud compute nodes. To create this system, each new compute node becomes a child (in the hierarchy) of a preexisting parent node. This way, while new compute nodes join, the system grows in a hierarchical manner. Even though this work targets geographically distributed compute nodes, the proposed algorithms do not consider an increasing number of nodes, which may create scalability concerns. In our work, we propose algorithms for organizing the compute nodes, and we examine the behavior of these algorithms when the system size grows.

Mortazavi et al. [32] present CloudPath which operates on a hierarchical structure due to assuming that network devices (e.g., routers or switches) between the edge and the cloud act as compute nodes. Thus, CloudPath does not actually create the hierarchical structure but nevertheless, the data is routed through a hierarchy with the edge nodes as leaves, the cloud as root, and various compute nodes in the middle. This approach does not measure the network proximity between nodes, but since compute nodes exist on the network path of the data, network proximity can be considered evident. Even though a similar assumption has also been proposed by Ascigil et al. [33], it should be noted that such approaches rely heavily on infrastructure support. For example, the network operator is required to route all the CloudPath traffic to specific nodes. In our approach, the nodes use distributed algorithms in order to self-organize into a (hierarchical or flat) structure without assuming cooperation from the network operator.

Nguyen et al. [34] present ICN-Fog which operates on a hierarchical (or flat) structure whereby each node has a logical link to the cloud, and also to compute nodes in proximity. The specific manner that this structure is created is not discussed, and the proximity between nearby nodes is assumed without any actual measurements. However, the authors mention that the communication of the nodes can be based on a wireless protocol, e.g., 6LoWPAN. Notably, assuming wireless communication simplifies the problem since the nodes can be organized based on discovery, e.g., each node forms logical links to the nodes in range. However, this solution does not apply to wide area distributed systems with nodes that are not within wireless range. Our approach on the other hand, aims at organizing the nodes of fog computing systems that may span over large geographical areas.

2.2. Flat structures

As mentioned above, most state-of-the-art systems in fog computing form hierarchical structures. Nevertheless, there are also approaches which utilize flat structures to organize the compute nodes of fog computing systems [16]. For instance, Casadei and Viroli [35] focus on the problem of coordinating distributed compute nodes at the edge of the network, and design an approach for dynamically partitioning the system in areas each one governed by an elected manager-node. Rabay'a et al. [36] present a fog computing system whereby the compute nodes are organized into a peer-to-peer overlay which provides performance benefits due to minimizing the interactions with the cloud. Yu et al. [37] present a fog computing system consisting of geographically distributed fog compute nodes which are organized into a flat-structured virtualization plane, and a hierarchical control plane, that aim at facilitating large-scale IoT.

Even though such approaches can be used for creating flat fog computing systems, they do not take into account the network proximity of the compute nodes. Moreover, none of the aforementioned works shows results from real-world implementations with geographically distributed compute nodes. In our work, we design algorithms for creating flat fog computing systems based on network proximity, and we also implement them in a real-world setting. Furthermore, we use logic similar to the aforementioned works as a baseline (in Section 4.2), in order to show the differences to our approach in real-world scenarios.

Regarding further related work, Song et al. [38] present PDS (Peer Data Sharing) which is a mechanism for organizing compute nodes at the edge of the network in order to exchange data. The organization of the nodes in PDS occurs opportunistically based on wireless discovery. Thus, this approach relies on wireless communication to enable the communication between nodes in proximity. Furthermore, PDS relies on the assumption that the target geographical area, and the number of participating compute nodes remain limited. Therefore, the authors do not address the problem of organizing the nodes of large-scale systems, or systems with nodes that span over large geographical areas. On the contrary, in our work we design algorithms aimed for geographically distributed compute nodes. Moreover, we consider large-scale systems by examining and evaluating scalability aspects.

Tato et al. [39] present Koala for organizing geographically distributed compute nodes. Each node in Koala maintains a routing table with the addresses of other nodes of the system. To create this table, every new node requests information from existing nodes, and measures proximity using latency and hop count. This work is presented only through experimental simulations. Hence, empirical results and technical information regarding measuring the network proximity between geographically distributed nodes are not provided. In our work, we address these matters, and we present the insights of our efforts. Moreover, we measure the overhead of organizing the nodes in order to analyze the system's scalability. This is not discussed in the context of Koala.

Jiang and Tsang [40] propose organizing the compute nodes of a fog computing system into an overlay network. This overlay is able to facilitate offloading of computational tasks to compute nodes, considering delay-related requirements in order to provide fog computing services. To account for the proximity between nodes, the proposed algorithms consider a function which outputs the disutility of the communication delay. In order to produce results for the evaluation of this work, the authors model the disutility function with a range of preconfigured values, and perform extensive experimental simulations. Therefore, this approach assumes that the communication delay between nodes is known a priori. In our work, we design algorithms that integrate network proximity measurements in order to create fog computing systems without any prior knowledge. Moreover, we deploy an implementation of our algorithms on geographically distributed compute nodes, and we show results from real-world fog computing systems.

Finally, our former work [41] proposes organizing the compute nodes of a fog computing system into a flat structure. To build this structure, each new node requests information about the system from existing nodes, and measures the network proximity using hop count. Again, this approach is presented only through experimental simulations. Thus, no information is provided on how to actually build a fog computing system consisting of geographically distributed compute nodes. Moreover, our former work does not produce any results to examine the effect of using hop count as an indicator of network proximity, or for comparing hierarchical to flat structures. In the work at hand, we design algorithms for both structures, and we elaborate on their differences.

To sum up the discussion, Table 1 provides an overview of the related work. As shown on this table, there are various approaches in the literature that can be used for organizing the compute nodes of a fog computing system into a hierarchical or a flat structure. However, most of these works do not consider actual network proximity measurements, and do not provide a scalability analysis. Furthermore, to the best of our knowledge, none of them provides a comparison of the two structures considering real-world scenarios with geographically distributed compute nodes. In the work at hand, we design algorithms for hierarchical and flat self-organization that target fog computing systems which span over large geographical areas, and we also compare them regarding communication latency, network bandwidth, and scalability.

Table 1
Overview of related work.

	Hierarchical structure	Flat structure	Proximity measurements based on	Scalability analysis	Presented system
Adhikari et al. [14]	✔				Simulated
Gao et al. [15]	✔				Simulated
Kiani et al. [16]	✔				Simulated
Skarlat et al. [18]	✔		Coordinates		Implemented
Saurez et al. [31]	✔		Coordinates		Implemented
Mortazavi et al. [32]	✔				Emulated
Nguyen et al. [34]	✔	✔			Not defined
Casadei and Viroli [35]		✔		✔	Simulated
Rabay'a et al. [36]		✔			Simulated
Yu et al. [37]	✔	✔			Not defined
Song et al. [38]		✔			Implemented
Tato et al. [39]		✔	Hops, latency	✔	Simulated
Jiang and Tsang [40]		✔			Simulated
Karagiannis et al. [41]		✔	Hops	✔	Simulated
Our work	✔	✔	Hops, latency	✔	Implemented

2.3. Self-organization

There is also work in the literature that does not target fog computing systems explicitly, but provides self-organizing algorithms for the organization of distributed nodes, and is therefore of relevance to our work. For example, Casadei et al. [21] propose creating a system structure for edge computing, using a self-organizing approach that divides the nodes into regions, and each region is coordinated by a leader node. However, such edge computing approaches focus on nodes that reside close to each other, and do not consider interactions with nodes in distant geographical areas, e.g., with remote cloud nodes. Leitão et al. [42] propose a distributed algorithm for peer-to-peer communication that allows multiple nodes to self-organize into groups which are referred to as super-peers. Nevertheless, such peer-to-peer approaches aim at facilitating information exchange among distributed nodes, and do not focus on leveraging the network proximity between the nodes in order to lower the latency, which is a prime goal of fog computing. Audrito et al. [22] propose a gradient algorithm for finding paths between nodes in order to create efficient self-organizing systems. In this approach, each node interacts with the other nodes through broadcasts. Thus, it is assumed that each node communicates with neighbor nodes without knowing their network addresses, since a broadcast reaches all the other nodes that are listening [43]. However, such approaches are not applicable to geographically distributed compute nodes that communicate over the Internet, which is our target environment. The reason is that Internet protocols such as TCP, provide end-to-end communication, and do not support broadcast operations (i.e., sending messages without knowing the destination address) [44,45].

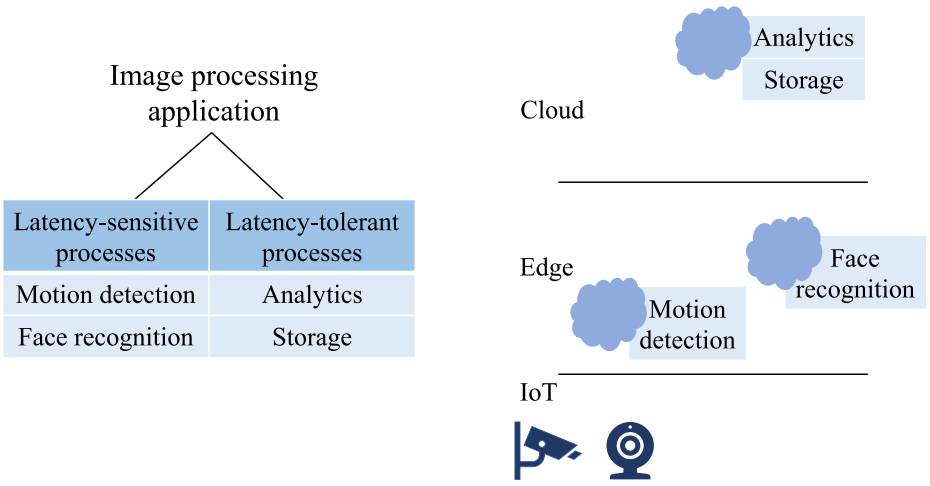
Additionally, there is related work from the field of self-organizing clustering in ad hoc networks. Ad hoc refers to self-organizing networks with a high degree of mobility which causes the network's topology to change rapidly and unpredictably [46,47]. For example, Liao et al. [48] propose a self-organizing clustering algorithm to enable hierarchical communication among nodes in wireless sensor networks. Johnen et al. [46] present a clustering algorithm that focuses on stability by ensuring that the network converges towards the desired system structure. Mitton et al. [47] propose a stabilization algorithm for clustering in ad hoc networks, which considers scalability aspects. However, the discussed approaches rely on the ability of the nodes to discover all the other nodes within wireless range. This does not apply to geographically distributed compute nodes that communicate with each other over the Internet (i.e., using also wired means), which is the scope of our work.

3. Organization of compute nodes in fog computing systems

In this section, we present the design and functionality of two algorithms for organizing the geographically distributed compute nodes of a fog computing system. To this end, we provide the application model in Section 3.1, the system model in Section 3.2, and a description of the proposed system functionality in Section 3.3. Afterwards, we present algorithms for organizing geographically distributed compute nodes into a hierarchical or a flat structure in Sections 3.4 and 3.5, respectively.

3.1. Application model

Our application model is based on a widely-used fog computing model from the literature [49]. In order to leverage the multiple compute nodes of a fog computing system, every application is executed in a distributed manner by many



(a) The processes of an image processing application.

(b) The placement of the processes on distributed compute nodes.

Fig. 3. Example of the application model based on an image processing application.

nodes [50]. For this reason, each application is assumed to consist of multiple processes. These processes are independently deployable, but may interact with each other through dependencies (i.e., the output of one process may become the input for another) in order to execute the functionality of the application as a process composition [49]. Thus, while running, each process performs application-related tasks such as: analytics, machine learning, aggregating, storage, etc. The input data of these processes is acquired from the data sources which can be either devices which integrated sensors (e.g., from the IoT) [51], or other processes. The final output of the processes can be a command for storage, notification, or actuation of a device that integrates an actuator.

Since fog computing aims at facilitating applications with processes that may require low communication latency, we distinguish the processes based on latency requirements [49]. Therefore, a process can be either latency-sensitive or latency-tolerant, e.g., for actuation commands or for sensor monitoring, respectively [52]. This distinction enables the handling of different processes in different ways. In general, latency-sensitive processes should be deployed on compute nodes close to the data source where the communication latency is low. Latency-tolerant processes on the other hand, can also be deployed on nodes outside the proximity of the data source, e.g., in a remote cloud node. For example, Fig. 3 shows the application model of an image processing application. Specifically, Fig. 3(a) shows the processes of the application, and Fig. 3(b) shows the placement of these processes on distributed compute nodes that span from the cloud to the edge of the network.

A more concrete example of such an image processing application (as shown in Fig. 3), is a smart doorbell. Such an application can operate as follows: First, the doorbell camera (as shown at the bottom of Fig. 3(b)) captures the image feed periodically, and sends pictures to a nearby compute node that executes the motion detection process. When this process detects the motion of a person, the image with the person is sent to the face recognition process. Then, the face recognition process retrieves the faces of the house residents from the cloud storage, and compares these faces with the image from the motion detection process. If there is a match, an actuation command is sent back in order to open the door automatically. Furthermore, the face recognition process sends all the comparisons to the cloud for storage and analytics purposes, e.g., to maintain a record of visitors. Notably, a smart doorbell may not be an application with stringent low latency requirements. Nevertheless, executing some of the involved processes with low latency (e.g., in order to open the door quickly in the face of residents), may increase the quality of service significantly. Other processes on the other hand, might not be affected by latency (e.g., performing monthly analytics regarding visitors). Thus, as latency-sensitive we consider the processes for which lower latency may result in higher quality of service, and as latency-tolerant we consider the processes for which lower latency does not affect the quality of service significantly. Hence, using this system model, and distinguishing between latency-sensitive and latency-tolerant processes, may benefit various IoT applications including those for which low latency is not a functional requirement (such as the smart doorbell application).

There are various approaches for the placement of the processes on the available compute nodes, most of which are based on optimization methods [53,54]. Such approaches can be considered complementary to the work at hand, since the system structure and the latency between nodes, are usually required as input of the placement algorithms [55,56].

Table 2
The basic notation of the system model.

Symbol	Description
$n_1, n_2, \ldots, n_N$	Set of all the compute nodes of the system.
n_{new}	New node to be added to the system.
n_{cp}	Existing node to be used as contact point.
n_{pro}	Existing node closest to n_{new}.
n_{far}	Existing node farthest away from its neighbors.
S	Maximum group size.
G_{new}	New group for adding n_{new} and n_{cp}.
G_{cp}	The group of the contact node n_{cp}.
G_{upd}	Updated group that includes n_{new}.
$P(n_i, n_j)$	Network proximity between n_i and n_j.

3.2. System model

Our system model is based on well-known system models from the literature [2,3]: We define a fog computing system as an environment that includes multiple geographically distributed compute nodes that span from the cloud to the edge of the network, along with resource-constrained devices which integrate sensors and actuators [51]. We refer to such devices as IoT devices [57].

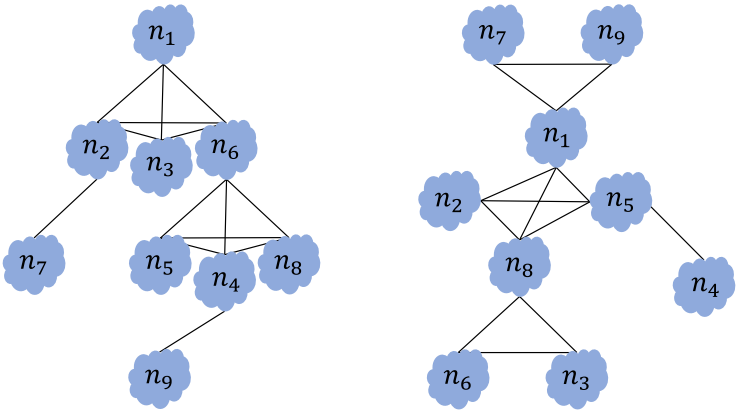
The amount of computational resources that each compute node provides can vary, commonly but not exclusively, based on the location of the node. This means that the compute nodes closer to the edge tend to integrate less resources than those closer to (or at) the cloud [50]. Typically, when a node integrates a large amount of computational resources, access to these resources is granted through dynamically created virtual machines which retain only a fraction of the initial computing power. In this work, we consider a compute node as a set of computational resources either physical or virtual, which are provided through an interface that can be accessed using the node's address (i.e., IP address and port number).

Moreover, we make the following assumptions: Since we target geographically distributed compute nodes, we assume that these nodes communicate with each other through the Internet. We also assume that the first compute node of the system, which is also the root in hierarchical structures (e.g., the top node in Fig. 2(a)) is always a cloud compute node, because the cloud can provide a global contact point with a static address that can be used by other nodes that want to join the system. Thus, all nodes are assumed to know this address, and use it to bootstrap the communication with the fog computing system. Additionally, we assume that all the compute nodes are non-mobile (i.e., the addresses of the nodes do not change during the runtime of the system), while the data sources might be mobile [21]. For the case that nodes join and leave (i.e., fail or disconnect) the system concurrently, we precondition a mechanism that maintains the system structure, as discussed in previous work [41]. By doing this, we decouple the problem of handling unexpected joining and leaving of compute nodes that are highly dynamic, and we focus on the problem of creating efficient system structures. We therefore assume that multiple compute nodes join the system sequentially in a transactional manner.

In the following, we provide the basic notation of the system model, which is also summarized in Table 2. The compute nodes that intend to join the system are denoted as $n_1, n_2, \ldots, n_N$. The first compute node n_1 is used as the initial contact point for other nodes to join. Each new node that requests to join the system, is denoted as n_{new}. The existing node that acts as the contact point is noted as n_{cp}.

In order to communicate with the rest of the compute nodes in the system, each node needs to select neighbors. Neighbors are defined as two nodes that are connected by a logical link, i.e., they know each other's addresses. Nodes may have more than one neighbor: When many nodes know each other's addresses, they are considered a group. A group is defined as a complete graph (or clique) of nodes, i.e., every pair of nodes in a group is connected by a logical link. For instance, the nodes n_1, n_2, n_3, and n_6 in Fig. 4(a) (and also the nodes n_1, n_7, and n_9 in Fig. 4(b)) belong to the same group, and consider each other neighbors. Groups are assumed to have a fixed maximum size S which can be used for controlling the layout of the structure. In Fig. 4(a) for example, $S = 4$ ensures a hierarchical tree-like structure with groups that can have up to 4 nodes. Furthermore, maintaining a fixed S for both hierarchical and flat structures enables creating comparable fog computing systems which differ only in the utilized system structure. This aids in drawing valuable conclusions regarding the benefits of each structure, as discussed in Section 4.

When a new group of nodes is created, e.g., due to new nodes joining the system, it is referred to as G_{new}. In order to communicate with other compute nodes, each node joins (at least) one group (which includes neighbors), and all the groups together constitute the fog computing structure, as shown in Fig. 4. Thus, a structure of nodes is defined as the high-level architecture of the system, which shows the manner whereby the compute nodes are interconnected and communicate with each other (as shown in Fig. 4) [20]. A hierarchical structure has a root (e.g., n_1 in Fig. 4(a)), and the nodes are organized in layers, whereas a flat structure does not have a root or layers [20]. Further information on hierarchical and flat structures is provided in Sections 3.4 and 3.5, respectively. The goal when creating either structure, is to select neighbors in close network proximity for each node, so that the communication through the structure occurs with low latency and high bandwidth.



(a) Hierarchical (b) Flat

Fig. 4. Examples of fog computing structures according to the proposed system model (when the compute nodes join sequentially, i.e., n_1 is the first, n_2 is the second, n_3 is the third, etc).

For organizing the compute nodes based on network proximity, which may be essential to fog computing systems (as discussed in Section 1), the following additional notation is used. The network proximity between two nodes n_i and n_j is denoted as $P(n_i, n_j)$, and represents the network distance, e.g., based on network latency (round-trip or one-way), or the number of hops, between n_i and n_j. The selection and implementation of specific network proximity indicators (i.e., number of hops, or network latency), are discussed further in our evaluation in Section 4.1. Thus, when we mention the proximity between compute nodes hereinafter, we refer to network proximity.

According to the proposed algorithms (which are presented in Sections 3.4 and 3.5), when joining, n_{new} selects the group that contains the node of the closest proximity, which is referred to as n_{pro}. When a group exceeds the capacity S, the node that resides the farthest away from the others (within the group), noted as n_{far}, leaves the group. For a group of size S, when the nodes of the group are $n_1, n_2, \ldots, n_S$, for n_{far} applies:

$$\sum_{i=1}^{S} P(n_{far}, n_i) = \max\left\{\sum_{i=1}^{S} P(n_1, n_i), \ldots, \sum_{i=1}^{S} P(n_S, n_i)\right\}$$

This means that n_{far} is the node for which the summary of the proximity values towards the rest of the nodes in the group, is the maximum among all the nodes of the group. Since all the neighbors have the exact same view of the group they belong to, even if this metric points to more than one node, it is possible to have a common strategy for selecting a unique n_{far}, e.g., using first occurrence.

3.3. Proposed system functionality

In Fig. 5, we show the high-level architecture of a compute node, along with the interactions with its neighbors, and with IoT devices. The IoT devices and the compute nodes of the system can be geographically distributed in various locations, but are expected to communicate with each other through the Internet. The goal is to process the generated IoT data by nodes in proximity thereby with low latency and high bandwidth. To execute an application, the IoT devices send the IoT data to nearby compute nodes which use this data as input for the hosted processes. In order to reach other nodes which reside farther away, the executed processes send data to the neighbor nodes, which is how data travels through the system.

To achieve this functionality, each compute node has four internal components, as shown in Fig. 5. The *Communication* component is used for the communication with other nodes and with the IoT devices. The *Self-organizing Algorithm* component runs the proposed algorithms in order to handle the self-organization of the compute nodes, and to select neighbors. The neighbors are stored in the *Groups* along with the proximity measurements. The *Processes* component runs the processes hosted by the node, and uses the Groups in order to send data to the neighbors (through the Communication component). Notably, the granularity of the components aims at facilitating the compatibility of alternative fog computing systems (e.g., [18] or [40]) with our algorithms. To achieve this, the Groups are also accessible through the Communication component. This way, an external component that handles the execution of the processes can also leverage the groups in order to discover compute nodes in proximity, and to abide by the system structure of the utilized algorithm (implemented in the Self-organizing Algorithm component). In the following, we design two distributed algorithms (one that results in a hierarchical structure, and one that results in a flat structure) which can be used as the logic of the Self-organizing Algorithm component.

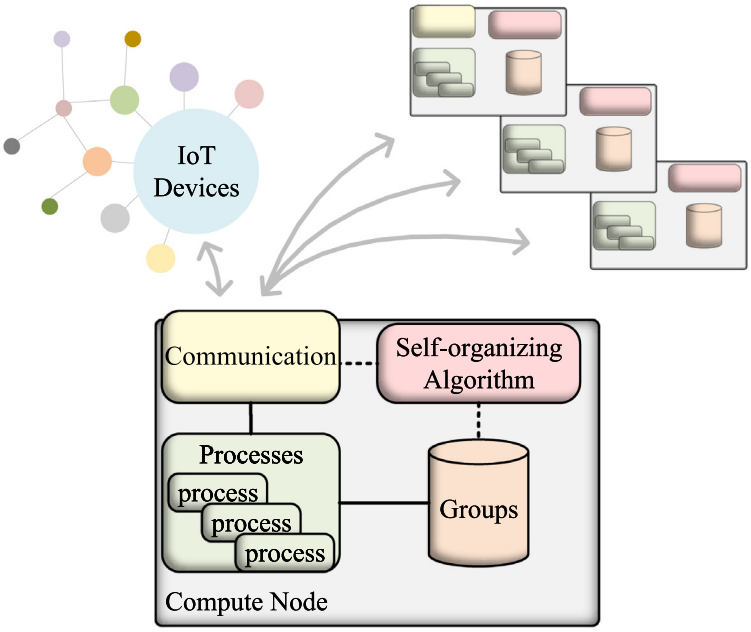


Fig. 5. High-level architecture of a compute node (dotted lines signify self-organization operations, whereas straight lines denote application execution operations) in the proposed fog computing system (lines with arrows signify interactions with neighbors and with IoT devices).

3.4. Hierarchical structures

As discussed in Section 2, various approaches have been proposed for organizing the compute nodes of a fog computing system in a hierarchical manner. In this section, we further advance the typical hierarchical structure by designing a distributed algorithm for the self-organization of geographically distributed compute nodes based on proximity. This can be essential for fog computing systems that span over large geographical areas since in such cases, close proximity cannot be assumed, as discussed in Section 1.

In the hierarchical structure, the compute nodes form a layered hierarchy, and communicate with the neighbors of adjacent layers, as shown in Fig. 4(a). Thus, by sending data to neighbors, the data travels up/down the hierarchy. Notably, Fig. 4(a) shows nine compute nodes that have joined the system sequentially (i.e., n_1 is the first, n_2 is the second, n_3 is the third, etc.). However, the arrangement of these nodes seems abstract. This can be the effect of organizing the compute nodes based on proximity. Every time a new node joins, this new node along with existing nodes self-organize, so that the new compute node has neighbors in close proximity. In the following, we define some additional notation which is exclusive to the hierarchical structure, and then we show how such a structure can be created.

In addition to the already mentioned definitions (presented in Section 3.2), for the hierarchical structure we use the terms parent, sibling (i.e., same parent), and child for neighbors that reside above, next to, or below a node, respectively. Notably, in the hierarchical structure each node (e.g., n_6 in Fig. 4(a)) belongs to (at most) two groups: one that includes a parent and siblings (e.g., n_1, n_2, n_3, n_6), and one that includes children (e.g., n_6, n_5, n_4, n_8). Exceptions are the root of the hierarchy, which belongs to one group due to not having a parent or siblings (e.g., n_1), and the leaves of the hierarchy, which belong to one group due to not having children (e.g., n_7, n_3, n_5, n_9, n_8).

To organize the compute nodes of a fog computing system into a hierarchical structure based on proximity, we design HPI (Hierarchical Proximity-Integrated) which is described in Algorithm 1 from the perspective of a new compute node which requests to join the system. HPI operates as follows: Initially, there is only one node which acts as the contact point n_{cp} for each new node n_{new} (Line 1). Upon request (Line 2), n_{cp} examines the group of its children. If this group is empty, then n_{cp} sends back an empty response (Line 3) which triggers n_{new} to create a new group with n_{cp} and n_{new} (Line 4), and then to notify n_{cp} (Line 5). This means that n_{new} is added as the first child of n_{cp}.

If the group of children of n_{cp} is not empty, n_{cp} responds with the group of its children (Line 6). Then, n_{new} joins this group as a child of n_{cp} (Line 7), and notifies the neighbors about the arrival of a new node (Line 8), i.e., the neighbors also add n_{new}. The arrival of a new node triggers all the siblings of the group to count the neighbors in the group that n_{new} joined. If the number of nodes in this group is smaller or equal to S, then n_{new} stays at its current position as a child of n_{cp} (Line 17). However, if the number of nodes is greater than S, i.e., the group exceeds its capacity (Line 9), then all the nodes of this group calculate the sibling that resides the farthest away from the others, i.e., n_{far} (Line 10), and remove n_{far} from the group (Line 15). Notably, the same functionality can be achieved if only one sibling calculates n_{far}, and shares this information with the rest of the nodes in the group. The difference is that the latter case utilizes less

Algorithm 1: HPI (Hierarchical Proximity-Integrated)

```
1  Procedure: joinSystemHPI(Address n_cp)
2      joinResponse = joinRequest(n_cp)
3      if joinResponse.isEmpty() then
4          G_new = new Group(n_new, n_cp)
5          notifyNeighbors(G_new)
6      else G_cp = joinResponse.getGroup()
7          addNeighbors(G_cp)
8          notifyNeighbors(G_cp)
9          if G_cp.size > S then
10             n_far = findNFar(G_cp)
11             if n_far == n_new then
12                 n_pro = findNPro(G_cp)
13                 joinSystemHPI(n_pro)
14             else
15                 G_cp.remove(n_far)
16             end
17         end
18     end
```

Algorithm 2: Perspective of existing nodes

```
1  Procedure: uponJoin(Group G_upd, Address n_cp)
2      if G_upd.size > S then
3          n_far = findNFar(G_upd)
4          G_upd.remove(n_far)
5          if n_far == this then
6              G_upd.remove()
7              joinSystem(Address n_cp)
8          end
9      end
```

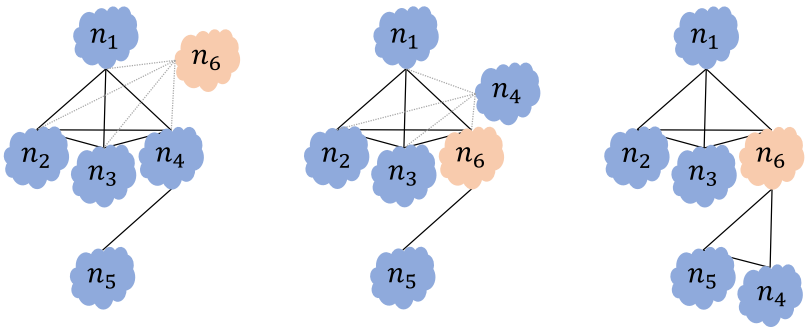
computational resources (because only one node calculates n_{far}), but more network resources (because the address of n_{far} needs to be sent to all the nodes in the group). Thus, since in such systems it is more common that the network becomes a bottleneck, and not the lack of computational resources, we propose that all nodes calculate n_{far} so that no additional network overhead is induced [58].

At this point, there are two cases for n_{far}. If n_{far} is the same as n_{new} (Line 11), then n_{far} finds the sibling with the closest proximity n_{pro} (Line 12), departs from its current group, and requests to join the system again using n_{pro} as contact point (Line 13). This way, n_{new} travels downwards the hierarchical structure, in order to find a group with nodes in close proximity. The second case is that n_{far} is one of the other siblings (i.e., not n_{new}). In this case, n_{far} departs from its current group, and joins the system again using the root as contact point. This way, the most distant node of each group departs and joins again from the root of the hierarchy in order to find siblings in close proximity.

Algorithm 2 shows what happens according to HPI from the perspective of the existing nodes. Upon the arrival of a new node, all the neighbors receive the updated group (Line 1) which is sent by the new node. Then, each node counts the neighbors in order to check if the group exceeds the capacity S (Line 2). If the capacity of the group is exceeded, then all nodes find n_{far} (Line 3), and remove it from the group (Line 4). Subsequently, the node that is n_{far} (Line 5) deletes the group (Line 6), and requests to join the system again using the initial contact point (Line 7), i.e., the root of the system.

To make the process of adding new compute nodes more comprehensible, Fig. 6 shows a system with 5 compute nodes, i.e., $n_1, n_2, \ldots, n_5$, and how a new compute node n_6 joins according to Algorithm 1. Initially, as shown in Fig. 6(a), n_6 requests to join from the root of the system. Then, n_1 responds with the addresses of its children (i.e., n_2, n_3, n_4), n_6 joins this group, and the neighbors (i.e., n_1, n_2, n_3, n_4) also add n_6. The addition of a new node triggers all the neighbors to count the size of the group. In this case, the size is 5 (i.e., larger than the maximum group size S that equals 4), which means that all nodes calculate n_{far} which is n_4. Thus, the new compute node n_6 stays in this group, and n_4 leaves and requests to join again through the root n_1, as shown in Fig. 6(b). Since $n_4 == n_{far}$ in the group of n_1, n_4 requests to join through the closest sibling which is n_6. Then, n_4 joins the group of n_6 as a child, as shown in Fig. 6(c). The size of this group is 3, i.e., smaller than S, which means that no further actions are required and the algorithm terminates.

This algorithm raises the question of why n_{far} is selected among the siblings, while the parent of the group is not considered. This is done in order to avoid that the following scenario occurs: n_{new} joins a group of the structure. Then, the parent n_{par} of that group, being the farthest away from its neighbors (i.e., n_{par} is calculated as n_{far}), leaves the group while n_{new} takes its place as parent. This results in n_{par} joining again through the root. If this happens, it might be the case that n_{par} ends up in the group in which n_{new} is a child. If in this group n_{new} is calculated as n_{far}, n_{par} will take the place of n_{new}, and n_{new} will join again through the root. This is exactly the initial state of the structure, which means that the same process will be repeated in a loop. Thus, n_{new} and n_{par} will continue leaving and joining the system indefinitely.



(a) New node n_6 requests to join from n_1, and joins group that includes n_1, n_2, n_3, and n_4.

(b) All nodes calculate n_{far} which is n_4. n_4 requests to join from the sibling in closest proximity n_6.

(c) n_4 joins the group as a child of n_6. Since the size of this group is smaller than S, the algorithm terminates.

Fig. 6. Example of the process of HPI step-by-step (for $S = 4$).

To avoid this, n_{cp} is prohibited from being considered in the calculation of n_{far}, thereby ensuring the normal operation of the algorithm.

A system that implements HPI self-stabilizes when all the operations that are triggered by a new compute node finish. The number of these operations depends on the system size. As described by Algorithm 1, new compute nodes start from the root, and travel downwards the hierarchy until a group with nodes in proximity is found, or a new group is created. Thus, the larger the number of layers in the hierarchy, the larger the number of groups that may need to be examined before the algorithm finishes. This means that a hierarchical structure that is long in depth may require more operations. Notably, what determines the number of layers in the hierarchy is the proximity of the nodes. If many nodes from the same location join, it is likely that every new node will follow the path of the previous ones (while traveling downwards the hierarchy), and force the hierarchical structure to grow in depth. However, assuming that the nodes are geographically scattered, the hierarchy will grow first in breadth and then in depth because the new nodes will follow different paths. This limits the operations required for the algorithm to finish, and the system stabilizes. Further evidence regarding the stabilization of a system that follows the approach of HPI is provided in Section 4.3.6.

Another aspect of HPI that should be discussed, is the distribution of the computational resources within groups. This is an important aspect because even though the data can travel up/down the hierarchical structure (as previously discussed), it is the compute nodes within the same group that provide additional computational resources with the lowest communication latency. Therefore, if a group has a high amount of computational resources (considering the summary of the resources of all the compute nodes within the group), it might be the case that this amount is enough to meet the requirements of the applications. Consequently, the data does not need to travel to other parts of the hierarchy, thereby reducing the communication latency by sending data only to nearby nodes. When organizing the compute nodes according to HPI, the groups tend to have nodes that reside close to each other. Thus, in systems that include multiple nodes in close proximity (e.g., at the edge of the network) with rather limited computational resources, HPI may lead to having some groups that have lower computational resources than others (i.e., groups which include only edge nodes). On the contrary, organizing the nodes in groups, e.g., randomly, would actually be more likely to distribute the computational resources evenly among the groups. Nevertheless, in fog computing systems the communication latency can contribute more to increased overall application execution delay than the lack of computational resources (as shown in Section 4.3.3). Thus, even though the approach of HPI might not distribute the computational resources evenly, it can reduce the overall application execution delay significantly (compared to a random approach), by lowering the communication latency within the groups. This is also shown in the evaluation in Section 4.3.

3.5. Flat structures

While hierarchical structures represent the common practice, flat structures have also been proposed for fog computing (as discussed in Section 2). In this section, we design a distributed algorithm that enables the nodes of a fog computing system to self-organize into a flat structure based on proximity. As shown in Fig. 4(b), a flat structure is different to a hierarchical structure in that: (*i*) There is no root. Thereby, the structure can expand towards any direction, contrarily to the hierarchical structure which expands downwards. (*ii*) In contrast to hierarchical structures whereby nodes cannot belong to more than two groups (as discussed in Section 3.4), in flat structures the number of groups that each node can belong to, is not limited [41]. As a result of these differences, the data does not travel exclusively towards the cloud. Instead, the data can travel through the structure in any direction. By doing this though, a cloud node can also be found.

Algorithm 3: FPI (Flat Proximity-Integrated)

```
1  Procedure: joinSystemFPI(Address n_cp)
2      GroupList = joinRequest(n_cp)
3      for i = 1 to GroupList.size() do
4          n_pro = findNPro(GroupList)
5          Gn_cp = findGroup(GroupList, n_pro)
6          addNeighbors(Gn_cp)
7          notifyNeighbors(Gn_cp)
8          if Gn_cp.size > S then
9              n_far = findNFar(Gn_cp)
10             if n_far == n_new then
11                 GroupList.remove(Gn_cp)
12             else
13                 Gn_cp.remove(n_far)
14                 joinFlag = true
15                 break
16             end
17         else
18             joinFlag = true
19             break
20         end
21     end
22     if joinFlag != true then
23         G_new = new Group(n_new, n_cp)
24         notifyNeighbors(G_new)
25     end
```

For this reason, the flat structure may be more appropriate when there are multiple geographically distributed cloud compute nodes in a fog computing system, which are preferred to be arranged among the other nodes, rather than on top of them (i.e., in a hierarchy).

Notably, Fig. 4(b) shows nine compute nodes that have joined the system sequentially. The arrangement of these nodes though, seems to be abstract again (similar to Fig. 4(a)) because when a new compute node joins, some of the nodes self-organize based on proximity, and the arrangement of the structure changes. Since the flat structure has no root, a new compute node can request to join from any existing node of the system as long as its address is known. Furthermore, since in flat structures the number of groups that each node belongs to is not limited, this number is decided dynamically according to the utilized (self-organizing) algorithm.

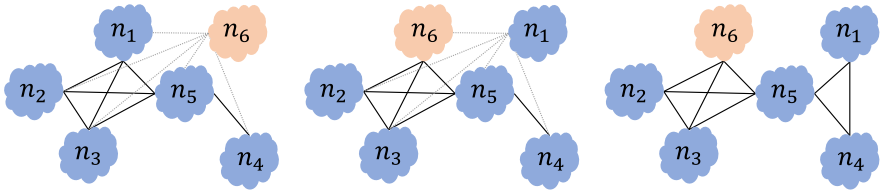
To organize the compute nodes of a fog computing system into a flat structure, we design FPI (Flat Proximity-Integrated) which is described in Algorithm 3 from the perspective of a new compute node which requests to join the system. In this algorithm, each new node n_{new} joins through n_{cp} (Line 1) which can be any existing cloud node of the system (as discussed in Section 3.2).

Upon request, n_{cp} responds with a list of the groups that it belongs to (Line 2), which includes all the neighbors of n_{cp}. Then, n_{new} measures the proximity to each one of these neighbors in order to find the node with the closest proximity n_{pro} (Line 4). Afterwards, n_{new} finds the group that includes n_{pro} (Line 5), joins this group (Line 6), and notifies the neighbors about the arrival of a new compute node (Line 7), i.e., the neighbors also add n_{new}. This triggers all the nodes of this group to count their neighbors. If the number of neighbors is smaller or equal to S (Line 17), n_{new} stays in this group and no further actions are required (i.e., a flag is raised in Line 18, and the process stops in Line 19).

If the number of neighbors is greater than S (Line 8), then the nodes calculate n_{far} (Line 9), and remove it from the group. Similar to Algorithm 1, n_{cp} is excluded from the calculation of n_{far}. If n_{far} equals n_{new} (Line 10), n_{new} removes this group from the list acquired by n_{cp} (Line 11), and examines the rest of the groups in the list by following the same process again (Line 3). If n_{new} equals n_{far} in all the groups of n_{cp}, n_{new} creates a new group that includes n_{new} and n_{cp} (Line 22). However, if n_{new} is not the most distant node in one of the groups of n_{cp} (Line 12), then n_{new} remains in that group and removes n_{far} (Line 13) since the group is at capacity. Then, n_{far} requests to join again by following the same process (Line 1), and using n_{cp} as contact.

Algorithm 2 shows what happens according to FPI from the perspective of the existing nodes. This process is similar to HPI, but the contact node used by n_{far} is different. Specifically, when a new node joins a group, all the nodes of this group receive the update (Line 1). Then, each node examines if the group capacity S is exceeded (Line 2). If it is, all nodes find n_{far} (Line 3), and remove it from the group (Line 4). The node that is n_{far} (Line 5) deletes the group (Line 6), and joins the system again through the contact point that was used by the new node which triggered n_{far} to leave (Line 7).

Fig. 7 shows the process of adding new compute nodes according to FPI graphically. In a system of 5 compute nodes, i.e., $n_1, n_2, \ldots, n_5$, a new compute node n_6 requests to join using n_5 as contact, as shown in Fig. 7(a). Then, n_6 joins the group of n_5 that contains the node in closest proximity n_{pro}, and the neighbors in this group (i.e., n_1, n_2, n_3, n_5) add n_6 as well. The addition of n_6 triggers all the neighbors to count the group size. In this case, the size is 5 (i.e., larger than the maximum group size S that equals 4). This means that all nodes calculate n_{far} which is n_1. Therefore, the new compute



(a) New node n_6 requests to join from n_5, and joins group that contains n_{pro}.

(b) All nodes calculate n_{far} which is n_1. n_1 leaves and joins again from n_5.

(c) n_1 being n_{far} in the group of n_6, joins the other group of n_5.

Fig. 7. Example of the process of FPI step-by-step (for $S = 4$).

node n_6 stays in this group, while n_1 leaves and requests to join again using n_5 as contact, as shown in Fig. 7(b). $n_1 == n_{far}$ in the group of n_5 that contains n_6, which means that n_1 will join the other group of n_5, i.e., the group that contains n_5, and n_4, as shown in Fig. 7(c). Since this group is not at capacity, i.e., the group size equals 3, no further actions are required and the FPI algorithm finishes.

This algorithm raises the question of why n_{far} joins through n_{cp} specifically, when rejoining the system. In general, the reason for having n_{far} replaced by a node in closer proximity (i.e., n_{new}), is based on the goal to select neighbors for each node so that the communication among them takes place with low latency and high bandwidth. For the same reason, we examine the option of having n_{far} (after leaving) join through its closest neighbor (n_{pro}). This seems reasonable because this way, both n_{new} and n_{far} are placed in groups with nodes in proximity. We also examine the case that n_{pro} is used as contact for n_{new} upon the join request, and until n_{cp} equals n_{pro}, in order to place each new node in the group of the closest neighbor.

However, after experimenting with large-scale systems (i.e., 1000–10,000 nodes), we notice that both of these options can trigger changes that spread throughout the system. This happens because when n_{far} leaves and joins another group at capacity, one of the preexisting nodes of that group leaves and joins again. This behavior continues until n_{far} joins a group which is not at capacity. However, in large-scale systems, the number of groups to be examined can be prohibitively large to allow this process to finish, resulting in additional overhead that may compromise the scalability of the system by creating bottlenecks. Therefore, to limit this overhead, n_{far} requests to join through n_{cp} which means that the resulting changes of a new node joining affect n_{cp} and its neighbors, while the rest of the system remains stable.

A system that follows FPI self-stabilizes when the new node has joined a group of the contact node (either preexisting or new), and no more operations are pending. Since in FPI, only the groups of the contact node are examined by a new node, the number of operations required for stabilization depends on the number of groups of the contact. This means that if the contact node belongs to many groups, many operations need to be completed before the system stabilizes, while a small number of groups means fewer operations. However, when the contact nodes are selected randomly, i.e., there is no global point of entry for new nodes (which is the case in FPI), the number of groups that each nodes belongs to, remains limited [41]. This means that the operations required for the algorithm to finish remain limited as well, and the system stabilizes. Additional evidence about the stabilization of a system that implements FPI is provided in Section 4.3.6.

Regarding the distribution of the computational resources within groups, FPI behaves similar to HPI. This means that when organizing the compute nodes according to FPI, the groups tend to have nodes that reside close to each other. Therefore, in systems with multiple compute nodes in close proximity (e.g., at the edge of the network) that have rather limited computational resources, FPI may result in having some groups that have lower computational resources than others. Nevertheless, the communication latency (and consequently the overall application execution delay, as shown in Section 4.3.3) is likely to be reduced significantly with FPI (compared to alternatives), due to organizing the compute nodes based on network proximity. This is also shown in the evaluation in Section 4.3.

4. Evaluation

To evaluate our algorithms, in this section we present an implementation of the proposed system, as presented in Section 3.3. The evaluation environment, including technical details of the implementation, and decisions regarding system parameters, is discussed in Section 4.1. Afterwards in Section 4.2, we describe the alternative methods which we use as baselines. Finally, we perform various experiments considering an image processing and a smart city use case, and we present our findings (which focus on communication latency, network bandwidth, utilization of computational resources, scalability, and stabilization) in Section 4.3. The source code of our implementation, along with executable binary files, numerical results, and additional documentation of the utilized APIs, can be found in the project repository[1] [59].

[1] https://bitbucket.org/BasilKaragiannis/self-organizing-fog-computing.

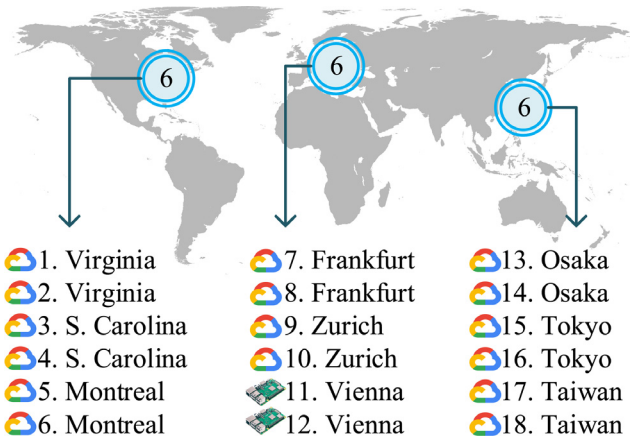


Fig. 8. Location of the compute nodes used in the evaluation (the cloud icon next to a location signifies a Google cloud node, and the Raspberry Pi icon signifies a Raspberry Pi node).

4.1. Evaluation environment

For this evaluation, we create fog computing systems using multiple geographically distributed compute nodes which are either virtual, i.e., using cloud services, or physical, i.e., using Raspberry Pi 4 single-board computers. Since our approach targets geographically distributed systems, we use 18 compute nodes in three regions around the world, namely: eastern America, central Europe, and eastern Asia, as shown in Fig. 8. In each region, we assume that there are: one cloud compute node (i.e., nodes: 1, 7, and 13 in Fig. 8), and five compute nodes at the edge of the network (i.e., the rest of the nodes in Fig. 8). Hereinafter, we refer to the cloud compute nodes as high-end nodes, and to the rest of the compute nodes as low-end nodes. We use this terminology because the computational resources at the edge of the network are expected to be limited compared to the cloud [50]. Notably, the actual resource capacities of the compute nodes do not affect the experiments significantly because we focus on network-related metrics. Nevertheless, considering the location and the resource diversity of the compute nodes of the system, aids at creating realistic scenarios for the experiments. Presumably, the goal is to enable these nodes to self-organize into a hierarchical or a flat structure, such that each compute node has neighbors in close proximity.

With this goal in mind, we develop a prototype that implements the proposed algorithms (presented in Section 3), and enables the compute nodes (shown in Fig. 8) to self-organize into a hierarchical or a flat structure in order to execute applications. Our prototype is developed using the Java Spring framework which produces an executable JAR file that can be executed on any node with a JVM (Java Virtual Machine). We do this for portability reasons, so that any compute node (virtual or physical) can run the prototype independently of the underlying operating system. Nevertheless, all the nodes in our system use the Debian 10 operating system (the Raspberry Pi computers use the Debian 10-based Raspbian). For the communication between the compute nodes, we use REST-based APIs due to the extensive adoption of REST (Representational state transfer) which provides interoperability with Internet services over HTTP [60].

Even though the proposed algorithms are agnostic of the utilized proximity measure, and can theoretically work with either network latency or number of hops as indicators of network proximity, in this evaluation we use the latter. The reason is that the number of hops between distributed compute nodes can be considered static, which makes it a steady indicator of network proximity [61,62]. On the contrary, proximity measures that rely on latency, may prove unreliable due to fluctuations that can follow random patterns [63]. In fact, it has been observed that in environments with multiple geographically distributed nodes, different latency measurements indicate different nodes as the closest [64]. It should be noted that hops might also cause irregularities, i.e., it is possible that nodes that reside nearby in terms of network hops, have high latency due to bandwidth bottlenecks. However, while nodes that reside far from each other (in terms of network proximity), are likely to have to deal with bandwidth bottlenecks [1], the communication between nodes that reside nearby is not as prone to such bottlenecks. Thus, connecting nodes according to network proximity can actually increase the available bandwidth [65]. This means that even though bandwidth bottlenecks between nodes in close network proximity might occur, they are rather unlikely. Therefore, we assume that the communication latency can be reduced by decreasing the number of hops between nodes, and increasing the network bandwidth (which has also been confirmed in the literature [66]). Further evidence to support this decision (i.e., using hops as an indicator of network proximity when creating fog computing systems) is provided by our results (in Section 4.3) which show reduced communication latency and increased network bandwidth when measuring the network proximity between compute nodes using network hops.

Thus, to measure the proximity between two compute nodes, we count the number of hops in the network path that connects these two nodes. In order to find this number, the prototype makes system-calls to Traceroute [67] (a system

command for network diagnostics integrated in all major operating systems such as Windows, Linux, and Mac). If a node belongs to an organization which uses a firewall that blocks the traffic from Traceroute, we consider the number of hops until this firewall as a representative indicator of network proximity for this node. This way, the prototype is able to count the network hops between any two nodes. To find the number of hops, Traceroute sends ICMP (Internet Control Message Protocol) messages that are processed by all the routers on-path [68]. Notably, only routers (or devices that operate on the OSI network layer) are counted. This means that switches, hubs, and other network devices (that operate on the OSI data link layer) do not affect the number of hops. This is very positive in our case, because even though the communication to the first router (within the local area network) may be very fast, it might involve a large number of network devices (e.g., wireless/wired bridges and hubs) that could rapidly increase the hop count without increasing the network latency. However, this is avoided by not considering such devices when measuring the number of hops.

Since in Section 3 we introduce the proposed structures with a group size that equals 4 (as shown in Fig. 4), initially, we retain the same group size for this evaluation in order to make the organization of the system more comprehensible. However, when scrutinizing the experiments in the next sections, we also discuss how the group size affects the results, and in general, how it affects the behavior of the system. The initial contact point of the system is always a cloud compute node (as discussed in Section 3.2), and is selected randomly (using the uniform distribution) among the cloud nodes. In the hierarchical structure, the same compute node acts as contact for all the other nodes to join, as discussed in Section 3.4. In the flat structure, the contact node may change, but is again selected randomly using the uniform distribution among the cloud nodes that have already joined, as discussed in Section 3.5.

4.2. Evaluation baselines

As discussed in Section 1, most state-of-the-art approaches in fog computing rely on the proximity of neighbor nodes without taking actual network proximity measurements. To represent such state-of-the-art hierarchical fog computing structures (such as [14–16] which are discussed in Section 2.1), we implement an algorithm whereby the nodes self-organize into a hierarchical structure without integrated proximity, which we refer to as Hierarchical Proximity-Agnostic (HPA). In HPA, each new node becomes a child in an existing group which is not at capacity. If all the groups are at capacity, a leaf node becomes a parent for the new node. This results in a hierarchical structure which grows first in breadth and then in depth. The arrangement of the compute nodes in this structure depends on the order that the nodes join.

To represent state-of-the-art flat fog computing structures that do not take network proximity measurements (such as [35–37] which are discussed in Section 2.2), we implement an algorithm which we refer to as Flat Proximity-Agnostic (FPA). In FPA, the nodes self-organize into a flat structure without integrated proximity. Similar to HPA, the arrangement of the nodes in FPA depends on the order that the nodes join, but also on the nodes which are used as contacts (which are selected randomly using the uniform distribution). The goal of both of these algorithms (i.e., HPA and FPA) is to serve as baselines in order to examine the potential benefits of applying self-organization with integrated proximity to organize geographically distributed compute nodes in fog computing.

Furthermore, we implement an algorithm which adds all the nodes to the same group. This means that every node has all the other nodes as neighbors. In contrast to the other approaches whereby each compute node can communicate directly only with a limited number of neighbors, in this algorithm each compute node can communicate directly with all the other nodes of the system. Thus, since in this approach all the nodes are directly connected to each other, this algorithm is expected to have the lowest communication latency. For this reason, we refer to this algorithm as an optimal latency algorithm (OPT). Similar to HPI and FPI, OPT is also proximity-integrated. Notably, the approach of OPT may not be realistic in fog computing because it assumes that every node maintains a global view of the system, which may require a large amount of computational resources (especially in large-scale systems). However, this might not be possible since the compute nodes at the edge of the network have limited computational and storage resources [50,69]. Nevertheless, we use OPT to represent the lowest possible communication latency.

4.3. Evaluation results

In this section, we perform various experiments which aim at showing how each algorithm performs regarding communication latency (Section 4.3.1), network bandwidth (Section 4.3.2), utilization of computational resources (Section 4.3.3), execution delay (Section 4.3.4), scalability (Section 4.3.5), and stabilization (Section 4.3.6). Since the order that each node joins the system may affect the arrangement of the structure, for each algorithm we perform 50 experiments with nodes that join randomly using the uniform distribution. This means that for each of the examined algorithms (i.e., HPA, HPI, FPA, FPI, and OPT), the 18 compute nodes (shown in Fig. 8) join the system sequentially (in a random order) 50 times, and form fog computing structures according to the corresponding algorithm. For each experiment, we take various measurements which are presented below. Even though we have experimented extensively with this setup, we present the results of 50 experiments (for each algorithm) because we found this number to be large enough to represent the general case.

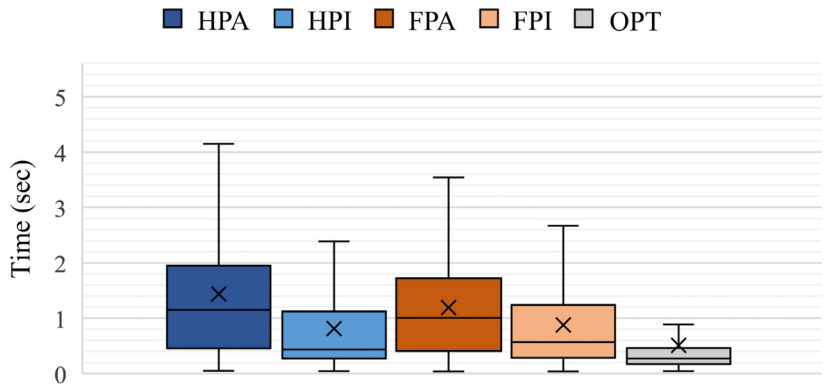


Fig. 9. Communication latency to nodes in proximity.

Table 3
Communication latency (in ms) to nodes in proximity for different file sizes.

		HPA	HPI	FPA	FPI	OPT
1.2 Mb	Average	893	479	795	533	126
	St. dev.	764	592	683	608	121
2.4 Mb	Average	1,065	581	947	627	201
	St. dev.	904	715	824	700	228
4.8 Mb	Average	1,435	811	1,193	876	507
	St. dev.	1,190	843	950	786	690

4.3.1. Communication latency

To evaluate the communication latency among the compute nodes for each algorithm, we assume an image processing use case (because such use cases are common for the evaluation of fog computing systems [70,71]). Specifically, in our experiments we consider a user device at the edge of the network, e.g., a smart doorbell, that takes a picture using an app (similar use case has been presented in Section 3.1). This app includes various latency-sensitive and latency-tolerant processes which can be related to pattern recognition, analytics, storage, etc. As discussed in Section 3.1, the processes for which low latency may increase the quality of service for the users, are considered latency-sensitive (e.g., pattern recognition). On the other hand, the processes for which low latency may not affect the quality of service significantly (e.g., analytics and storage), are considered latency-tolerant.

Instead of executing the app on the user device (which may have limited computational resources or power supply), the app's processes are distributed on the compute nodes of the fog computing system (i.e., the computations of the app are offloaded to the compute nodes). The latency-sensitive processes are hosted on the compute nodes in proximity, while the latency-tolerant processes are hosted on compute nodes which reside farther away.

Each low-end node receives an image file (from a user device) that needs to be processed by processes that are hosted on the compute nodes. Thus, this file is sent through the structure to all the other compute nodes of the system. Presumably, the nodes that receive the file first are the neighbors which are provided by our algorithms. To take latency measurements, we measure the delay from the time that the initial compute node sends the file until the time each of the other nodes receives the file. This delay includes communication latency but excludes further processing from the processes.

To examine the incurred delay for latency-sensitive processes, for each low-end node we measure the time period until the image reaches the nodes in proximity. As nodes in proximity, we consider the three nodes which receive the file first. The reason that we consider three nodes specifically, is that the group size equals 4 (as discussed in Section 4.1), i.e., one sender and three receivers. Notably, the group size does not apply to OPT since there is only one group that includes all the compute nodes. Nevertheless, we still consider as compute nodes in proximity the three closest nodes in order to make the results comparable among all the algorithms.

The latency measurements of the latency-sensitive processes which are hosted on compute nodes in proximity are shown in Fig. 9, for each one of the algorithms. Each box plot considers all 50 values (from the 50 experiments). However, the outliers are omitted because due to their distance from the interquartile range, they change the scale of the figure, and impair the visual representation of the results. Notably, the outliers have higher values than the shown maximum, but do not actually change the interpretation of the shown results. The values of Fig. 9 are the outcome of using an image file of 4.8 megabytes (Mb), which is a reasonable size of an image taken for pattern recognition purposes. Nevertheless, in order to produce results that apply to various cases, we also perform experiments with different file sizes which are shown in Table 3 along with the average values and the standard deviations.

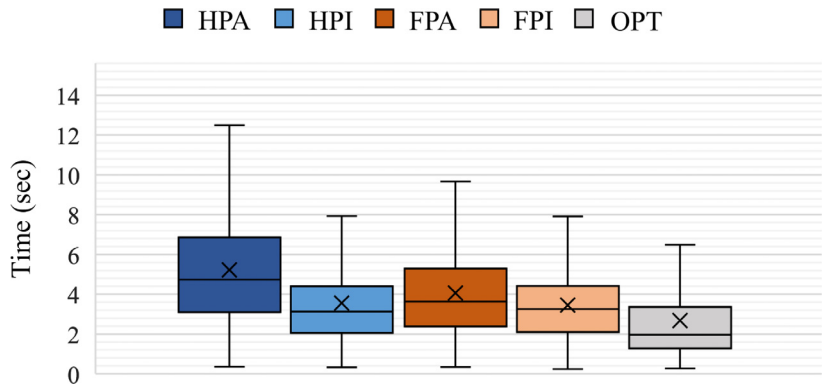


Fig. 10. Communication latency to the rest of the nodes in the system.

Table 4
Communication latency (in ms) to the rest of the nodes in the system for different file sizes.

		HPA	HPI	FPA	FPI	OPT
1.2 Mb	Average	3,250	2,236	2,665	2,329	1,359
	St. dev.	1,679	1,213	1,362	1,200	783
2.4 Mb	Average	4,021	2,694	3,348	2,821	1,839
	St. dev.	2,299	1,548	1,896	1,549	1,254
4.8 Mb	Average	5,220	3,564	4,061	3,456	2,673
	St. dev.	2,804	2,649	2,134	1,731	2,067

As shown in Fig. 9, HPA has an average delay of 1,435 milliseconds (ms). By applying self-organization with integrated proximity in the hierarchical structure, as implemented in HPI, the average delay drops to 811 ms which indicates a reduction of ca. 43%. By using a flat structure instead of the hierarchy, FPA has an average delay of 1,193 ms which means ca. 17% less than HPA. When applying self-organization with integrating proximity in the flat structure, FPI has an average delay of 876 ms. This means ca. 39% less than HPA, ca. 27% less than FPA, and ca. 7% more than HPI. Furthermore, by examining the average values for different file sizes (which are shown in Table 3), we note that HPI and FPI provide similar latency benefits independently of the size of the file.

Finally, OPT has an average delay of 507 ms which means ca. 65% less than HPA, ca. 37% less than HPI, ca. 58% less than FPA, and ca. 42% less than FPI. Notably, in Fig. 9 we notice that the average delay of OPT is above the upper quartile. This happens because there are many outliers which are numerically distant from the interquartile range. The reason that OPT exhibits this behavior is that initiating transmissions to many nodes at the same time may cause an (upload) bandwidth bottleneck which can increase the communication latency of the transmissions invoked by latency-sensitive processes. This shows that in fog computing, trying to improve performance by connecting many compute nodes to each other, may cause the opposite effect. Further experimentation shows that when increasing the size of the system, this behavior becomes increasingly noticeable. For this reason, the notion of a group (i.e., nodes that maintain a limited number of neighbors), and algorithms to create the groups – such as the algorithms proposed in this paper – become crucial.

We also measure the delay for reaching the rest of the nodes in the system, which host latency-tolerant processes. Even though the communication latency may not affect latency-tolerant processes, low latency still improves the quality of experience of the users. Fig. 10 shows the delay to reach all the nodes of the system apart from the nodes in proximity (which are shown in Fig. 9). The average values and the standard deviations along with additional file sizes are shown in Table 4.

As shown in Fig. 10, HPA has an average delay of 5,220 ms. By applying self-organization with integrated proximity, HPI lowers the delay to 3,564 ms which indicates a reduction of ca. 32%. With a flat structure, FPA has an average delay of 4,061 ms which means ca. 22% less delay than HPA. FPI which applies self-organization with integrated proximity in the flat structure, has an average delay of 3,456 ms which is ca. 34% less than HPA, ca. 3% less than HPI, and ca. 15% less than FPA. Finally, when all the nodes are connected to each other, OPT has an average delay of 2,673 ms which is ca. 49% less than HPA, ca. 25% less than HPI, ca. 34% less than FPA, and ca. 23% less than FPI.

Therefore, we note that applying self-organization with integrated proximity reduces the communication latency of latency-sensitive processes for both hierarchical and flat structures by 27%–43%. Furthermore, for latency-tolerant processes, self-organization with integrated proximity reduces the communication latency by 15%–32%. In addition, with this experiment we note that even though hierarchical structures represent the common practice in fog computing, FPA outperforms HPA for both latency-sensitive (ca. 17% reduction) and latency-tolerant processes (ca. 22% reduction), while FPI performs slightly better than HPI for latency-tolerant processes (ca. 3% reduction), but mildly worse for latency-sensitive processes (ca. 7% reduction). The reason that flat structures perform better is that in a flat structure, each node

Table 5
Network bandwidth (in Mb/sec) to nodes in proximity for different file sizes.

	HPA	HPI	FPA	FPI	OPT
1.2 Mb	1.344	2.505	1.509	2.251	9.524
2.4 Mb	2.254	4.131	2.534	3.828	11.94
4.8 Mb	3.345	5.919	4.023	5.479	9.467

Table 6
Network bandwidth (in Mb/sec) to the rest of the nodes in the system for different file sizes.

	HPA	HPI	FPA	FPI	OPT
1.2 Mb	0.369	0.537	0.45	0.515	0.883
2.4 Mb	0.597	0.891	0.717	0.851	1.305
4.8 Mb	0.92	1.347	1.182	1.389	1.796

can have more neighbors than in the hierarchical (even with the same group size) due to the unconstrained number of groups (as discussed in Section 3.5). Thus, the chances of finding neighbors in close proximity increase. Nevertheless, in HPI every new node travels through the hierarchy in search of nodes in proximity, which is why more groups are examined than in FPI. This means that even though each node in HPI, may have fewer neighbors that in FPI, the nodes that do become neighbors are more carefully selected, thereby achieving the lowest communication latency for latency-sensitive processes.

When experimenting with various group sizes and different number of nodes in the system, we have only observed slight changes in the communication latency. Nevertheless, a pattern in these changes can be noticed. If the number of nodes in the system is smaller or equal to the group size, there are no differences among the examined algorithms since there is only one group of nodes. However, when the number of nodes increases, and the number of groups in the system becomes equal to the group size, then the proposed algorithms start showing significant differences. This happens because when the system reaches this size, enough nodes have selected neighbors according to the utilized algorithm for the system to form the target structure. For instance, by using a group size that equals 4, the results we present in this section apply when the system grows to more than 16 nodes. After this threshold, the percentages of latency reduction remain similar with slight increase when the system size grows.

4.3.2. Network bandwidth

Estimating the bandwidth capacity between two nodes in fog computing scenarios can be challenging because the various devices and applications which utilize the same network affect the available network bandwidth [72]. This happens because the bandwidth capacity of the links that connect the participating compute nodes may be shared among all the nodes in the network [73]. Thus, in our fog computing environment which includes geographically distributed compute nodes that communicate with each other through the Internet, measuring the bandwidth between nodes may produce unreliable results since other nodes of the Internet may be using the same network links. For this reason, instead of measuring the bandwidth capacity between compute nodes for each algorithm (i.e., HPA, HPI, FPA, FPI, and OPT), we examine the changes in the available network bandwidth based on the required time to transfer files through the structure.

To examine the changes in the available network bandwidth for latency-sensitive processes, we analyze the amount of time required to transfer files of different sizes to the compute nodes in proximity. In Table 3, we notice that, e.g., HPA requires an average of 893 ms to send a 1.2 Mb file to the nodes in proximity. This indicates that the involved network links are able to transfer 1.2 Mb in 893 ms, or 1.34 Mb per 1 s. Similarly, we can estimate the size of the data per second (Mb/sec) that can be transferred to compute nodes in proximity for all the algorithms. Table 5 shows these measurements.

By examining Table 5, we note that HPI which applies self-organization with integrated proximity, increases the available network bandwidth compared to the simple hierarchical approach (i.e., HPA), by ca. 76%–86% for all evaluated file sizes. Similarly, FPI increases the available network bandwidth compared to FPA, by ca. 36%–51% for all evaluated file sizes. OPT has the highest bandwidth due to utilizing only direct communication links between nodes.

The reason that HPI and FPI increase the available network bandwidth of latency-sensitive processes (by ca. 36%–86%) is because the bandwidth capacity of the links that connect nodes in close proximity, tends to be higher than when the nodes reside farther apart. This is a crucial observation of fog computing that current research often claims without actual empirical results, and without quantifying the extend of the potential bandwidth gains [2,74].

Similar to Table 5 which is based on the average values of Table 3, we also create Table 6 which is based on Table 4, and shows bandwidth measurements for latency-tolerant processes. According to Table 6, HPI increases the available network bandwidth by ca. 46%–49% compared to HPA, while FPI increases the available network bandwidth by ca. 14%–19% compared to FPA. Again, OPT which uses only direct communication links between nodes, has the highest network bandwidth.

Therefore, by further analyzing the time required to send files of various sizes to the compute nodes of the system, we show that the proposed algorithms enable the compute nodes to communicate with higher network bandwidth.

Table 7
Execution delay (in ms) of image processing applications for different file sizes.

		Face detection	Body detection	Smile detection
1.2 Mb	Average	296	231	315
	St. dev.	27	23	21
2.4 Mb	Average	518	423	622
	St. dev.	13	11	11
4.8 Mb	Average	960	793	1,275
	St. dev.	26	14	15

This means that using self-organization with integrated proximity (i.e., HPI and FPI) increases the available network bandwidth for both hierarchical and flat structures. Specifically, we note that self-organization increases the available network bandwidth of latency-sensitive processes by ca. 36%–86%, and for latency-tolerant processes by ca. 14%–49%.

4.3.3. Utilization of computational resources

To ensure that the proposed algorithms do not interfere with the execution of the applications, we examine CPU and RAM utilization. We do this on the nodes which run the proposed algorithms on single-board computers (Raspberry Pis), because these are the nodes with the least amount of computational resources.

Using Top [75] (a system command for monitoring the utilized computational resources of the running processes), we monitor the resource utilization of the processes that handle the self-organization according to each of the proposed algorithms. CPU utilization remains at 1% with occasional spikes which do not exceed 10%, and RAM utilization remains steady around 9%. Since the proposed algorithms are executed in a distributed manner, no node has to perform all the organizational operations, and resource utilization remains low. Hence, these results support our decision to use low-end nodes for creating fog computing systems as presented in Section 3.3.

4.3.4. Execution delay

Since the benefits of the proposed algorithms stem from reducing the communication latency, in this section we examine the execution delay of the processes in order to put the communication latency benefits into perspective. Notably, if the execution delay is very high compared to the communication latency, it means that reducing the communication latency results in a small reduction of the overall latency of executing processes in the system (since the overall latency includes both communicate latency and execution delay). However, if the execution delay is small compared to the communication latency, it means that the overall latency depends more on the communication latency. Hence, a significant reduction in the communication latency (such as 27%–43% as presented in Section 4.3.1) can cause a noticeable reduction in the overall latency.

The execution delay represents the time required by a compute node to execute a process. Thus, this delay depends on the resource capacities (e.g., CPU and RAM) of the utilized compute node, and on the computations that are performed by the process. The processes that we execute for this experiment are related to image processing which is the examined use case of this evaluation. Specifically, we execute a face detection process (e.g., for face tracking), a body detection process (e.g., for counting people), and a smile detection process (e.g., for emotion recognition). All of these processes are implemented to use machine learning-based Haar cascade classifiers which are provided pre-trained by the OpenCV library [76].

Since the execution delay depends on the computational resources of the utilized node, we measure this delay again on the compute nodes with the least amount of computational resources (i.e., the Raspberry Pis). This way, the measurements represent the maximum execution delay which can be reduced if the processes are executed on nodes with more computational resources, e.g., in the cloud. To acquire results which are representative of the general case, we execute each process 50 times. The results include the average values and standard deviations of executing each process for different file sizes, and are shown in Table 7.

Notably, each process has different execution delay because a different cascade classifier is used, i.e., the computations are different for each process. Specifically, we note that for an image file of 1.2 Mb, the face detection process requires, by average, 296 ms, the body detection process needs, by average, 231 ms, and the smile detection process takes, by average, 315 ms. We also note that the execution delay grows when the file size increases. This happens because larger files require more time to be processed, which increases the execution delay. Nevertheless, the previous observation that the execution delay of detecting bodies is, by average, less than detecting faces which is less than detecting smiles, always applies.

By comparing the execution delay of the processes (shown in Table 7) with the communication latency required to reach compute nodes in proximity (shown in Table 3), and remote compute nodes (shown in Table 4), excluding the approach of OPT (which may not produce realistic results, as discussed in Section 4.2), we note the following: The communication latency of remote compute nodes is always much higher than the execution delay. For an image file of 1.2 Mb, for example, the communication latency of HPI (2,236 ms) is about 6.5 times larger than the execution delay

of the face detection process (296 ms). The communication delay of compute nodes in proximity, on the other hand is not as high. Nevertheless, for an image file of 1.2 Mb, the execution delay of all the processes is still lower than the communication latency of any approach (i.e., HPA, HPI, FPA, FPI). For an image file of 2.4 Mb, the execution delay is also lower, with the exception of the smile detection process which has execution delay (622 ms) that is approximately 7% larger than the communication delay of HPI (581 ms). Finally, for an image file of 4.8 Mb, the execution delay is lower or comparable to the communication latency, with the exception of the smile detection process which has execution delay (1,275 ms) that is approximately 57% higher than the communication latency of HPI (811 ms), 6% higher than FPA (1,193 ms), and 45% higher than FPI (876 ms).

Therefore, by considering these results, and by taking into account that these execution delays represent maximum values, we note that in a system with geographically distributed compute nodes, the communication latency is much higher than the execution delay. However, if the computations of the executed processes are very intensive, or if the input data is very large, then the execution delay may become comparable to the communication latency. Notably, in many IoT scenarios the input data is rather small, e.g., sensor values, which may be inferred as even lower execution delay. Hence, we conclude that the communication latency plays a critical role in the overall latency of executing applications in the system. This further advocates that algorithms which enable the compute nodes to self-organize and communicate with low communication latency, are crucial for achieving low latency in fog computing systems.

4.3.5. Scalability

While performing the experiments regarding communication latency (in Section 4.3.1) and network bandwidth (in Section 4.3.2), we noticed that the incurred network traffic of the nodes that use OPT, is slightly higher than with the other algorithms. However, despite the extensive experimentation, with a system of 18 nodes we were not able to produce results which show a pattern that explains this observation. Even though slight changes in the network traffic seem trivial in small-scale systems, in larger systems, such changes may scale and eventually lead to scalability concerns due to network bottlenecks. For this reason, in this section we examine the scalability of the proposed algorithms considering the incurred network overhead.

To evaluate scalability, we examine the network overhead in a system with a growing size, i.e., a system in which new nodes are added consecutively, and follow the proposed algorithms in order to self-organize into the desired system structure. Systems with a growing size may find real-world use within the context of smart cities. Smart cities may integrate an increasing number of nodes due to new nodes being added for facilitating new applications, but also, due to the city itself expanding geographically [77]. Furthermore, we consider this a good example of a growing system because smart cities have been proposed as a target use case for fog computing [11].

In order to take scalability measurements that can be extrapolated to large-scale systems, first we use software to extend the fog computing structures of the proposed algorithms with a large number of compute nodes, and we measure the organizational network overhead. Then, we use regression (which is a predictive method) in order to model this overhead, and to create a function which outputs the overhead of our algorithms based on the system size. By examining the slope of this function, we are able to draw conclusions regarding the sensitivity of the network overhead when the system size changes. Therefore, we still use the same code of the prototype, but each node assumes that it is part of a larger system. To achieve this, we benchmark the code of the prototype into a simulation which creates fog computing systems similar to the ones of the previous experiments. However, unlike the previous experiments, we are now able to examine the behavior of our algorithms in a setting with a configurable number of participating compute nodes. The simulator we use for this purpose is based on a simulator for hierarchical and flat fog computing systems from the literature [20]. The code of our simulator, which is written in Java, is available in the project repository [59] along with all the numerical results, and the regression analysis.

For this experiment, we create fog computing systems with 1000 nodes which self-organize according to the proposed algorithms. We consider 1000 nodes to be a satisfactory system size for the regression analysis due to the high coefficients of determination in the regression results. Specifically, with the overhead data from 1000 nodes, both HPI and OPT have coefficients of determination with values higher than 0.9. The coefficient of determination is a widely-used statistical measure, also known as R^2, which shows how close is the regression line to the data. This statistical measure has a value between 0 and 1, and shows how well the model fits the data, i.e., values close to 1 indicate that the model explains the variability of the data. Usually, a coefficient of determination with a value higher than 0.6 is considered adequate [78,79].

To gather the overhead data, the system starts with one compute node, and new nodes join dynamically according to each one of the examined algorithms, until the system reaches 1000 nodes. For each new node that joins the system, we count the number of control messages which need to be exchanged in order to find neighbors. Thus, this experiment allows us to measure the overhead of the required control operations without considering the application traffic. When counting control messages, we consider requests to join the system, to notify neighbors, to measure proximity, etc. (as described by Algorithms 1 and 3 in Section 3). Since Traceroute sends messages incrementally to count the hops between two nodes, the number of required messages to emulate Traceroute equals to the number of hops between these nodes.

For the proximity-agnostic algorithms (i.e., HPA and FPA), the number of control messages remains negligible even with 1000 nodes. This happens because these algorithms are designed to serve as baselines, and only minimal effort is put into the self-organization of the nodes, which results in low overhead. However, in OPT, HPI, and FPI this overhead increases due to the multiple control messages which are required for the self-organization.

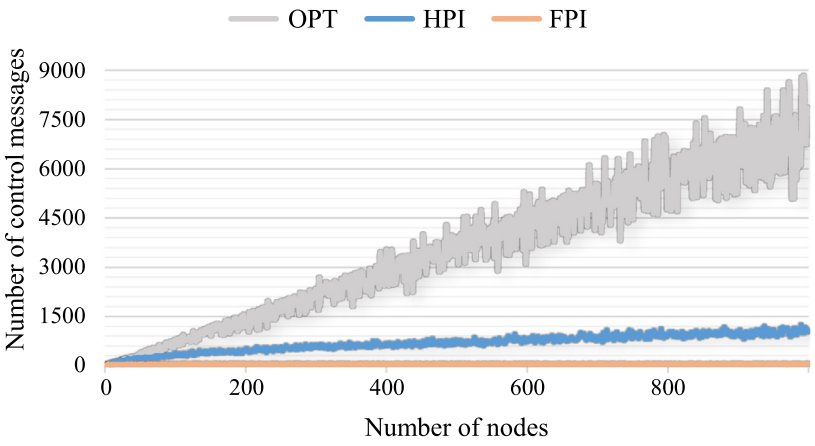


Fig. 11. Number of control messages for each new node that joins the system.

Table 8
Regression analysis of the data in Fig. 11.

	OPT	HPI	FPI
R^2	0.956	0.931	0.029
Standard Error	440.526	66.682	5.046
Variance	4,473,771	64,484	26
Intercept	−26.861	263.127	45.687
Slope	7.166	0.848	0.003

Fig. 11 shows the number of required control messages for each new node that joins the system according to OPT, HPI and FPI. Since these results exhibit a similar pattern which resembles a straight line, we perform a linear regression analysis (using the least squares method) with two goals in mind. First, we want to examine if there is a relationship between the number of nodes (as independent variable) and the overhead from control messages (as dependent variable). If such a relationship exists, which seems to be the case for OPT, there might be scalability concerns since the system may need to scale to a point that the overhead creates network bottlenecks, and interferes with the application traffic. Second, regression can be used as a prediction method for estimating future values, i.e., with even more than 1000 nodes, in order to examine how steeply the overhead changes when the system scales.

In Table 8, we show the regression analysis results which are based on the overhead from OPT, HPI, and FPI in Fig. 11. OPT and HPI have high coefficients of determination meaning that the regression line explains the data well. However, FPI has a coefficient of determination which is close to zero. This is a surprising result because by visualizing the data (in Fig. 11), it is evident that all the points of FPI follow the pattern of a straight line. To interpret the coefficient of determination of FPI, we examine the definition: $R^2 = 1 - \frac{SSres}{SStot}$ with SS_{res} being the residual sum of squares (i.e., difference between actual data and predicted data), and SS_{tot} being the total sum of squares of the differences between dependent variable and its mean. In linear models, SS_{tot} can also equal to the residual sum of squares plus the explained sum of squares. The explained sum of squares measures the variation of the data.

If SS_{res} equals zero, it means that the data fall perfectly on the regression line (i.e., predicted values), and $R^2 = 1$ (similar to OPT and HPI). When SS_{res} grows, it means that the data start deviating from the regression line, and R^2 decreases. However, when the data have very low variance, the explained sum of squares decreases. This means that SS_{tot} becomes similar to SS_{res} and R^2 approaches zero. This is why it is possible to have a prediction model with high accuracy even with low R^2 as long as the variance of the data is very low. To verify that this is the case for FPI, we examine the variance of the data, but also the standard error which shows how far are the data from the regression line. In Table 8, we note that FPI has very low variance and a low standard error, which indicate the accuracy of the regression line for estimating the overhead of the current and future values. OPT and HPI have higher standard errors due to the variance of the data.

Thus, the combination of two statistical measures (R^2 and standard error) indicates that the overhead of each algorithm can be predicted according to the regression lines. Each regression line can be created based on the values of Table 8, and considering that for each new node that joins the system (independent variable X), the resulting overhead (dependent variable Y) will be $Y = Slope \cdot X + Intercept$. The slope, which shows the rate of increase, indicates that in OPT with a slope that equals 7.1, the overhead grows radically which may lead to scalability concerns very fast. In HPI, the slope equals to 0.8 which is not as high as OPT, but still affects the overhead noticeably. Finally in FPI, the slope is 0.003 meaning that the overhead remains almost stable while the system size grows. This advocates scalability.

We also perform regression analysis for different group sizes. By doing this, we note that the statistical measures remain very similar independently of the group size. Specifically, OPT and HPI maintain high R^2, variances, and standard

errors while FPI maintains low R^2, variance, and standard error. This means that the regression lines can be used to explain both actual and predicted data. The slopes of the regression lines are also similar, with OPT and HPI having growing overhead, while in FPI the overhead remains almost stable. However, we also note that the intercept increases along with the group size. This happens because in larger group sizes, the proximity to more nodes needs to be measured, and more nodes need to be notified when a new node joins. The messages required for these operations increase the overhead. Therefore, when the group size grows, the overhead grows, and the intercept grows as well. Essentially, this means that for larger group sizes, the values of the *y*-axis in Fig. 11 increase, but the lines of the figure remain very similar.

Hence, in all the algorithms (i.e., OPT, HPI, and FPI) a larger group size increases the resulting overhead. In addition, in OPT and HPI, there is a dependency between the number of the nodes and the overhead, which indicates that while the system size grows, the overhead grows as well. However, this dependency is exceedingly weak in FPI. This means that while the system size grows, the overhead remains almost stable.

The reason that FPI exhibits this behavior is that there is no central entity in the system, and when new nodes join, only the neighbors are affected while the rest of the system remains stable independently of the number of nodes. On the contrary, when new nodes join in OPT and HPI, this might trigger changes that spread all over the system, which is why the overhead grows. This is a significant finding because it shows that when a system uses a centralized approach, even in a hierarchical manner which is the common case in state-of-the-art approaches to organize fog computing systems, it is possible that the overhead accumulates. This might lead to scalability concerns. However, as suggested by our results, the increase in the overhead can be reduced significantly by using a flat structure.

4.3.6. Stabilization

As discussed in Section 4.3.5, for Fig. 11 we add compute nodes sequentially, and for each one, we count the control messages that need to be exchanged in the system until the new node has joined, and no further operations are pending. This means that Fig. 11 also shows the number of messages that need to be exchanged until the system stabilizes and a new compute node can be added.

When using OPT, we note that while the system size grows, the system requires an increasing number of messages (and thereby operations) to stabilize. This means that a large number of nodes may interfere with the normal operation of the algorithm, if a new node is added before the system has stabilized. When using HPI, the rate of increasing operations drops significantly, which means that the system can scale to a large degree before stabilization becomes a concern. Finally, when using FPI, the number of required operations is the lowest, and the rate of increasing is close to zero (as discussed in Section 4.3.5). This means that FPI stabilizes before the other approaches, and that the stabilization is not affected by the system size.

Notably, these results align with the discussions of stabilization in Sections 3.4 and 3.5. In HPI, the system stabilization is affected by the number of layers in the hierarchy, which is why when the hierarchical structure grows, the algorithm requires more operations for the system to stabilize. In FPI, on the other hand, the stabilization does not depend on the system size, which is why the number of required operations for the system to stabilize, does not increase when the system grows.

5. Conclusion

In this paper, we design and implement distributed algorithms for self-organizing fog computing systems that span over large geographical areas. By applying the proposed self-organizing algorithms with integrated proximity, we show that fog computing systems can execute applications with lower communication latency and higher network bandwidth than alternative methods. Specifically, according to the experiments we perform considering actual use cases, we show that the proposed algorithms reduce the communication latency of latency-sensitive processes by 27%–43%, and increase the available network bandwidth by 36%–86%. We also examine the scalability of our algorithms based on empirical results and using predictive methods, and we show that flat structures can scale better than the commonly used hierarchical structures, due to producing less overhead when the system size grows. However, hierarchical structures are shown to be particularly effective at reducing the communication latency for latency-sensitive processes. Therefore, we reach the conclusion that flat structures with integrated proximity provide solid benefits regarding communication latency, bandwidth, and scalability, while hierarchical structures provide similar or better latency and bandwidth benefits, but exhibit a lower degree of scalability. Considering all the provided results, we claim that self-organization has the potential to further advance the efficiency of fog computing systems despite the utilized system structure.

In our future work, we plan to improve the proposed algorithms by investigating aspects of fog computing systems that change at runtime, i.e., while the system is running. Specifically, one promising research direction is to focus on the size of the groups, and design mechanisms that enable the compute nodes to calculate the most appropriate group size dynamically based on the proximity of the neighbors, and their resource capacities. Furthermore, we plan to consider scenarios with compute nodes that leave the system, as well as cases with potential churn, i.e., when many nodes join/leave the system concurrently. Another promising research direction, with regard to the execution of applications, is to examine the effect of node failures on fog computing systems, and the impact that this may have on meeting latency- and bandwidth-related application requirements. For this reason, we plan to investigate novel techniques for ensuring that the application requirements remain satisfied when various compute nodes fail or disconnect from the system unexpectedly. Therefore, we consider that the work-at-hand can be regarded as the basis for creating more advanced fog computing systems.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation.

References

[1] M. Satyanarayanan, How we created edge computing, Nat. Electron. 2 (1) (2019) 42.
[2] W. Shi, G. Pallis, Z. Xu, Edge computing [scanning the issue], Proc. IEEE 107 (8) (2019) 1474–1481.
[3] K. Oh, N. Qin, A. Chandra, J. Weissman, Wiera: Policy-driven multi-tiered geo-distributed cloud storage system, IEEE Trans. Parallel Distrib. Syst. 31 (2019) 294–305.
[4] C. Cicconetti, M. Conti, A. Passarella, Architecture and performance evaluation of distributed computation offloading in edge computing, Simul. Model. Pract. Theory 101 (2020) 102007.
[5] T.X. Tran, D. Pompili, Adaptive bitrate video caching and processing in mobile-edge computing networks, IEEE Trans. Mob. Comput. 18 (9) (2018) 1965–1978.
[6] M. Satyanarayanan, The emergence of edge computing, IEEE Comput. 50 (1) (2017) 30–39.
[7] P. Bellavista, J. Berrocal, A. Corradi, S.K. Das, L. Foschini, A. Zanni, A survey on fog computing for the internet of things, Pervasive Mob. Comput. 52 (2019) 71–99.
[8] Y. Li, A.-C. Orgerie, I. Rodero, B.L. Amersho, M. Parashar, J.-M. Menaud, End-to-end energy models for edge cloud-based IoT platforms: Application to data stream analysis in IoT, Future Gener. Comput. Syst. 87 (2018) 667–678.
[9] D. Rapone, R. Quasso, S.B. Chundrigar, S.T. Talat, L. Cominardi, A. De la Oliva, P.-H. Kuo, A. Mourad, A. Colazzo, G. Parmeggiani, A.Z. Orive, C. Lu, C.-Y. Li, An integrated, virtualized joint edge and fog computing system with multi-rat convergence, in: International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), IEEE, 2018, pp. 1–5.
[10] D. Wang, Y. Peng, X. Ma, W. Ding, H. Jiang, F. Chen, J. Liu, Adaptive wireless video streaming based on edge computing: Opportunities and approaches, IEEE Trans. Serv. Comput. 12 (5) (2019) 685–697.
[11] C. Mouradian, D. Naboulsi, S. Yangui, R.H. Glitho, M.J. Morrow, P.A. Polakos, A comprehensive survey on fog computing: State-of-the-art and research challenges, IEEE Commun. Surv. Tutor. 20 (1) (2017) 416–464.
[12] Y. Li, W. Gao, MUVR: Supporting multi-user mobile virtual reality with resource constrained edge cloud, in: Symposium on Edge Computing (SEC), IEEE, 2018, pp. 1–16.
[13] U. Ramachandran, H. Gupta, A. Hall, E. Saurez, Z. Xu, Elevating the edge to be a peer of the cloud, in: International Conference on Cloud Computing (CLOUD), IEEE, 2019, pp. 17–24.
[14] M. Adhikari, M. Mukherjee, S.N. Srirama, DPTO: A deadline and priority-aware task offloading in fog computing framework leveraging multi-level feedback queueing, IEEE Internet Things J. (2020).
[15] X. Gao, X. Huang, S. Bian, Z. Shao, Y. Yang, Pora: Predictive offloading and resource allocation in dynamic fog computing systems, IEEE Internet Things J. 7 (2019) 72–87.
[16] A. Kiani, N. Ansari, A. Khreishah, Hierarchical capacity provisioning for fog computing, IEEE/ACM Trans. Netw. 27 (3) (2019) 962–971.
[17] T.-D. Nguyen, E.-N. Huh, M. Jo, Decentralized and revised content-centric networking-based service deployment and discovery platform in mobile edge computing for IoT devices, IEEE Internet Things J. 6 (3) (2019) 4162–4175.
[18] O. Skarlat, V. Karagiannis, T. Rausch, K. Bachmann, S. Schulte, A framework for optimization, service placement, and runtime operation in the fog, in: International Conference on Utility and Cloud Computing (UCC), IEEE, 2018, pp. 164–173.
[19] C. Ge, N. Wang, W.K. Chai, H. Hellwagner, Qoe-assured 4k HTTP live streaming via transient segment holding at mobile edge, IEEE J. Sel. Areas Commun. 36 (8) (2018) 1816–1830.
[20] V. Karagiannis, S. Schulte, Comparison of alternative architectures in fog computing, in: International Conference on Fog and Edge Computing (ICFEC), IEEE, 2020, pp. 1–10.
[21] R. Casadei, D. Pianini, M. Viroli, A. Natali, Self-organising coordination regions: a pattern for edge computing, in: International Conference on Coordination Languages and Models, Springer, 2019, pp. 182–199.
[22] G. Audrito, R. Casadei, F. Damiani, M. Viroli, Compositional blocks for optimal self-healing gradients, in: International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE, 2017, pp. 91–100.
[23] S. Jamin, C. Jin, A.R. Kurc, D. Raz, Y. Shavitt, Constrained mirror placement on the internet, in: Conference on Computer Communications (INFOCOM), 1, IEEE, 2001, pp. 31–40.
[24] P. Wydrych, P. Cholda, ISP-Supported traffic reduction for application-level multicast, in: International Conference on Communications (ICC), IEEE, 2011, pp. 1–6.
[25] J.F. Kurose, K.W. Ross, Computer networking: A top-down approach featuring the Internet, Addison Wesley, 2010.
[26] F. Nawab, D. Agrawal, A. El Abbadi, Dpaxos: Managing data closer to users for low-latency and mobile applications, in: International Conference on Management of Data (SIGMOD), ACM, 2018, pp. 1221–1236.
[27] F. Lamnabhi-Lagarrigue, A. Annaswamy, S. Engell, A. Isaksson, P. Khargonekar, R.M. Murray, H. Nijmeijer, T. Samad, D. Tilbury, P. Van den Hof, Systems & control for the future of humanity, research agenda: Current and future roles, impact and grand challenges, Annu. Rev. Control 43 (2017) 1–64.
[28] A. Cohen, X. Shen, J. Torrellas, J. Tuck, Y. Zhou, S. Adve, I. Akturk, S. Bagchi, R. Balasubramonian, R. Barik, M. Beck, R. Bodik, A. Butt, L. Ceze, H. Chen, Y. Chen, T. Chilimbi, M. Christodorescu, J. Criswell, C. Ding, Y. Ding, S. Dwarkadas, E. Elmroth, P. Gibbons, X. Guo, R. Gupta, G. Heiser, H. Hoffman, J. Huang, H. Hunter, J. Kim, S. King, J. Larus, C. Liu, S. Lu, B. Lucia, S. Maleki, S. Mazumdar, I. Neamtiu, K. Pingali, P. Rech, M. Scott, Y. Solihin, D. Song, J. Szefer, D. Tsafrir, B. Urgaonkar, M. Wolf, Y. Xie, J. Zhao, L. Zhong, Y. Zhu, Inter-disciplinary research challenges in computer systems for the 2020s, Tech. rep, National Science Foundation, USA, 2018.
[29] K. Vermeulen, B. Ljuma, O. Fourmaux, T. Friedman, R. McGeer, Internet measurements on edgenet, in: Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2019, pp. 915–916.

[30] V. Karagiannis, Compute node communication in the fog: Survey and research challenges, in: Workshop on Fog Computing and the IoT, ACM, 2019, pp. 36–40.
[31] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, Incremental deployment and migration of geo-distributed situation awareness applications in the fog, in: International Conference on Distributed and Event-Based Systems (DEBS), ACM, 2016, pp. 258–269.
[32] S.H. Mortazavi, M. Salehe, C.S. Gomes, C. Phillips, E. de Lara, Cloudpath: A multi-tier cloud computing framework, in: Symposium on Edge Computing (SEC), ACM, 2017, p. 13.
[33] O. Ascigil, T.K. Phan, A.G. Tasiopoulos, V. Sourlas, I. Psaras, G. Pavlou, On uncoordinated service placement in edge-clouds, in: International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2017, pp. 41–48.
[34] D. Nguyen, Z. Shen, J. Jin, A. Tagami, ICN-Fog: An information-centric fog-to-fog architecture for data communications, in: Global Communications Conference (GLOBECOM), IEEE, 2017, pp. 1–6.
[35] R. Casadei, M. Viroli, Coordinating computation at the edge: a decentralized, self-organizing, spatial approach, in: International Conference on Fog and Mobile Edge Computing (FMEC), IEEE, 2019, pp. 60–67.
[36] A. Rabay'a, E. Schleicher, K. Graffi, Fog computing with p2p: Enhancing fog computing bandwidth for IoT scenarios, in: International Conference on Internet of Things (IThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, 2019, pp. 82–89.
[37] R. Yu, G. Xue, V.T. Kilari, X. Zhang, The fog of things paradigm: Road toward on-demand internet of things, IEEE Commun. Mag. 56 (9) (2018) 48–54.
[38] X. Song, Y. Huang, Q. Zhou, F. Ye, Y. Yang, X. Li, Content centric peer data sharing in pervasive edge computing environments, in: International Conference on Distributed Computing Systems (ICDCS), IEEE, 2017, pp. 287–297.
[39] G. Tato, M. Bertier, C. Tedeschi, Koala: Towards lazy and locality-aware overlays for decentralized clouds, in: International Conference on Fog and Edge Computing (ICFEC), IEEE, 2018, pp. 1–10.
[40] Y. Jiang, D.H. Tsang, Delay-aware task offloading in shared fog networks, IEEE Internet Things J. 5 (6) (2018) 4945–4956.
[41] V. Karagiannis, S. Schulte, J. Leitão, N. Preguiça, Enabling fog computing using self-organizing compute nodes, in: International Conference on Fog and Edge Computing (ICFEC), IEEE, 2019, pp. 1–10.
[42] J.C.A. Leitão, L.E.T. Rodrigues, Overnesia: a resilient overlay network for virtual super-peers, in: International Symposium on Reliable Distributed Systems (SRDS), IEEE, 2014, pp. 281–290.
[43] C. Esposito, M. Ficco, A. Castiglione, F. Palmieri, H. Lu, Loss-tolerant event communications within industrial internet of things by leveraging on game theoretic intelligence, IEEE Internet Things J. 5 (3) (2018) 1679–1689.
[44] M. Polese, R. Jana, M. Zorzi, TCP And MP-TCP in 5g mmwave networks, IEEE Internet Comput. 21 (5) (2017) 12–19.
[45] M. Schindewolf, O. Mattes, W. Karl, Thread creation for self-aware parallel systems, in: Facing the Multicore-Challenge, Springer, 2010, pp. 42–53.
[46] C. Johnen, et al., Robust self-stabilizing weight-based clustering algorithm, Theoret. Comput. Sci. 410 (6–7) (2009) 581–594.
[47] N. Mitton, E. Fleury, I.G. Lassous, S. Tixeuil, Self-stabilization in self-organized multihop wireless networks, in: International Conference on Distributed Computing Systems Workshops (ICDCSW), IEEE, 2005, pp. 909–915.
[48] Y. Liao, H. Qi, W. Li, Load-balanced clustering algorithm with distributed self-organization for wireless sensor networks, IEEE Sensors 13 (5) (2012) 1498–1506.
[49] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, B. Koldehofe, Mobile fog: A programming model for large-scale applications on the internet of things, in: SIGCOMM Workshop on Mobile Cloud Computing, ACM, 2013, pp. 15–20.
[50] R. Ranjan, O. Rana, S. Nepal, M. Yousif, P. James, Z. Wen, S. Barr, P. Watson, P.P. Jayaraman, D. Georgakopoulos, M. Villari, M. Fazio, S. Garg, R. Buyya, L. Wang, A.Y. Zomaya, S. Dustdar, The next grand challenges: Integrating the internet of things and data science, IEEE Cloud Comput. 5 (3) (2018) 12–26.
[51] V. Karagiannis, Building a Testbed for the Internet of Things, Alexander Technological Educational Institute of Thessaloniki, 2014, pp. 1–92.
[52] C. Fiandrino, A. De la Oliva, J. Widmer, K. Kogan, Pdcell: An end-to-end transport protocol for mobile edge computing architectures, in: International Conference on Distributed Computing and Networking (ICDCN), ACM, 2019, pp. 71–80.
[53] P. Varshney, Y. Simmhan, Characterizing application scheduling on edge, fog, and cloud computing resources, Softw. - Pract. Exp. 50 (5) (2020) 558–595.
[54] J. Bellendorf, Z.Á. Mann, Classification of optimization problems in fog computing, Future Gener. Comput. Syst. 107 (2020) 158–176.
[55] V. Karagiannis, A. Papageorgiou, Network-integrated edge computing orchestrator for application placement, in: International Conference on Network and Service Management (CNSM), IEEE, 2017, pp. 1–5.
[56] C. Avasalcai, C. Tsigkanos, S. Dustdar, Decentralized resource auctioning for latency-sensitive edge computing, in: International Conference on Edge Computing (EDGE), IEEE, 2019, pp. 72–76.
[57] L. Valerio, M. Conti, A. Passarella, Energy efficient distributed analytics at the edge of the network for IoT environments, Pervasive Mob. Comput. 51 (2018) 27–42.
[58] M. Zimmermann, U. Breitenbücher, K. Képes, F. Leymann, B. Weder, Data flow dependent component placement of data processing cloud applications, in: International Conference on Cloud Engineering (IC2E), IEEE, 2020, pp. 83–94.
[59] Repository: self-organizing fog computing, myehosthttps://bitbucket.org/BasilKaragiannis/self-organizing-fog-computing, (Accessed 1 June 2020).
[60] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, J. Alonso-Zarate, A survey on application layer protocols for the internet of things, Trans. IoT Cloud Comput. 3 (1) (2015) 11–17.
[61] J. Mineraud, S. Balasubramaniam, J. Kangasharju, W. Donnelly, Fs-pgbr: a scalable and delay sensitive cloud routing protocol, ACM SIGCOMM Comput. Commun. Rev. 42 (4) (2012) 301–302.
[62] M. Pourvali, H. Bai, F. Gu, K. Shaban, M. Naeini, J. Crichigno, M. Hayat, S. Khan, N. Ghani, Virtual network mapping for cloud services under probabilistic regional failures, in: International Conference on Cloud Networking (CloudNet), IEEE, 2014, pp. 407–412.
[63] C. Wang, A. Jayaseelan, H. Kim, Comparing cloud content delivery networks for adaptive video streaming, in: International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 686–693.
[64] Z. Wu, H.V. Madhyastha, Understanding the latency benefits of multi-cloud webservice deployments, ACM SIGCOMM Comput. Commun. Rev. 43 (2) (2013) 13–20.
[65] J. Yu, M. Li, Cbt: A proximity-aware peer clustering system in large-scale bittorrent-like peer-to-peer networks, Comput. Commun. 31 (3) (2008) 591–602.
[66] S. Maheshwari, D. Raychaudhuri, I. Seskar, F. Bronzino, Scalability and performance evaluation of edge cloud systems for latency constrained applications, in: 2018 IEEE/ACM Symposium on Edge Computing (SEC), IEEE, 2018, pp. 286–299.
[67] Traceroute command, https://manpages.debian.org/buster/traceroute/traceroute.1.en.html, (Accessed 1 June 2020).
[68] R. Fontugne, C. Pelsser, E. Aben, R. Bush, Pinpointing delay and forwarding anomalies using large-scale traceroute measurements, in: Internet Measurement Conference (IMC), ACM, 2017, pp. 15–28.
[69] T.X. Tran, K. Chan, D. Pompili, Costa: Cost-aware service caching and task offloading assignment in mobile-edge computing, in: Annual International Conference on Sensing, Communication, and Networking (SECON), IEEE, 2019, pp. 1–9.

[70] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, G. Agha, Droplet: Distributed operator placement for IoT applications spanning edge and cloud resources, in: International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 1–8.
[71] C. Cicconetti, M. Conti, A. Passarella, Low-latency distributed computation offloading for pervasive environments, in: International Conference on Pervasive Computing and Communications (PerCom), IEEE, 2019, pp. 1–10.
[72] Y.-L. Jiang, Y.-S. Chen, S.-W. Yang, C.-H. Wu, Energy-efficient task offloading for time-sensitive applications in fog computing, IEEE Syst. J. 13 (3) (2018) 2930–2941.
[73] C. Cicconetti, M. Conti, A. Passarella, D. Sabella, Toward distributed computing environments with serverless solutions in edge systems, IEEE Commun. Mag. 58 (3) (2020) 40–46.
[74] M. Satyanarayanan, N. Davies, Augmenting cognition through edge computing, IEEE Comput. 52 (7) (2019) 37–46.
[75] Top command, https://manpages.debian.org/buster/procps/top.1.en.html, (Accessed 1 June 2020).
[76] P. Viola, M. Jones, Rapid object detection using a boosted cascade of simple features, in: Conference on Computer Vision and Pattern Recognition (CVPR), 1, IEEE, 2001, pp. 1–9.
[77] M.M. Rathore, A. Paul, W.-H. Hong, H. Seo, I. Awan, S. Saeed, Exploiting IoT and big data analytics: Defining smart digital city using real-time urban data, Sustainable Cities Soc. 40 (2018) 600–610.
[78] D. Mendez, M. Labrador, K. Ramachandran, Data interpolation for participatory sensing systems, Pervasive Mob. Comput. 9 (1) (2013) 132–148.
[79] D. Hasenfratz, O. Saukh, C. Walser, C. Hueglin, M. Fierz, T. Arn, J. Beutel, L. Thiele, Deriving high-resolution urban air pollution maps using mobile sensor nodes, Pervasive Mob. Comput. 16 (2015) 268–285.