Assessing the impact of unbalanced resources and communications in edge computing

Luiz Angelo Steffenel, Manuele Kirsch Pinheiro, Carine Souveyet

Version of Record: https://www.sciencedirect.com/science/article/pii/S1574119220301450
Manuscript_95ec6ebce65600b2d5b13c58d5d63053

Assessing the Impact of Unbalanced Resources and Communications in Edge Computing

Luiz Angelo Steffenel[a,*], Manuele Kirsch Pinheiro[b], Carine Souveyet[b]

[a] *Universit de Reims Champagne Ardenne, CReSTIC EA 3804*
Reims, France
[b] *Universit Paris 1 Panthon-Sorbonne, Centre de Recherche en Informatique*
Paris, France

Abstract

Edge computing relies on devices at the edge of the network to best meet the application's needs, such as low communication latency and data caching. However, unbalanced networks and resource heterogeneity are challenging factors when optimizing data transfers and applications' performances in the edge. In this paper, we explore these two concerns through a comprehensive set of benchmarks and illustrate their importance with the help of two elements that should be part of any edge computing toolkit: data locality and context awareness. Thanks to the benchmarks, we highlight some pitfalls and opportunities that may be met when trying to deploy applications on the edge.

Keywords: Data locality, Context-awareness, distributed scheduling, DHT storage, performance evaluation, communication bottleneck, edge computing, fog computing, cloud computing

1. Introduction

Edge computing appears as an emerging paradigm enabling the use of proximity resources for data processing. It advocates for the deployment at the network edge of services traditionally placed on cloud platforms. Just

*Corresponding author
Email addresses: angelo.steffenel@univ-reims.fr (Luiz Angelo Steffenel), manuele.kirsch-pinheiro@univ-paris1.fr (Manuele Kirsch Pinheiro), carine.souveyet@univ-paris1.fr (Carine Souveyet)

like the almost similar concept of fog computing, this paradigm is presented as an alternative to "pure cloud" solutions, particularly when considering telecommunications (5G networks, for example) or IoT (Internet of Things) applications. These applications can be very data-intensive, producing and transferring data to cloud platforms for further analysis or storage.

Many problems may arise from this pure cloud model, as underlined by several authors [1, 2, 3]. Among these problems, we may cite security and privacy concerns, but also latency issues. Data transfers between IoT devices and cloud platforms may be subject to considerable latency, which may negatively affect application behavior. For example, the data transfer to a remote facility may induce non-negligible delays and slow down data processing and decision-making. For this reason, next-generation telecommunication networks such as 5G rely on services running at the very antenna stations to provide very low latency applications to the final users [4]. Hence, Edge computing is a paradigm in which substantial computing and storage resources are placed at the Internet's edge in close proximity to mobile devices or sensors [5].

Applications that rely on distant services may also freeze or fail if the network connection is unstable or faulty. Placing such applications on edge devices may allow at least a partial operation even in cases of network failure, enough to respect SLA (Service-Level Agreement) commitments. By using resources at the edge of the network, edge computing also promotes the idea of exploiting resources near to data production devices, reducing amounts of data transferred between IoT devices and cloud platforms. Also, edge computing emphasizes QoS (Quality of Service) through the proximity to end-users, enabling superior users' experience and reduced latencies [6].

To be able to provide such service level, edge platforms should ensure that the services are deployed close to the end devices and that the computing resources are assigned according to their capabilities at run time. For this, they have to cope with some "defining characteristics" pointed out by Bonomi et al. [7], including location awareness, mobility, and heterogeneity. On the one hand, keeping the data local produces two main benefits: reducing wireless access traffic and easing Internet cloud load [6]. Achieving data locality is then one of the key objectives for edge computing [8]. On the other hand, nodes heterogeneity and mobility will affect application execution. Edge nodes are heterogeneous, possibly displaying a vast set of RAM (Random Access Memory) capacities, CPU (Computer Processing Unit) performances, storage supports, and network bandwidth. Also, connectivity to

edge nodes may suffer from unreliable or dynamic networks (Bluetooth, mobile, LoRaWAN, etc.) contrarily to cloud servers [9]. Sometimes, resources are even scattered over intermittently connected personal and vehicular platforms [6].

In other words, to fulfill edge promises and achieve reduced latency, we need to reinforce data locality through a conscious exploration of heterogeneous resources at the network edge. One question arises from this situation: how much the heterogeneity of resources and networks may compromise the performance of edge applications and platforms, and how can we prevent it? To tackle this question, two main aspects should be considered: (i) first, the effects of heterogeneity on communications and data management; and (ii) the effects of heterogeneity on the tasks' execution and placement.

Therefore, this paper explores these questions by performing practical experiments on a real edge platform, introduced in previous works [10]. This experimental platform allows us to set very heterogeneous environments, connecting HPC (High-Performance Computing) servers and cloud nodes to small nanocomputers, allowing us to investigate the heterogeneity impact in communications, data storage, and service execution. Hence, the current paper uses that platform to develop the preliminary ideas for data locality control in P2P (Peer-to-Peer) networks initially presented in [11, 12]. Thanks to the data locality control, we are able to evaluate the performance of uplink and downlink communications involved in a DHT (Distributed Hash Table) storage process, and at several degrees (from the local network to the cloud). All these observations allow us to compile a set of recommendations to optimize the deployment of edge computing and storage services.

Also, this paper presents a new section dedicated to task scheduling in a distributed computing edge environment. Using real nodes and detailed task-level tracking, we show how the overload of a node can impact the execution of tasks, and how context information may be used to improve the efficiency of the schedulers.

Therefore, the contributions of this paper include:

- A comprehensive benchmark of edge $\leftrightarrow$ cloud and edge $\leftrightarrow$ edge performances for data transmission and storage, obtained in real environments
- Benchmarks with devices presenting a variety of performance profiles and capabilities

- The evaluation of a data-locality strategy for DHT storage and the assessment of its impact on low-end edge devices
- The implementation of a context-aware scheduler and its benchmark on a network with heterogeneous computing devices

The remainder of this paper is organized as follows: Section 2 presents the background works and definitions around edge (and fog) computing, and discuss related work on edge computing benchmarks. Section 3 presents the main architecture and operation mechanisms that lie at the core of CloudFIT, the distributed computing framework used as execution platform for this work. Section 4 discusses data locality issues, presenting strategies to embrace data storage inside an edge computing platform, followed by real benchmarks and the analysis of the performances from these strateiges. Section 5 continues the discussion towards resource heterogeneity, suggesting strategies for context-awareness scheduling on the edge and presenting a set of experiments that illustrate the importance of context-aware scheduling on task execution on the edge. Finally, section 6 presents our conclusions and future research directions.

2. Background and Related Work

The dissemination of nearby devices with non-negligible computational capabilities promotes the production of large amounts of data as well as the integration of these devices into the data processing. More and more services that once were restricted to powerful servers are now being relocated close to the final users, such as in the case of artificial intelligence [13] or virtual reality games [14]. Over the last decade, several initiatives like pervasive grids [15], opportunistic edge computing [8], mobile cloud computing [6], mobile edge computing [16], cloudlets [17], edge-centered computing [18] or fog computing [7] have been proposed to move some applications and services closer to the end user. On the other hand, edge computing is a paradigm in which substantial computing and storage resources (cloudlets, micro datacenters, or fog nodes) are placed at the Internets edge in close proximity to mobile devices or sensors [5].

Today, both edge and fog computing domains share most definitions [19, 3, 9] and challenges. For example, the term "fog" was coined to express the idea of services surrounding users and data sources [7], enabling computing

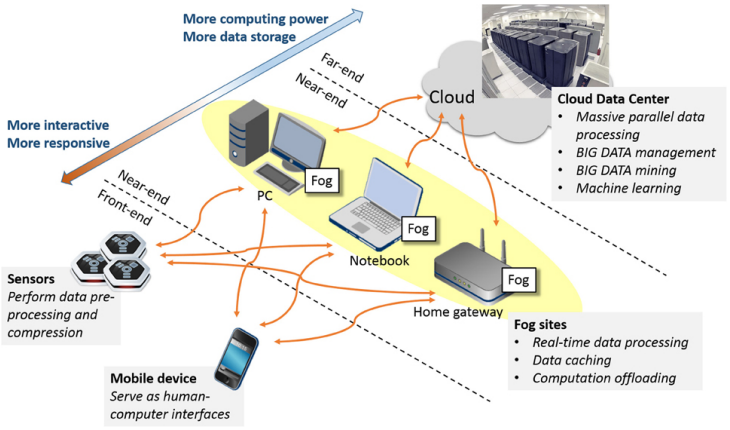

Figure 1: Conceptual architecture from a fog/edge infrastructure [20]

services to reside at the edge of the network, emphasizing proximity to end-users and resulting in superior user experience and redundancy in case of failure [6]. The similarities between edge and fog computing can be seen in the scenario from Figure 1, which depicts the relationship between IoT, fog nodes (which are located at the edge of the network), and the cloud [20].

Hence, from a more systemic point of view, edge computing can be defined as an application deployment strategy to explore data processing power at the border of a network, avoiding moving unnecessary data to a central data center. Fog computing complements this view by reasoning on how applications and data can be efficiently allocated in the continuum from the cloud to the edge and the end-user nodes.

As a result, the literature often struggles between two different visions. First, we have those works, such as [21, 22], who advocate that the main goal of edge platforms is to minimize communication with the cloud, without necessarily eliminate this remote element. In this vision, the cloud is always part of the equation, with the edge nodes acting to minimize the drawbacks of cloud computing (latency) or the limitations of mobile devices (computing power, for example). A second vision, suggested by [9, 17, 23], considers using edge resources for moving data processing and services to the edge of the network, without necessarily counting on cloud resources. They propose to take profit of nearby resources mainly for processing data nearest to their sources, minimizing or even discarding the use of cloud platforms. On this vision, the edge is not deployed to compensate the cloud drawbacks, its design

is driven by the new opportunities that surrounding resources may offer. According to Bonomi et al. [7], fog/edge computing enables a new breed of applications and services between the cloud and the edge, particularly when it comes to data management and analytics.

Despite their differences, both visions agree on the use of resources on the edge of the network for data processing and services execution, potentially reducing costs and latency and, hopefully, improving the user experience. They require leveraging resources that may not be continuously connected to a network and that may be limited in their computing power, storage space but also their battery lifetime [6]. Therefore, we consider in this paper that the term *edge computing* groups the many challenges of these weakly coupled environments, characterized by the heterogeneity of resources and tasks but also by their strategical situation, right between the cloud and the end users.

Indeed, edge computing is still a vast research domain where low-end devices and SOCs (System-on-a-Chip) are often evoked as proximity servers [24, 25], despite some efforts to re-brand and advertise edge computing as a pure industrial domain (such as in telco edge platforms, with edge servers and micro-clouds data centers installed in cell towers or street cabinets). Since nearby devices can perform basic services and data processing on the edge, applications and services need to be redefined to minimize the transfer of large amounts of data to cloud infrastructures. As IoT edge gateways or sink server can also be considered as edge nodes, applications and platforms deployed on these nodes must also handle heterogeneous network performances (WiFi, Bluetooth, LoRaWAN, mobile networks) and energy consumption constraints [5]. Therefore, edge computing is all about developing applications that shall be distributed to locations at the edge of the network to best meet the application's needs.

As noticed by [26], the challenges on edge computing include the deployment, the orchestration, the migration of tasks/services, the management of the networks, and so on. Most works try to minimize this complexity by relying on specific hardware (e.g., [27]) or on a centralized server to coordinate the resources (e.g., [28]). Other works rely on a "cloudification" approach, proposing edge architectures based on virtualization [17], micro-clouds [29], micro-services [30] or workflows [9]. Some standardization initiatives like *Open Edge Initiative* [31], *OpenFog Consortium* [32] or *Mobile Edge Computing* [33] also contribute to this problem, as they usually impose a complex software dependency or a non-negligible memory footprint that may prevent

the deployment in low-end devices. In our understanding, bringing the cloud to the edge thanks to the massive use of virtualization (with either hypervisors or containers) only postpones the handling of real edge-related problems.

When considering the challenges in an edge environment, the heterogeneity of edge devices is on the first line, requiring suitable data and task placement [34]. Indeed, frequent or heavy data transfers may negatively impact the network and the application performance on low-end devices, as we demonstrate on [35]. Transferring data is not just a network bandwidth/throughput issue but also depends on the nodes' capacities to handle data volumes. Quite often, uncoordinated communications may cause network bottlenecks or slow down applications due to the data handling overhead. As a consequence, the notions of data locality and context-awareness become key factors when considering job placement as they may also help coordinate/compartmentalize communications and reduce the overhead.

The challenge here is how to manage the resources and deploy services, as the edge needs to efficiently interpose between end users and the cloud without becoming dependent on the latter. Solutions relying on specific hardware [27] or centralized servers often limit the scalability and compatibility of the solutions. The same problem also affects solutions based on hybrid hierarchical topologies, such as [36, 34, 37, 38], in which a server, broker, or proxy is in charge of a set of nodes in a given perimeter. To prevent such drawbacks, P2P-based platforms can be seen as interesting alternatives for the interconnection, deployment and management of edge computing devices, as their loose coupling brings the flexibility, scalability, and fault tolerance required on edge environments [39, 40, 38].

In spite of a rich literature on fog and edge computing proposing infrastructures and platforms, a reduced number of works are dedicated to Edge computing benchmarking. Indeed, a recent survey[1] from [41] tries to identify these works and classify their contributions. It obseves that several works focus on developing benchmark tools that are widely inspired by other grid or cloud benchmark tools. In other cases, some works "*rely on either trace data obtained from simulators or simulators for evaluation (which is contrary to the classic definition of benchmarking)*" [41]. Furthermore, most works in the last years focus on specific hardware (Graphical Purpose Units - GPUs, Field Programmable Gate Arrays - FPGAs, mobile devices) or specific usages

[1]Still in pre-print status during the redaction of this paper

such as container migration, Artificial Intelligence offloading, etc.

Only a few works are sufficiently general to tackle the entire edge computing stack. Hence, [42] compares different Edge computing solutions from major cloud providers (Amazon AWS Greengrass and Microsoft Azure IoT Edge), but with a focus on edge-cloud exchanges inside each provider API (Application Programming Interface) that excludes inter-edge communications. One of the closest works to our paper is [43], which compares the performance of object-store systems such as Apache Cassandra or Ceph RADOS in a fog/edge scenario. Contrarily to the present paper, however, the work from [43] only focuses on the benchmark of storage solutions and does not account for other important aspects of Edge computing, such as scheduling and deployment of tasks and services.

3. The CloudFIT platform

In this section, we present the distributed computing platform CloudFIT, developed to be an experimental and lightweight platform for fog and edge computing research. CloudFIT is inspired by the FIIT (Finite number of Independent and Irregular Tasks) paradigm [44]. By definition, a FIIT problem can be broken down into a set of tasks that meet the following conditions:

1. a task can't make assumptions about the execution of another one task;
2. the execution time of a task is not predictable;
3. the same algorithm is used by all tasks, only the input changes.

This computation paradigm allows the representation of most parallel computing problems that do not require a strong dependency between tasks. It should be noted that this restriction can be circumvented if we synchronize jobs or tasks:

- **Job Synchronisation** Two or more jobs are executed in sequence, allowing a dependency check at the end of each run. This model corresponds to the BSP (Bulk-Synchronous Parallel) programming model [45], which is based on the succession of supersteps synchronized with a *barrier*. In the case of CloudFIT, if no assumption is made on the execution order of the tasks, the only constraint is that the data necessary for the next superstep is available at the time of the synchronization.

- **Task Synchronisation** This fine-grain synchronization provides dependency between tasks, like a *Directed Acyclic Graph* (DAG). To do

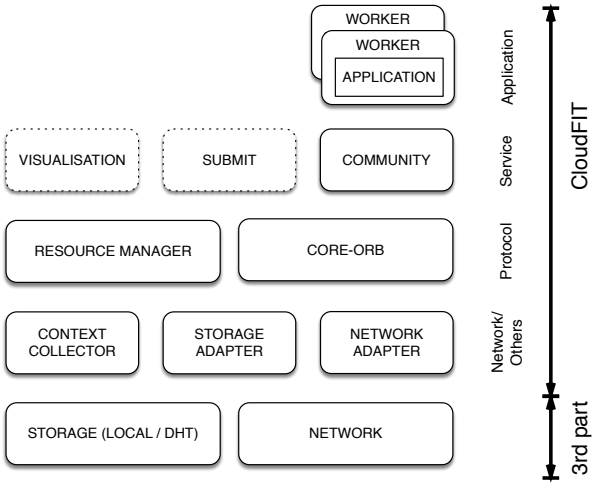

Figure 2: Simplified representation of the CloudFIT stack

this, we simply modify the task scheduler to take into account the status of the tasks and a list of dependencies: a task will only be started if the tasks on which it depends have already been completed. Although this violates the first property of the FIIT model (independence between tasks), its implementation is simple and allows the deployment of other types of applications.

3.1. Architecture

The CloudFIT architecture was designed as a software stack, inspired by the TCP/IP and OSI (Open Systems Interconnection) network models. Thus, we defined four layers representing the different functionalities of the platform, **Network**, **Protocol**, **Service** and **Application**.

Despite its name, the **Network** layer is responsible for all interactions with third-party systems on which CloudFIT relies(P2P overlays, operating system, storage systems). For example, the `Network Adapter` class performs the basic encapsulation and decapsulation of messages, using primitives designed according to the capabilities of the subordinated P2P overlays (*send*,*sendAll*,*receive*, and so on). The same principle applies to the `Storage Adapter` class, where primitives like *read*,*write*, *delete*, *lookup* interface with different possible storage solutions (local files, DHTs, databases, cloud storage). We finally find the classes related to the context collection, which

allow the scheduler to match the resources with the tasks/jobs constraints or requirements.

The **Protocol** layer is responsible for the management of messages and resource management. Thus, the module `Core-ORB` routes messages received from the Network layer to the appropriate services of the higher layer. Thanks to a *publish-subscriber* mechanism, different services can register with the `Core-ORB`, obtaining a unique identifier used for the reception of the messages but also for communications between the services within the same node.

The `Resource Manager` class, on its side, analyzes the information extracted by the context collector to check if the resources present in the machine are compatible with the application's needs. These needs are expressed as properties (required memory, disk space, etc.) provided by the application.

The **Service** layer contains the services needed to run distributed applications. This layer includes the `Community` class, an abstraction of a group of machines that drives the deployment of applications and manages events related to the nodes (input, output, message retransmission, etc.). CloudFIT defines a default community with all nodes on the network, but additional instances can be set to create subnets for specific needs (node partitioning, location awareness, etc.).

Other services in this layer include submission or visualization interfaces. These services are particularly useful when interacting with IoT devices that cannot run an instance of CloudFIT, such as Arduino micro-controllers. In this case, it is sufficient to provide access to a CloudFIT node through a REST (Representational State Transfer) or JSON (JavaScript Object Notation) interface for application triggering or data storage.

Finally, the **Application** layer contains the elements needed to execute the application, especially the application interface that must be implemented by the user. The application interface is quite simple and intuitive, following the principles of the FIIT paradigm. Thus, the developer only needs to write the following methods:

- **numberOfBlocks()** - method that returns the number of tasks to launch. This method is called during the configuration of the `Task Scheduler`;
- **executeBlock(taskID, required[])** - method that starts the actual execution of a task, delegated to one of the `Workers`. The *taskID* allows the task to customize its execution, and the *required [taskID]* element

indicates any dependencies for this task, a parameter used by the `Task Scheduler` to manage the execution order to respect the dependencies;

- **finalizeApplication()** - optional method that is executed by the `Task Scheduler` once the task set is completed. This method allows the aggregation of results, like a *Reduce* phase in the *MapReduce* paradigm.

3.2. Execution Model and Scheduling

Multiple applications can be concurrently executed over the CloudFIT network, according to the available resources and application requirements. Both regular applications (mono-server) and distributed applications can be submitted, as CloudFIT coordinated the deployment and communication in the network. To do so, an application must be expressed as a finite set of independent irregular tasks that share the same execution code but work on different data segments, just like an MPI (Message Passing Interface) application. Please note that this strategy can be extended by adding dependencies between applications (like in the BSP programming model) or tasks (like in a Directed Acyclic Graph - DAG), allowing for example the implementation of *map-reduce* applications.

To implement this, each community is associated with a `Job Scheduler`, which manages the submitted application queue (*jobs*). The `Job Scheduler` is in tight collaboration with the `Ressource Manager`, both for the scaling of the `Workers` pool and the verification of the requirements of the job. Also, a `Task Scheduler` is instantiated for each started job, allowing specific scheduling policies. Figure 3 shows in a simplified way the interaction between the `Task Scheduler`, the application, and the other elements of the Service layer: when a new job is received, it is assigned to the `Job Scheduler` and then to the `Task Scheduler`, which is responsible for deploying the tasks.

Please note that the `Task Scheduler` class is extensible and customizable. By default, CloudFIT provides a simple scheduler that it can be replaced by schedulers more elaborate or particularly adapted to the needs of applications. The default scheduler performs a random redistribution of tasks, a simple technique that reduces the risk of duplicate work between nodes. Examples of more sophisticated schedulers include those that support dependencies between tasks (in the case of a DAG application) or that use contextual elements, such as the location of a node to minimize the data access time.

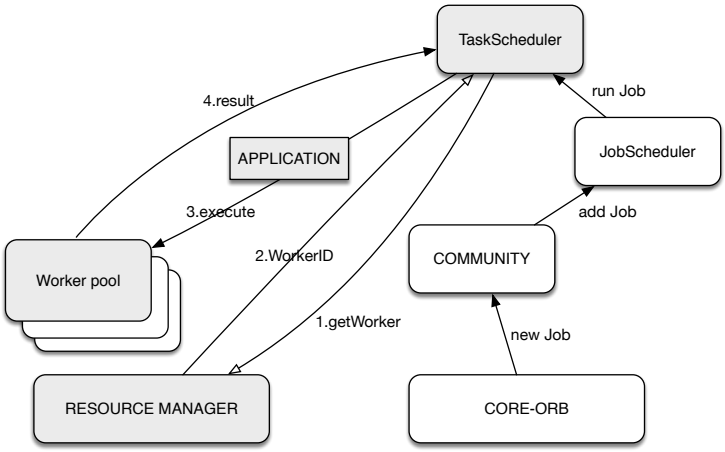

Figure 3: Coordination between Job and Task schedulers

The choice of each job and task to execute is performed by a distributed scheduling mechanism using a two-layer scheduler. This two-layered scheduler is partially inspired by the *Fair Scheduler* strategy from Apache Hadoop [46], which allows multiple jobs to run concurrently instead of blocking the execution queue in case of resource starvation.

In CloudFIT, the first layer is the "job" scheduler, which is enriched with a context manager that helps verifying requirements expressed by each job. Context information is used to establish if a node is currently able to handle a job or a task. This context information is composed of the nodes' current status (idle or working, CPU usage, available memory), the nodes location, and the node specifications (CPU type and speed, total memory, disk space). Node specifications are used to create predefined communities corresponding to typical application requirements, so a job can also express its needs in terms of "required communities". Once validated, the application is deployed over one or more nodes that currently match the requirements, otherwise it remains in a waiting queue until resources become free. While the primary objective of the job scheduler is to guarantee resources that fit the job requirements, it also allows load balancing, QoS, and concurrency management. By distributing jobs according to the nodes' current execution context, i.e., if two communities match the minimal requirements, the scheduler will deploy the job on the community that is less overloaded even if it is not the "most powerful" community.

On the second layer, we found the "tasks" scheduler, a per-job sched-

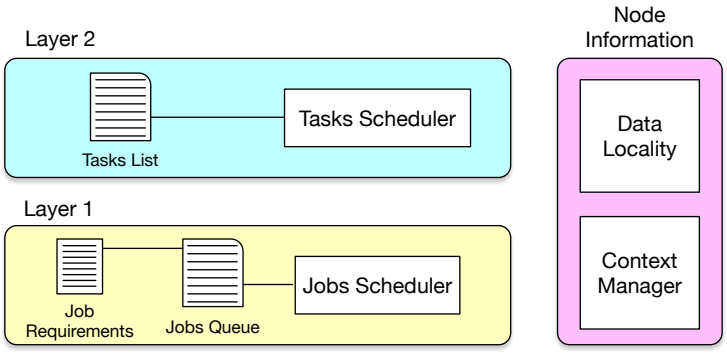

Figure 4: Coordination between Job and Task schedulers and context information

uler that coordinates the execution of the tasks on each node. The basic scheduler algorithm in CloudFIT considers tasks as fully independent, so each node randomly starts a task and then diffuses the status of the tasks to the other nodes (*in execution*, *completed*), updating the list of tasks still available for execution. However, as this scheduler is inherent to each job, it can be extended in several ways and applied according to the job's requirements. Therefore, to implement a data-locality aware scheduling, the presence of a data resource in the local storage is used as a factor to determine the tasks' execution order. By assigning dependencies between tasks and data resources, we instrument the scheduler to prioritize tasks that have the required datasets already at hand. As shown in Fig. 4, both layers base their scheduling decisions on data locality and context information, which are constantly updated according to the nodes' conditions.

Additional context elements such as the currently available memory, CPU load, and disk space also permit the task scheduler to decide at runtime if a task can be started, avoiding penalizing an application because of an overloaded node. This context information is collected by a context collector like the one presented in Fig. 5, which is integrated into the CloudFIT stack. This collector is based on the standard Java monitoring API, which allows us to easily access the real characteristics of a node. It allows collecting different context information like the number of processors (cores) and the system memory using a set of interface and abstract/concrete classes that generalize the collecting process. Besides, due to its design, it is easy to integrate new collectors and improve the resources available for the scheduling process, providing data about the CPU load or disk usage, for example [47].

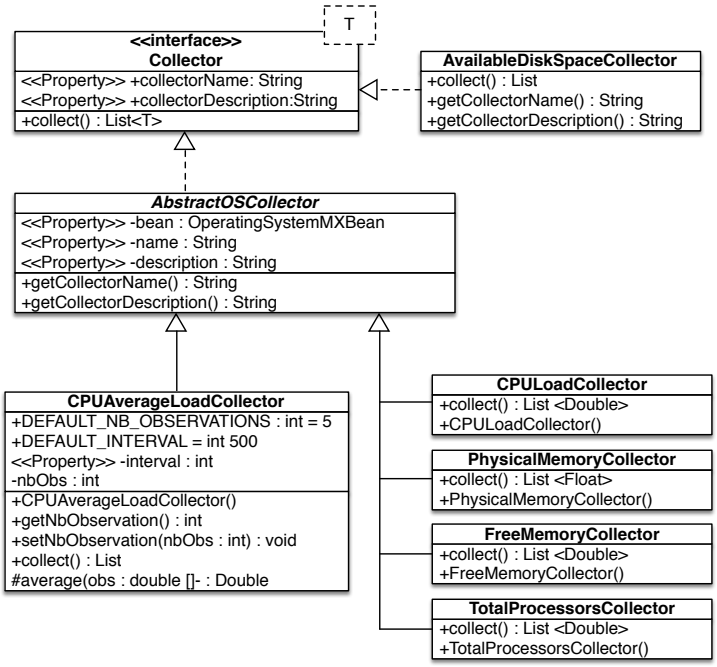

Figure 5: Context collector structure [47]

3.3. Distributed Coordination

The previous section illustrates the structuring and interaction of modules within a CloudFIT instance. However, a distributed computing platform must guarantee exchanges between the different nodes on the network. In the case of CloudFIT, the choice to use a third-party P2P overlay simplifies peer discovery operations, network management (inputs, outputs), message routing, etc. This allows us to focus on intrinsic platform communication, such as application deployment, execution progress monitoring, and result distribution/retrieval.

Everything starts with the submission of a job, carried out directly by a node already connected to the network or through a submission interface. This submission contains the application code, the target community, and a list of properties required for the job to run properly. A message containing the parameters and properties of the submission is broadcasted over the network, thanks to the P2P overlay communication mechanisms.

To automatically deploy the application on the executing nodes, we chose to use the DHT storage associated with the P2P overlay. Indeed, DHT provides an access to objects and files, from the moment the nodes know

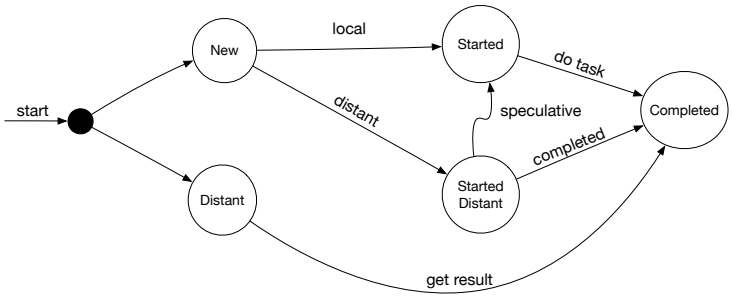

Figure 6: Lifecycle of a Job/Task submission in CloudFIT

the DHT key to these resources. Thus, submitting a job involves saving a jar file containing the application code and its storage key. When the job is launched, the `JobScheduler` retrieves the file and extracts its classes, which will be loaded using a *classloader*.

DHT storage can also be used to provide input data for the applications, especially when a large set of data needs to be made available to the applications. This is not mandatory, since applications also can obtain the data from external services (Uniform Resource Locators - URLs, cloud storage, databases).

When a job starts on a machine, it is tagged STARTED and a Task Scheduler associated with the job is launched. Tasks have 5 possible states: NEW, STARTED, STARTED_REMOTE, COMPLETED, and REMOTE, as illustrated by the lifecycle shown in Figure 6.

When a task is launched, a message containing the job and task ID is sent to the other nodes in the Community. This informs other nodes that this task is being processed by someone and thus minimizing the risk of duplicated work: "remote" tasks are marked as STARTED_REMOTE and are moved to the end of the execution queue. A task marked as STARTED _REMOTE will only be executed when all NEW tasks have been exhausted and, of course, if no other message has come up indicating that the task has been completed. In fact, the Task Scheduler sends a second message at the end of the execution of a task, indicating the change of its status to COMPLETED and also indicating the result values or the coordinates to retrieve this result, whether it is stored in the DHT or in an external resource.

If a node finishes executing all its NEW tasks, it can speculatively start tasks marked STARTED_REMOTE. This mechanism guarantees the termination of all tasks (if the original node fails, for example) and even speeds

up the termination of the computing if the original node is too slow.

In addition to updating the other nodes, these message exchanges also have the role of notifying a new node joining the network. By seeing "task completed" type messages, nodes can ask a neighbor to forward the message with the job description to them. Its TaskManager will then proceed to the recovery of REMOTE jobs thanks to specific requests. This mechanism guarantees the integration of the nodes in a volatile environment and the durability of the results. It suffices that one machine remains in the network to keep results accessible.

When all tasks are completed, the TaskManager gathers all results, optionally performing additional operations, and the status of the job is set to COMPLETED. These results remains available to any application or node that wishes to query them.

4. Data Transfer in Unbalanced Networks

When running applications the edge, it is important to consider the data access and data management performance, since most operations involve the collection, transformation, and analysis of data. Indeed, the cost of data transfers is an essential factor for edge applications. When looking at Big Data solutions, which are confronted with similar situations, we observe that platforms such as Apache Hadoop [46] use the concept of data locality to prioritize computing on nodes that hold a copy of the data to be processed.

Besides data transfer issues, the heterogeneity of edge environments can also impact the application deployment since devices may have very different capabilities and constraints. In [10], we observed the effects of the heterogeneity on the application performance, highlighting the importance of observing the nodes' status before assigning jobs for execution. Thus, we identified two key points to be addressed by an edge computing middleware: (*i*) the support for data locality and (*ii*) its integration with the task scheduling. In this section, we introduce some strategies to improve the control over data locality and present several benchmarks for data storage in local, edge, and cloud servers. The analysis of these results demonstrates that both locality and nodes' capacity need to be considered when designing an edge application.

4.1. Data Locality in a P2P overlay

When handling data on the edge, the first factor to consider is where data lies (the data locality). For any advanced application, edge nodes are not

only gateways to the cloud but, instead, play an active role in data sharing, replication, and load balancing [48].

To reduce the dependency on cloud storage services, P2P overlays can be used to provide decentralized data management [40] with the help of a Distributed Hash Table (DHT). DHTs use cryptographic hashing functions such as Message-Digest 5 (MD5) or Secure Hash Algorithm (SHA), providing a lookup service in which any node can efficiently retrieve the value associated with a given key. In the case of DHT storage, hashing is used to assign resources (data) to nodes that will hold a copy of this data.

Most DHT overlays use plain hashing functions that uniformly distribute and replicate their data across the network. Uniform hashing allows load balancing among the nodes and, in most cases, helps to prevent data losses when clients disconnect. However, it makes data transfer optimization harder since data locality information is lost [49, 43]. While some non-uniform hashing functions exist [50], basic DHT storage is insufficient for data management in edge platforms, as we seek to control where data is stored.

In recent years, the development of geographical databases and NoSQL boosted the research for performance improvement through co-located data [51]. Most of these works concern specific data types like those found on geographical databases, whose queries can benefit from grouping data in a given area. Hence, one can express the data location as a set of coordinates, used to generate a geohash key [52], or to identify zones like in CAN (Content Addressable Network) [53], quad-tree hashing [54] or in a Voronoi overlay network [55]. [43] propose the use of external storage services such as Cassandra (a P2P database), which can be configured to segment the network in data centers. In all these works, the network partition scheme is often predefined, which created a strong dependency between the partition, the nodes, and the applications that can be executed on these nodes.

As none of the cited works efficiently deal with dynamic environments and ephemeral applications deployment, we proposed in [11] a pure P2P solution that ensures that data segments for the application are distributed preferentially among the nodes that will execute that application, all in a dynamic environment where nodes can join or leave the network.

To do this, we need to extend the concept of data locality to accommodate the principle of group membership. The group membership (that we call "community") represents a subset of the system nodes and may be statically or dynamically defined. In the first case, previous knowledge of the network and the application requirements drives the assignment of a group

of nodes to perform some specific role. Indeed, a community may include the nodes covering a specific area (used to host a proximity service), nodes that present similar computing performances (same CPU type, for example), etc. In the second case, dynamic factors such as the status and context of the nodes, or even the evolution of the network (nodes joining or leaving), are used to form temporary associations between nodes in order to execute an application. In both cases, different communities may be created to fulfill specific requirements (nodes capabilities, location or security requirements, etc.), and a node may belong to one or more communities.

As both static and dynamic communities require data locality to perform correctly, we proposed to manipulate the hash key so that data is not randomly spread among all nodes but attached to a community. While a typical DHT uses a single hashing function that computes the data hash (the content key) and the node ID (the location key), we rely on a double hashing function that decouples the location and the content key for a resource: first, the content key is obtained through a traditional hashing method. Later, the location key is computed to map only among the community nodes. To query a resource, two strategies can be applied. If the community is statically defined, one can rely on the same $hash_{ca}()$ function to find the resource. If the membership varies with the time (nodes joining or leaving the community), an index file keeps track of the resources' location, despite the eventual mapping skew when the group changes. Both operations (write and read) are schematized in Figures 7 and 8, where we consider that applications and storage services are interconnected by a "DHT" component.

To illustrate how this strategy reinforces data locality, we present in Figure 9 a P2P network with several nodes and a community C_1 composed by nodes with IDs n_2, n_3 and n_6 (the yellow blob). We also assume that a node n is the primary storage site for a resource with location key kl_n. For instance, in a traditional P2P storage with a single location key, a resource R could be stored in any node in the network, depending on the hash result (e.g.: $hash(R) = kl_n$). On the contrary, by using a location key and a content key with a community-aware hash function $hash_{ca}()$, we can bound the location key to the nodes in the community, while properly identifying a resource. Hence, for a resource R and the community C_1, we can compute a community-aware location key kl_3 that points to a node from this community. This way, the primary copy of the resource R will be located in the node n_3. As a result, the proposed mapping procedure improves the probability that the primary copy of a resource lies in a node belonging to the group.

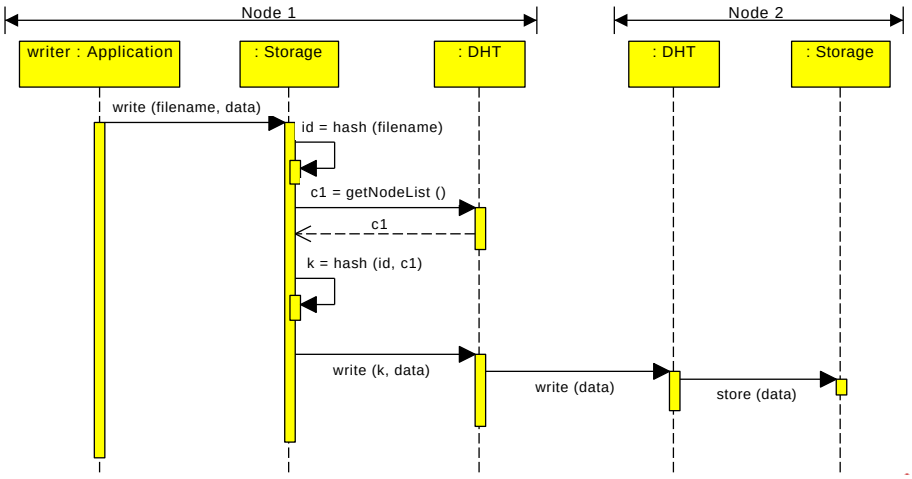

Figure 7: Schematic representation of a write operation with data locality

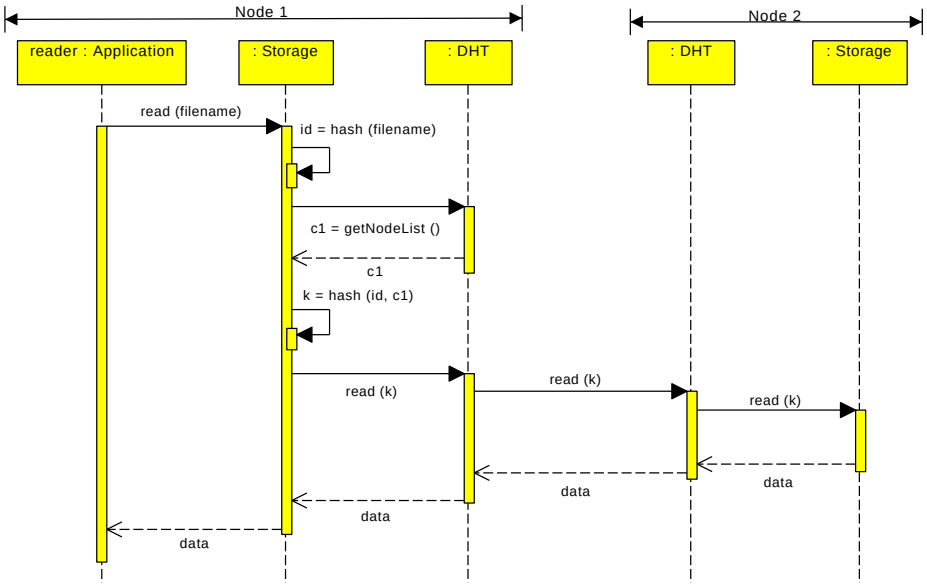

Figure 8: Schematic representation of a read operation with data locality

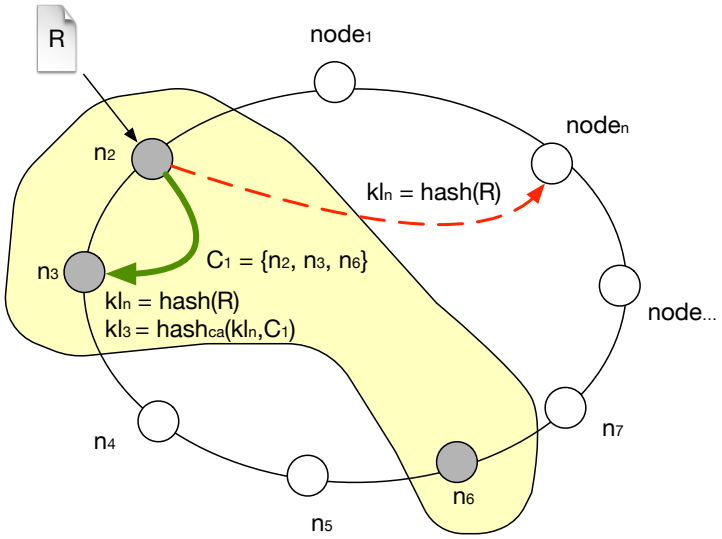

Figure 9: Location key remapping to reinforce data-locality [11]

Please note that this procedure does not restrain the creation of additional replicas on other nodes, even those outside the community, just ensures that at least one copy of the data is located within the community.

This mapping can be implemented on any DHT overlay through the use of specific hashing functions. To validate this approach, we implemented it using our distributed computing platform CloudFIT [10]. CloudFIT is a lightweight middleware adapted for edge and fog environments, structured around collaborative nodes connected over a P2P overlay network that provides communication, fault-tolerance, and distributed storage. The TomP2P[2] overlay network provides discovery, communication, and storage services, which uses up to four keys k_l, k_d, k_c, k_v to identify a resource, instead of a single key:

1. k_l - location key, which determines the node ID closest to the hash key;
2. k_d - domain key, used for namespacing;
3. k_c - content key, which identifies different resources stored in the same location; and
4. k_v - version key, which allows the management of different versions of the same resource.

[2] `https://tomp2p.net/`

The next sections present benchmarks conducted to study the impact of data locality on the storage of datasets on an edge environment. We also extend this experiment towards low-end devices such as Raspberry Pi nodes, whose storage and network performances have a strong influence on storage performance.

4.2. Round 1: Local vs Cloud Storage

While Big data strongly relies on data locality to improve its performance, it usually considers quasi-homogeneous networks with high-performance interconnections. When dealing with edge computing, however, data locality has a potentially higher impact as not only communications and data transfers can span from the local network to the cloud, but also because devices are inherently heterogeneous. We present here different experiments to evaluate the impact of network latency on the performance of a DHT. By default, CloudFIT (the fog/edge platform used for the benchmarks) supports a robust distributed scheduling based on work-stealing, allowing nodes to execute different tasks in parallel when they can communicate to each other or, in the worst case, to complete all the computation by itself. This original scheduling was modified to include the possibility to reinforce data locality, as proposed in Section 4.1, for better control on the data storage and retrieval operations.

In a first experiment, we used the CloudFIT platform to interconnect nodes in a local cluster and nodes in the Google Cloud Platform (GCP). The local machines are identical (AMD Opteron 6164 HE, 12 cores, 48 GB RAM) and interconnected by a Fast Ethernet network. GCP machines are of type *n1-standard-2* (2vCPU, 7.5 GB RAM) and located at the *us-central1-c* zone.

Three situations were considered: data storage/access in the same machine, in a close node (same LAN or enterprise network), and over distant nodes lying in the cloud. The first case represents the situations where the application can obtain data directly from its own DHT instance, or when it wishes to store data locally to favor a subsequent task (therefore, using data locality). The *same network* situation represents an intermediate case where the data is located in a nearby server (perhaps a node that has more storage capabilities and serves other nodes in the network). Finally, the last situation represents the cases where the data lies far from the node, like when data-locality techniques were not used.

The experiments performed write and read operations on the DHT, with file sizes of 1 kB, 10 kB, 100 kB, 1 MB, 10 MB, and 100 MB. These file sizes cover the typical Big data storage block sizes: for example, Hadoop HDFS

uses 64 MB or 128 MB data blocks. At each run, the CloudFIT platform was shut down and the files were removed. Also, the order of the files was shuffled to prevent *pipeline* effects on the network, and at least 10 runs were executed for each situation.

Figure 10 represents the average for each combination of read or write operations, data size, and situation (same node, same network, on the cloud). We can observe that both local node and network cases present similar performances when reading small messages. For large data blocks, the reading time over the LAN becomes more important, scaling to levels almost comparable to those of remote nodes. This slowdown evidences not only the network limitations (a 100 Mbps network) but the overhead of requesting a resource through the DHT.

Regarding the write operation from Figure 11, an interesting effect is observed: up to 1 MB, it is more expensive to write files in the local node than in another node on the LAN. This could be explained by the DHT management overhead, as the cost of data handling (serialization $\rightarrow$ transmission $\rightarrow$ deserialization) is shared between different nodes. Also, communication policies for small messages (buffering, Nagle's algorithm, TCP_NODELAY option, etc.) may delay transmission is some cases and not in others. Even the cloud seems more efficient when writing small messages (less than 10kB), but latency and throughput rapidly increase the cost of cloud operations. As such discrepancy only is observed on really small filesizes, we believe that its impact on applications that rely on DHT data storage is limited.

Both read and write observations encourage the use of data locality to improve the performance of edge applications. Indeed, if we can control the community boundaries, it is possible to finely adjust the distribution pattern according to in/out performances and data sizes.

4.3. Round 2: The Case of Low-end Devices

The previous scenario allows us to understand how reading and writing performances are affected by the data locality and the importance of locality-awareness during the data placement and the scheduling of tasks. However, those experiments were conducted with "regular" machines on both the local cluster and the cloud. In this scenario, we are interested in the performances of low-end devices based on SoCs such as a Raspberry Pi.

Several works on fog and edge computing suggest the use of SoC devices as "proximity relays" or gateways for the users and IoT devices. Most times, these devices perform basic computing and data processing, but more and

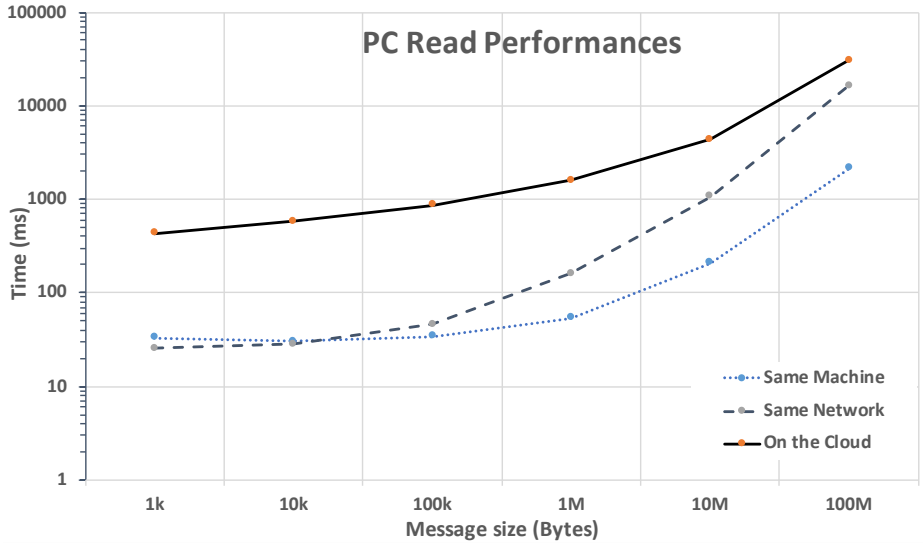

Figure 10: Performance comparison when reading data from the DHT

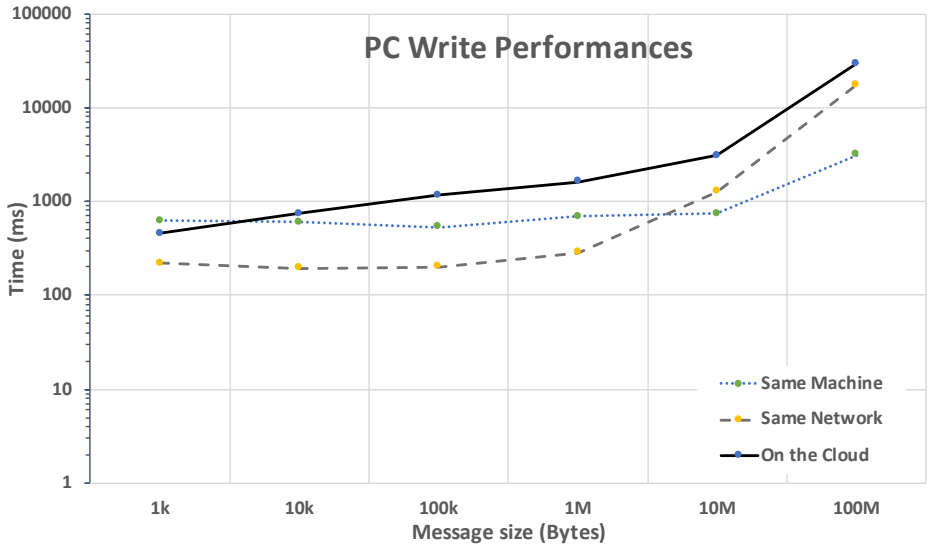

Figure 11: Performance comparison when writing data from the DHT

more works try to deploy advanced data processing thanks to artificial intelligence algorithms. The relatively small cost and size of these devices favor their dissemination and integration in the environment, complementing the fog ecosystem. Even if new SoC generations tend to integrate more advanced features like GPUs or even AI accelerators, they still have several limitations (CPU speed, memory, I/O) that must be considered.

Therefore, this experiment interconnects a Raspberry Pi 3 (4-core ARM Cortex-A53, 1.2 GHz, 1 GB RAM) to a PC (AMD Opteron 6164 HE, 12 cores, 48 GB RAM) and to a cloud instance of type *n1-standard-2* (2 vCPU, 7.5 GB RAM), located at the Google Cloud Platform *us-central1-c* zone. Like before, the experiments focus on performing read/write to and from the Raspberry Pi, according to the "same machine" and "same cluster" and "cloud" locality scenarios previously described.

Besides, we conducted the experiences using different storage supports on the Raspberry Pi: the built-in MicroSD memory card (class 10), an external USB 2.0 hard disk, and memory-only storage. The comparison between the MicroSD and the external USB disk drive is motivated by their relative performance differences: a class 10 MicroSD should support at least at 10 MB/s when writing (with a reading speed depending on the constructor), while a USB external disk on a Raspberry Pi 3 can reach at most 20 MB/s.

Further, as both methods suffer from bottlenecks due to the Raspberry Pi chipset configuration, we also included memory-only storage tests in our experiments. This latter option could benefit volatile datasets that will be discarded after being processed, like in the example of "gateway" nodes that store intermediate data and preprocess it before sending it to other nodes. The experiment protocol (number of runs, DHT reset, etc.) is similar to the previous benchmark.

Taking the "same machine" case, Figure 12 shows the performances when the Raspberry Pi tries to save data to the DHT address at its own location, and when it reads data from that address. With no surprise, the memory storage option is faster on writing, as it does not pay the overload of accessing the persistent storage. Concerning the MicroSD and the USB disk storage, their performances are in the same order of magnitude. Nevertheless, a closer look at the performance numbers shows that the USB disk should be preferred, as it needs 22.7s on average to write 100 MB on the DHT, against 27.3s for the MicroSD (a 16% improvement).

Concerning the DHT access (reading), all three storage options show similar performances. One possible explanation is that the DHT implementation

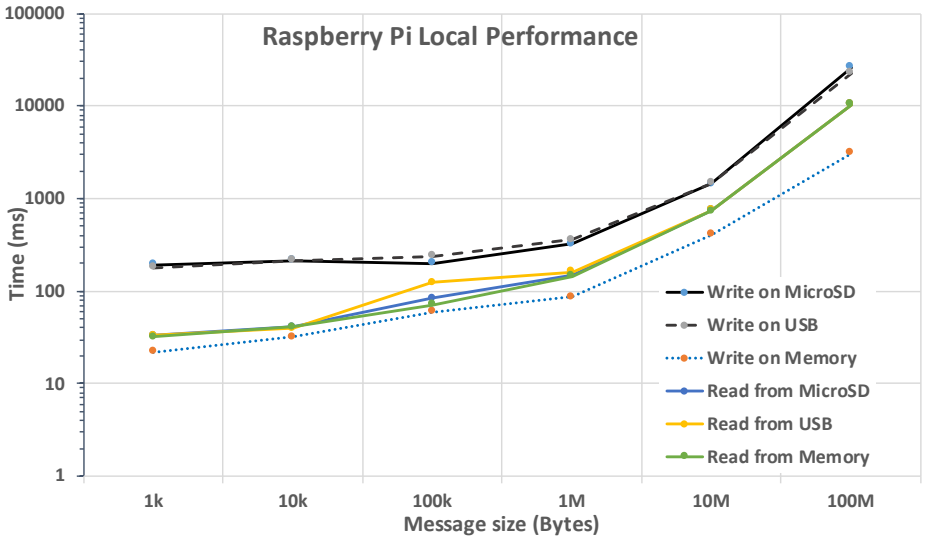

Figure 12: Raspberry Pi read/write performances on local host

keeps recent data in the memory, hiding the extra cost of accessing the storage units. Further improvements on the benchmarking protocol should help study this result, and also to identify why writing on the memory is faster than any read operation.

In the case of the "same cluster" scenario, we performed benchmarks in two directions: the Raspberry Pi performing read/write in the other machine and the remote machine performing read/write on the Raspberry Pi. This is due to the performance difference between the machines (processing, memory, and storage speeds), which leads to asymmetric performances according to the transfer direction. Hence, Figs. 13 and 14 show the writing and reading performances when the Raspberry access the other node in the same network, while Figs. 15 and 16 present the performances when the other node access the Raspberry Pi.

For the writing operations initiated by the Raspberry Pi (Figure 13), we observe that all storage supports on the PC or the cloud (disk and memory) behave similarly for large data sets. This means that, at least for 100 MB of data, the DHT storage caused no bottleneck on the disk access. This scenario is quite different in the opposite direction (Figure 15), where the cost of storing data in the Raspberry Pi is much higher if it involves persistent storage. Knowing this information is useful as it can help identify the best role for SoC devices in the edge network.

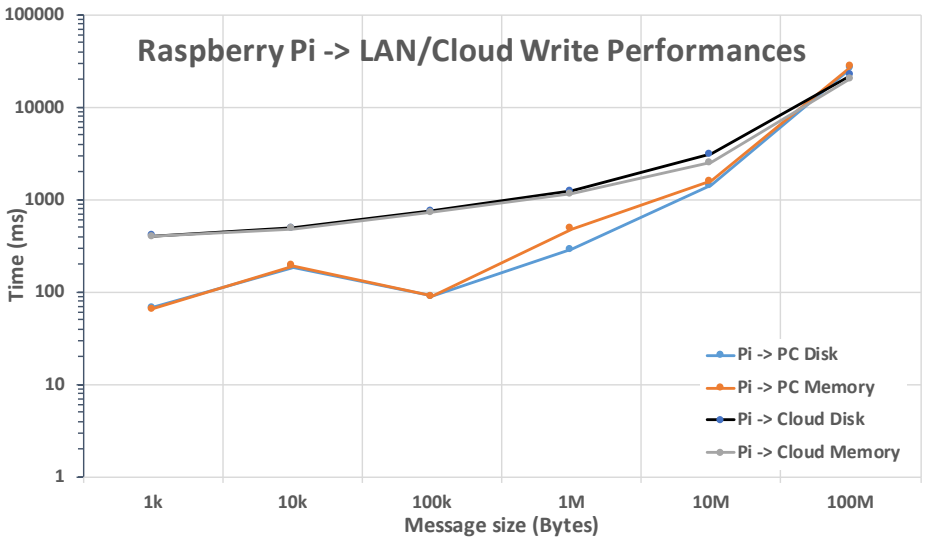

Figure 13: Raspberry Pi to PC: Write performance

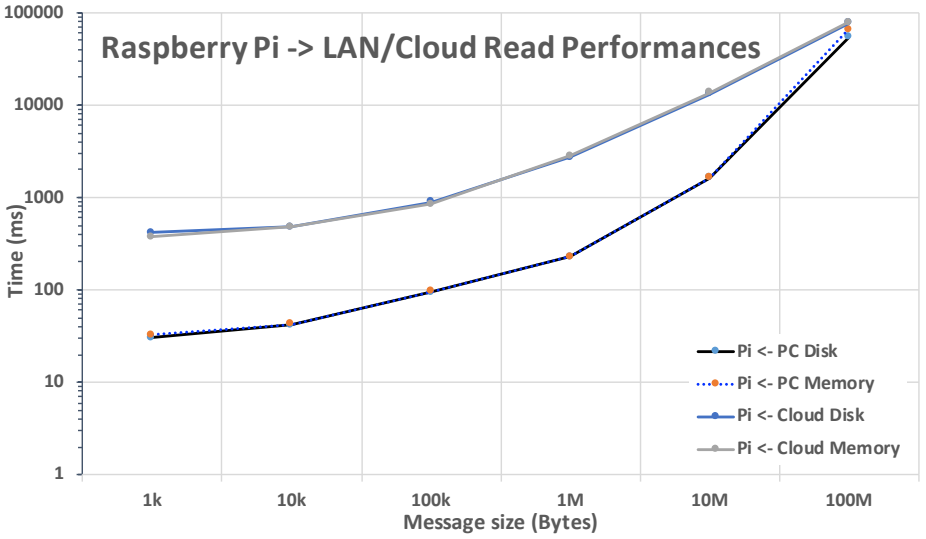

Figure 14: Raspberry Pi to PC: Read performance

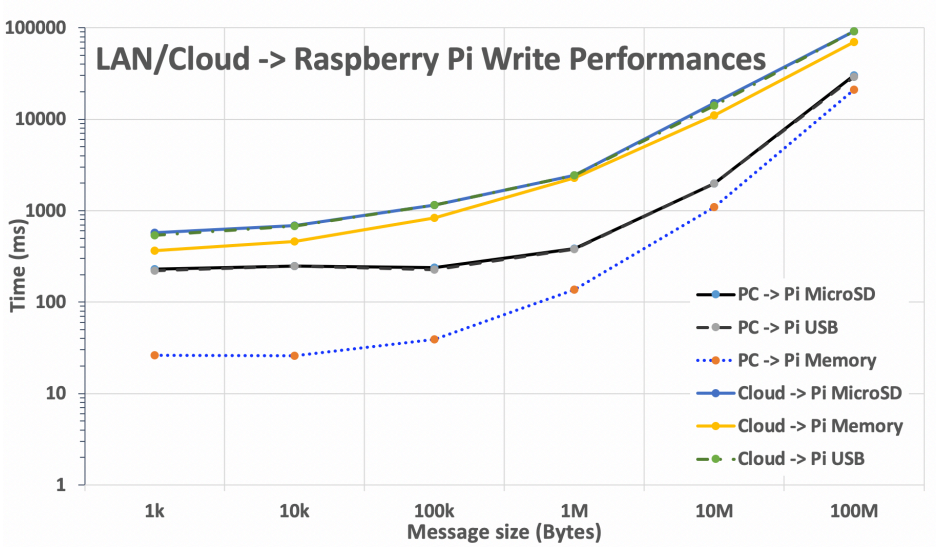

Figure 15: PC to Raspberry Pi: Write performance

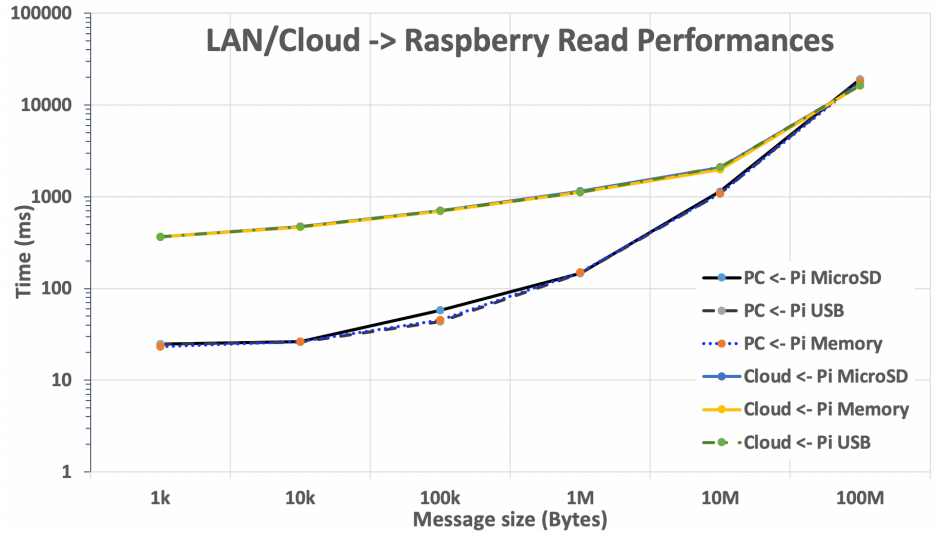

Figure 16: PC to Raspberry Pi: Read performance

This asymmetrical performance is also seen in the read benchmark. While the reading performance is similar whether we use disk or memory storage, the absolute values are several orders different according to the direction of the operation (PC $\rightarrow$ Pi or Pi $\rightarrow$ PC). Hence, the cost for a Raspberry Pi to read 100 MB from an external node is almost three times higher than when reading from the Raspberry Pi. As this difference disappears for data smaller than 1 MB, we suspect that the overhead is related to the buffering and serialization/deserialization operations on the receiver node. While further investigation should help understand the causes of this unbalance and help to minimize it, we believe that being aware of the capabilities and limitations of the devices at the edge would allow the development of better resource managers and scheduling algorithms.

For instance, the results from these benchmarks suggest that low-end devices are better suited to slowly gather data from nearby devices (sensors, IoT), perform small computations, and serve this data to other nodes, instead of receiving large data volumes all at once. If possible, the aggregated data should be sent to other nodes as soon as possible (i.e., while their amount is still limited), or at least be stored locally on the low-end device for others to read, as access from the outside is less expensive for large files than sending them out. Furthermore, data size is a key aspect to consider, and the experiments point that, at least with the TomP2P DHT, data chunks should not exceed 10 MB as all scenarios exhibit substantial cost overheads when manipulating large datasets.

To conclude this section, these experiments demonstrate that edge computing should not only be concerned by the latency impact on the services but also on the capability of each computing node to perform its tasks efficiently. Choosing whether to deploy an application is a complex task, with several strategies and optimization heuristics to consider [56]. Hence, as artificial intelligence applications become faster and less demanding in resources, IoT, mobile, and edge devices are now in the front line. Similarly to the work of [57] on energy-awareness, intelligent planning on the data flow in edge environments can lead to significant improvements in the data storage, transfer, and processing performances, benefiting both the users and the service providers.

5. Scheduling with Context-Awareness

After considering data locality and network performance issues, we can now address another topic related to the heterogeneity of the edge. As the computing performance varies from device to device, context information such as processing power, available memory, or current CPU load can be useful to improve the execution performance, as demonstrated by [16]. Therefore, efficiently matching the tasks with the resources capabilities and their locations is a fundamental element for the optimization of performance-sensitive edge applications [58]. In the next sections we discuss this issue (section 5.1) and analyzed it through practical experiments (Section 5.2).

Finally, due to the heterogeneity that may characterize edge nodes, data and service placement must also consider runtime conditions on each node, preferring nodes that have available resources instead of nodes that are already overloaded by other tasks. In other words, nodes execution *context* should be considered for scheduling purposes.

5.1. Heterogeneity and Context-Awareness

As stated before, edge environments present important variations on their nodes' characteristics. Contrary to cloud resources, the resources at the edge can be: (i) constrained, i.e., limited computational resources that cannot be easily scaled; (ii) heterogeneous, including processors with different architectures; and (iii) dynamic, since their workloads change, and applications compete for the limited resources [59].

Edge platforms have to cope with such dynamic and heterogeneous pool of resources, whose composition and status may vary during execution, and which are not necessarily dedicated to these platforms, since these resources may house multiple applications from multiple users (following classification given by [59]). Such heterogeneity makes resource management a challenging task, particularly when compared to cloud environments [59, 60]. Since applications executing on edge platforms can be spread over multiple heterogeneous nodes, deciding where to schedule computational tasks is more difficult when compared to cloud computing [9]. Indeed, data and service placement must consider runtime conditions on each node to consume resources efficiently, preferring nodes that have available resources instead of nodes that are already overloaded by other tasks.

Edge platforms should then carefully consider the capabilities of available nodes and their current state before scheduling tasks on those nodes. As

recalled by [34], "Task scheduling decisions cannot be made in a statical way but need to be done dynamically upon task creation". Indeed, in a heterogeneous environment, scheduling tasks without considering the current status of the nodes may easily lead to a waste of available resources.

More than ever, context awareness [61] becomes a key property for handling appropriately the heterogeneity and dynamicity that characterize edge nodes. Context-awareness can be defined as the capability a system has to observe its execution context and to adapt its behavior consequently [61]. Context includes any information that can be used to characterize the situation of an entity (a person, a place, an object, etc.) considered as relevant to the interaction between a user and a system [62]. Different information can be considered as context, according to the system purposes [63]: available memory, CPU load, available storage, network connection, etc. Works such as [64, 47] demonstrate the interest of considering context information for scheduling purposes in such heterogeneous environments, while [34, 37] have tackled this question in fog environments. Both [34, 37] collect and use context information for estimating nodes stability and reliability in order to better place tasks [37] and data replicas [34]. Unfortunately, in both cases, the presence of a broker can be noticed, acting as a central manager. According to Ghobaei-Arani et al. [60], a centralized manager suffers from a single point of failure, which makes it vulnerable. Also, since a centralized core is responsible for processing and managing decisions in centralized approaches, it limits scalability based on its bandwidth and processing capacity. On the opposite, decentralized approaches are highly scalable, but they introduce a certain communication overhead.

In this work, we have thus considered the possibility of adopting a totally distributed approach and studying the possible effects that variations on the execution context may have on task execution. For doing so, we also base our experiments on the CloudFIT platform.

As detailed in section 3.2, Context-awareness is achieved by allowing nodes to decide by themselves about their capability to execute a task. By exchanging information with other nodes, they can drive the placement of replicas and tasks, resulting in a distributed architecture in which nodes collaborate to satisfy the user's demands. The scheduling mechanism we use was initially proposed in [12] and is structured in two layers that observe both the execution and the data locality contexts. We call "job" the application to be executed, which in turn describes its needs and may split its execution into several "tasks" that may be carried independently by the platform

nodes.

Hence, in the first layer of the scheduling mechanism, a "job" scheduler observes the execution context of the node, verifying whether current conditions allow a job (and its tasks) to be run w.r.t. job requirements. In the second layer, a "task" scheduler considers the execution of each task, considering the availability of the resources. In the case of this work, the two-layer scheduling is associated with context information, allowing an opportunistic use of available resources and the minimization of data access costs. Indeed, while the "basic" scheduler only checks if there are free resources (CPUs, cores) with the required computing capabilities for a task, the context-aware scheduler may also include information such as the presence of data chunks on the local storage, which avoids data transfers over the network, or the current load of the node.

In the next section, we present a set of experiments we have performed using the two-layered scheduler, and analyze the impact of observing context execution conditions on the application performance.

5.2. Evaluating a context-aware scheduling

As mentioned in the previous section, the scheduler presented in [12] uses context information to decide whether it executes a task locally or not, following a simple *best-effort* philosophy. Every scheduler is independent, deciding on its own which task to run. Since the task list is shared among the community nodes, any node that decides to execute a task will prevent the other nodes about its decision; similarly, every time a task is completed, the node communicates the execution results (and the "complete" status) to its neighbor nodes using the communication module. Context information is observed using a lightweight context manager [47], which keeps track of several context elements, including CPU average load, CPU process usage, available memory, and storage.

These experiments were conducted using four nodes of type *n1-standard-1* (one virtual processor, 3.75 GB of memory) from the Google Cloud Platform. We use this configuration since it is one of the most basic ones. Such low-level configuration (one single processor, low memory capacity) represents the low-end devices we may find on edge environments. We decided to use a cloud platform to have nodes within a controlled and reproducible environment in which we could concentrate our analysis on the effect of context information on scheduling without risking unpredictable external influences. For the same reasons, we decided to focus the experiments presented here on

a single context element, the CPU average load, which hints the occupancy of the node. Even if the context manager keeps observing other available context information, such as available memory, it is the CPU information that guided our experiments. As a result, we chose a benchmark application that is CPU intensive. This benchmark application is a Monte Carlo simulation composed of 50 independent tasks with the same number of iterations, deployed over the network. Despite the same input parameters (iterations, operation to execute), the running duration for each task depends on the input data it receives and also the computing capabilities and context of the executing node.

Several parameters could be used to characterize a heterogeneous execution environment but some of them are quite subtle to control. Instead, to illustrate the impact of context-aware scheduling and keep the scenario the most reproducible as possible, we decided to focus on the execution performance of the nodes under stress. Hence, we overload part of the nodes and compare the scheduling behavior with and without context-awareness. To overload the machines, we use the *stress-ng* tool[3] and the parameters `stress-ng -c 0 -l 50` to occupy 50% of the available CPU utilization. The metric used in the experiment is the CPU system load average over the last minute, a popular metric from Linux systems, being present in tools such as *top*. This metric represents the number of processes being or waiting to be executed over the covered period. In a single (virtual) processor node, any value above 1 indicates a high concurrency for the CPU, which may delay the execution of the applications.

In the first experiment, we execute the set of 50 tasks in a network where two machines (nodes *m3* and *m4*) are overloaded, and no context-awareness is in use. The Figure 17 represents in a Gantt chart this execution. The lines represent the current load average (in solid red) and the 100% threshold (in dashed blue). We can observe that tasks in the overloaded nodes are slower than in the other nodes and that the average CPU usage raises to 166% on node *m3*. From these numbers, we can extrapolate that not only tasks' execution is interfering with other services in the node but also that the CPU temperature will rise, incurring extra energy consumption if this was a real edge node.

Figure 18 uses the same scenario, but now the context information is

[3] `https://kernel.ubuntu.com/~cking/stress-ng/`

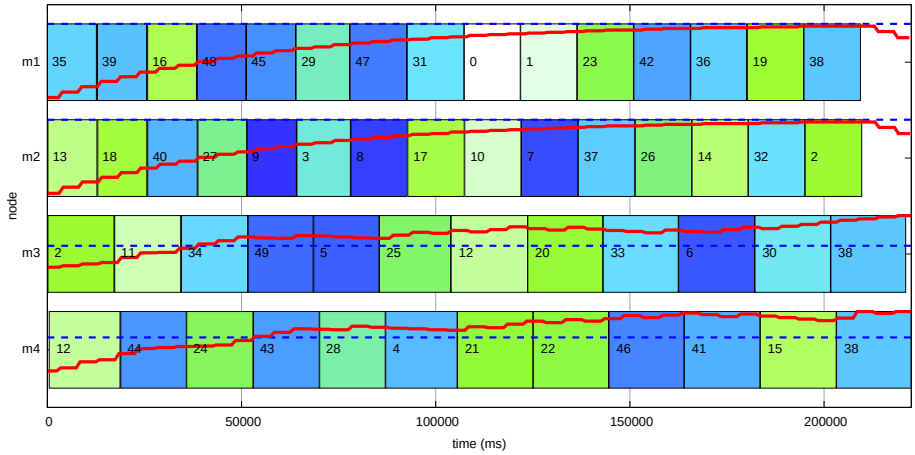

Figure 17: Distributed computing scheduling without context awareness

taken into account by the scheduler thanks to tasks with a required context `"CPU average load <= 1.0"`. When such task requirements are no longer satisfied (CPU overloaded), the scheduler refrains from starting a new task, which is perceptible through the white spaces in Figure 18. Although the application takes a bit longer to complete its execution (239.2s vs. 222.4s), the context-aware scheduling allows us to preserve the CPU in all machines, which means reduced CPU temperature and energy consumption, as well as fairer sharing of resources with other services in the same node.

This effect is even more perceptible when we increase the charge on a node. Figure 19 illustrates a third experiment in which the charge of nodes m3 and m4 was artificially improved up to 90% using *stress-ng*. We may observe that, in this case, these nodes very often abdicate to execute our tasks because of their overcharged execution context. Of course, this behavior has an impact on the total execution time, but all nodes have globally respected their overall charge, which is not the case in the first experiment (Figure 17). As the solid red line in Figure 17, representing the CPU average load, demonstrates, these nodes (*m3* and *m4*) spend more time above the CPU threshold (represented by the dashed black line), while in Figures 18 and 19 this threshold is better respected.

Finally, we proceed to the last experiment focusing on the network partition problem. In this scenario, a node is disconnected from its neighbors, simulating a disconnection or a network partition. This is a common problem in dynamic environments such as edge environments, in which nodes may lose

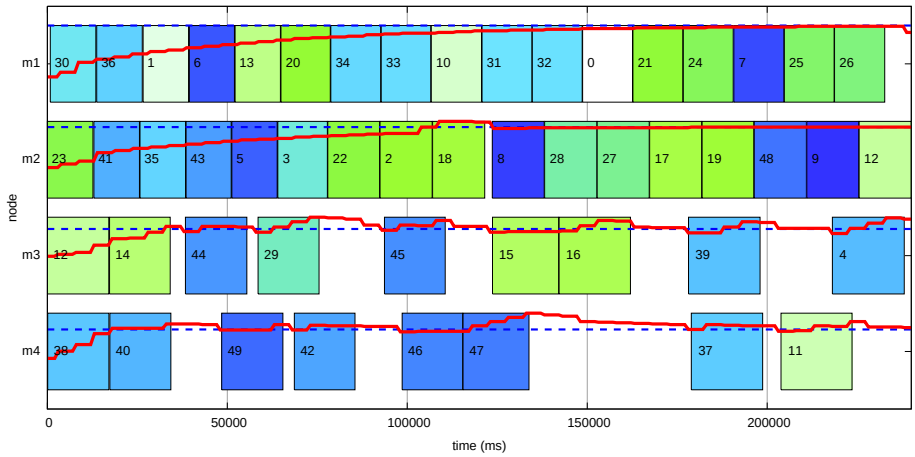

Figure 18: Distributed computing scheduling with context awareness

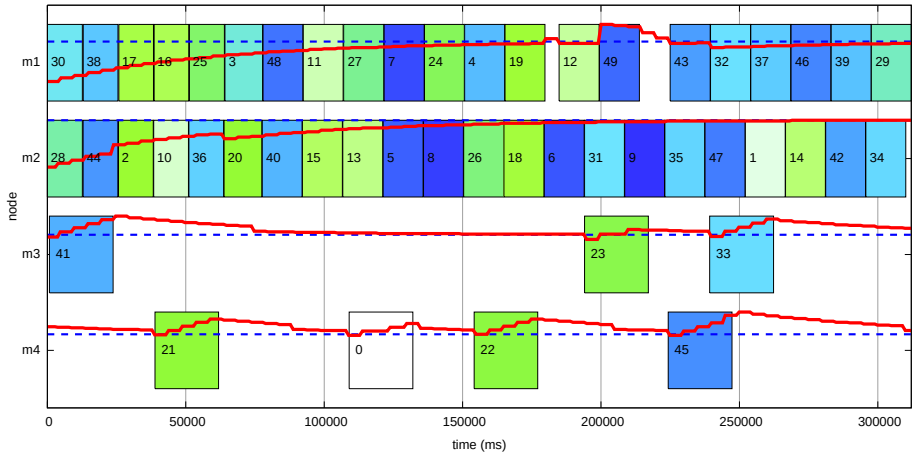

Figure 19: Context-aware scheduling applied to nodes with up to 90% of charge.

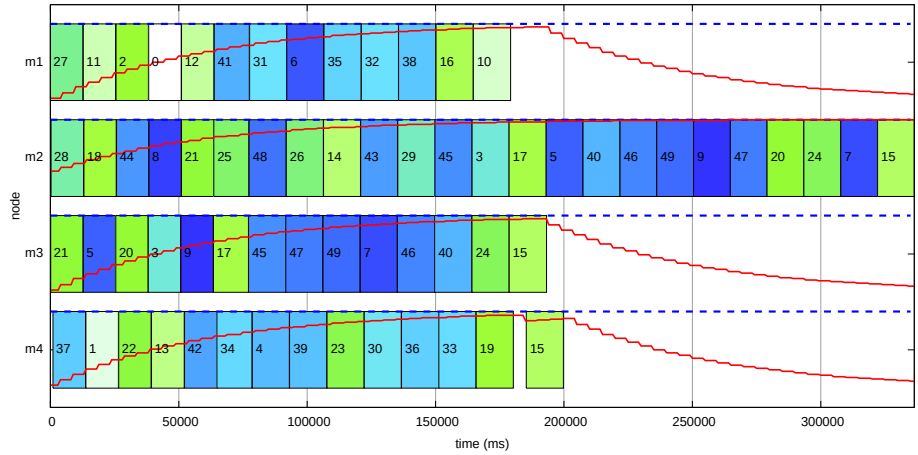

Figure 20: Partitioned environment in which node m2 is isolated from the other nodes.

contact with other nodes at any moment, for many different reasons. The decentralized scheduling algorithm we use has the advantage of ensuring that tasks will be completed despite network connection problems. However, since nodes can become unreachable, a phenomenon of "multiple executions of the same task" may happen. Figure 20 represents the execution of this scenario, in which node *m2* has been disconnected from the community and stop receiving information from the other nodes. Without information from the other nodes, the disconnected node does not know that tasks were finished elsewhere, so it tries to complete the remaining tasks. Although this has no impact on the consistency of the results, this represents a waste of energy that can be critical for low-end devices running on batteries.

Multiple executions can also happen when many nodes have available resources and a limited list of remaining tasks. In this example, both nodes *m3* and *m4* execute task 15 at the end of their runs. This happens because the distributed scheduler applies a "speculative execution" strategy for tasks that have been running for some time but not yet finished. Also present in Apache Hadoop, this strategy helps to complete tasks that otherwise would run too slowly or even hang the application completion due to a failing node. Just like in the case of network partition, the consistency of the results is not at risk, as any duplicate result will be ignored by the application.

Many other metrics can be used as context information, allowing fine-tuning of requirements/constraints in a distributed computing platform. Some metrics such as resource capacity and computing power can be easily ac-

quired, while other metrics such as link availability, latency, and bandwidth require active probing of the network (unless the developers chose to use static parameters for standard configurations). In the case of platforms with heterogeneous resources such as edge platforms, context awareness allows developers to set rules for better resource usage and to deploy tasks according to other metrics such as proximity to the end-users or Quality of Service (QoS) and resiliency requirements.

6. Conclusion

This work focused on the experimental evaluation of an edge computing platform to address two main concerns when deploying applications in heterogeneous environments: the impact of unbalanced communications, which affect data transmission, and the impact of heterogeneous (and sometimes dynamic) resources, which affect the applications' performance. Through the observation of the behavior from applications and devices, we identified several performance bottlenecks that may hinder an edge application: the difficulty to place the data where it needs to be, the need for awareness about data locality, and the node runtime context during the application deployment. Also, the asymmetrical performances of the network on low-end nodes may present interesting challenges for the optimization of edge services. We also propose some strategies to deal with data locality and context-awareness, highlighting the interest of controlling data locality as well as the characteristics of the resources in edge environments.

The experimental benchmarks have considered scenarios involving both real cloud and edge devices. This close relation between cloud and edge computing is also perceptible on numerous definitions from the literature. These results let us foresee possibilities for combining both cloud and edge environments, according to the capabilities and context of involved devices. We hope the bottlenecks identified here may help designers to better prepare their solutions. Also, the techniques we proposed may help improve the scalability of data-intensive applications deployed on the edge, as we try to embrace the decentralization and heterogeneity that characterizes edge computing.

While the next step in our work leads towards a comparison with other edge platforms, many other challenges that remain open. Among them, energy consumption must be addressed, especially if it diverts devices from their original purpose (as in the case of using mobile devices as computing

relays). Extending context-awareness to energy management is probably the next milestone for edge scheduling, together with task migration and the definition of metrics and heuristics to decide whether to deploy a service in the edge or the cloud.

References

[1] P. Hofmann, D. Woods, Cloud computing: The limits of public clouds for business applications, Internet Computing, IEEE 14 (6) (2010) 90–93. doi:10.1109/MIC.2010.136.

[2] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, G. P. Nolan, Computational solutions to large-scale data management and analysis, Nature Reviews Genetics 11 (9) (2010) 647–657. doi:10.1038/nrg2857.

[3] S. Chen, T. Zhang, W. Shi, Fog computing, IEEE Internet Computing 21 (2) (2017) 4–6. doi:10.1109/MIC.2017.39.

[4] F. van Lingen, M. Yannuzzi, A. Jain, R. Irons-Mclean, O. Lluch, D. Carrera, J. L. Perez, A. Gutierrez, D. Montero, J. Marti, R. Maso, a. J. P. Rodriguez, The unavoidable convergence of nfv, 5g, and fog: A model-driven approach to bridge cloud and edge, IEEE Communications Magazine 55 (8) (2017) 28–35. doi:10.1109/MCOM.2017.1600907.

[5] M. Satyanarayanan, The emergence of edge computing, Computer 50 (1) (2017) 30–39. doi:10.1109/MC.2017.9.

[6] D. Huang, H. Wu (Eds.), Mobile Cloud Computing, Morgan Kaufmann, 2018.

[7] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the 1st MCC Workshop on Mobile Cloud Computing, MCC '12, ACM, New York, NY, USA, 2012, pp. 13–16. doi:10.1145/2342509.2342513.

[8] R. Olaniyan, O. Fadahunsi, M. Maheswaran, M. F. Zhani, Opportunistic edge computing: Concepts, opportunities and research challenges, Future Generation Computer Systems 89 (2018) 633 – 645. doi:10.1016/j.future.2018.07.040.

[9] Z. Hao, E. Novak, S. Yi, Q. Li, Challenges and software architecture for fog computing, IEEE Internet Computing 21 (2) (2017) 44–53. doi:10.1109/MIC.2017.26.

[10] L. A. Steffenel, M. Kirsch-Pinheiro, When the cloud goes pervasive: approaches for IoT PaaS on a mobiquitous world, in: Springer (Ed.), EAI International Conference on Cloud, Networking for IoT systems (CN4IoT 2015), no. 169 in LNICST, Rome, Italy, 2015, pp. 347–356.

[11] L. A. Steffenel, Improving the performance of fog computing through the use of data locality, in: 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2018, pp. 217–224. doi:10.1109/CAHPC.2018.8645879.

[12] L. A. Steffenel, M. K. Pinheiro, Improving data locality in p2p-based fog computing platforms, Procedia Computer Science 141 (2018) 72 – 79, the 9th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2018) / The 8th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2018) / Affiliated Workshops. doi:10.1016/j.procs.2018.10.151.

[13] E. Lagerspetz, J. Hamberg, X. Li, H. Flores, P. Nurmi, N. Davies, S. Helal, Pervasive data science on the edge, IEEE Pervasive Computing 18 (3) (2019) 40–49. doi:10.1109/MPRV.2019.2925600.

[14] M. Viitanen, J. Vanne, T. D. Hmlinen, A. Kulmala, Low latency edge rendering scheme for interactive 360 degree virtual reality gaming, in: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1557–1560. doi:10.1109/ICDCS.2018.00168.

[15] M. Parashar, J.-M. Pierson, Pervasive grids: Challenges and opportunities, in: K. Li, C. Hsu, L. Yang, J. Dongarra, H. Zima (Eds.), Handbook of Research on Scalable Computing Technologies, IGI Global, 2010, pp. 14–30. doi:10.4018/978-1-60566-661-7.ch002.

[16] S. Dey, A. Mukherjee, H. S. Paul, A. Pal, Challenges of using edge devices in iot computation grids, in: Int. Conf. on Parallel and Distributed Systems, 2013, pp. 564–569. doi:10.1109/ICPADS.2013.101.

[17] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies, The case for vm-based cloudlets in mobile computing, IEEE Pervasive Computing 8 (4) (2009) 14–23.

[18] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, E. Riviere, Edge-centric computing: Vision and challenges, SIGCOMM Comput. Commun. Rev. 45 (5) (2015) 37–42. doi:10.1145/2831347.2831354.

[19] O. Vermesan, P. Friess, P. Guillemin, R. Giaffreda, H. Grindvoll, M. Eisenhauer, M. Serrano, K. Moessner, M. Spirito, L.-C. Blystad, E. Z. Tragos, Internet of things beyond the hype: Research, innovation and deployment, in: Internet of Things - From Research and Innovation to Market Deployment, River Publishers, 2014, pp. 15–118.

[20] J. K. Zao, T.-T. Gan, C.-K. You, C.-E. Chung, Y.-T. Wang, S. J. Rodríguez Méndez, T. Mullen, C. Yu, C. Kothe, C.-T. Hsiao, S.-L. Chu, C.-K. Shieh, T.-P. Jung, Pervasive brain monitoring and data sharing based on multi-tier distributed computing and linked data technology, Frontiers in Human Neuroscience 8 (2014) 370. doi:10.3389/fnhum.2014.00370.

[21] J. K. Zao, T.-T. Gan, C.-K. You, C.-E. Chung, Y.-T. Wang, S. J. Rodríguez Méndez, T. Mullen, C. Yu, C. Kothe, C.-T. Hsiao, S.-L. Chu, C.-K. Shieh, T.-P. Jung, Pervasive brain monitoring and data sharing based on multi-tier distributed computing and linked data technology, Frontiers in Human Neuroscience 8 (2014) 370. doi:10.3389/fnhum.2014.00370.

[22] A. M. Toihir, H. Bencherif, V. Sivakumar, L. El Amraoui, T. Portafaix, N. Mbatha, Comparison of total column ozone obtained by the IASI-MetOp satellite with ground-based and OMI satellite observations in the southern tropics and subtropics, Annales Geophysicae (Sep. 2015). doi:10.5194/angeo-33-1135-2015.

[23] L. Steffenel, O. Flauzac, A. S. Charao, P. P. Barcelos, B. Stein, S. Nesmachnow, M. K. Pinheiro, D. Diaz, Per-mare: Adaptive deployment of mapreduce over pervasive grids, in: 8th Int. Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'13), Compiegne, France, 2013.

[24] C. Pahl, S. Helmer, L. Miori, J. Sanin, B. Lee, A container-based edge cloud paas architecture based on raspberry pi clusters, in: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), 2016, pp. 117–124. doi:10.1109/W-FiCloud.2016.36.

[25] Z. Yunzhou, Z. Mo, L. Haoqi, Z. Gang, Innovative architecture of single chip edge device based on virtualization technology, Pervasive and Mobile Computing 52 (2019) 100 – 112. doi:10.1016/j.pmcj.2018.12.004.

[26] S. Yi, Z. Hao, Z. Qin, Q. Li, Fog computing: Platform and applications, in: IEEE (Ed.), Third IEEE Workshop on Hot Topics in Web Systems and Technologies, 2015, pp. 73–78.

[27] C. Devnet, What is IOx? [cited October 2018].
URL `https://developer.cisco.com/site/iox/docs/`

[28] D. F. Willis, A. Dasgupt, S. Banerjee, Paradrop: a multi-tenant platform for dynamically installed third party services on home gate- ways, in: ACM SIGCOMM workshop on Distributed cloud computing, 2014, pp. 43–44.

[29] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, E. Rivière, On using micro-clouds to deliver the fog, IEEE Internet Computing 21 (2) (2017) 8–15. doi:10.1109/MIC.2017.35.

[30] M. Villari, M. Fazio, S. Dustdar, O. Rana, R. Ranjan, Osmotic computing: A new paradigm for edge/cloud integration, IEEE Cloud Computing 3 (6) (2016) 76–83. doi:10.1109/MCC.2016.124.

[31] O. E. Initiative, Open edge initiative webpage, https://www.openedge-computing.org/ (Jul 2020).

[32] O. Consortium, Openfog reference architecture, https://www.openfog-consortium.org/ (Sep 2017).

[33] E. S. Dahmen-Lhuissier, Multi-access edge computing, http://www.etsi.org/technologies-clusters/technologies/multi-access-edge-computing.

[34] M. Breitbach, D. Schäfer, J. Edinger, C. Becker, Context-aware data and task placement in edge computing environments, in: 2019 IEEE International Conference on Pervasive Computing and Communications (PerCom, 2019, pp. 1–10. doi:10.1109/PERCOM.2019.8767386.

[35] L. A. Steffenel, M. Kirsch Pinheiro, Leveraging data intensive applications on a pervasive computing platform: The case of mapreduce, in: The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), Vol. 52 of Procedia Computer Science, Elsevier, 2015, pp. 1034 – 1039. doi:10.1016/j.procs.2015.05.102.

[36] M. A. López Peña, I. Muñoz Fernández, Sat-iot: An architectural model for a high-performance fog/edge/cloud iot platform, in: 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), 2019, pp. 633–638. doi:10.1109/WF-IoT.2019.8767282.

[37] J. Edinger, D. Schäfer, C. Krupitzer, V. Raychoudhury, C. Becker, Fault-avoidance strategies for context-aware schedulers in pervasive computing systems, in: 2017 IEEE International Conference on Pervasive Computing and Communications (PerCom), 2017, pp. 79–88. doi:10.1109/PERCOM.2017.7917853.

[38] F. Rodrigo Duro, J. Garcia Blas, D. Higuero, O. Perez, J. Carretero, Cosmic: A hierarchical cloudlet-based storage architecture for mobile clouds, Simulation Modelling Practice and Theory 50 (2015) 3 – 19, special Issue on Resource Management in Mobile Clouds. doi:10.1016/j.simpat.2014.07.007.

[39] D. Tracey, C. Sreenan, How to see through the fog? using peer to peer (p2p) for the internet of things, in: 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), 2019, pp. 47–52. doi:10.1109/WF-IoT.2019.8767275.

[40] R. S. Carbajo, C. M. Goldrick], Decentralised peer-to-peer data dissemination in wireless sensor networks, Pervasive and Mobile Computing 40 (2017) 242 – 266. doi:10.1016/j.pmcj.2017.07.006.

[41] B. Varghese, N. Wang, D. Bermbach, C.-H. Hong, E. de Lara, W. Shi, C. Stewart, A survey on edge benchmarking (2020). arXiv:2004.11725.

[42] A. Das, S. Patterson, M. Wittie, Edgebench: Benchmarking edge computing platforms, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 175–180. doi:10.1109/UCC-Companion.2018.00053.

[43] B. Confais, A. Lebre, B. Parrein, Performance Analysis of Object Store Systems in a Fog and Edge Computing Infrastructure, Transactions on Large-Scale Data- and Knowledge-Centered Systems (Aug. 2017). doi:10.1007/978-3-662-55696-2_2.

[44] M. Krajecki, An object oriented environment to manage the parallelism of the FIIT applications, in: V. Malyshkin (Ed.), Parallel Computing Technologies, 5th International Conference, PaCT-99, Vol. 1662 of Lecture Notes in Computer Science, Springer-Verlag, St. Petersburg, Russia, 1999, pp. 229–234.

[45] L. G. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990) 103–111.

[46] T. White, Hadoop: The definitive guide, 2nd Edition, Yahoo Press, O'Reilly, 2010.

[47] G. W. Cassales, A. Schwertner Charão, M. Kirsch-Pinheiro, C. Souveyet, L.-A. Steffenel, Improving the performance of apache hadoop on pervasive environments through context-aware scheduling, Journal of Ambient Intelligence and Humanized Computing (2016) 1–13doi:10.1007/s12652-016-0361-8.

[48] P. Bellavista, J. Berrocal, A. Corradi, S. K. Das, L. Foschini, A. Zanni, A survey on fog computing for the internet of things, Pervasive and Mobile Computing 52 (2019) 71 – 99. doi:10.1016/j.pmcj.2018.12.007.

[49] D. Wu, Y. Tian, K.-W. Ng, Aurelia: Building locality-preserving overlay network over heterogeneous p2p environments, in: Proc. of the International Conference on Parallel and Distributed Processing and Applications, ISPA'05, Springer, 2005, pp. 1–8. doi:10.1007/11576259_1.

[50] A. Rowstron, P. Druschel, Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility, in: ACM Symposium on Operating Systems Principles (SOSP'01), 2001.

[51] M. Ben Brahim, W. Drira, F. Filali, N. Hamdi, Spatial data extension for cassandra nosql database, Journal of Big Data 3 (1) (2016) 11. doi:10.1186/s40537-016-0045-4.

[52] A. Fox, C. Eichelberger, J. Hughes, S. Lyon, Spatio-temporal indexing in nonrelational distributed databases, in: IEEE International Conference in Big Data, 2013, pp. 291–299.

[53] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, ACM SIGCOMM Computer Communication Review 31 (4) (2001) 161–172.

[54] E. Tanin, A. Harwood, H. Samet, Using a distributed quadtree index in peer-to-peer networks, The VLDB Journal 16 (2) (2007) 165–178. doi:10.1007/s00778-005-0001-y.

[55] S.-Y. Hu, J.-F. Chen, T.-H. Chen, Von: a scalable peer-to-peer network for virtual environments, IEEE Network 20 (4) (2006) 22–31. doi:10.1109/MNET.2006.1668400.

[56] Y. Xu, S. Helal, Application caching for cloud-sensor systems, in: Proceedings of the 17th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM 14, Association for Computing Machinery, New York, NY, USA, 2014, p. 303306. doi:10.1145/2641798.2641814.

[57] J. Wang, Y. Wang, D. Zhang, S. Helal, Energy saving techniques in mobile crowd sensing: Current state and future opportunities, IEEE Communications Magazine 56 (5) (2018) 164–169. doi:10.1109/MCOM.2018.1700644.

[58] S. Shekhar, A. Gokhale, Dynamic resource management across cloud-edge resources for performance-sensitive applications, in: 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 707–710. doi:10.1109/CCGRID.2017.120.

[59] C.-H. Hong, B. Varghese, Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms, ACM Computing Surveys 52 (5) (Sep. 2019). doi:10.1145/3326066.

[60] M. Ghobaei-Arani, A. Souri, A. A. Rahmanian, Resource management approaches in fog computing: a comprehensive review, Journal of Grid Computing (Sep 2019). doi:10.1007/s10723-019-09491-1.

[61] M. Baldauf, S. Dustdar, F. Rosenberg, A survey on context-aware systems, International Journal of Ad Hoc and Ubiquitous Computing 2 (4) (2007) 263–277. doi:10.1504/IJAHUC.2007.014070.

[62] A. Dey, Understanding and using context, Personal and Ubiquitous Computing 5 (1) (2001) 4–7.

[63] M. K. Pinheiro, C. Souveyet, Supporting context on software applications: a survey on context engineering, Modélisation et utilisation du contexte 2 (1) (2018). doi:10.21494/ISTE.OP.2018.0275.

[64] K. A. Kumar, V. K. Konishetty, K. Voruganti, G. V. P. Rao, Cash: context aware scheduler for hadoop, in: International Conference on Advances in Computing, Communications and Informatics, ICACCI '12, New York, NY, USA, 2012, pp. 52–61.