LIGHTKONE

Lightweight computation for networks at the edge

Project no.	732505
Project acronym:	LightKone
Project title:	*Lightweight computation for networks at the edge*

D5.2: Report on Generic Edge Computing

Deliverable no.:	D5.2
Title:	Report on Generic Edge Computing
Due date of deliverable:	June 30, 2018
Actual submission date:	July 6, 2018
Lead contributor:	NOVA
Revision:	1.0
Dissemination level:	PP
Start date of project:	January 1, 2017
Duration:	36 months

This project has received funding from the H2020 Programme of the European Union

Revision Information:

Date	Ver	Change	Responsible
6/7/2018	1.0	First submitted version	NOVA

Revision information is available in the private repository https://github.com/LightKone/WP5.

Contributors:

Contributor	Institution
João Leitão	NOVA
Nuno Preguiça	NOVA
Henrique Domingos	NOVA
Sérgio Duarte	NOVA
Pedro Ákos Costa	NOVA
Pedro Fouto	NOVA
Guilherme Borges	NOVA
Bernardo Ferreira	NOVA
Maria Cecília Gomes	NOVA
Peter Van Roy	UCL
Roger Pueyo Centelles	UPC
Felix Freitag	UPC
Leandro Navarro	UPC
Roc Messeguer	UPC
Ali Shoker	INESC TEC
João Marco Silva	INESC TEC
Carlos Baquero	INESC TEC
Giorgos Kostopoulos	GLUK
Adam Lindberg	STRITZINGER

Contents

1 Executive summary

Allowing large-scale applications and systems to leverage on general purpose computations in the edge is a significant challenge that has to be tackled to pave the way for a novel generation of *edge-enabled applications*. This new generation has the potential to provide enriched interactivity, better user-experience, lower operational costs, and even increased scalability. This will effectively fulfill the ever growing requirements and needs of modern distributed systems such as the Internet of Things (IoT) and their areas of application like smart cities, smart grid, etc.

To this end, one needs to have a concrete vision of how this new generation of edge-enabled applications will leverage and exploit resources and networks in the edge. This includes understanding what are the effective computational resources and network components available in the edge, what are their key properties, and what forms of computations can be supported by these devices. Furthermore, since these resources can be as numerous as heterogeneous, one should also understand how they can be effectively managed.

This deliverable reports the continuous efforts of the LightKone consortium in tackling these challenges in its mission to enable a new generation of large-scale edge-enabled applications, with emphasis on light-edge scenarios. It builds upon the results reported in Deliverable 5.1 [11], and presents the progress achieved in the last five months of the project.

In the way to empowering a new generation of edge-enabled large-scale applications, key challenges are related to the execution, management, and monitoring of application components deployed in edge environments, and how these components interact among themselves and other components located in centralized infrastructures such as cloud data centers. To tackle these challenges in a systematized way, we present a view on the different types of resources (computational/network devices) that can be found in the edge, and look at them in terms of their different characteristics and their potential uses in designing new edge-enabled applications.

To continue addressing these challenges, we introduce a new distributed data aggregation protocol that allows many nodes in a system to collaboratively compute aggregation functions, such as average, maximum, minimum, over input values owned by different nodes. This solution is a first step in the direction of building scalable and effective distributed monitoring services that can operate in the edge. Furthermore, we present a possible architecture to allow the dynamic management of application components, enabling dynamic migration and replication of these components between cloud infrastructures and edge locations by exploiting microservice architectures.

Finally, devising, implementing, and testing distributed protocols that support the operation of edge infrastructures and application components in the edge is a continuous challenge. Particularly, when considering that some edge environments are highly restrictive regarding existing network infrastructures. We have continued our efforts in building tools for enabling the design and implementation of applications for wireless ad hoc networks, a particularly challenging edge scenarios. This effort is reflected in multiple lines of work. The first is the further development of programming and execution frameworks as well as hardware (and associated software support) that enables direct execution in the

edge; the second is porting Lasp to integrated devices, in particular the GRiSP board. Finally, we also present some ongoing work on exploring schemes to automatically find a balance between the quality of data acquired and the energy consumption in the context of Wireless Sensor Networks.

The report presents a set of main contributions and on-going work achieved in the context of WP5. It also discusses the relation of results with the overreaching goals of the project. The main results and contributions reported here can be summarized as follows:

Generic Edge Computing Vision: While there have been many proposals on how to leverage the edge to boost the performance and scalability of distributed application, we believe that current approaches are still somewhat limited regarding their exploitation of edge resources (i.e., computational and networking devices that exist in the edge). Furthermore, many of the existing approaches are biased towards applications in the IoT domain. To overcome this, we present our own vision of the potential that exists for exploiting edge computing in a more holistic way. This vision aims at improving distributed applications in other domains, such as user-centric applications (e.g., multi-player games, social networks/applications, etc.). To do so, we explore the characteristics of existing edge resources and their potential utilization in building novel edge-enabled applications. We further illustrate our vision by describing two potential application use cases and discuss how they could be materialized by taking advantage of the different edge resources.

Improvements to the GRiSP Platform: In the period reported in this deliverable we made efforts on improving the GRiSP platform that had been previously presented in Deliverable 5.1 [11]. In particular, there were improvements performed over the software stack that supports the GRiSP operation, as well as new and improved drivers that enable GRiSP that improve the stability of the platform and also support additional (and relevant) sensors. We finally report on dissemination activities conducted regarding the GRiSP platform.

Evolution of the Yggdrasil Framework: We also have conducted significant work on the Yggdrasil framework that was previously presented in Deliverable 5.1 [11]. The key improvements of the framework are: a new interface for designing protocols, that allows the programmer to only write the handler for the different types of events that are relevant for their protocol (or application); a new execution model that allows multiple protocols to be executed by a shared execution thread, which avoids the programmer to deal with complexities that arise from resource sharing; and a new set of protocols, particularly aggregation protocols implemented in the framework[1]. Furthermore, we present plans for the future evolution of Yggdrasil that will enable its practical deployment in the context of a use case (reported in Deliverables D2.1 and D2.2).

MiRAge, a new aggregation protocol for Wireless environments: To complement the results presented in Deliverable 5.1, we have designed, implemented (leveraging

[1] Data aggregation in the edge was the central thematic of the Deliverable 5.1.

Yggdrasil), and evaluated a new data aggregation protocol particularly suitable for wireless environments, where processes interact through a wireless ad hoc network. Contrary to previous state of the art, this protocol is suitable for commodity devices, supports continuous aggregation, and is fault-tolerant. This protocol paves the way for designing distributed monitoring schemes that can operate in wireless networks, which are essential for designing more complex edge-enabled applications in this context[2].

Additionally, we also report the following on-going activities, that have started in the period reported by this deliverable:

Self-adaptive Microservices in the Edge: Enabling new edge-enabled applications and systems that can take full advantage of a myriad of computational and network resources beyond data centers boundaries is not trivial. Particularly when considering that the components of such systems should be dynamically allocated, executing in the core infrastructure (i.e., data centers) or closer to end-clients. These decisions cannot be easily made at deployment time since they are affected by run-time conditions, such as the location of clients, load and activities of clients, among others. To overcome part of these challenges, we are exploring how to build microservice based solutions, with autonomic features, that enable components of applications/systems built as independent microservices, to be moved or replicated between cloud infrastructures and edge locations. Our proposal is based on combining three different aspects, managing the logic plane (i.e., microservices), managing the data plane, and an integrated monitoring scheme that captures relevant information allowing the two previous components to make adequate adaptations on the system configuration.

Lasp Applications for Wireless Edge with GRiSP: A second on-going line of work is related with enabling Lasp (and hence applications written using Lasp) to execute on integrated devices expanding its use to additional edge scenarios. This is particularly relevant for applications where integrated devices are employed for data acquisition, processing, and reporting. To this end, we have started to explore how to port Lasp to GRiSP boards. Both Lasp and GRiSP are results of WP5 that have been previously reported in Deliverable 5.1 [11]. We further plan to build applications on top of this model, as to showcase the practical benefits of combining these two results.

Adaptive Sensing in Wireless Sensor Networks with LiteSense: A final on-going line of work addresses the particularly challenging field of Wireless Sensor Networks. In particular, the challenge being addressed here is to find adequate schemes that can find (automatically) a balance between the quality of data being sensed in these networks and the amount of energy consumed by this process (with a direct implication on the maximum life time of sensors). Out proposal is to take into consideration, in a first step, the variation of the values being sensed, and based on that

[2]The work reported here was accepted for publication at the International Symposium on Reliable and Distributed Systems (SRDS). For the convenience of the reader we reproduce the (non-final) camera ready version of the publication in Appendix D.

adapt the frequency of data acquisition (and propagation throughout the network). A second step is to enrich this approach by also taking into consideration the energy available to the sensor. Hence making a more judicious selection of when to consume additional resources to improve the accuracy of sensed data.

1.1 Software Artifacts

Similarly to Deliverable 5.1, we report on software artifacts produced in the context of WP5. Given the shorter time span covered by this deliverable (compared to the previous one) we deliver revised versions of previous software artifacts produced in the context of the work package. In particular, the new version of the Yggdrasil framework. As we detail further ahead in the report, this will not be the last version of the framework, as we have concrete plans to further evolve it to support more general purpose edge environments. Additionally, we present the enriched software stack for supporting the GRiSP platform.

The software discussed in this report is currently available in the public Git repository of the work package[3].

[3] LightKone WP5 public repository:https://github.com/LightKone/wp5-public

2 Introduction

This deliverable reports on the main results and on-going activities of the LightKone consortium on addressing the multiple and varied challenges of light edge scenarios. The light edge intuitively captures edge computing applications where the communication among components is dominated by direct interactions of components executing in the edge of the system. This is a broad definition that can be materialized by many existing distributed architectures such as *Peer-to-Peer (P2P)* [32], *fog computing* [29], and *mist computing* [5]. Therefore, the mission of WP5 is to lead the research and development of solutions, protocols, and tools, to further exploit light edge scenarios. This includes not only improving existing architectural patterns, but also proposing new ones that can allow other application domains to benefit from edge computing.

This deliverable is focused on discussing solutions for supporting generic edge computations in (large scale) distributed systems. In this context, we define *generic edge computations* as computations that go beyond what has been traditionally achieved by existing solutions. Namely, we observe that many existing edge-based solutions (many of which are biased towards supporting IoT or Internet of Everything (IoE) applications) focus on supporting *data filtering* and *data aggregation* [3, 5][4]. Supporting general-purpose edge computations however, is a non-trivial challenge. To achieve it, one has to fully understand the different nature and capabilities of edge computational and networking resources, and how they can actively support different forms of computation at the edge.

To take fully advantage of different types of edge computing and network resources identified in the vision discussed above, one has to address other challenges. In particular, there is a clear need to devise mechanisms that allow the decomposition of (traditional) distributed applications that execute in cloud computing environments, enabling some of these computations to be pushed towards the edge of the system. Additionally, executing application components in the edge requires the creation of a execution environment and adequate packing of such components. General purpose computations delegated to edge resources may potentially need additional data sources, other than the ones that already exist locally in the edge. To tackle these challenges we have started to explore how to adapt microservice architectures as to enable (dynamic) migration or replication of applications components (i.e., individual or small sets of microservices) between cloud infrastructures and edge locations. We further explore how we can devise distributed data storage solutions that can be leveraged to support such a dynamic execution model.

Fully exploiting the potential of edge computing requires tools and runtime support for components of *edge-enabled* applications on edge devices. Such tools and runtime support should provide programmers with abstractions and mechanisms that simplify the development of correct software artifacts and systems in the edge. To this end, we continuous to focus on the edge levels farther from cloud data centers, where no physical network infrastructure exists, and hence, devices are restricted to communicate via wireless ad hoc networks. We argue that this is a particularly challenging point in the edge spectrum. This is tackled by further developments in the Yggdrasil framework. This is a framework (previously presented in Deliverable 5.1) whose goal is to simplify the development of distributed protocols and applications in this extreme edge scenario. In the

[4] We further discuss this forms of computation in Section 3.1

last few months we have refactored a portion of the framework to provide new abstractions for developers. Furthermore, we have enriched the execution model offered by the framework to provide additional control over the device resource (in particular Central Processing Unit (CPU)s) usage. We have also started the development (and concluded a first prototype) of a mechanism to simplify the evaluation and validation of distributed protocols in real scenarios running on real hardware, and further extended the number of distributed protocols implementations offered alongside the framework.

Another relevant aspect to take full advantage of the opportunities created by edge computing is specialized hardware, and support software stacks, that simplify the development, prototyping, and validation of new applications for the edge. We have continued our efforts in providing support for the GRiSP platform (presented in [11], particularly in its support software stack. Improvements performed in the platform recently improve its robustness while providing a new functionality.

Additionally, we have started a line of work to combine the results that were previously reported in [11]. In particular we have started to explore how to port the Lasp framework to GRiSP boards. This line of work will enable novel edge applications, that benefit from both the synchronization-free programming abstractions, offered by Lasp, and the flexibility of the GRiSP board to prototype and execute Erlang software directly on bare metal.

Finally, the previous deliverable produced in this work package was focused on aggregation computations in the light edge. In [11] we have implemented (on top of Yggdrasil) multiple distributed aggregation protocols. Based on what we have learned with this development (and also the implementation of a few additional aggregation protocols found in the literature) we have proposed and implemented a novel aggregation protocol, particularly tailored for wireless ad hoc environments. The design of this protocol exploits some design principles of *hybrid gossip* [17, 27] to build a robust and self-healing spanning-tree covering a (dynamic) set of wireless devices. This allowed us to build a *continuous aggregation protocol* that we named Multi-Root Aggregation, or simply MiRAge. The use of the Yggdrasil framework and its abstractions was crucial for the development of MiRAge.

2.1 Structure of the Deliverable

The remainder of the report is structured as follows:

Section 3 presents the main contributions reported as part of this deliverable in detail.

Section 4 discusses ongoing work being conducted by the LightKone consortium in the context of WP5.

Section 5 discusses the relationship of the results produced by WP5 with the work being conducted in the context of other work packages.

Section 6 reports on current exploitation plans to apply results achieved in the context of WP5 to the use cases of industrial partners.

Section 7 summarized the software artifact that is part of this deliverable.

Section 8 lists the papers and publications produced by in the context of WP5.

Section 9 concludes this report.

We note that the results reported here are focused on the achievements of the LightKone consortium since the delivery of Deliverable 5.1 [11]. Some of these results directly build upon previous results reported there.

3 Main Deliverable Contributions

In this Section we present the main results achieved in the context of WP5. The work captured by these results was developed since the previous deliverable produced by this work package [11].

The main results presented are:

Section 3.1 presents a vision for the future of edge computing. Here we analyze how different edge resources, either computational or network resources, that can be leveraged to build novel edge applications.

Section 3.2 presents the evolution of the GRiSP Platform that had previously been introduced in Deliverable 5.1, the additions and changes to the software stack and the device drivers. It also summarizes the various dissemination activities since the last report.

Section 3.3 details the evolution of the Yggdrasil framework as a tool for building applications and protocols for a concrete edge scenario, where devices communicate via wireless ad hoc networks.

Section 3.4 reports the developments since the previous deliverable regarding aggregation for light edge scenarios, in particular a novel distributed aggregation protocol, named Multi-Root Aggregation, or simply MiRAge.

3.1 Generic Edge Computing Vision

As discussed in the previous deliverable of this work package [11], edge computing can take many forms. However, the support infrastructure for edge computing is yet to be clearly defined. Fog computing [9, 29, 39], a materialization of edge computing, considers fog servers to be in the vicinity of IoT devices, where these fog servers are used to pre-process data. Given this model, fog computing is usually presented as having three tiers [3], the cloud, the fog servers, and the IoT devices. Mist computing, an evolution of the fog computing model that has been adopted by industry [5], proposes to push computations towards sensors in IoT applications, enabling sensors themselves to perform data filtering computations.

While these recent architectures exploit the potential of edge computing, they do so in a limited way, requiring specialized hardware and not taking a significant advantage of computational devices that already exist in the edge. Furthermore, and as noted for instance in [3] and [5], all of these proposed architectures are highly biased towards IoT applications. These architectures focus on filtering and mostly simple aggregation computations, which are far from our vision on general purpose computations in the edge.

Considering this, our vision is that edge computing also offers the opportunity to build novel *edge-enabled* applications, whose use of edge resources go beyond what has been achieved in the past, and in particular beyond proposals such as fog and mist computing [3, 5]. We believe that edge computing will enable the creation of significantly more complex distributed applications, both in terms of their capacity to handle client requests and data processing, and in terms of the scale regarding the number of components. This

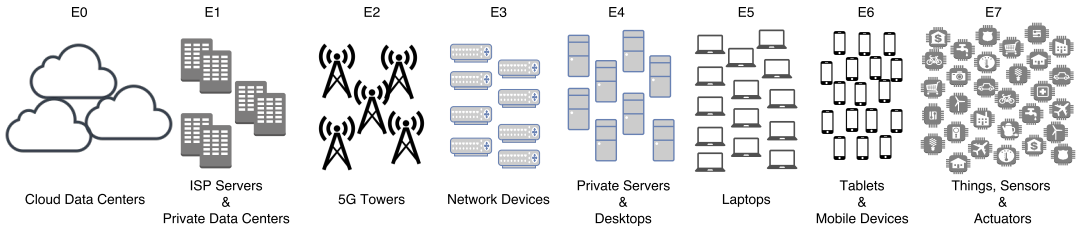


Figure 3.1: Edge Components Spectrum

will empower the design of user-centric applications that promote additional and enriched interactivity among users and between users and their (intelligent) environment(s).

(a) Overview of the Edge

To fully realize the potential of edge computing, we start by identifying the computational resources that lie beyond the cloud boundary, and try to systematically identify their limitations and potential benefits for edge-enabled applications. Figure 3.1 provides a visual representation of the different edge resources that we consider. We represent these edge resources as being organized in different levels, starting with level zero, that represents cloud data centers (one of the ends of the edge spectrum which is central to heavy edge scenarios). Edge-enabled applications are not however, required to make use of resources in all presented edge levels. Nevertheless, we expect data to move mostly between components that are adjacent in the spectrum. Levels can be skipped and different application data may follow different routes among edge components.

To better *characterize* the different levels in the edge resource spectrum, we consider three main dimensions: *i*) **capacity**, which refers to the processing power, storage capacity, and connectivity of the device; *ii*) **availability**, which refers to the probability of the resource to be reachable (due to being continuously active or the prevalence of hardware/network faults); and *iii*) **domain**, which captures if the device supports the operation of an edge-enabled application as a whole (*application domain*) or just the activities of a given user[5] within an edge-enabled application (*user domain*).

We further classify the *potential uses* of the different edge resources considering two main dimensions: *i*) **storage**, and *ii*) **computation**. The first one refers to the ability of an edge resource to store and serve application data. Devices that can provide storage can do so by either storing *full application state*, *partial application state*, or by providing *caching*: the former two enable state to be modified by that resource, and the later only enables reading (of potentially stale) data. The second one refers to the ability of performing data processing. Here, we consider three different classes of data processing, from the more general to the more restrictive: *generic computations*, *aggregation and summarization*, and *data filtering*.

We start by observing that, as we move farther from the cloud (i.e., to higher edge levels), the capacity and availability of each individual resource tends to decrease, while the number of devices increases. We now discuss each of these edge resources in more detail. We further note that resources could be presented with different granularity. How-

[5]We refer to user in broad terms, meaning an entity that uses an edge-enabled application, either an end-user or a company.

		E0 Cloud DCs	**E1** ISP Servers & Priv. DCs	**E2** 5G Towers	**E3** Network Devices
Characterization	**Capacity**	High	Large	Medium	Low
	Availability	High	High	High	High
	Domain	Application	Application	Application	Application
Potential uses	**Storage**	Full State	(Large) Partial	(Limited) Partial	None/Caching
	Computation	Generic	Generic	Generic	Filtering

		E4 Priv. Servers & Desktops	**E5** Laptops	**E6** Tablets & Mobiles	**E7** Things
Characterization	**Capacity**	Medium	Medium	Low	Varied
	Availability	Medium	Low	Low	Limited
	Domain	User	User	User	User
Potential uses	**Storage**	(User) Partial	Caching	(User) Caching	(Local) Caching
	Computation	Generic	Aggregation	Aggregation	Filtering

Table 3.1: Characteristics of Edge Devices Per Level

ever, here we focus on a presentation that allows to distinguish computational resources in terms of their properties and potential uses within the scope of future edge-enabled applications.

E0: Cloud Data Centers Cloud data centers offer pools of computational and storage resources that can be dynamically scaled to support the operation of edge-enabled applications. The existence of geo-distributed locations can be used as a first edge computing level, by enabling data and computations to be performed at the data center closest to the client. These resources have *high capacity and availability* and operate at the *application domain*. They offer the possibility to store *full application state* and perform *generic computations*.

E1: ISP Servers & Private Data Centers This edge resource represents private regional data centers and dedicated servers located at Internet Service Providers (ISPs) facilities or exchange points that can operate over data produced by users in a particular area. These servers operate at the *application domain*, presenting *large capacity* and *high availability*. They offer the possibility to store *(large) partial application state* and perform *generic computations*.

E2: 5G Towers The new advances in mobile networks will introduce processing and storage power in towers that serve as access points for mobile devices (and tablets) as well as improved connectivity. While we can expect these edge resources to have *medium capacity*, they should have *high availability* operating at the *application domain*. These computational resources can execute *generic computations* over stored *limited partial application state* enabling further interactions among clients (e.g., mobile devices) in close vicinity.

E3: Network Devices Network devices (such as routers, switches, and access points) that have processing power capabilities, offer *low capacity* and *high availability*. From

the storage perspective, they offer either *caching* capacity or none at all. Devices in close vicinity of the user will operate at the *user domain* while equipment closer to servers might operate at the *application domain*. We expect these devices will mostly enable in-network processing for edge-enabled applications in the form of *data filtering* activities over data produced by client devices being shipped towards the center of the system.

E4: Private Servers & Desktops This is the first layer (and more powerful in terms of capacity) of devices operating exclusively in the *user domain*. Private servers and desktop computers can easily operate as logical gateways to support the interaction and perform computations over data produced by levels E5-E7. While individual edge resources have *medium capacity* and *medium availability* they can easily perform more sophisticated computing tasks if the resources of multiple devices are combined together. These edge resources are expected to store *(user-specific) partial application state* while enabling *generic computations* to be performed. Private servers in this context are equivalent to *in-premises servers*, frequently referred as part of fog computing architectures [3].

E5: Laptops User laptops are similar to resources in the E4 level, albeit with *low availability*. Low availability in this context is mostly related with the fact that the up-time of laptops can be low due to the user moving from location to location. Because of this, we expect these devices to be used for performing *aggregation and summarization* computations and eventually provide *(user-specific) caching* of data for components running farther from the cloud. Laptops might act as application interaction portals, enabling users to use such devices to directly interact with edge-enabled applications.

E6: Tablets & Mobile Devices Tablets and Mobile devices are nowadays preferred interaction portals, enabling users to access and interact with applications. We expect this trend to become dominant for new edge-enabled applications since users expect continuous and ubiquitous access to applications. These devices have *low capacity* and *low availability*, the latter is mostly justified by the fact that the battery life of these devices will shorten significantly if the device is used to perform continuous computations. These devices, however, can be used as logical gateways for devices in the E7 level in the *user domain* context. They can provide *user-specific caching* storage and perform either *aggregation and summarization* or *data filtering* for data produced by E7 devices in the context of a particular user.

E7: Things, Sensors, & Actuators These are the most limited devices in our edge resource spectrum. These devices will act in edge-enabled applications mostly as data producers and consumers. They have *extremely limited capacity* and *varied availability* (in some cases low due to limited power and weak connectivity). They operate in the *user domain*, and can only provide extremely limited forms of *caching* for edge-enabled applications. Due to their limited processing power they are restricted to perform *data filtering* computations. Devices in the E7 layer with computational capacity are the basis for Mist computing architectures [5].

Table 3.1 summarizes the different characteristics and potential uses of edge resources at each of the considered levels. We expect application data to flow along the edge re-

source spectrum, although different data might be processed differently at each level (or skip some entirely).

(b) Envisioned Case Studies

We now briefly discuss two envisioned case studies of novel edge-enabled applications, and argue how edge resources in different levels of the edge spectrum can be leveraged to enable or improve these case studies.

Mobile Interactive Multiplayer Game Consider an augmented reality mobile game that allows players to use their mobile devices to interact with augmented reality objects and non-playing characters similar to the popular Pókemon Go game [6]. Such game could enable direct interactions among players, (e.g., to trade game objects or fight against each other) and allow players to interact in-game with (local) third party businesses that have agreements with the company operating the game (e.g., a coffee shop that offers in-game objects to people passing by their physical location).

Pókemon Go only recently started to support in-game trades (among users that are registered friends in the application) and does not support the remaining discussed interactions, with some evidence [4] pointing to one of the main reasons being the inability of cloud-based servers to support such interactions in a timely manner due to the large volume of traffic produced by the application. However, edge computing offers the possibility to enable such interactions, by leveraging on edge resources on some of the levels discussed above. Considering that the game is accessed primarily through mobile phones, one could resort to computational and storage capabilities of *5G Towers (E2)* to mediate direct interactions (e.g., fights) between players. One could also leverage on regional *ISP and Private Data centers (E1)* to manage high throughput of write operations (and inter-player transactions) to enable trading objects. Some trades could actually be achieved by having transaction executed directly between the *Tablets & Mobile Devices (E6)* of players and synchronizing operations towards the *Cloud (E0)* later. Special game features provided by third party businesses could be supported by *Private servers (E4)* being accessed through local networks (supported by *Network Devices (E3)*) located on their business premises.

Intelligent Health Care Services Consider an integrated and intelligent medical service that inter-connects patients, physicians (in hospitals and treatment centers), and emergency response services [7], that can leverage on wearable devices (e.g., smart watches or medical sensors), among other IoT devices (e.g., smart pills dispensers), to provide better health care including, handling medical emergencies, and tracking health information in the scope of a city, region, or country.

These systems are not a reality today due to, in our opinion, two main factors. The first one is the large amounts of data produced by a large number of health monitors. The second one is related with privacy issues regarding the medical data of individual patients. Edge computing and the clever usage of different edge resources located in

[6] https://www.pokemongo.com/

[7] A significative evolution of the Denmark Medical System briefly described in [36].

different levels (as discussed previously) can assist in realizing such applications. In particular, *Wearables and medical sensors (E7)* can cooperate among themselves and interact with users' *Mobile Devices (E6)* and *Laptops (E5)*, which can archive and perform simple analysis over gathered data. The analysis of data in these levels could trigger alerts, to notify the user to take a medicine, to report unexpected indicators, or to contact emergency medical services if needed. This data could be further encrypted and uploaded to *Private Servers (E4)* of hospitals, so that physicians could follow their patients' conditions. Additionally, health indicators' aggregates could be anonymously uploaded to *Private Data Centers (E1)* for further processing, enabling overall health monitoring at the level of cities, regions, or countries to identify pandemics or to co-relate frequent medical conditions with environmental aspects.

(c) Discussion

Having a clear understanding of the computational and network devices in the edge that can support edge applications is essential to the success of the WP5 goals of fully exploiting the light-edge scenarios. Furthermore, it is also relevant to understand that these devices have fundamental different characteristics which might limit the type of computations and support that they can provide to edge applications.

Here we present a first systematic effort to understand the different characteristics and potential benefits in edge resources that, mostly, are currently available. We characterize these different resources in levels, and discuss their properties in two high level dimensions: *characteristics* and *potential benefits*. We argue that tapping on different existing resources in the edge will enable applications in new domains to take advantage of edge computing. This is illustrated by two envisioned use cases, a mobile multiplayer game and a medical application. These applications go beyond common IoT applications, mainly in the sense that they these are user-facing applications and highly interactive, whereas IoT applications mostly provide monitoring and data aggregation (not necessarily in real time).

This view of edge resources can be further evolved in the future, particularly by taking into account the deployment of concrete case studies in the edge, which we expect to pursue in the near future. Furthermore, the presented model for the edge also establishes the need to build support to exploit all of these edge resources. Yggdrasil, that we present next, is focused on supporting the development of applications and protocols executing on edge resources, primarily located in levels E5 and E7 of the proposed edge spectrum. As we detail further ahead we expect to expand the coverage of Yggdrasil to support edge resources in other levels of our edge spectrum.

3.2 Improvements to the GRiSP Platform

The GRiSP platform has seen many improvements since the last deliverable of the work package [11].

(a) Improvements of the Software Stack

The software stack has been greatly improved, bringing it in line with more modern environments and enabling new classes of embedded applications to be developed using GRiSP. One feature that was previously missing was access to low-level cryptographic functions. This has been alleviated by supporting the native cryptographic library that comes with Erlang/OTP, `crypto`. This makes other useful high-level libraries that come with Erlang/OTP usable, such as Secure Socket Layer (SSL), Secure Shell (SSH) and advanced features of Inets.

In addition to supporting `crypto`, we have also added the new releases of Erlang/OTP as possible target platforms for GRiSP development. This includes Erlang/OTP version 20[15] and the newest version, 21[14]. They include many improvements to Erlang, including but not limited to performance improvements and features such as a new distribution API that can be extended to create new ways of interacting with devices.

We have also upgraded Real-Time Executive for Multiprocessor Systems (RTEMS), the underlying Operating System (OS) platform we use to enable Erlang running on bare metal directly on the CPU. The newest version is RTEMS 5.0 and the GRiSP platform now uses it by default. Among other things, improvements have been made to the memory card access which should speed up the loading of embedded edge applications developed on the GRiSP platform.

(b) New and Improved Drivers

Apart from the underlying software stack itself, we have also made many improvements to the runtime platform. This platform provides active support to applications running on GRiSP hardware and includes system managing code and special device drivers for peripheral components.

The 1-Wire driver has been improved with timeout fixes making it more reliable. A new device driver for the 1-Wire device DS18B20 has been added. The DS18B20 is a 1-Wire device that implements a digital thermometer which is very useful when implementing control systems that need temperature, such as agricultural systems or home automation systems.

Furthermore, we have improved the older Pmod MAXSONAR driver making it more compatible with the current runtime structure. We have also upgraded the driver for DS2480, a 1-Wire 8-channel port expander, to be more reliable in the presence of other 1-Wire devices.

(c) Dissemination Activities

Our showcase of GRiSP and its activities in the Lightkone project have made its rounds at various conferences and events. The year started with GRiSP being presented at three talks over two conferences at the same time, BOBkonf Berlin, Germany, and LambdaDays in Kraków, Poland. Nadezda Zryanina presented a well received talk at BOBkonf 2018, showing the GRiSP platform as a functional bare metal platform for IoT applications with a hands-on demo using an autonomous robot with sonar vision [40].

At LambdaDays 2018, the platform was presented in two separated talks. One talk by Peer Stritzinger and Kilian Holzinger that presented a prototype for functional reactive

programming and talked about the process to approach hard real-time using Erlang/OTP and the GRiSP platform [18]. Claudia Doppioslash and Adam Lindberg showed a home automation prototype for an IoT edge system using 1-Wire temperature sensors and a real-time dashboard served from the edge device itself [13].

Later in March there was the popular CodeBEAM SF in San Francisco, United States. Here, Sébastien Merle presented GRiSP positioning it as a way to get closer to edge networks with capable hardware and software backing for applications [31] when developing home automation and robotics projects. At the sister conference CodeBEAM STO in Stockholm, Sweden. In the end of May, Peer Stritzinger and Adam Lindberg presented our work to scale Erlang distribution to 1000 nodes or more, which is needed to operate on large IoT networks while still leveraging native Erlang networking primitives allowing applications to scale without too many internal changes [28].

Finally, we went back to Kraków in June to show GRiSP at the Erlang meetup, give a lecture about industrial uses of Erlang for students in the AGH University and to give a half-day tutorial to both students and professors.

(d) Discussion

The GRiSP platform is well aligned with the efforts of this work package to provide additional support for edge applications, taking advantage of computational resources that are farther away from cloud infrastructures (in the opposite extreme of the edge spectrum as discussed previously). This line of work however, focus on a different perspective, that of providing specialized hardware (and associated software stacks).

We perceive this as an important effort by the LightKone consortium, which we plan to continue to make in the future. In fact, this effort is on-going, since as we discuss in the next Section, we have started to conduct efforts in integrating existing tools produced by the consortium into this platform.

3.3 Yggdrasil framework

Yggdrasil was designed to address a particularly challenging edge scenario, that of commodity devices that have to interact and execute computations in an environment where there is no network infrastructure (typically E5 and E6 considering the edge spectrum presented above). This forces processes that run on these devices to resort to wireless ad hoc networks as a means of communication. However, and as we discussed in the previous deliverable for WP5 [11], there is a significant lack of tools and support to design, implement, and test wireless ad hoc distributed protocols and applications.

Meanwhile, we have made multiple improvements to the Yggdrasil framework, which include code refactoring, enriching the API provided by the framework, improve the execution model and interface for distributed protocols (and applications), implementing a larger set of distributed protocols that operate on Yggdrasil, and finally we have started the development (and completed a first prototype) of a control process that simplifies the task of conducting experiments using Yggdrasil. In the following we discuss each of these improvements, and then discuss future plans for the evolution of Yggdrasil.

(a) Refactoring of Yggdrasil

The original prototype of Yggdrasil had historic code that had been developed during the early months of the LightKone project, even before the framework got its name. This lead to some *pollution* of the code with data types being named with the letters "LK" (which stand for LightKone) as well as some functions exposed by the API of the framework to be labeled with the same letters. We have since refactored the code to brand it adequately under the name Yggdrasil. Therefore, data types and methods exposed by the framework API are now labeled with the "YGG" letters instead of "LK". While this was a minor modification we believe it important to promote the visibility of the framework at a latter point by providing a more clear self identity.

(b) Enriching the API and Functionality

Core Data Structures Manipulation. We noticed that Yggdrasil provided very few abstractions to manipulate its core data structures (the ones that encode each event type in Yggdrasil, such as Messages, Timers, etc). To address this, we enriched the Yggdrasil API with auxiliary methods that provide easier mechanisms to add and extract elements to the payload of core data structures. We further added functions to initialize and release these data structures. In particular, when a core data structure is initialized, all its fields are also transparently initialized. The identifier of the event represented by the data structure is set according to the value of an argument of the initialization function, and the payload fields are set to represent an empty payload (`NULL`). When an item is added to the payload, memory for that item is allocated accordingly. The functions that are used to release the resources used by the core data structures, transparently frees the memory that was previously allocated. Some data structures also have associated functions that reset their internal state to its initial state. These mechanisms allow to write protocols and applications with fewer lines of code, and minimize common mistakes associated with memory management in C.

Timer Protocol and Events. The timer protocol and the timer data structure were modified to support nanosecond precision instead of only second precision, as we found that in some cases this was not enough to support all protocol operations. This was achieved by modifying the timer data structure to add a field representing the nanoseconds. As nanoseconds can reach very high numbers (possibly not fitting in an unsigned long variable), additional functions to manipulate the time interval associated with a timer data structure were implemented to avoid overflows and ensure the correct operation of timer events. The Timer protocol that belongs to the core of Yggdrasil was also modified to accommodate these changes.

Initialization Interface and Protocol Management. In the previous version of Yggdrasil, upon the initialization of the runtime, the number of Yggdrasil support protocols, user defined protocols, and applications had to be explicitly specified by the developer. This was originally intended as an optimization, as it allowed the Yggdrasil core to use a static data structure to represent these elements. Unfortunately, we have since noted that

this limited Yggdrasil from having a dynamic set of protocols associated with an application. This makes it impossible to easily enable or disable protocols in reaction to external events or runtime conditions. In the current version, we have only partially addressed this limitation. Protocols still need to be added/registered in the runtime during the initialization of Yggdrasil. However, Yggdrasil's runtime no longer requires the programmer to specify the number of different protocols and applications. Additionally, protocols that follow the newly provided execution model (described below) can be pre-registered in the runtime. The current runtime version provides start and stop functions that can be used for these pre-registered protocols hence, enabling a simple form of dynamic control.

Data Structure Manipulation. While implementing additional protocols on Yggdrasil, we noticed that the neighbors list was a recurrent data structure used in many protocols. Most of the protocols that we have developed using Yggdrasil required to keep track of the current neighbors. This is especially true for the neighbor discovery protocols, fault detection protocol, aggregation protocols, among others. This implied that many protocols had to repeat large blocks of code (differing only on a few lines) to define and maintain the data structures responsible to encode the protocol state regarding neighbors. To avoid this overhead, we implemented a simple library that exports a data structure that can be used by any protocol (or application) to maintain information regarding neighbors, as well as utility functions to manipulate and search this list. In a bit more detail, this library provides a representation for a neighbor (i.e., node) that has a unique identifier (which is a long random bit string), the MAC address, and a generic attribute. The generic attribute is protocol specific (e.g., an aggregation protocol might need to store, associated with each neighbor, a data structure containing the last information received from that neighbor).

Protocol/Application Programming Interface. A relevant improvement that was made to Yggdrasil is related with the programming interface to implement protocols or applications. In the previous version, protocols were defined by writing a function that encoded the main control loop of the protocol/application. This function had to initialize the internal state of the protocol, and then continually wait for new events (Messages, Timer, Notification, Request/Replies) received through it's event queue, test the type of event, and handle it accordingly. While this interface provides fine grained control over the specification of the protocol, in the general case this leads to repetitive and error-prone code to be written whenever a new protocol was implemented.

To mitigate this, in the current version, developers have an additional programming interface available to implement protocols. Instead of writing a function as discussed above, developers can simply write a (much smaller) function to handle each type of event that is relevant to their protocol/application. Then, the developer simply has to write a *specialized* initialization function that has two responsibilities: *i*) initialize the protocol's state; and *ii*) return a predefined data structure that provides pointers to each of the event handlers of the protocol/application, or a `Null` pointer when that event is not processed. Additionally, this data structure should also provide the protocol's unique numeric identifier, a string containing the protocol's name, a function to free the protocol's state (i.e., a destruct function), and finally, the notifications that the protocol will

produce and consume (in the previous version this was done explicitly by the application developer in the application code, which forced the developer to be aware of operation details of the protocols employed by her).

(c) Execution Model of Protocols/Applications

In this version of Yggdrasil we have added a new protocol that is automatically initialized along with the Dispatcher protocol and the Timer protocol by the framework upon initialization. This is the *Executor* protocol. This protocol behaves as a meta-protocol, where other protocols are able to be registered and executed in the context of a shared execution thread managed by the Executor protocol. This enriches Yggdrasil by allowing additional flexibility, in particular, instead of each protocol having its own execution thread, now the developer can chose between that option or having multiple protocols executed in a single thread context.

This is exposed to the developer by the API functions used to register the protocol in the Yggdrasil runtime. This relies on the structure that encodes all information relevant for the operation of a protocol (that we described above). In the current version, the runtime checks this data structure and decides either to prepare a thread to run the protocol, if the definition contains the main loop function, or to register the event handlers in the executor protocol, if the event handlers are specified. If both are present, the runtime will issue a warning and default to use a dedicated execution thread for that protocol.

When the protocol is registered within the Executor protocol, the registered protocol becomes associated with the Executor's event queue. Every event that is destined to the registered protocol will then be delivered to the Executor. The Executor is responsible for executing the function of the appropriate protocol that handles the type of received event.

(d) New Protocols in Yggdrasil

Distributed Aggregation Protocols. Since the writing of Deliverable 5.1 [11], we have leveraged Yggdrasil to implement some aggregation protocols found in the literature. These include Push Sum [23]; LiMoSense [16]; Distributed Random Grouping (DRG) [10]; Flow Updating [21]; and the Generic Aggregation Protocol (GAP) [12]. Additionally, we also developed and implemented a novel aggregation protocol, named MiRAge, that we describe in detail further ahead in this document. All of these protocols are now provided with the Yggdrasil framework, both as examples and protocols that can be used to build applications. The implemented aggregation protocols include Push Sum [23], LiMoSense [16], DRG [10], Flow-Updating [21], and GAP [12].

For completeness we provide a brief description of these protocols in Appendix A. The implementation of all protocols benefited (and exercised) from the abstractions and mechanisms provided by the framework.

Multi-Hop Routing Protocol. We have also implemented a variant of the popular ad hoc routing protocol *Better Approach To Mobile Ad hoc Networking* , or simply B.A.T.M.A.N. [22]. This protocol builds a routing table that reflects the most stable

link to forward a message to each other node in the network (i.e., each possible destination). To identify these links, each node periodically broadcasts an announce message containing a sequence number, that is incremented at each broadcast by the originator[8]. Each node that receives such an announce stores the originator node identifier (e.g., an IP address, or some other unique identifier), the message sequence number, and the node from which it received the announce. The received sequence numbers are maintained in a sliding window, and the most stable link to a destination (i.e., the next hop for every possible destination), is the neighbor from whom the local node received more broadcast messages with sequence numbers within the sliding window. This window moves independently for each distinct destination whenever a higher sequence number than the sliding window's limit is received.

(e) Yggdrasil Control Process

Besides serving as a framework to implement and execute distributed protocols and applications in wireless ad hoc networks, Yggdrasil also offers the opportunity to serve as a benchmark and validation tool for this class of protocols. This means that Yggdrasil can be used to run (controlled) experiments that exercise different protocols, extracting metrics of their performance which we believe is an invaluable tool for both researchers and practitioners developing their solutions for this particular edge environment. We ourselves were faced with this problem when conducting experimental assessment of the MiRAge aggregation protocol (described further ahead in Section 3.4).

To simplify this task, we have started to develop, and implemented an initial prototype, of the *Yggdrasil Control Process*, that allows researchers to scatter devices running this process (in particular we have added this process to the `init.d` of our fleet of 24 Raspberry Pi 3 - Model B [2]) and remotely launch and stop other Yggdrasil protocols/applications in a controlled fashion. This does not require any infrastructure (i.e., it does not require devices to be connected to a wired network) operating on top of the ad hoc network. Furthermore, this Control Process also provides an interface, that allows to block communication between specific pairs of devices (simulating link failures) or simulate nodes crashes. This is useful to evaluate the behavior of protocols or application in different faulty scenarios.

The Yggdrasil Control Process is currently composed of a set of three protocols that we call Control protocols. The three protocols are a specialized *discovery protocol*, the *external input protocol* that allows commands to be issued by a client application (through TCP sockets), and the *core control protocol*. In addition to these protocols we have developed a set of simple client applications that issue commands to the external input protocol. The list of commands that we currently support are reported in Appendix B.

Discovery Protocol: The discovery protocol designed to support the Yggdrasil Control Process is very similar to the discovery protocol that we reported in the first version of Yggdrasil on Deliverable 5.1 [11]. The key difference is that, since we use TCP connections in the design of the *core control protocol*, we have created a discovery protocol that also propagates the IP address of the wireless interface (which is

[8] The node that created the announce.

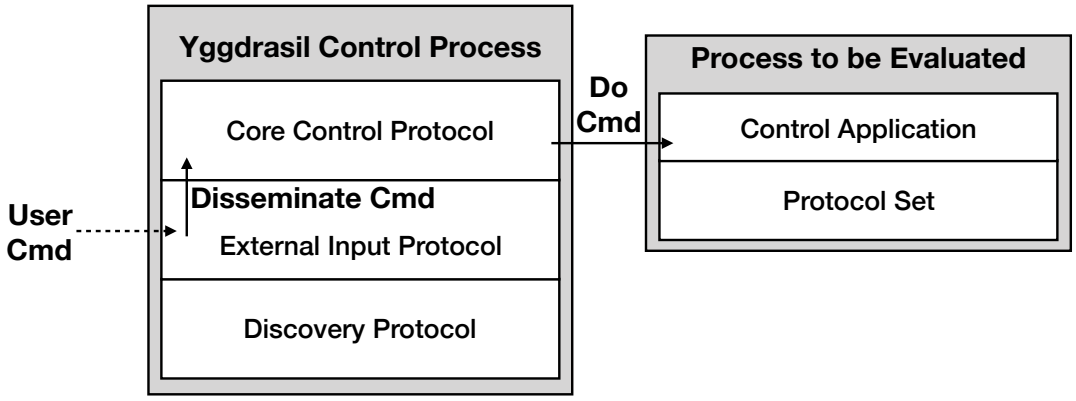


Figure 3.2: Execution mode where independent processes are spawned

generated by DHCP through a local process) on the announcement messages periodically issued by the protocol. Moreover, this protocol was also enriched with support for special disable and enable operations, that respectively deactivate and activate the transmission of announcements, which is relevant to ensure that during experiments we minimize interference due to the activity of the Control Process.

External Input Protocol: The external input protocol fundamentally waits for incoming TCP connections on any network interface and processes user operations issued through these connections (through a client application). These operations, as stated before, can be requests to start a particular protocol or application, terminate an application, simulate a link failure, recover from a link failure, etc. Some of these commands can be tagged with a set of nodes identifiers, in which case only those nodes execute the requested action. Otherwise the command is executed by all processes. Independently of the targets of the command, whenever this protocol receives a request from the user, it issues it to the core control protocol for dissemination and processing (for some commands it also waits for a reply from the core control protocol that is redirected to the client).

Core Control Protocol: The core control protocol is the main protocol of the Yggdrasil Control Process. This protocol has two main goals. The first is to disseminate commands to all other Yggdrasil Control Processes in the experimental benchmark (which are discovered by the discovery protocol, although we should note that this solution allows for multi-hop network configurations). This is is achieved by a broadcast protocol, that operates on top of TCP connections, whose design is inspired by the PlumTree [27] protocol. This broadcast protocol currently also offers a mechanism to gather responses from processes that execute disseminated commands to produce a reply for the client. This mechanism is however not fully stable in the current prototype, and will be improved in the following months. The second goal of the core control protocol is the (local) execution of commands issued by users.

A key goal of the Yggdrasil Control Process is to run protocols during experiments in real settings and with real hardware. Currently this can be achieved in two possible modes.

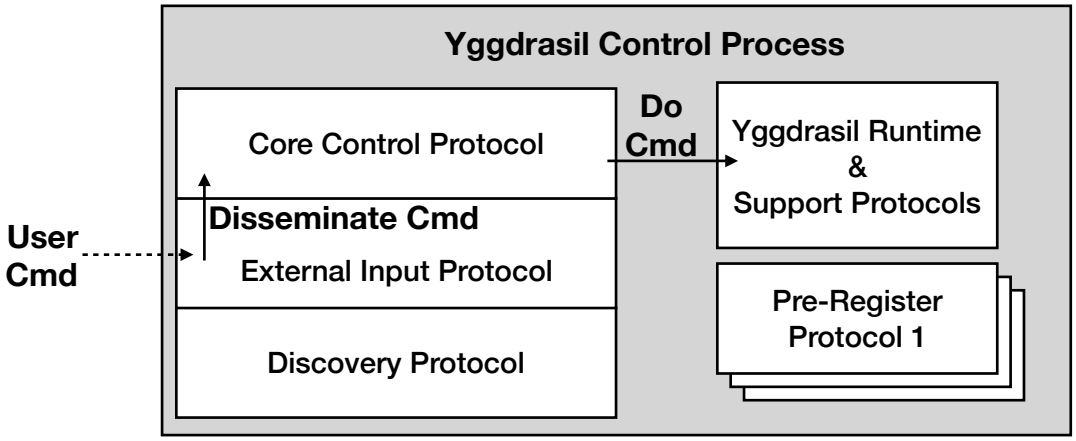


Figure 3.3: Execution mode where protocols are executed in the context of the Yggdrasil Executor (single process)

The first mode (illustrated in Figure 3.2) runs independent Yggdrasil process (that use the protocol or protocols being evaluated). To enable the core control protocol to interact with application being tested, the additional process should create an instance of an application that we named *Control Application*[9]. This application maintains an active pipe with the core control protocol for receiving specific control commands (such as blocking all communication to another device in the network).

The second mode (illustrated in Figure 3.3) executes the protocol(s) being tested in the context of the control process itself, by means of the Executor (meta) protocol discussed earlier. This allows the core control protocol to directly interact with the protocol being tested, while also allowing to directly emulate link failures or simulate process crashes. This however, requires protocols to be written under the new programming interface presented previously in this document.

(f) Discussion and Future Development

The new version of Yggdrasil presents multiple improvements (and some bug fixes not reported here) that strive to make the framework easier to use, particularly by reducing the among of code that has to be written to implement a protocol, while also providing additional abstractions and flexibility.

We currently have plans for the future development of the Yggdrasil framework, as we believe this is an appropriate tool for building and testing new distributed protocols and applications for edge computing scenarios.

More precisely we consider the following action points:

- We will continue development of the Yggdrasil Control Process as to further stabilize the current prototype and add other features to simplify the experimental validation and evaluation of protocols and applications developed in Yggdrasil.
- We will generalize the framework to also support other network modes. In particular, we plan to also support protocols and applications running on TCP/IP or

[9] We remind the reader that a Yggdrasil process can run multiple applications simultaneously.

UDP/IP stacks on wired networks. This is a key goal to allow Yggdrasil to be employed on scenarios such as the Server Monitoring use-case of UPC, which is also in our short term plans (this use case was originally presented in D2.1 and formalized in D2.2). This should be a fairly simple task, since Yggdrasil was built to allow this flexibility. In particular we plan to achieve this by writing a new low level library (for IP networks) and a new dispatcher protocol, that should enable existing protocols to operate seamlessly in these environments.

- We also plan on building other families of protocols in Yggdrasil. We are particularly interested in building a family of overlay network protocols and associated dissemination protocols based on gossip that operate on top of these networks.

- We will remove existing limitations of Yggdrasil, for instance limitations related with the maximum size of payloads in messages exchanged among different Yggdrasil processes through the network.

- We also plan on building a library offering general purpose data structures, such as lists, maps, etc. This is motivated by the fact that during the development of protocols for the framework we have noted that many lines of code are dedicated to specify and manage such data structures.

- We are currently starting to incorporate mechanisms to allow the encoding of messages exchanged by Yggdrasil processes in a common format to simplify its integration with other tools and frameworks designed in the project or tailored for other edge scenarios. In particular we are incorporating in the framework a set of tools to simplify the encoding (and decoding) of message payloads in JSON format.

3.4 Wireless Aggregation with MiRAge

The Multi Root Aggregation protocol, or simply MiRAge, provides efficient continuous aggregation, being particularly designed to take advantage of one-hop broadcast primitives that are available in ad hoc wireless networks. MiRAge implicitly paves the way for new edge-enabled applications where (some) edge components execute in devices that interact through a wireless channel, without access to an infrastructured network. This complements the work previously presented in [11] regarding the study on aggregation protocols for wireless ad hoc networks, by proposing a novel protocol that is more adequate for relevant applications (such as monitoring) in the context of edge computing.

In the following we discuss the motivation and context for MiRAge (Section (a)). For self-containment we introduce the problem of Aggregation and in particular discuss particular aspects of Continuous Aggregation (Section (b)). MiRAge design is then introduced (Section (c)) followed by a brief discussion on future applications and uses (Section (d)).

(a) Motivation and Context

Edge computing implies performing computations outside of the data center boundary, on devices located closer to clients of a system [34]. Therefore, edge computing can take

many different forms depending on four main factors: *i*) the devices being leveraged to perform computations; *ii*) the communication medium used by those devices; *iii*) the interaction pattern of those devices with the remaining components of a system; and *iv*) the nature of the computations being performed at the edge of the system.

In this line of work, and given the vast nature of the edge, we focus in a concrete edge scenario, that of multiple commodity devices being used to perform distributed computations over a given geographical area, where a network infrastructure is not available. This can be the case for some applications in the domain of smart cities and smart spaces [30] as well as IoT [8]. A concrete example of this is monitoring traffic within a city through a set of devices and enabling localized decisions regarding traffic signals to alleviate areas of high traffic density in a timely fashion [35], without the need of time-consuming centralized control.

The realization of this scenario would require a large number of devices with wireless capability. The absence of network infrastructure, and the non-negligible costs associated with its installation (i.e., adding access points), motivates the need to have these devices create an infrastructure-less network, or ad hoc network [38]. In such a network, devices would interconnect forming a multi-hop network. This network can have routing capabilities (creating a mesh network [7]), but this would hinder the possibility to leverage the network's capacity to perform in-network processing, as messages routing through the network would be controlled by a routing protocol, instead of some application level protocol capable of modifying/operating over the contents of messages. Furthermore, routing protocols have a non-negligible overhead, which further reinforces the benefits of using a simple ad hoc network.

To allow these systems to gather relevant information regarding their operation, deployment and execution environment, or even application-level data, efficient and reliable distributed monitoring solutions are required. Key to the design of monitoring solutions is the capacity to aggregate information produced or managed independently by large numbers of devices using a distributed aggregation protocol [20].

In the context of monitoring operational aspects of distributed systems, the individual values owned by each node are not static. In fact, in many use cases the values being aggregated change over time. Hence, we need to consider a particular form of aggregation named *continuous aggregation* [6] where the value being computed by the distributed aggregation protocol is continuously updated to reflect modifications in the individual input values or changes in the system affiliation (i.e., no longer taking into consideration the value of a node that leaves the system or fails).

MiRAge answers these needs, by providing an aggregation protocol that can compute any of the frequent aggregation functions over input data that is generated by each individual node in the system, and enabling efficient and robust operation in ad hoc networks. Furthermore, MiRAge was particularly designed to support continuous aggregation, enabling the design and implementation of efficient and reliable monitoring infrastructures for these edge environments.

(b) Aggregation Overview

Aggregation is an essential building block in many distributed systems [37]. A distributed aggregation protocol coordinates the execution of an aggregation task among devices of a

system. In short, a distributed aggregation protocol should compute, in a distributed fashion, an aggregation function over a set of input values, where each input value is owned by a node in the system. Typical aggregation functions include *count*, *sum*, *average*, *minimum*, and *maximum*.

Not all of these aggregation functions are equivalent regarding their distributed computation. Where count, sum, and average are sensitive to input value duplication, maximum and minimum are not. Distributed aggregation protocols must take measures to deal with these aspects, ensuring that the correct aggregate is computed.

Distributed aggregation protocols for edge systems, and in particular for settings where edge nodes interact through a multi-hop wireless ad hoc network, should be designed to address some key challenging aspects that are prevalent on this environment:

Continuous aggregation: As discussed before, the input values used for computing the aggregate function may change over time. Consider the example of computing the average CPU utilization among a collection of edge nodes, the individual usage of CPU by each node will vary accordingly to the tasks being executed by that device. To cope with this aspect, we argue in favor of using protocols that can perform *continuous aggregation*. Continuous aggregation was initially coined in [6] as being a distributed aggregation problem where, periodically, every node receives a new input value for the aggregation discarding the previous one. However, this notion implies that all nodes periodically restart the protocol [24]. In practice, not all input values change at the same time, and the change of a single value should not require an effort as significant as restarting the whole aggregation process. Instead, the protocol should naturally incorporate input value variations continually and with minimal overhead.

Fully decentralized: Distributed aggregation in the context of edge computing is paramount to enable the self-management of edge computing platforms. To promote timely management decisions and overall system availability, one should favor local reconfiguration decisions with minimal coordination among nodes. This has two relevant implications in the properties of aggregation protocols. First, every node in the system should have local access to the result being produced by the aggregation protocol. Second, the algorithm should not depend on the activity of specialized nodes in the system, and operate in a fully decentralized fashion.

Fault tolerance: Distributed algorithms should be tolerant to node failures, which are unavoidable in any realistic distributed setting. Since we are considering a particular edge setting where nodes communicate through a multi-hop ad hoc network, external interference is inevitable, which can be a result from having other devices in close vicinity using the wireless medium. This implies that a successful aggregation distributed protocol for this setting should be highly robust to message loss.

Minimal overhead: Considering the particular case of supporting distributed monitoring schemes, the execution of the aggregation protocol should have minimal overhead as to minimize its impact on any other services or applications being executed on edge devices. In particular, communication should have a low cost regarding the number of messages exchanged among nodes.

(c) The Design of MiRAge

In this section we discuss the design of MiRAge. We start by noting that our protocol is inspired in the design of GAP [12], that relies on a spanning tree with a fixed root node to coordinate the distributed computation of an aggregation function (the design of this protocol is discussed in Appendix A). Our solution, however, generalizes this design to remove the dependence on a single root. To this end, our protocol leverages on a self-healing spanning tree to *support* efficient continuous aggregation. In MiRAge all nodes compete to build a tree rooted on themselves. This competition is controlled by the identifier of each node (a large random bit string) and a monotonic sequence number (i.e., a timestamp) controlled by the corresponding root node. Additionally, our protocol was designed to ensure that all nodes in the system are able to continually compute and update the result of the aggregation function.

System Model We assume a distributed system where nodes communicate via message exchange. Furthermore, we assume that devices are equipped with a WiFi radio capable of operating in ad hoc mode. Each node is pre-configured to join a single ad hoc network. We do not assume any routing algorithm or infrastructure access. Devices can transmit messages using one-hop-broadcast, where the message can be received (with some probability) by all or a subset of the devices in the transmission range of the sender.

No node is aware of the total number of nodes in the system. However, we assume that each node has a unique identifier (this can be achieved by having each node generate a large random bit string at bootstrap). We do not make any assumption regarding clock synchronization, although, we assume that each node perceives the passing of time at a similar (albeit, not necessarily equal) rate.

Finally, we assume that each device runs a discovery protocol, where periodically the node transmits (in one-hop broadcast) an announcement containing its own identifier. The period of this transmission is controlled via a parameter ΔD. This protocol is also used by each node as an unreliable failure detector where, if the announcement of a known node is not received for a consecutive number of transmission periods large enough, the node becomes suspected of having failed, generating a notification to the aggregation protocol. The number of transmissions that a node can miss before suspecting another one is a parameter denoted K_{fd}. This is an assumption made by many other aggregation protocols including GAP [12], LiMoSense [16], Flow-Updating [21], among others.

Aggregation Mechanism Algorithm 1 describes the local state maintained by each node executing MiRAge and the components of the protocol related with the aggregation computation at each node.

The intuition of MiRAge is quite simple. Nodes organize themselves in a fault-tolerant *support* spanning tree that is used to control the aggregation mechanism. Nodes periodically propagate to their neighbors the local estimate of the aggregation result, that is obtained by applying the aggregation function over their own input value and the (latest) estimates received from neighbors with whom they share a tree link.

In MiRAge, each node owns a unique node identifier (Alg. 1 line 2) and its own input value for the aggregation (Alg. 1 line 3). Additionally, each node maintains its current

estimate of the aggregate value (Alg. 1 line 4) and a set of known neighbors (containing for each its node identifier; its latest received aggregated value; its current status in the support spanning tree, being `Active` if the neighbor is considered to be part of the tree through the local node, `Passive` otherwise; the identifier of the tree that the neighbor is connected to; and its level in that tree (Alg. 1 line 5).

Each node also owns a set of local variables that capture its current position and configuration in the support spanning tree. This includes: the identifier of the tree to which the node is currently connected (tree identifiers are the identifier of their root node), its level (the root of the tree has level zero), the highest timestamp observed, and the identifier of the parent node (Alg. 1 lines $6-9$). In addition, each node stores a map that associates tree identifiers to the last locally observed timestamp for that tree (Alg. 1 line 10).

When a node is initialized (Alg. 1 line 11) it has no knowledge regarding existing neighbors. Because of this, it assumes that the result of the aggregation function is its own value, and initializes the state related with the support spanning tree to reflect a tree rooted on itself (the tree identifier being its own identifier). Additionally, the node setups a periodic function named `Beacon` that is executed every ΔT which corresponds to the main aggregation logic of our algorithm. Typical values for ΔT are one or two seconds.

When the Beacon procedure is executed, a node will start by updating its local estimate. This is achieved through the execution of the `updateAggregation` procedure, which applies the aggregation function (operator $\oplus$ in Alg. 1 line 32) to the input value of the node with the received estimates of neighbors whose link with the local node has been marked as belonging to the node's current tree (i.e., status $=$ `Active`).

After updating its local estimate, the node will prepare a message to be disseminated through one hop broadcast. This message contains, for each known neighbor, a tuple including the neighbor identifier and the locally computed aggregated value without the effects of the last contribution received from that neighbor (operator $\ominus$ in Alg. 1 line 26). This tuple is computed independently of the status of that neighbor. The message is then tagged with the local node identifier, and the information on the support tree that the node is currently attached to, including the tree identifier, the level of the node, the highest tree timestamp observed, and the identifier of the node's parent (Alg. 1 line 27).

Upon receiving one of these messages (Alg. 1 line 33), a node checks if there is a tuple in the message tagged with its own identifier. If so, then it uses the `updateNeighborEntry` procedure to either create or update the entry for that neighbor in its neighbor set. The information that is updated is the latest aggregate value received, the identifier of the tree to which the neighbor is currently attached, and its current tree level (the status of the node remains unchanged and is set to `Passive` if this was the first message received from that node).

Tree Management Mechanism We now discuss how MiRAge manages the support spanning tree used by the aggregation strategy discussed previously. As noted before, the tree implicitly defines the data path used for performing and propagating the aggregation information. In MiRAge there is no specialized sink node nor a pre-configured root node. Instead, all nodes strive to become the root of a tree covering all nodes in the system. However, at all times, each node only belongs to a single tree. Interestingly,

Algorithm 1 MiRAge: Aggregate Function Computation

1: **Local State:**
2: N_{id} //Node identifier
3: Value //current input value
4: Aggregation //Current result of aggregation
5: Neighs //Set: (N_{id}, value, status, T_{id}, T_{lvl})
6: T_{id} //Unique identifier of the tree to which the node is currently attached (N_{id} of tree root)
7: T_{lvl} //Level of the node in its current tree
8: T_{ts} //Higher timestamp of current tree
9: P_{id} //Identifier of current tree parent
10: Trees //Map: $Tress[T_{id}] \rightarrow$ Timestamp

11: **Upon Init ( ) do:**
12: Value $\longleftarrow$ *initValue*() //initial input value
13: $T_{id} \longleftarrow N_{id}$
14: $T_{lvl} \longleftarrow 0$
15: $T_{ts} \longleftarrow$ *now*(); //now() = current time
16: $P_{id} \longleftarrow N_{id}$
17: Neighs $\longleftarrow \{\}$
18: Aggregation $\longleftarrow$ Value
19: **Setup Periodic Timer Beacon** (ΔT)

20: **Upon Beacon do:** //every ΔT
21: **Call** updateAggregation()
22: **if** ($T_{id} = N_{id}$) **then**
23: $T_{ts} \longleftarrow$ now()
24: msg $\longleftarrow \{\}$
25: **foreach** (id, val, $stat$, tid, $tlvl$) $\in$ Neighs **do**
26: msg $\longleftarrow$ msg $\cup$ (id, Aggregation $\ominus$ val)
27: **Trigger** OneHopBCast ($< N_{id}, T_{id}, T_{lvl}, T_{ts}, P_{id}$, value, msg$>$)

28: **Procedure** updateAggregation()
29: Aggregation $\longleftarrow$ Value
30: **foreach** ($Neigh$, V_{neigh}, $Status$, T_{neigh}, L_{neigh}) $\in$ $Neighs$ **do**
31: **If** ($Status$ = Active) **then**
32: Aggregation $\longleftarrow$ Aggregation $\oplus$ V_{neigh}

33: **Upon** Receive ($<$ id, tid, $tlvl$, tts, pid, val, msg $>$) **do:**
34: **if** $\exists(N_{id}, RecvVal) \in$ msg **then**
35: **Call** updateNeighborEntry(id, tid, $tlvl$, $RecvVal$)
36: **Call** updateTree(tid, tts, $tlvl$, pid, id, val)

37: **Procedure** updateNeighborEntry(id, tid, $tlvl$, val)
38: **if** ($id \notin$ Neighs) **then**
39: Neighs $\longleftarrow$ Neighs $\cup$(id, val, Passive, tid, $tlvl$)
40: **else**
41: Neighs $\longleftarrow$ Neighs $\setminus$(id, _, $stat$, _, _) $\cup$ (id, val, $stat$, tid, $tlvl$)

the management of the support spanning tree in MiRAge does not require any additional exchange of messages.

In a similar fashion to the GAP protocol, we rely on the level of nodes in a tree to establish a tree topology (avoiding cycles). The level of a node in a tree is defined as being the level of its parent plus one. The root of a tree has a level with a value of zero (and for convenience of notation, the parent of the root is defined as being the node itself). When the root of a tree fails, maintaining that tree becomes impossible, as electing a new root involves too much synchronization among nodes (and it would require nodes to have more than local knowledge about the tree topology impairing scalability). Instead, our tree stabilization mechanism allows nodes to switch trees (and/or parent) in some

conditions.

Upon initialization (as denoted in Alg. 1 line 13), each node joins the tree rooted on itself. Whenever nodes exchange aggregation information, they also propagate information on their current tree and their position in the tree, in particular the identifier of their parent in the tree and their current level.

Whenever a node processes an aggregation message (as denoted in Alg. 1 line 33) it takes advantage of that information to run a local stabilization mechanism to manage the support tree. This is achieved through the procedure `updateTree` which is presented in Alg. 2.

In a nutshell, this procedure has the following goals: *i*) a node *a* switches tree if some node *b* belongs to a tree with a lower identifier and *b* becomes parent of *a*; *ii*) if the parent node switches tree, then the node decides to either switch to that tree or to try to establish its own tree as the dominating tree; *iii*) if a cycle is detected, the current tree is considered failed and abandoned; *iv*) if some node *a* considers *b* as its parent then *b* should perceive *a* as being `Active`; *v*) if node *a* is not parent nor child of *b*, *b* perceives *a* as `Passive`. Additionally, some invariants are also enforced, such as the level of a node being the level of the parent plus one and the parent node always being considered as `Active`.

A node decides to switch to another tree if it receives a message from some neighbor that belongs to a tree with a tree identifier that is lower than its current tree (Alg. 2 lines $28-34$). This could lead a node to switch to a tree whose root has failed. To avoid this, each node stores and propagates a timestamp associated with their current tree. This timestamp is only increased by the corresponding tree root (when it executes the periodic `Beacon` task). This acts as a form of heartbeat for the tree root. Nodes store the highest timestamp that they have observed for every tree (Alg. 2 lines $37-38$). This information can be garbage-collected after enough time has passed without receiving any message from a neighbor belonging to that tree.

The use of these timestamps allows a node to only switch to a tree with a smaller identifier if this is the first time it becomes aware of that tree, or if the received message reports a timestamp that is higher than the last timestamp locally observed for that tree. When a node switches to another tree it adopts the node that sent information about that tree as its parent (by updating its local variable P_{id} and setting the state of that node to `Active` in its neighbor list).

Moreover, nodes perform another set of verifications and adaptations whenever they receive a message from a neighbor that enforces the correctness of the tree topology. Whenever a node receives a message from a neighbor for the first time, that neighbor is marked as not being connected to the tree, by setting its state to `Passive` in the receivers neighbor list (Alg. 2 lines $2-3$).

Furthermore, if the node receives a message from a neighbor that considers itself as being its parent (Alg. 2 line 6) belonging to the same tree (Alg. 2 line 8) then the status of that neighbor is set to `Active`. If two nodes believe at some point to be the parent of each other, a cycle has been created, potentially because of the root failure. In this case nodes switch to the trees rooted on themselves (Alg. 2 lines $10-13$).

When a node receives a message from the neighbor that it considers as its current parent in the current tree or in one with lower identifier (Alg. 2 line $14-15$), the node simply reinforces that node as being its parent, by marking it as `Active`, and updating its current tree level to the level of that node plus one (Alg. 2 lines $16-20$). If, however,

Algorithm 2 MiRAge: Tree Management

1: **Procedure** updateTree(tid, tts, $tlvl$, pid, id, val) **do:**
2: **if** ($id \notin$ Neighs) **then**
3: Neighs $\longleftarrow$ Neighs $\cup(id, val,$ `Passive`$, tid, tlvl)$
4: **if** ($T_{id} = tid \wedge tts > T_{ts}$) **then**
5: $T_{ts} \longleftarrow tts$
6: **if** ($N_{id} = pid$) **then**
7: **if** ($P_{id} \neq id$) **then**
8: **if** ($T_{id} = tid$) **then**
9: **Call** changeNeighborStatus (id, `Active`)
10: **else** `//There is a loop in the tree`
11: $T_{id} \longleftarrow N_{id}$
12: $T_{lvl} \longleftarrow 0$
13: $P_{id} \longleftarrow N_{id}$
14: **else if** ($P_{id} = id$) **then**
15: **if** ($T_{id} = tid \vee tid < N_{id}$) **then**
16: **Call** changeNeighborStatus (id, `Active`)
17: $T_{id} \longleftarrow tid$
18: $T_{lvl} \longleftarrow tlvl + 1$
19: **if** ($T_{ts} < tts$) **then**
20: $T_{ts} \longleftarrow tts$
21: **else**
22: $T_{id} \longleftarrow N_{id}$
23: $T_{lvl} \longleftarrow 0$
24: $P_{id} \longleftarrow N_{id}$
25: **else**
26: **if** ($T_{id} \leq tid$) **then**
27: **Call** changeNeighborStatus (id, `Passive`)
28: **else** `//his tree is lower than mine`
29: **if** (Trees$[tid] = \perp \vee tts >$ Trees$[tid]$) **then**
30: **Call** changeNeighborStatus (id, `Active`)
31: $P_{id} \longleftarrow id$
32: $T_{id} \longleftarrow tid$
33: $T_{lvl} \longleftarrow tlvl + 1$
34: $T_{ts} \longleftarrow tts$
35: **else**
36: **Call** changeNeighborStatus (id, `Passive`)
37: **if** ($tid \neq N_{id} \wedge$ (Trees$[tid] = \perp \vee$ Trees$[tid] < tts$)) **do**
38: Trees$[tid] \longleftarrow tts$
39: **Call** checkTreeTopology()

40: **Procedure** checkTreeTopology( ) **do:**
41: **foreach** (id, _, $stat$, tid, $tlvl$) $\in$ Neighs **do**
42: **if** ($stat =$ `Passive` $\wedge\ tid = T_{id} \wedge tlvl < (T_{lvl} - 1)$) **then**
43: **Call** changeNeighborStatus (P_{id}, `Passive`)
44: $P_{id} \longleftarrow id$
45: $T_{lvl} \longleftarrow tlvl + 1$
46: **Call** changeNeighborStatus (P_{id}, `Active`)

47: **Upon NeighborDown** (id) **do:**
48: Neighs $\longleftarrow$ Neighs $\setminus(id, \_, \_, \_, \_)$
49: **if** $(P_{id} = id)$ **then**
50: $P_{id} \longleftarrow N_{id}$
51: **foreach** (id, _, $stat$, tid, $tlvl$) $\in$ Neighs **do**
52: **if** ($stat =$ `Passive` $\wedge tid = T_{id} \wedge tlvl < (T_{lvl} - 1)$) **then**
53: $P_{id} \longleftarrow id$
54: $T_{lvl} \longleftarrow tlvl + 1$
55: **if** $(P_{id} = N_{id})$ **then**
56: $T_{id} \longleftarrow N_{id}$
57: $T_{lvl} \longleftarrow 0$
58: $T_{ts} \longleftarrow$ now()
59: **else**
60: **Call** changeNeighborStatus (P_{id}, `Active`)

61: **Procedure** changeNeighborStatus(id, $stat$) **do:**
62: Neighs $\longleftarrow$ Neighs $\setminus$ (id, val, s, tid, $tlvl$)
63: Neighs $\longleftarrow$ Neighs $\cup$ (id, val, $stat$, tid, $tlvl$)

the parent belongs to a tree with an identifier higher than local node's identifier, then the previous tree's root has failed. In this case the local node decides to switch to the tree rooted on itself, and will try to establish this tree as the new support tree (Alg. 2 lines 21 – 24).

If a node receives a message from a neighbor that belongs to a tree whose identifier is not lower than the tree to which that node currently belongs, it simply marks that neighbor as not being connected to the tree through itself, setting that neighbor status to `Passive` in its local neighbor list.

Finally, an optional optimization mechanism can be employed by a node (denoted by procedure `checkTreeTopology` in Alg. 2 lines 40 – 46) which allows it to switch its parent to the neighbor in that tree with minimal level. In this case, the previous parent is marked as having a state of `Passive`, the new parent is marked, conversely, to have a state of `Active`, and the current tree level is updated to reflect the change.

Essential to the correction of this algorithm is the capacity of a node to detect whenever a node has failed. This is captured in Alg. 2 by the processing of the *neighbor down notification* that is triggered by the local unreliable failure detector (Alg. 2 line 47). In this case the information for the suspected node is removed from the neighbor set, and if the suspected neighbor was the local node's parent, it will try to locate a suitable replacement in its neighbor list. A suitable replacement is a neighbor that is connected to the same tree with a status of `Passive` and a level bellow the local node's own level (to avoid the accidental formation of cycles). If a suitable candidate is found, then the local state of the node is adjusted to reflect the new parent (Alg. 2 line 52 – 54; 60), otherwise the node switches to the tree rooted on itself (Alg. 2 line 55 – 58).

This algorithm ensures the convergence of the state in each node of the system to a configuration where a single tree covers all nodes, if the underlying ad hoc network is connected. This allows nodes to continually exchange their local contribution and their local perception of the aggregated value being computed by each neighbor with all their neighbors, which in turns allows the algorithm to converge to the correct aggregated value, even in cases where the individual contributions of nodes change frequently over time.

(d) Discussion

Supporting the operation of large-scale systems in edge environments requires an effort in monitoring the system, not only to enable system administrators to manage the system, but also to allow the design of autonomic management schemes that can significantly boost the overall performance and user experience. MiRAge posits itself as an algorithm that enables this, even in a complex edge scenario such as ad hoc wireless networks. However, we note that MiRAge will also operate correctly (potentially even better) in other edge scenarios, for instance, if nodes would be interconnected by a infrastructured network running the TCP/IP stack. This use case however, might offer some additional space for improvements and optimizations in the protocol. Therefore, we plan to explore how to better generalize the use of MiRAge on hybrid edge settings, where some nodes have access to infrastructure while others do not.

We also note that the development of MiRAge was conducted using the Yggdrasil framework (described previously). This has two implications, the first is that the im-

plemented protocol is a self-contained module that can be reused. The second is that the protocol follows general assumptions made by the Yggdrasil framework, which for instance involves not having specialized local configuration. For MiRAge to operate, it suffices that a set of devices are active within radio range, where the binary and configuration files of all processes are identical. This, in our opinion, will simplify the task of deploying MiRAge in realistic scenarios.

MiRAge was accepted for publication at the International Symposium on Reliable and Distributed Systems (SRDS). For the convenience of the reader we reproduce the (non-final) camera ready version of the publication in Appendix D

4 Ongoing Work

In this section we discuss multiple lines of work that have been started in the context of WP5. While these works are not mature enough to be considered completed results, they have concrete goals that we discuss here. Additionally, we expect them to become complete in the next few months.

This Section is organized as follows:

Section 4.1 presents a line of work that aims at exploring autonomic microservices. In this Section we discuss how microservices architectures can be leveraged to build self-adaptive systems whose components flow between the cloud and the edge.

Section 4.2 details on-going work regarding Lasp and GRiSP boards. This work explores the exploitation of the Lasp framework for building edge applications in embedded systems.

Section 4.3 describes work that is being conducted for wireless sensor networks, exploring the trade-offs between data accuracy and energy consumption.

4.1 Self-adaptive Microservices in the Edge

(a) Context & Motivation

We expect future edge-enabled applications to have components both in cloud infrastructures and edge devices. These components should be highly dynamic and be able to freely migrate between these two extremes of the edge spectrum. The resulting systems will no longer be purely cloud-based or fundamentally edge-based: they will be hybrid in the sense that they operate on a *hybrid cloud/edge infrastructure*. Fundamentally, this captures applications that can leverage on edge resources scattered throughout the levels E0 to E7 considering the edge spectrum previously presented in Section 3.1.

As of the writing of this document, this line of work is currently evolving. There is still a significant effort to be conducted both in terms of research and engineering. We expect however to have demonstrators built in one year.

(b) Summary of Current Development

This line of work is being pursued by a team at NOVA, in collaboration and coordination with the Lightkone consortium. The team is composed by senior members (i.e., faculty) João Leitão, Maria Cecília Gomes, Nuno Preguiça, and Vitor Duarte, and master students Pedro Ákos Costa, David Mealha, André Carrusca, and André Lameirinhas. The size of the core team is justified by the fact that the overarching goal of the work involves the combination of very different competences, that range from distributed system monitoring, distributed data management, distributed systems architecture, and autonomic computing.

Currently we have conceptually devised a set of three complementary aspects that we consider essential to build highly robust and efficient applications for hybrid cloud/edge infrastructures. Furthermore we have also devised a basic design for the architecture of

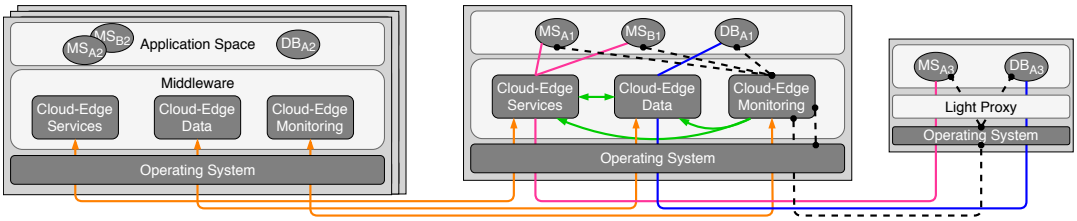


Figure 4.1: Autonomic Microservice Architecture

the support middleware for these applications. In the following, we discuss the three key components of our envisioned solution, and provide a brief description on the architecture of the middleware.

Overview Our proposal is based on building an autonomic microservice architecture that covers cloud and edge infrastructures. Application components, materialized in the form of individual microservices (and, potentially, their own storage solutions). These components will have their life cycles and deployments controlled in an automatic fashion by the support runtime to improve their performance, according to runtime aspects, such as resource availability/consumption and evolving workloads, among others.

A clear challenge in devising an autonomic Microservice Architectures (MSA) is to identify which aspects of the application configuration can be autonomously managed, the information required to execute such reconfigurations, and being able to operate with localized (and possibly incomplete) information. Self-managing applications composed of a large number of components (microservices and database instances) require accessing large volumes of data to guide reconfiguration decisions (as well as significant coordination overhead). This implies that management must be achieved in a decentralized fashion relying on partial and localized information.

We argue that, to make a microservice application autonomic, one has to be able to dynamically control: *i*) the application logic plane, which entails controlling the life cycle of individual instances of microservices, including deciding where new instances should be deployed and how these instances interconnect among each other (i.e., when a microservice has to access another microservice, which instance should it contact). We name this aspect of our proposal *Cloud-Edge Services*; *ii*) the application data plane, which entails controlling the location of data storage replicas used in the operation of individual microservices. This involves not only controlling the life cycle of these replicas, but also understanding which fraction of the microservice state is relevant to be maintained in each of these replicas, and the consistency guarantees (enforced by replication protocols) provided by different replicas located in different cloud and edge resources. We name this aspect of our proposal *Cloud-Edge Data*; and finally, *iii*) an adaptive and distributed monitoring mechanism that can efficiently gather relevant information. Besides information regarding used and available resources, the monitoring component needs to gather the necessary information to guide the dynamic and partial replication of data, and provide this information to all components in the system that make reconfiguration decisions. We name this component of our architecture *Cloud-Edge Monitoring*.

Integration and Implementation through a distributed Middleware We now discuss possibilities that pave the way for realizing and implementing the autonomic microservice architecture discussed above. Figure 4.1 provides an overview of the envisioned architecture. We expect to build a distributed middleware layer that combines the three main components and exposes APIs to simplify the development of applications. This middleware will interact directly with the applications' components being managed (microservice instances in the application logic plane and database instances in the application data plane).

Each component interacts only with the other components within the same middleware instance (green lines), and with the components of the same type in other middleware instances (orange lines). Furthermore, communication among middleware instances is restricted to those that are in close vicinity. This implies that each component operates in a localized fashion, where it exchanges information and coordinates with a limited number of other entities. Additionally, we consider the use of light proxies for edge nodes with limited capacity, enabling the remote and direct management and monitoring of applications' components by a more powerful (nearby) node.

In each middleware instance the three components interact among them. The Cloud-Edge Monitoring component provides information gathered locally and from close-by nodes to the Cloud-Edge Services and Cloud-Edge Data. These take into consideration the current configuration of each one to make adaptation decisions for the application logic and application data planes, respectively.

To interconnect different middleware instances across different cloud/edge nodes, we can use a lightweight and robust overlay network, whose topology is defined to take into consideration the proximity between nodes (i.e., in terms of latency or administrative domains), as well as the needs of each of the three components of our architecture [19, 25, 26]. For instance, the overlay links can map the hierarchical relations between different database instances (i.e., replicas) and/or the flow patterns of monitoring information. This overlay can then be used to efficiently exchange information among the components residing in the middleware instances by leveraging on lightweight and robust gossip mechanisms [26, 27]. The configuration and specification of applications can also be updated by operators, leveraging on this overlay to efficiently and reliably disseminate such modifications.

Evidently, not all edge resources support the direct execution of software developed for cloud environments, in particular those that are tightly coupled with software stacks that are specific to some cloud infrastructures. This is particularly true for storage services, and has two implications. The first is that our middleware layer must offer the execution environment for microservices being managed by the Cloud-Edge Services component. This evidently depends on the capacity of the hardware at the edge. In some cases its materialization may resort to containers (using, for instance, Docker), while in other cases microservices must be executed directly on the operating system. The last option might require multiple implementations of the same microservice. The second implication is that not all data storage solutions can be executed in arbitrary edge hardware, creating the need to develop lightweight versions of these storage solutions that can efficiently execute in hardware with limited capacity. These *edge database instances* will only replicate small fractions of the data. Specialized replication protocols to manage the interactions of edge data storage replicas with counterparts executing in cloud

infrastructures and other edge nodes will need to be developed.

This proposal has been detailed in a paper submitted to the SOCC 2018 conference entitled "*A Case for Autonomic Microservices for Hybrid Cloud/Edge Applications*". For the convenience of the reader, the integral text of this submission is reproduced in Appendix D.

4.2 Lasp Applications for Wireless Edge with GRiSP

As explained in previous deliverables D3.1 and D5.1, Lasp is a programming model and runtime system for writing large-scale coordination-free edge applications with Erlang. Lasp is provided as a library for Erlang developers to integrate into their existing applications. Since the delivery of deliverables D3.1 and D5.1, there were some efforts to continue on improving the stability of Lasp (and its communication library Partisan, as discussed in D3.1). The next challenge to be addressed in the context of Lasp is to explore its utilization in the context of light edge scenarios, in particular to support computations in lightweight devices located very close to end-users.

(a) Context, Motivation, and Goals

To explore the use of Lasp for edge computation, we have started in the second period of the project to pursue the combination of *Lasp with GRiSP boards*. GRiSP is an embedded board designed by partner Stritzinger, as explained in D5.1, which was first manufactured at the end of 2017. Each GRiSP board consists of a processor running Erlang directly on the hardware and a large set of connectors for standard Pmod sensors. Additionally the GRiSP platform included a set of utilities and libraries that simplify the interaction with Pmod devices. As discussed previously in Section 3.2, GRiSP has been improved recently, particularly regarding its software stack.

This work is being pursued by UCL, by a team composed of Peter van Roy and three UCL master's students (Igor Kopestenski, Dan Martens, and Alexandre Carlier) in coordination and collaboration with the Lightkone consortium.

The goals of this work are, in a nutshell, to build a platform for edge computation, to write applications on this platform, and to compare the performance of the devised platform with a traditional cloud-based architecture. The use-case application envisioned to be used in the context of this work, is a data acquisition systems that collects (and potentially co-relates) sensor information gathered from multiple devices. Experimentally, the plan is to evaluate the overall system performance and correctness in a network of 12 GRiSP embedded systems boards augmented with multiple Pmod sensors.

As of the writing of this document, the project is still ongoing, with expected completion at the beginning of August 2018.

(b) Current Development

As part of this line of work, an initial effort was made to port Lasp to the GRiSP embedded system. This was a relatively easy task since GRiSP already has the capacity to run Erlang directly on the bare metal.

Data Management The first use of Lasp in the envisioned use case is to manage data acquired and manipulated by the application. Due to this, there was the need to integrate the data acquisition from Pmod sensors into the Lasp-based application running on edge devices. Sensors are visible at the Erlang level as standard Erlang processes. This makes it easy to write highly reactive edge applications on GRiSP. Each GRiSP board hosts one Lasp process (that represents a node). In the context of the envisioned use-case applications, Lasp is used both as a key/value store and for computation. The first use of Lasp is therefore as a replacement for cloud storage. Because of the replication and convergent coherence among the different Lasp processes (running on different GRiSP boards), this store is highly resilient despite running on an unreliable edge network.

Computational Model The second use of Lasp is for managing edge-computations. To this end, we extended Lasp with a simple task model that allows to distribute arbitrary computations over a network composed of GRiSP devices. Computations can be done in one of two ways, or as a combination of both: (1) as local computations done on each node (standard Erlang computations), or (2) as Lasp computations performed over data encoded in the form of CRDTs [33] on the Lasp storage (according to Lasp's model of CRDT composition). Local computations are stored in the Lasp storage as Erlang higher-order functions, called *tasks*. The result of a local computation can be another task, which is again stored in Lasp. The Lasp storage therefore serves as a resilient shared coordination layer between the computations done on the different GRiSP nodes. Each node runs in a loop, reading tasks from Lasp, running them, and storing resulting tasks in Lasp again. Multiple nodes can do the same task, which provides resilience in case a node fails or becomes unreachable. If a node crashes before it can store its result, since other nodes will continue the computation, the correctness of the system is not compromised.

Using this task model, we are currently experimenting with data streaming and aggregate computations using different kinds of sensors, including navigation, temperature, sonar, ambient light, and so on. GRiSP boards have limited speed (300 MHz) and storage (64 MB), which means that we need to carefully manage time and space resources. In addition to this, it is also planned to explore automatic load balancing mechanisms by using a heterogeneous network where some nodes with additional resources exist in addition to GRiSP nodes.

4.3 Adaptive Sensing in Wireless Sensor Networks with LiteSense

(a) Context & Motivation

The multitude of Wireless Sensor Networks (WSN)s environments, being typically resource-constrained, clearly benefit from properties such as adaptiveness. In particular, such properties can support the operation of highly demanding data gathering applications, as to allow the extension of the lifetime of sensors, among other things. This is a relevant scenario for the context of this work package, since sensors (as well as actuators and things) are at the end of the edge spectrum farther from cloud platforms (see Section 3.1).

To address directly this edge scenarios, a line of work is being pursued to devise new adaptive data sampling schemes for WSNs. This as lead to the proposal of LiteSense, an

adaptive sampling scheme oriented to WSNs aiming at improving the trade-off between capturing data accurately and saving energy to enhance operational lifetime of sensors.

(b) Summary of Current Development

The development of LiteSense is being led by INESC TEC, in particular by João Marco C. Silva and his collaborators.

LiteSense relies on self-regulation of sensing events in order to reduce the amount of data acquired and transmitted without human intervention. It uses the temporal variation in the observed scalar physical quantities in order to self-adjust the interval between two consecutive sensing events.

In a nutshell, when the sampled values of the observed parameter do not vary significantly, the interval between two sensing events is increased, reducing its frequency, which leads to less computational efforts and consequently, less energy consumption. Conversely, if a significant variation in the sampled parameter is observed, the time scheduling for the next sensing event is decreased improving the accuracy in identifying its temporal fluctuation.

A proof-of-concept has provided a demonstration that adaptive sampling can be a robust approach to significantly reduce the number of sensing events and power consumption, while maintaining an accurate view of WSN activity and behavior.

As a second objective, we aim at expanding this scheme as to further increase the efficiency of WSNs data gathering processes, by enriching the previous strategy with information regarding the available energy at a sensor. Effectively, our goal is to devise an efficient energy-aware adaptive sensing scheme that is capable of balancing accuracy in the acquired data with energy conservation.

This scheme leads to the evolution of LiteSense to take into consideration specific WSN data gathering requirements, while extending the sensors lifetime and, consequently, the overall network utility. This is achieved through the use of low-complexity rules, that are particularly specified to optimize the sensing process, the data processing, and the communication overhead. The practical consequence of this approach is that the data acquisition rate is now, not only dependent on the observed variation in scalar physical quantities measured, but also on the perceived devices residual battery level over time.

This line of work has already generated a short paper entitled "*LiteSence: An Adaptive Sensing Scheme for WSNs*" at the IEEE Symposium on Computers and Communications 2018. Another paper entitled "*Flexible WSN Data Gathering through Energy-aware Adaptive Sensing*" is currently under submission to the IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD). For the convenience of the reader, the integral copy of these publications are provided in Appendix D.

5 Relationship with Results from other Work Packages

We now briefly discuss the relationship between the contributions presented and documented here and the work being conducted in the other Work Packages of the Lightkone

project. Some of this relationships have been previously discussed in Deliverable 5.1 [11], we however have decided to lead these discussions here for the convenience of the reader.

WP1: This work package is concerned with data protection and privacy, and it articulates with all other work packages including WP5. The results presented here are not explicitly addressing issues related with data protection and privacy. We expect to address such challenges in the second half of the project, particularly when implementing concrete use-cases and demonstrators on top of the tools and solutions described in this document. We note however, that we discuss issues related with the data protection and privacy in the work line related with the vision for novel edge-enabled applications. We believe that novel edge technology can enable systems and applications that provide additional guarantees in this context.

WP2: The contributions and tools presented here will assist in supporting the design of novel solutions and applications some of which will directly address the use cases reported (and further studied in the last five months) by WP2. In particular, the Yggdrasil framework is currently planned to be applied to two of the use cases selected by WP2, in particular the *Network Monitoring* presented by UPC, and the *Irrigation Control System* presented by GLUK. We discuss these applications further ahead in this document.

WP3: This work package focuses on designing solutions and methodologies to allow the inter-operation of components of edge-enabled (distributed) applications for heavy edge and light edge. A way to enable the inter-operation of application/system components that are scattered across different execution environments is to allow different frameworks and runtime support tools devised by the project consortium to co-exist and interact directly. Yggdrasil, presented here, is planned to evolve to address additional network environments, enabling its easier re-utilization in both heady-edge and light-edge scenarios.

WP4: This work package is focused on devising semantics and programming abstractions for supporting edge applications, whose components might exist in heavy or light edge scenarios. The proposal presented here regarding the design and implementation of novel edge-enabled applications, as well as the on-going work discussed on the use of microservice-based architectures, both create new challenges and opportunities to build correct and adequate abstractions for a new generation of edge applications. Due to this, both of these results are also reported in the context of Deliverable4.2 produced by WP4.

WP6: While WP5 focuses on light edge scenarios, this work package focuses on the complementary heavy edge scenarios. Naturally, we expect future edge-applications to showcase components that operate over both edge scenarios. Our on-going work on developing a microservice-based architecture to support edge applications whose components can, dynamically, be executed on heavy edge and light edge scenarios can be paramount in allowing the natural co-existence of solutions at both ends of the spectrum.

WP7: The WP7 is focused on the evaluation of solutions and systems produced by the Lightkone consortium. In particular, the experimental validation and evaluation of Legion, Lasp (both presented in D5.1) and also of Yggdrasil and MiRAge protocol presented here have been conducted and reported in the context of WP7 (see Deliverable 7.1).

6 Exploitation of Results by Industrial Partners

This section briefly present existing plans for exploitation of the results generated so far by WP5 to some of use cases of the industrial partners. This does not aims at being an exhaustive discussion of all the possible venues for exploitation. Instead the goal is to present concrete plans that we will effectively pursue during the second half of the Lightkone project. In particular, we omit the already on-going efforts to exploit the synergies between the Lasp framework and the GRiSP platform. GRiSP is being continuously used by industrial partner STRITZINGER to build new products and demonstrators. Additionally, we also plan on exploring the possibility of using this platform to address challenges of other industrial partners, such as GLUK (as discussed below in more detail).

6.1 UPC

(a) Relevant use cases

Monitoring systems for Guifi.net: The distributed monitoring system use case for Guifi.net described in D2.1 and D2.2 is presented as a set of monitoring servers sharing a distributed database and network devices (i.e., the monitored devices). The servers, under permanent operation, continuously adjust their monitoring assignment according to the evolution of the overall monitoring system configuration. To achieve this, each monitoring server makes decisions in a decentralized way, based on the information pushed by the remaining monitoring servers to the database. In this scenario, the communication of the information on the actions taken by the different servers is done only indirectly, over the updated states written to the distributed database. However, if direct communication between servers was available, this would allow exchanging additional information. The benefit is that such information could be included into the local decision-making processes of each server, improving their capacity to respond to certain situations in a better way.

Cloudy microcloud computing platform: In Cloudy[10], the microcloud computing platform which hosts (among others) the monitoring server applications, Serf [11] is currently used as the messaging framework that interconnects the different computing devices.

Serf is currently, and successfully, used for services publication and discovery between Cloudy devices, fulfilling most of the current needs, but has shown some limitations that limit further growth, in particular:

1. The payload to be transmitted inside a Serf message is limited to what fits within a single UDP packet. While with small pieces of information this limitation is not relevant, the Cloudy platform faces problems if too many local services are published, which would be the case when the vision of microservices is fully rolled

[10] http://cloudy.community/
[11] https://www.serf.io/

out. The design of Serf is not suitable to easily overcome this limitation as to meet the current needs.

2. The overlay built between Serf nodes has very limited customization possibilities, allowing only to select a value across two possible options to define the global fanout across all overlay nodes. The obtained overlay is not an optimal fit to the heterogeneous network conditions which are found in community networks. One of the consequences of this overlay construction and management strategy is that more than the strictly needed resources (e.g. bandwidth) are spent for message dissemination. Future versions of Serf are not expected to improve on this aspect, since the main use case target of Serf is for its usage within data centers, where the network characteristics are very homogeneous.

3. Serf is mainly used as a mechanism to transfer messages between nodes, without doing any operation on the information contained within payload of exchanged messages (e.g., in-network computations).The potential of such a capability has currently not been taken into consideration, but could become relevant for the monitoring use case and to enable future new applications and use-cases. For instance, performing operations such as aggregation along the message dissemination tree could have advantages with regards to resource usage efficiency. Other operations customized to our specific scenario could further increase the usage scope and capabilities of the monitoring system.

(b) Exploitation of Yggdrasil

The proposed Yggdrasil framework developed within LightKone (and reported in this document and Deliverable 5.1 [11]) may offer more flexibility to surpass the mentioned limitations of Serf and could be applied as a messaging framework for the communication between servers, and between the microservices running on them. Applying gossip dissemination strategies over such a communication layer could support the monitoring servers with additional information about the global system state, while facing less technical constraints. Enabling in-network processing over the content of messages being disseminated may create the opportunity for new scenarios for which these capabilities could be paramount.

Yggdrasil aims at being a framework to support the implementation and execution of distributed protocols, which allows for the creation of efficient communication strategies that operate directly at the network edge. Furthermore, Yggdrasil was designed with the flexibility that enables exploiting the options and techniques discussed above.

The next steps to be taken for the exploration of this direction includes deploying Yggdrasil among a set of nodes deployed in Guifi.net and conduct tests to understand Yggdrasil's capabilities and potential under realistic conditions. The results should provide feedback to be considered in further developments of the framework.

6.2 GLUK

(a) Precision Agriculture

The precision agriculture use case presented by Gluk and originally described in D2.1 (and formalized in D2.2), discusses the use of a wireless sensor network to assist in the everyday operation of a vineyard. In this use case, the sensors collect data about production factors such as soil humidity and periodically (every ten minutes) send their collected data to a gateway, which is responsible for sending this data to the core of the system hosted in a cloud infrastructure. On the cloud, all collected data is analyzed and results are reported back to the gateway. These results are used to manage the agriculture system, such as increasing irrigation of the field. The current development of this system follows the typical architecture of current sensor networks. However, the presented design poses limitations for the future of such systems. Particularly, if such a system is to become more complex, such as having additional types of sensors and actuator, or simply by having a significant increase in the managed area. The limitations here are related with the significant increase in the produced data at the edge of the system, as well as the increasing dependence on the cloud infrastructure as more control is delegated to the system.

To overcome these limitations, some of the data processing tasks could be moved from the cloud infrastructure to the edge of the system. A possible way to achieve this is to rely on the gateway to perform data processing lowering the burden on the network and identify, in a more timely fashion, critical actions that can be decided locally.

Having a single gateway can limit the amount of data processed in this way, potentially limiting the scalability of the system. This implies that the system should be enriched with multiple gateways that should directly coordinate in the end of the system in managing different zones. In particular, neighboring gateways can share the load of managing zones, and also provide some form of redundancy in case of faults. In the following we discuss how the results produced so far by WP5 can be exploited to assist in addressing the challenges associated with this use case.

(b) Exploitation of Yggdrasil and Lasp

There are multiple technologies and results from WP5 that can be exploited to improve this use case. Particularly, the Yggdrasil framework and the MiRAge aggregation protocol are suitable solutions to materialize some of the aspects discussed above. MiRAge can support the aggregation of data from multiple sensors towards gateways, while also enabling gateways to compute aggregates that capture sensed data across multiple zones. An additional interesting feature, is that the natural operation of MiRAge also provides nodes with the current estimates of neighboring nodes. This can provide a means to achieve efficient fault-tolerance among gateways, by ensuring that neighboring gateways have recent information collected and processed.

The Yggdrasil framework can, not only support the operation of MiRAge, but also offer significant flexibility in terms of deployment, as gateways may operate with limited access to infrastructured network. Gateways could communicate by forming a wireless ad hoc network, with only a few of them having up-link connections to cloud infrastructures, lowering the cost of operation. Furthermore, implementing the state machines described

in D2.2 that models the envisioned operation of this use case, would be a simple task, considering the event driven programming model provided by Yggdrasil.

Finally, gateways must synchronize with cloud infrastructures as to allow more sophisticated data analytics to be performed and also to store collected data (or periodic summaries) for archiving. This could be achieved in practice by leveraging on the Lasp framework and runtime. Since gateway nodes in this use case have limited capability (see D2.1) the on-going work on combining Lasp with GRiSP boards can be a point of departure to exploit Lasp in this context.

More precisely, the next steps to be taken in this context are: *i*) to explore how to build a more lightweight version of Yggdrasil that can be ported into both GRiSP and Arduino [1], the later is being currently used in production on this use case. This will imply adapting Yggdrasil to operate in a RTEMS execution environment; *ii*) optimize the execution of distributed protocols (and in particular MiRAge) in this environment; *iii*) Design the demonstrator using Raspberry Pis [2] as Gateways, and leveraging on the existing protocols offered by Yggdrasil; and *iv*) use Lasp to integrate the designed system with components in the cloud.

7 Software Artifacts

Due to the limited span of time between the previous deliverable for WP5 and the current deliverable, we only present evolutions of some of the software artifacts presented in Deliverable 5.1. These artifacts include: the new version of the Yggdrasil framework that includes, not only the modifications reported previously (Section 3.3), but also implementations of a new set of distributed protocols; and the new version of the software associated with the GRiSP platform, previously discussed in Section 3.2.

The new protocols featured in Yggdrasil include a set of aggregation protocols implemented in Yggdrasil, which also contains our own proposal for a new continuous aggregation protocol, named MiRAge (presented in Section 3.4). Another component of this software artifact is the first prototype of a control protocol that simplifies the use of Yggdrasil as a benchmark platform. We however, expect to revise the design and implementation of this prototype to improve its stability and also to adapt its operation to take advantage of new features that will be added to Yggdrasil.

The software for the GRiSP platform includes all improvements discussed in this document. The software itself combines: *i*) the GRiSP Erlang Runtime Library that exports to Erlang operations directly related with the GRiSP board; and *ii*) the rebar3 grisp that provides the Rebar plug-in for the GRiSP project.

The code for the new version of Yggdrasil (and protocols designed for the framework) can be publicly accessible through the Lightkone WP5 public git repository at: https://github.com/LightKone/wp5-public.git

8 Papers and publications

Some of the results reported in this document have been made public through the following publications. For convenience of the reader, we provide the full text of these publications in Appendix D. We note that some of these are, at the time of the writing of this report, under submission, while one is a public technical report available in the Arxiv platform.

- Pedro Ákos Costa and João Leitão. Practical Continuous Aggregation in Wireless Edge Environments. Proceedings of 37th IEEE International Symposium on Reliable Distributed Systems (SRDS'18). Salvador, Brazil, 2018 (accepted for publication).
- João Marco C. Silva, Kall Araujo Bispo, Paulo Carvalho, and Solange Rito Lima. LiteSense: An adaptive sensing scheme for WSNs. Proceedings of the IEEE Symposium on Computers and Communications (ISCC), Heraklion, 2017, pp. 1209-1212.
- João Leitão, Pedro Ákos Costa, Maria Cecília Gomes, and Nuno Preguiça. Towards Enabling Novel Edge-Enabled Applications. Technical Report arXiv:1805.06989. https://arxiv.org/abs/1805.06989. May 2018.
- João Leitão, Maria Cecília Gomes, Nuno Preguiça, Pedro Ákos Costa, Vitor Duarte, David Mealha, André Carrusca, and André Lameirinhas. A Case for Autonomic Microservices for Hybrid Cloud/Edge Applications. Submitted as a *Vision Paper* to the ACM Symposium on Cloud Computing (SOCC) 2018.
- João Marco Silva, and Kalil Bispo. Flexible WSN Data Gathering through Energy-aware Adaptive Sensing. Submitted to the IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks 2018 (CAMAD).

9 Final Remarks

This deliverable reports the results achieved by the Lightkone consortium on devising new solutions and tools to support edge-enabled applications, with a particular emphasis of solutions tailored for light-edge scenarios. These scenarios are composed by system/application components whose communication and interactions are dominated by components that lie closer to the end-user. Overall, the goal of this report is to discuss contributions toward the support of efficient and robust general purpose computations in light edge scenarios.

To this end, we have presented our own view of the highly heterogeneous executions environments that we expect to see in future edge-enabled applications and systems. We also complemented this vision by discussing possible uses for these different execution environments for applications and systems. We illustrate this medium-term vision with a few application use-cases.

Improvements over the GRiSP platform were made, mainly on the software stack including available drivers that support developers using GRiSP. The GRiSP platform was also highly divulged through numerous talks. These efforts will help popularize GRiSP as a platform to build novel edge-enabled applications that take advantage of embedded systems. Additionally, we have reported on the continued effort to develop Yggdrasil, a framework to build efficient and correct distributed protocols and applications for edge scenarios, where nodes have to resort to ad hoc networks for interacting among them. We further discussed how we plan to evolve Yggdrasil to make it suitable for other execution environments.

We also report the design of a novel aggregation protocol for ad hoc networks (that was developed using Yggdrasil) and that, in some sense, complements the work reported previously on D5.1 [11] on data aggregation in the edge. This protocol features interesting properties, such as being fault-tolerant and supporting *continuous aggregation*, where the view of aggregate values by each node evolves over time when the input values of individual processes change.

Finally, we have reported on three lines of work that have been started recently and that further explore the support of applications in the edge. The first explores the potential use of microservice architectures, enriched with autonomic features, to allow application/systems components to naturally move or be replicated between data centers, and execution locations closer to end-users. Another on-going line of work aims at building a demonstrator for the possibility of executing applications developed on top of Lasp in the GRiSP platform, making the first suitable for integrated systems. Both Lasp and GRiSP have been previously introduced in D5.1. Finally, another on-going effort is exploring methodologies to dynamically adjust the trade-offs between data accuracy and energy consumption in wireless sensor networks. Since sensor are highly resource constrained devices, research towards enabling data sampling techniques to self-adapt is crucial to extend the life of such networks.

As part of this deliverable we also present software artifacts that are now publicly available through the work package public git repository.

References

[1] Arduino - Products. https://www.arduino.cc/en/Main/Products.

[2] Raspberry Pi 3 Model B - Raspberry Pi. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[3] Cisco. Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf, 2015. Accessed: 2018-05-16.

[4] Hu Yan (Huawei iLab). Research Report on Pokémon Go's Requirements for Mobile Bearer Networks. http://www.huawei.com/~/media/CORPORATE/PDF/ilab/05-en, 2016. Accessed: 2018-05-16.

[5] Raka Mahesa (IBM). FHow cloud, fog, and mist computing can work together. https://developer.ibm.com/dwblog/2018/cloud-fog-mist-edge-computing-iot/, 2018. Accessed: 2018-05-16.

[6] Sebastian Abshoff and Friedhelm Meyer auf der Heide. Continuous aggregation in dynamic ad-hoc networks. In Magnús M. Halldórsson, editor, *Structural Information and Communication Complexity*, pages 194–209, Cham, 2014. Springer International Publishing.

[7] I. F. Akyildiz and Xudong Wang. A survey on wireless mesh networks. *IEEE Communications Magazine*, 43(9):S23–S30, 9 2005.

[8] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 10 2015.

[9] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[10] Jen-Yeu Chen, G. Pandurangan, and Dongyan Xu. Robust computation of aggregates in wireless sensor networks: Distributed randomized algorithms and analysis. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):987–1000, 9 2006.

[11] LightKone Consortium. D5.1: infrastructure support for aggregation in edge computing. https://github.com/LightKone/wp5-public/blob/master/deliverables/D5.1.pdf, feb 2018.

[12] Mads Dam and Rolf Stadler. A generic protocol for network state aggregation. *self*, 3:411, 2005.

[13] Claudia Doppioslash and Adam Lindberg. Visualizing Home Automation with GRiSP. LambdaDays 2018, 2018.

[14] Ericsson AB. Download OTP 21.0. https://www.erlang.org/downloads/21.0, 2017.

[15] Ericsson AB. Download OTP 20.0. https://www.erlang.org/downloads/20.0, 2018.

[16] Ittay Eyal, Idit Keidar, and Raphael Rom. Limosense: live monitoring in dynamic sensor networks. *Distributed computing*, 27(5):313–328, 2014.

[17] Mário Ferreira, João Leitão, and Luis Rodrigues. Thicket: A protocol for building and maintaining multiple trees in a p2p overlay. In *Proc. of SRDS'10*. IEEE, 2010.

[18] Kilian Holzinger and Peer Stritzinger. Realtime Functional Reactive Programming with Erlang. LambdaDays 2018, 2018.

[19] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Comput. Netw.*, 53(13):2321–2339, August 2009.

[20] P. Jesus, C. Baquero, and P. S. Almeida. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys Tutorials*, 17(1):381–404, 1 2015.

[21] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. Fault-tolerant aggregation by flow updating. In Twittie Senivongse and Rui Oliveira, editors, *Distributed Applications and Interoperable Systems*, pages 73–86, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[22] David Johnson, Ntsibane Ntlatlapa, and Corinna Aichele. Simple pragmatic approach to mesh routing using batman. In *2nd IFIP International Symposium on Wireless Communications and Information Technology in Developing Countries (WCITD2008)*. IFIP, 2008.

[23] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 482–491, 10 2003.

[24] Oliver Kennedy, Christoph Koch, and Al Demers. Dynamic approaches to in-network aggregation. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 1331–1334. IEEE, 2009.

[25] J. Leitão, J. P. Marques, J. Pereira, and L. Rodrigues. X-bot: A protocol for resilient optimization of unstructured overlay networks. *IEEE TPDS*, 23(11), 11 2012.

[26] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429, June 2007.

[27] João Leitão, José Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 301–310. IEEE, Oct 2007.

[28] Adam Lindberg and Peer Stritzinger. 1000 Nodes, Large Messages, We Want It All! Prototype With New OTP 21 API. CodeBEAM STO 2018, 2018.

[29] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions, 2018.

[30] Rashid Mehmood, Jrn Schlingensiepen, Lars Akkermans, M. Cecilia Gomes, Jacek Malasek, Lee McCluskey, Rene Meier, Florin Nemtanu, Angel Olaya, and Mihai-Cosmin Niculescu. Autonomic systems for personalised mobility services in smart cities. Technical report, King Khalid University, et. al., 2014.

[31] Sébastien Merle. From Cloud to Edge Networks. CodeBEAM SF 2018, 2018.

[32] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, October 2010.

[33] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proc. of 13th SSS'11*, Berlin, Heidelberg, 2011. Springer-Verlag.

[34] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[35] William Tärneberg, Vishal Chandrasekaran, and Marty Humphrey. Experiences creating a framework for smart traffic control using aws iot. In *Proc. of the 9th International Conference on Utility and Cloud Computing*, UCC'16, pages 63–69, New York, NY, USA, 2016. ACM.

[36] Gonçalo Tomás, Peter Zeller, Valter Balegas, Deepthi Akkoorath, Annette Bieniusa, João Leitão, and Nuno Preguiça. Fmke: A real-world benchmark for key-value data stores. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '17, pages 7:1–7:4, New York, NY, USA, 2017. ACM.

[37] Robbert Van Renesse. The importance of aggregation. In *Future Directions in Distributed Computing*, pages 87–92. Springer, 2003.

[38] Y. Yan, N. H. Tran, and F. S. Bao. Gossiping along the path: A direction-biased routing scheme for wireless ad hoc networks. In *2015 IEEE Global Communications Conference*, pages 1–6, 12 2015.

[39] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, Mobidata '15, pages 37–42, New York, NY, USA, 2015. ACM.

[40] Nadezda Zryanina. GRiSP, Bare Metal Functional Programming. BOBkonf 2018, 2018.

A Description of Aggregation Protocols implemented in Yggdrasil

- Push Sum [23] is a well know aggregation protocol where nodes continuously exchange their internal state. The internal state of a node is composed of two components, the value being aggregated and an associated weight. Periodically, each node chooses a random neighbor to whom it will send information. The node proceeds to split its value and weight in half, transmits one half, and keep the other. Once the randomly chosen neighbor receives the value and the weight, it incorporates them into its own local state. Nodes repeat this process indefinitely (or until some predefined stop criteria is met), and an aggregation result can be obtained through the division of the value by the weight.

- LiMoSense [16] is a variant of the Push Sum algorithm, that is enriched with fault tolerance. The LiMoSense protocol proceeds in the same fashion as Push Sum but keeps track of the values that have been sent and received by a given neighbor. This allows LiMoSense to recover the values that would have been lost in the case of message loss or node failures. The protocol also has mechanisms to deal with variation of the input value of nodes for the aggregation process, hence supporting a form of continuous aggregation.

- DRG [10] works in iterative steps similarly to Push Sum however, instead of nodes simply exchanging values, they create random local groups at each iteration. To this end, whenever a node starts its iterative step it decides with some probability to become a group leader. The leader transmits a one-hop broadcast message to establish a group. The nodes that receive such message, acknowledge it by sending a join message to that group leader containing their current aggregated value. After receiving several join messages for a period of time, the leader node computes a local average of the group and one-hop broadcasts the result to all group members, terminating the group. As nodes will randomly form groups, the computed average will be propagated throughout the network, allowing every node to compute a global average.

- Flow-Updating [21] is another iterative approach where, contrary to Push-Sum and variants, nodes exchange and maintain flows to all their neighbors that encode the difference between the local estimation of the aggregate result of a node and that of its neighbor. Flows are continually updated to reflect changes in the computed local estimate. Due to the maintenance of state for each neighbor, this protocol is robust to both message loss and node crashes.

- All previously mentioned aggregation protocols function over a random network topology. GAP [12] however, does not. GAP operates by establishing a tree topology over the existing ad hoc network. The tree is used to establish relationships between nodes, being either `Parent`, `Child`, or `Peer` and computing the aggregation function using the values provided by the child nodes and propagating the partial result that combines the values received by all children with the local node

input value to its parent nodes. This allows the root node to obtain the global aggregate results. Nodes communicate by issuing messages to all neighbors (which can leverage one-hop broadcast in wireless ad hoc networks). These messages are sent periodically and in reaction to external events (such as the input value of a node changing). These messages contain the local perception of the node, including its current parent and its local aggregated value. Each node also transmit its level on the tree topology (the root node having level 0). The level is used to ensure the correctness of the tree topology.

- In GAP only the root node is able to compute the aggregation result. To enable all nodes to become aware of the aggregation result, the root node must broadcast the result to all the nodes in the system. We have implemented a variant of GAP that does this. This broadcast process is achieved by piggybacking on other control messages issued by GAP.

B Commands Supported by the Yggdrasil Control Process

Start Experience: This command begins an experience. According to the operation mode of the Yggdrasil Control Process (described in Section 3.3), this command will either create a child process given the path to the binary that is provided as an argument to the command, or ask the runtime to start a given protocol (also passed as argument to the command). Additionally, this command will redirect the standard output of the executing process or protocol to a statically defined file.

Stop Experience: A command used to terminate an ongoing experience. According to the operation mode of the Yggdrasil Control Process, this command will either kill the previously created child process, or ask the runtime to stop the given protocol. A second effect of the execution of this command, is that the output file being used by the terminated process or protocol is moved to a different location, which is provided by the user as an argument to the command.

Change Link: This command will change the status of a link between two nodes. Yggdrasil has a protocol that is capable of blocking incoming and outgoing communication from/to a particular device (i.e., a link). By defaults, all links are active. This command allows to switch the state of a link, if ti was active, then its state is changed to blocked, and all messages sent to, or received from, that neighbor are transparently dropped. If the link was already block the opposite happens. This feature is currently only used when testing protocols or applications, as to simulate network partitions or interference in the wireless medium. However, in the future, this could also be used to block unstable/highly unreliable links.

Change Value: This command will change the input value of a node, that is used in the context of the execution of a distributed aggregation protocol. Similarly to the previous command, the change value command is only used to perform experiments, in particular, for aggregation protocols.

Setup Tree: The setup tree command is a utility that forces the transmission of a message with only a control code, that effectively forces the control core protocol to define the (distributed) spanning tree among nodes. This command is usually employed to accelerate the establishment of the spanning tree among all nodes of a deployment, prior to the start of an experiment.

Check Tree: This command allows the user to test the number of processes currently connected to the spanning tree employed by the control core protocol to disseminate commands. This is a debug utility that is typically employed to ensure that all nodes are in standby before starting an experiment.

Disable Discovery: This command will disable the Yggdrasil Control Process discovery protocol. This is typically performed to minimize the noise produced by the periodic transmission of announcement messages during experiments.

Enable Discovery: This command will enable the Yggdrasil Control Process discovery protocol. Effectively reverting the effects of the previously document command.

Debug This is, as the name implies, a debug operation, that leads any node that receives the command to print to its log, his current perception of his neighbors and if the link between the local node and each of its neighbors is part of the spanning tree used by the control core protocol. Contrary to all other commands, which are disseminated throughout all nodes of an experimental deployment, this command is not disseminated and hence, only affects the local node.

C List of Acronyms

API Application Programming Interface
CPU Central Processing Unit
GAP Generic Aggregation Protocol
IoE Internet of Everything
IoT Internet of Things
ISP Internet Service Provider
MSA Microservice Architectures
OS Operating System
P2P Peer-to-Peer
RTEMS Real-Time Executive for Multiprocessor Systems
SSH Secure Shell
SSL Secure Socket Layer
WP2 Work Package 2
WP4 Work Package 4
WP5 Work Package 5
WP7 Work Package 7
WSN Wireless Sensor Networks

D Relevant Publications

Practical Continuous Aggregation in Wireless Edge Environments

Pedro Ákos Costa and João Leitão
NOVA LINCS & DI/FCT/NOVA University of Lisbon, Lisboa, Portugal
pah.costa@campus.fct.unl jc.leitao@fct.unl.pt

***Abstract*—The edge computing paradigm brings the promise of overcoming the practical scalability limitations of cloud computing, that are a result of the high volume of data produced by Internet of Things (IoT) and other large-scale applications. The principle of edge computing is to move computations beyond the data center, closer to end-user devices where data is generated and consumed. This new paradigm creates the opportunity for edge-enabled systems and applications, that have components executing directly and cooperatively on edge devices.**

Having systems' components, actively and directly, collaborating in the edge, requires some form of distributed monitoring as to adapt to variable operational conditions. Monitoring requires efficient ways to aggregate information collected from multiple devices. In particular, and considering some IoT applications, monitoring will happen among devices that communicate primarily via wireless channels. In this paper we study the practical performance of several distributed continuous aggregation protocols in the wireless ad hoc setting, and propose a novel protocol that is more precise and robust than competing alternative.

***Index Terms*—Aggregation Protocols, Wireless ad hoc, Distributed Fault-Tolerant Protocols, Edge Computing**

I. Introduction

Edge computing has emerged as a paradigm to overcome some of the scalability issues of cloud computing [1]. These limitations arise from the limited capability of cloud infrastructures to receive and process vast amounts of data in a timely fashion. This interest has been boosted by emerging new applications in the domain of Internet of Things (IoT) [2]. The rising popularity of these novel applications will lead to a tremendous growth in data production that will surpass the network's capacity for data producers and consumers to effectively transfer data to cloud data centers [3].

In a nutshell, edge computing implies performing computations outside of the data center boundary, on devices closer to end clients of a system [4]. Therefore, edge computing can take many different forms depending on four main factors: i) the devices being leveraged to perform computations; ii) the communication medium used by those devices; iii) the interaction pattern of those devices with the remaining components of a system; and iv) the nature of the computations being performed at the edge of the system.

However, given the vast nature of the edge, we are particularly interested in a concrete edge scenario, that of multiple commodity devices being used to perform distributed computations over a given geographical area, where a structured network infrastructure is not available. This can be the case for some applications in the domain of smart cities and smart spaces [5]. One such materialization of this could be, for instance, monitoring transit within a city enriched with local decisions regarding traffic signals to aleviate areas of high traffic density in a timely fashion [6], without the need of time consuming centralized control.

The realization of this scenario would require a large number of devices with wireless capability. Although, the absence of network infrastructure, and the non-negligible costs associated with its installation (i.e, adding access points), motivates the need to have these devices create an infrastructure-less network, or ad hoc network [7]. In such a network, devices would interconnect forming a multi-hop network. This network can have routing capabilities (creating a Mesh network [8]), but this would hinter the possibility to leverage the network's capacity to perform in-network computation, as messages transversing the network would be controlled by a routing protocol, instead of some application level protocol that is capable of modifying the contents of messages. Furthermore, routing protocols have non-negligible overhead, which further reenforces the benefits of using a simple ad hoc network.

To allow these systems to gather relevant information regarding their operation, deployment and execution environment, or even application-level data, efficient and reliable distributed monitoring solutions are required. Key to the design of monitoring solutions is the capacity to aggregate information produced or managed independently by large numbers of devices using a distributed aggregation protocol [9].

In the particular case of monitoring operational aspects of distributed systems, the individual values owned by each node are not static. In fact, in many use cases the values being aggregated change over time. Hence, we need to consider a particular form of aggregation named *continuous aggregation* [10] where the value being computed by the distributed aggregation protocol is continually updated to reflect modifications in the individual input values or changes in the system affiliation (i.e, no longer taking into consideration the value of a node that leaves the system or fails).

In this paper we study the practicality of performing distributed continuous aggregation, paving the way for novel edge computing applications that can leverage on large amounts of commodity devices that are seamlessly interconnected through a wireless ad hoc network. To do this, we start by discussing the requirements of a distributed aggregation protocol for

The work presented here was partially supported by the Lightkone European H2020 Project (under grant number 732505) and NOVA LINCS (through the FC&T grant UID/CEC/04516/2013).

ad hoc networks (Section II). We then survey some of the most popular distributed aggregation protocols (Section III). We propose a novel distributed continuous aggregation protocol that we named *MiRAge* (Section IV). The practical performance of our protocol is experimentally validated and compared with the some solutions found in the literature on a real deployment using a fleet of 24 Raspberry Pi3 - Model B (Section V). Finally, we conclude this paper by identifying interesting venues for future research (Section VI).

II. Aggregation in the Edge

Aggregation is an essential building block in many distributed systems [11]. A distributed aggregation protocol coordinates the execution of an aggregation task among devices of a system. In short, a distributed aggregation protocol should compute, in a distributed fashion, an aggregation function over a set of input values, where each input value is owned by a node in the system. Typical aggregation functions include *count*, *sum*, *average*, *minimum*, and *maximum*.

Not all of these aggregation functions are equivalent regarding their distributed computation. Where count, sum, and average are sensitive to input value duplication, maximum and minimum are not. Distributed aggregation protocols must take measures to deal with these aspects, ensuring that the correct aggregate is computed.

Distributed aggregation protocols for edge systems, and in particular for settings where edge nodes interact through a multi-hop wireless ad hoc network, should be designed to address some key challenging aspects of this environment:

Continuous aggregation: As discussed in the previous section, the input values used for computing the aggregate function may change over time. Consider the example where one is computing the average CPU utilization among a collection of edge nodes, the individual usage of CPU by each node will vary accordingly to the tasks being executed by that device. To cope with this aspect, we argue in favor of using protocols that can perform *continuous aggregation*. Continuous aggregation was initially coined in [10] as being a distributed aggregation problem where periodically, every node receives a new input value for the aggregation discarding the previous one. However, this notion implies that periodically all nodes restart the protocol [12]. In practice, not all input values change at the same time, and the change of a single value should not require an effort as significant as restarting the whole aggregation process. Instead the protocol should naturally incorporate input value modifications continually and with minimal overhead.

Fully decentralized: Distributed aggregation in the context of edge computing is paramount to enable the self-management of edge computing platforms. To promote timely management decisions and overall system availability, one should favor local reconfiguration decisions with minimal coordination among nodes. This has two relevant implications in the properties of aggregation protocols. First, every node in the system should have local access to the result being produced by the aggregation protocol. The second, is that the algorithm should not depend on the activity of specialized nodes in the system, and operate in a fully decentralized fashion.

Fault tolerance: Distributed algorithms should be tolerant to node failures, which are unavoidable in any realistic distributed setting. Since we are considering a particular edge setting where nodes communicate through a shared multi-hop ad hoc network, external interference is inevitable, which can be a result from having other devices in close vicinity using the wireless medium. This implies that a successful aggregation distributed protocol for this setting should be highly robust to message loss.

Minimal overhead: Considering the particular case of supporting distributed monitoring schemes, the execution of the aggregation protocol should have minimal overhead as to minimize its impact on any other services or applications being executed on edge devices. In particular, communication should have a low cost regarding the number of messages exchanged among nodes.

III. Related Work

Distributed aggregation algorithms have been widely studied in the past, particularly in the context of sensor networks [13]–[15] and peer-to-peer systems [16]. In the context of sensor networks, most solutions strive to propagate aggregation results towards a special node in the network, called *sink*, potentially performing in-network computation on the nodes alongside the route to the sink. In peer-to-peer systems aggregation protocols were mostly dedicated to counting the number of nodes in the system.

There are different classes of distributed aggregation algorithms that differ on the precision of the computed aggregate value, the supported aggregation functions, and how they deal with duplicated input values.

Some protocols are designed to minimize the overhead by exploiting sampling techniques that compute approximate values for an aggregate function by only gathering information from a small subset of nodes. Examples of this technique include Random Tour and Sample & Collide [16] as well as Randomized Reports [17]. Unfortunately, the precision of these solutions is highly sensitive to the distribution of input values among the nodes of the system.

Other distributed aggregation algorithms resort to somewhat complex data structures in order to collect more information from the system than simply computing an aggregation function. For instance, Q-Digest [18] and Equi-Depth [19] can compute histograms with distributions of input values in the network by having nodes exchange collections of values among them. These solutions are much more computational demanding and the distribution computed by these solutions can have errors that make it impossible to use it to compute a precise aggregate. Additionally, Q-Digest can only compute the distribution at a special sink node. Extrema Propagation [20], on the other hand, transforms the problem of computing an average in computing a minimum vector among all nodes in the system by having nodes iteratively exchange information, which is not sensitive to input duplicates. Still the solution can only compute an approximate result.

Many solutions rely on having nodes continually exchange information among them to compute estimates of the aggregate

result that become increasingly precise with the number of iterations. Distributed Random Grouping (DRG) [15] has nodes iteratively form groups with a leader. The leader gathers the current estimates of the group members, compute a new estimate from those contributions and his own estimate that is then propagated to all elements of the group.

Push-Sum [21] is a very well known protocol that operates by having each node iteratively exchange their input value and an additional parameter called *mass*. At each communication step, a node divides its local value and mass and transfer those to a random peer, that incorporate those into their local value and mass, respectively. At any moment a node can compute its local estimation of the result of the aggregation function by dividing its current value by the locally stored mass. This protocol can only be used to compute the average or sum/count. The correctness of the protocol depends on no mass being lost from the system, which implies that it is not robust neither to message loss nor node crashes. There are however, variants of this protocol that try to deal with this issue. LiMoSense [22] employs the same principles of Push-Sum, but stores the mass received and sent to each peer, being then able to restore it in the case of failures. Additionally, LiMoSense has nodes maintain a copy of their input value, allowing them to modify their input during the execution of the protocol, an aspect not explicitly supported by Push-Sum.

Flow-Updating [23] is another iterative approach where, contrary to Push-Sum and variants, nodes exchange and maintain flows to all their neighbors that encode the difference between the local estimation of the aggregate result of a node and that of its neighbor. Flows are continually updated to reflect changes in the computed local estimate. Due to the maintenance of state for each neighbor, this protocol is robust to both message loss and node crashes. Unfortunately, this protocol can only compute precise aggregations of the average function, being unclear how to generalize this approach to other aggregation functions.

Finally, some aggregation protocols leverage on tree topologies to enable efficient aggregation avoiding the duplication of input values. The most well known of these solutions is the Tiny Aggregation (TAG) protocol [13], that was developed as part of the TinyOs [24] to enable efficient aggregation in sensor networks using a sink node (usually not a sensor). The tree is built by having the sink node broadcast a message to the sensor nodes that retransmit it and set as their parent in the tree the node from whom they receive the broadcast for the first time. In TAG the tree is constructed by having nodes that are connected in the tree to schedule their radios to be active in overlapping periods (which saves energy for resource restricted sensors). Aggregation happens by having nodes report to their tree parent the result of the aggregation of its own input value with the aggregation result of all its tree children. TAG also features a fault-tolerance mechanism that relies on a per node cache with previous values received from their children, that can be re-used in case of failure.

The Directed Acyclic Graph [14] protocol, further improves the fault tolerance of TAG by building a multi-path tree rooted in the sink node, assigning a grandparent node to every node and leveraging on these grandparent nodes to effectively compute partial aggregation values. This allows computations to proceed even if some nodes fail during the propagation of input values and partial aggregates towards the sink node.

Generic Aggregation Protocol [25] is another algorithm that relies on a tree topology however, contrary to TAG and DAG, the management of the tree in GAP happens naturally with the exchange of values among nodes to compute the aggregate (i.e, without needing the broadcast from a sink node). The process to build the tree is governed by an additional parameter maintained by each node called its *level*, which initially is set to an arbitrary large value. The tree is formed by an appointed root (that operates like a sink node) that has a virtual neighbor named *virtual root* with a constant level of -1. Each node maintains a set with information about all nodes with whom they exchange information (either receive or send). This set contains, for each other node, the current level of the node, and its relative relation with the local node in the tree that can be either `PARENT`, `CHILD`, or `PEER`, and the last aggregate value observed from that node. Each message sent by a node contains information that enables the receiver to update this data structure and also for the receiver to update its local perception of the (current) tree topology, for instance, who is the current parent in the tree of a node. This is achieved by enforcing a set of invariants, for instance, a node will always consider as its parent in the tree, the node that has the lowest level. Additionally, if a node receives information from a node that believes to be its child, it marks the node accordingly in its local data structure. The aggregate value of a node is computed by combining the aggregated values of all its children in the tree and its own input value. Due to the decentralized nature of the tree management algorithm employed by GAP, the protocol is fault tolerant as long as the root of the tree does not fail.

All the solutions based on trees discussed above require the existence of a sink to build and manage the tree supporting the execution of the aggregation. If the sink becomes unavailable the protocols are no longer capable of operating and computing an aggregate result. Furthermore, in these solutions only the sink node becomes aware of the aggregation result as part of the execution of the protocol. If this result is relevant to the remainder nodes of the system, it has to be broadcasted by the sink node after its computation.

IV. The Design of MiRAge

In this section we discuss the design of our own distributed aggregation protocol. Our protocol is inspired in the design of GAP [25] discussed previously, but generalizes its design to remove the dependence of a single root. To this end our protocol leverages on a self-healing spanning tree to *support* efficient continuous aggregation. In our protocol, that we named Multi Root Aggregation protocol, or simply MiRAge, all nodes compete to build a tree rooted on themselves. This competition is controlled via the identifier of each node (a large random bit string) and a monotonic sequence number (i.e, a timestamp) controlled by the corresponding root node.

Additionally, our protocol was designed to ensure that all nodes in the system are able to continually compute and update the result of the aggregation function.

In the following, we discuss the system model assumed in the design and implementation of MiRAge. We then discuss in detail the design of our algorithm. We start by explaining how the aggregate value is computed by each node. Then we explain how the natural evolution of the protocol via the exchange of messages between nodes, allows to maintain the self-healing spanning tree that support the aggregation.

A. System Model

We assume a distributed system where nodes communicate via message exchange. Furthermore, we assume that devices are equipped with an WiFi radio capable of operating in ad hoc mode. Each node is pre-configured to join a single ad hoc network. We do not assume any routing algorithm or infrastructure access. Devices can transmit messages using one-hop-broadcast, where the message can be received (with some probability) by all or a subset of devices in the transmission range of the sender.

No node is aware of the total number of nodes in the system. However, we assume that each node has a unique identifier (this can be achieved by having each node generate a large random bit string at bootstrap). We do not make any assumption regarding clock synchronization, although, we assume that each node perceives the passing of time at a similar (albeit, not necessarily equal) rate.

Finally, we assume that each device runs a discovery protocol, where periodically the node transmits (in one-hop broadcast) an announce containing its own identifier. The period of this transmission is controlled via a parameter ΔD. This protocol is also used by each node as an unreliable failure detector, where if the announce of a known node is not received for a consecutive number of transmission periods large enough, the node becomes suspected of having failed, generating a notification to the aggregation protocol. The number of transmissions that a node can miss before suspecting another one is a parameter denoted K_{fd}. This is an assumption made by many other aggregation protocols including GAP [25], LiMoSense [22], Flow-Updating [23], among others.

B. Aggregation Mechanism

Algorithm 1 describes the local state maintained by each node executing MiRAge and the components of the protocol related with the aggregation computation at each node.

The intuition of MiRAge is quite simple. Nodes organize themselves in a fault tolerant *support* spanning tree that is used to control the aggregation mechanism. Nodes periodically propagate to their neighbors the local estimate of the aggregation result, that is obtained by applying the aggregation function over their own input value and the (latest) estimates received from neighbors with whom they share a tree link.

In MiRAge, each node owns a unique node identifier (Alg. 1 line 2) and its own input value for the aggregation (Alg. 1 line 3). Additionally, each node maintains its current estimate of the aggregate value (Alg. 1 line 4) and a set of known neighbors (containing for each its node identifier; its latest received aggregated value; its current status in the support spanning tree, being `Active` if the neighbor is considered to be part of the tree through the local node, `Passive` otherwise; the identifier of the tree that the neighbor is connected to; and its level in that tree (Alg. 1 line 5).

Each node also owns a set of local variables that capture its current position and configuration in the support spanning tree. This includes: the identifier of the tree to which the node is currently connected (tree identifiers are the identifier of their root node), its level (the root of the tree has level zero), the highest timestamp observed, and the identifier of the parent node (Alg. 1 lines $6-9$). In addition, each node stores a map that associates tree identifiers to the last locally observed timestamp for that tree (Alg. 1 line 10).

When a node is initialized (Alg. 1 line 11) it has no knowledge regarding existing neighbors. Due to this, it assumes that the result of the aggregation function is its own value, and initializes the state related with the support spanning tree to reflect a tree rooted on itself (the tree identifier being its own identifier). Additionally, the node setups a periodic function named `Beacon` that is executed every ΔT which corresponds to the main aggregation logic of our algorithm. A typical value for ΔT is one or two seconds.

When the Beacon procedure is executed, a node will start by updating its local estimate. This is achieved through the execution of the `updateAggregation` procedure, which applies the aggregation function (operator $\oplus$ in Alg. 1 line 32) to the input value of the node with the received estimates of neighbors whose link with the local node has been marked as belonging to node's current tree (i.e, status $=$ `Active`).

After updating its local estimate, the node will prepare a message to be disseminated through one hop broadcast. This message contains, for each known neighbor, a tuple containing the neighbor identifier and the locally computed aggregated value without the effects of the last contribution received from that neighbor (operator $\ominus$ in Alg. 1 line 26). This tuple is computed independently of the status of that neighbor. The message is then tagged with the local node identifier, and the information on the support tree that the node is currently attached to, including the tree identifier, the level of the node, the highest tree timestamp observed, and the identifier of the node's parent (Alg. 1 line 27).

Upon receiving one of these messages (Alg. 1 line 33), a node checks if there is a tuple in the message tagged with its own identifier. If so, then it uses the `updateNeighborEntry` procedure to either create or update the entry for that neighbor in its neighbor set. The information that is updated is the latest aggregate value received, the identifier of the tree to which the neighbor is currently attached, and its current tree level (the status of the node remains unchanged and is set to `Passive` if this was the first message received from that node).

Algorithm 1: MiRAge: Aggregate Function Computation

```
1: Local State:
2:    N_id //Node identifier
3:    Value //current input value
4:    Aggregation //Current result of aggregation
5:    Neighs //Set: (N_id, value, status, T_id, T_lvl)
6:    T_id //Unique identifier of the tree to which the
          node is currently attached (N_id of tree root)
7:    T_lvl //Level of the node in its current tree
8:    T_ts //Higher timestamp of current tree
9:    P_id //Identifier of current tree parent
10:   Trees //Map: Tress[T_id] → Timestamp

11: Upon Init ( ) do:
12:   Value ⟵ initValue() //initial input value
13:   T_id ⟵ N_id
14:   T_lvl ⟵ 0
15:   T_ts ⟵ now(); //now() = current time
16:   P_id ⟵ N_id
17:   Neighs ⟵ {}
18:   Aggregation ⟵ Value
19:   Setup Periodic Timer Beacon (ΔT)

20: Upon Beacon do: //every ΔT
21:   Call updateAggregation()
22:   if (T_id = N_id) then
23:      T_ts ⟵ now()
24:   msg ⟵ {}
25:   foreach (id, val, stat, tid, tlvl) ∈ Neighs do
26:      msg ⟵ msg ∪ (id, Aggregation ⊖ val)
27:   Trigger OneHopBCast (< N_id, T_id, T_lvl, T_ts, P_id, value, msg>)

28: Procedure updateAggregation()
29:   Aggregation ⟵ Value
30:   foreach (Neigh, V_neigh, Status, T_neigh, L_neigh) ∈ Neighs do
31:      If (Status = Active ) then
32:         Aggregation ⟵ Aggregation ⊕ V_neigh

33: Upon Receive ( < id, tid, tlvl, tts, pid, val, msg > ) do:
34:   if ∃(N_id, RecvVal) ∈ msg then
35:      Call updateNeighborEntry(id, tid, tlvl, RecvVal)
36:   Call updateTree( tid, tts, tlvl, pid, id, val )

37: Procedure updateNeighborEntry( id, tid, tlvl, val)
38:   if ( id ∉ Neighs ) then
39:      Neighs ⟵ Neighs ∪(id, val, Passive, tid, tlvl)
40:   else
41:      Neighs ⟵ Neighs \(id, _, stat, _, _) ∪ (id, val, stat, tid, tlvl)
```

C. Tree Management Mechanism

We now discuss how MiRAge manages the support spanning tree used by the aggregation strategy discussed previously. As noted before, the tree implicitly defines the data path used for performing and propagating the aggregation information. In MiRAge, there is no specialized sink node nor pre-configured root node. Instead, all nodes strive to become the root of a tree covering all nodes in the system. However, at each moment each node only belongs to a single tree. Interestingly, the management of the support spanning tree in MiRAge does not requires any additional exchange of information.

We rely on the level of nodes in a tree to establish a tree topology (avoiding cycles). The level of a node in a tree is defined as being the level of his parent plus one. The root of a tree has a level with a value of zero (and for convenience of notation, the parent of the root is defined as being the node itself). When the root of a tree fails, maintaining that tree becomes impossible, as electing a new root involves too much synchronization among nodes (and it would require nodes to have more than local knowledge about the tree topology impairing scalability). Instead, our tree stabilization mechanism allows nodes to switch trees (and/or parent) in some conditions.

Upon initialization (as denoted in Alg. 1 line 13), each node joins the tree rooted on itself. Whenever nodes exchange aggregation information, they also propagate information on their current tree and their position in the tree, in particular the identifier of their parent in the tree and their current level.

Whenever a node processes an aggregation message (as denoted in Alg. 1 line 33) it takes advantage of that information to run a local stabilization mechanism to manage the support tree. This is achieved through the procedure `updateTree` which is presented in Alg. 2.

In a nutshell, this procedure has the following goals: i) a node a switches tree if some node b belongs to a tree with lower identifier and b becomes parent of a; ii) if the parent node switches tree, then the node decides to either switch to that tree or to try to establish its own tree as the dominating tree; iii) if a cycle is detected the current tree is considered failed and abandoned; iv) if some node a considers b as his parent then b should perceive a as being `Active`; v) if node a is not parent nor child of b, b perceives a as `Passive`. Additionally, some invariant are also enforced, such as the level of a node being the level of the parent plus one and the parent node always being considered as `Active`.

A node decides to switch to another tree if it receives a message from some neighbor that belongs to a tree with a tree identifier that is lower than its current tree (Alg. 2 lines $28-34$). This could lead a node to switch to a tree whose root has failed. To avoid this, each node stores and propagates a timestamp associated with their current tree. This timestamp is only increased by the corresponding tree root (when it executes the periodic `Beacon` task). This acts as a form of heartbeat for the tree root. Nodes store the highest timestamp that they have observed for every tree (Alg. 2 lines $37-38$). This information can be garbage collected after enough time has passed without receiving any message from a neighbor belonging to that tree.

The use of these timestamps allows a node to only switch to a tree with a smaller identifier if this is the first time it becomes aware of that tree, or if the received message reports a timestamp that is higher than the last timestamp locally observed for that tree. When a node switches to another tree it adopts the node that sent information about that tree as its parent (by updating its local variable P_{id} and setting the state of that node to `Active` in its neighbor list).

Moreover, nodes perform another set of verifications and adaptations whenever they receive a message from a neighbor that enforces the correctness of the tree topology. Whenever a node receives a message from a neighbor for the first time, that neighbor is marked as not being connected to the tree, by setting its state to `Passive` in the receivers neighbor list (Alg. 2 lines $2-3$).

Algorithm 2: MiRAge: Tree Management

```
1: Procedure updateTree( tid, tts, tlvl, pid, id, val ) do:
2:    if ( id ∉ Neighs ) then
3:       Neighs ⟵ Neighs ∪(id, val, Passive, tid, tlvl)
4:    if ( T_id = tid ∧ tts > T_ts ) then
5:       T_ts ⟵ tts
6:    if ( N_id = pid ) then
7:       if ( P_id ≠ id ) then
8:          if ( T_id = tid ) then
9:             Call changeNeighborStatus ( id, Active )
10:      else //There is a loop in the tree
11:         T_id ⟵ N_id
12:         T_lvl ⟵ 0
13:         P_id ⟵ N_id
14:   else if ( P_id = id ) then
15:      if ( T_id = tid ∨ tid < N_id ) then
16:         Call changeNeighborStatus ( id, Active )
17:         T_id ⟵ tid
18:         T_lvl ⟵ tlvl + 1
19:         if ( T_ts < tts ) then
20:            T_ts ⟵ tts
21:      else
22:         T_id ⟵ N_id
23:         T_lvl ⟵ 0
24:         P_id ⟵ N_id
25:   else
26:      if ( T_id ≤ tid ) then
27:         Call changeNeighborStatus ( id, Passive )
28:      else //his tree is lower than mine
29:         if ( Trees[tid] = ⊥ ∨ tts > Trees[tid]) then
30:            Call changeNeighborStatus ( id, Active )
31:            P_id ⟵ id
32:            T_id ⟵ tid
33:            T_lvl ⟵ tlvl + 1
34:            T_ts ⟵ tts
35:         else
36:            Call changeNeighborStatus ( id, Passive )
37:   if ( tid ≠ N_id ∧ ( Trees[tid] = ⊥ ∨ Trees[tid] < tts ) ) do
38:      Trees[tid] ⟵ tts
39:   Call checkTreeTopology()

40: Procedure checkTreeTopology( ) do:
41:   foreach ( id, _, stat, tid, tlvl) ∈ Neighs do
42:      if ( stat = Passive ∧ tid = T_id ∧ tlvl < (T_lvl − 1)) then
43:         Call changeNeighborStatus ( P_id, Passive )
44:         P_id ⟵ id
45:         T_lvl ⟵ tlvl + 1
46:         Call changeNeighborStatus ( P_id, Active )

47: Upon NeighborDown (id) do:
48:   Neighs ⟵ Neighs \(id, _, _, _, _ )
49:   if (P_id = id) then
50:      P_id ⟵ N_id
51:      foreach ( id, _, stat, tid, tlvl) ∈ Neighs do
52:         if ( stat = Passive ∧tid = T_id ∧ tlvl < (T_lvl − 1)) then
53:            P_id ⟵ id
54:            T_lvl ⟵ tlvl + 1
55:      if (P_id = N_id) then
56:         T_id ⟵ N_id
57:         T_lvl ⟵ 0
58:         T_ts ⟵ now()
59:      else
60:         Call changeNeighborStatus ( P_id, Active )

61: Procedure changeNeighborStatus( id, stat ) do:
62:   Neighs ⟵ Neighs \ ( id, val, s, tid, tlvl )
63:   Neighs ⟵ Neighs ∪ ( id, val, stat, tid, tlvl )
```

Furthermore, if the node receives a message from a neighbor that considers itself as being his parent (Alg. 2 line 6) belonging to the same tree (Alg. 2 line 8) then the status of that neighbor is set to `Active`. If two nodes believe at some point to be the parent of each other, a cycle as been created, potentially due the root failure. In this case nodes switch to the trees rooted on themselves (Alg. 2 lines $10-13$).

When a node receives a message from the neighbor that it considers as his current parent in the current tree or in one with lower identifier (Alg. 2 line $14-15$), the node simply reinforces that node as being its parent, by marking it as `Active`, and updating its current tree level to the level of that node plus one (Alg. 2 lines $16-20$). If however, the parent belongs to a tree with an identifier higher than local node's identifier, then the previous tree's root has failed. In this case the local node decides to switch to the tree rooted on itself, and will try to establish this tree as the new support tree (Alg. 2 lines $21-24$).

If a node receives a message from a neighbor that belongs to a tree whose identifier is not lower than the tree to which that node belongs currently, it simply marks that neighbor as not being connected to the tree through itself, setting that neighbor status to `Passive` in his local neighbor list.

Finally, an optional optimization mechanism can be employed by a node (denoted by procedure `checkTreeTopology` in Alg. 2 lines $40-46$) which allows it to switch its parent to the neighbor in that tree with minimal level. In this case, the previous parent is marked as having a state of `Passive`, the new parent is marked, conversely, to have a state of `Active`, and the current tree level is updated to reflect the change.

Essential to the correction of this algorithm is the capacity of a node to detect whenever a node has failed. This is capture in Alg. 2 by the processing of the *neighbor down notification* that is triggered by the local unreliable failure detector (Alg. 2 line 47). In this case the information for the suspected node is removed from the neighbor set, and if the suspected neighbor was the local node's parent, it will try to locate a suitable replacement in its neighbor list. A suitable replacement is a neighbor that is connected to the same tree with a status of `Passive` and a level bellow the local node's own level (to avoid the accidental formation of cycles). If a suitable candidate is found, then the local state of the node is adjusted to reflect the new parent (Alg. 2 line $52-54$; 60), otherwise the node switches to the tree rooted on itself (Alg. 2 line $55-58$).

This algorithm ensures the convergence of the state in each node of the system to a configuration where a single tree covers all nodes, if the underlying ad hoc network is connected.

V. Evaluation

In this Section we present an extensive experimental evaluation of MiRAge comparing its performance against state-of-the art solutions for continuous aggregation.

We start by noting that, contrary to the largest body of the literature regarding wireless ad hoc protocols, our experimental evaluation does not resort to simulation. Instead we have implemented prototypes of both MiRAge and the relevant competing baselines using the C language. All protocols rely

on similar code bases, use the same fault detector mechanism, and execute in similar conditions. We believe this is an essential step to demonstrate the applicability of our solution.

In the following, we discuss in more detail our experimental methodology and present our experimental results composed of two different deployments that exercise these protocols in varied conditions. These include fault-free, input value changes, and multiple node failures scenarios.

A. Experimental Methodology

As discussed above, we have conducted our experimental evaluation by implementing prototypes of both our protocol and relevant baselines that represent different alternatives found in the literature. All protocols were implemented in an event driven way, having a dedicated thread that handles events. These events can either be timers (for executing periodic tasks), message reception, or notifications of failures from a failure detector. This implementation strategy minimizes problems that arise from concurrency within the scope of the protocol execution. All protocols used in our evaluation resort to the same unreliable failure detector, which operates as described previously in Section IV-A configured with $\Delta D = 1s$ and $K_{fd} = 10$. In practice this means that each node disseminates an announcement with its own identifier every second, and that a node a is suspected to have failed by node b when b is unable to receive an announcement from a for a period longer than 10 seconds.

The baselines employed in our experimental work are the **Flow-Updating** [23] which is a protocol that can compute the average function; a version of **LiMoSense** [22] published by the authors, where counters maintained by nodes are never garbage collected. LiMoSense is a representative of the well known Push-Sum protocol [21] that can compute the sum, count, and average functions, that is enriched to ensure fault tolerance; **GAP** [25] which is the protocol that mostly resembles our own solution, being able to compute any aggregation function but unfortunately, unable to tolerate the failure of its static tree root. Since GAP does not enable every node in the network to obtain the aggregated value, we also developed a simple variant of GAP, that we named **GAP+Bcast** where the root of the tree also piggybacks its aggregated value in all messages disseminated by it. This value is then propagated in piggyback along the tree used by GAP, enabling all nodes to learn the result of the aggregation. All protocols were configured to perform their periodic communication step every two seconds. In both GAP and GAP+Bcast experiments the root is fixed in experiments with faults, and randomly assigned in experiments with no faults.

All experiments reported here were conducted by executing each protocol in a fleet of 24 Raspberry Pi3 - Model B. All communication among nodes is performed through a wireless ad hoc network using one-hop broadcast.

While MiRAge can easily be employed to compute any arbitrarily aggregation function, in our experiments every protocol was configured to compute the average. The initial input values of nodes were fixed, being the numbers 1 to 24 attributed statically to each of the 24 Raspberry Pis. Hence, the average value computed based on the initial input values is 12.5. Each experiment reported here was executed three times. Protocols were rotated between these executions to amortize the effects of external and uncontrollable factors. Results show averages of results obtained across the multiple runs.

We have conducted experiments in two different settings:

Disperse Deployment: In this deployment we have positioned the nodes across multiple rooms in two hallways in our department building. Figure 1a illustrates the distribution of nodes in the space. Each of the hallways has approximately $30m$. This is a particularly challenging environment, as there are multiple factors that affect transmissions among nodes in a very dynamic fashion. This deployment attempts to illustrate a scenario where the radio signal strength is highly variable among devices, which we have observed that could trigger our fault detector multiple times.

Dense Deployment with Overlay: In this deployment we have positioned all nodes within a single room however, we have added to all tests prototypes a filter that restricts at each node the set of nodes from which it can receive messages. Effectively, this produces a logical network that defines (potential) neighboring relationships among nodes. This overlay is represented graphically in Figure 2. We note that in this setting transmissions by any device can still produce collisions. This setting tries to illustrate a scenario where there are multiple sources of interference that can produce a somewhat higher number of collisions in the wireless medium.

B. Experimental Results

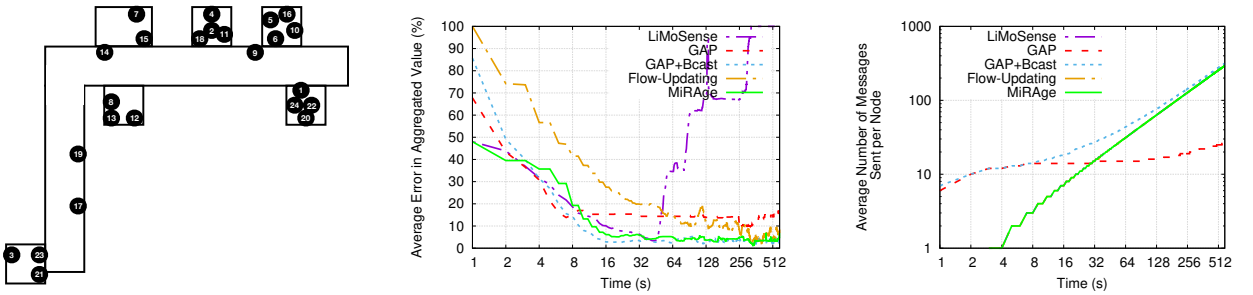
We now report our experimental results. In our experimental work we focus on the *Average Error in the Aggregated Value*, abriviated *AvgErr*, that illustrates how far on average are all nodes from the correct average value, being defined as:

$$AvgErr = \frac{\sum_{i=1}^{n}(|Avg_{real} - Avg_i|)}{n \times Avg_{real}} \times 100$$

where n represents the total number of nodes, Avg_{real} is the current average value considering all input values, and Avg_i represents the current average computed by node i. We present this normalized for the real average value. Intuitively, in a scenario where all nodes have computed the correct average, the *AvgErr* will be 0% which is the ideal scenario. On the other hand, when the real average value is 12.5 and the average computed value by all nodes is 25, the *AvgErr* will be 100%, where an average computed value of 50 would yield a *AvgErr* of 300%. Additionally, we also measure the total number of messages transmitted by all nodes in function of time. This is a measure of the overhead produced by each protocol.

1) Disperse Deployment: Figure 1 presents the schematics and results for our experiments in the disperse deployment. In this experiment we have deployed the nodes as represented in Figure 1a. Each experiment was conducted for a period of 10 minutes (600 seconds).

Figure 1b reports the measured *AvgErr* across all nodes as the experience progresses. Note that the x-axis is in logarithmic scale, as the main point of this plot is to show the



(a) Deplyment Configuration (b) Average Error in the Aggregated Value (c) Total number Messages sent by all nodes

Fig. 1. Disperse Deployment

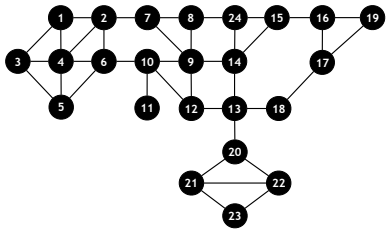


Fig. 2. Deplyment Configuration and Overlay Topology

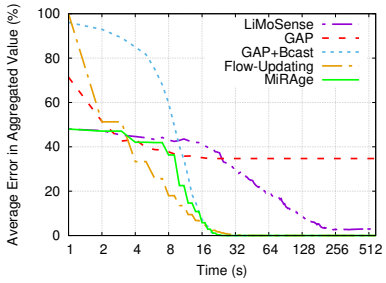


Fig. 3. Average Error in the Aggregated Value in Fault-free Scenario

convergence of nodes towards the correct average, a process that is more noticeable at the start of the experiment. Results show that Flow-Updating is the protocol that converges more slowly towards the correct average, having an *AvgErr* in the order of 10% even after more than one minute of execution.

GAP converges relatively fast towards an *AvgErr* close to 15% and stabilizes in this value. This is expected, as GAP was not designed to provide all nodes in the system with the aggregated value. GAP+Bcast and MiRAGE show the best performance, as they converge quite fast to an *AvgErr* of approximately 4%. Both algorithms are unable to compute the correct value due to the fact that the communication among many pairs of nodes is highly unstable, leading to a high level of message loss (we have confirmed this by inspecting logs). However, we note that, contrary to MiRAge, GAP+Bcast is not fault-tolerant, as the failure of the root would make it impossible for any node to compute the correct value.

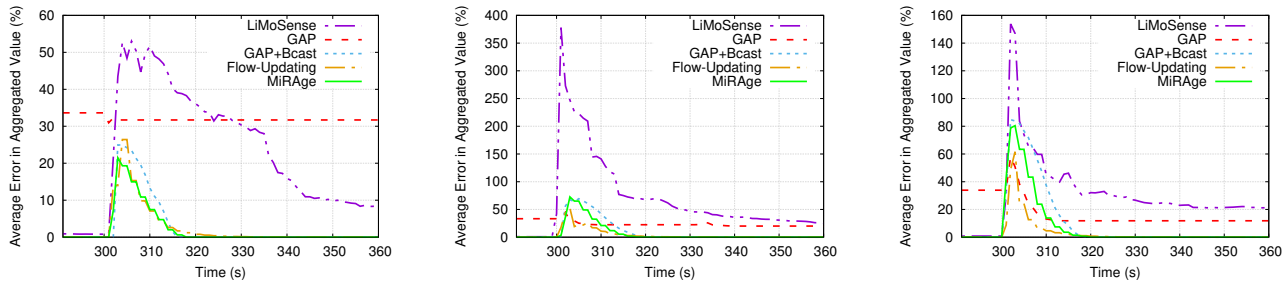
Finally, and interestingly, LiMoSense appears to converge to a similar value to those computed by MiRAge and GAP+Bcast, but at some point the *AvgErr* starts to increase without the protocol being able to regain an adequate *AvgErr*. We inspected this case carefully and discovered that this happens due to an aspect in the design of LiMoSense, that tries to compensate transferred values when a node failure is detected. Unfortunately, in this environment, and due to the instability of the wireless medium, nodes detect each other as failed in an asymmetric way. This leads one of the nodes to apply the compensation mechanism while the other does not. This produces an error that propagates towards the entire network, leading all nodes to compute an incorrect average of zero. While LiMoSense is indeed fault-tolerant, it was not designed to tolerate asymmetric communication links.

Figure 1c shows the total number of messages sent over time for each protocol. The results show that MiRAge, Flow-Updating, and LiMoSense have exactly the same cost. This is expected as all protocols were configured to exchange information at the same rate. GAP and GAP+Bcast issues more messages at the start of the experiment, as they send bootstrap messages to newly found nodes. GAP stops transmitting messages when values become stable, reducing GAP's communication overhead. However, this could lead to missed updates due to message loss, and as such, GAP+Bcast was modified to cope with this issue by always transmitting messages, converging to a slightly higher cost than the rest of the protocols.

In all experiments that we conducted, communication overhead always followed this pattern and hence, we omit those results from the following sections.

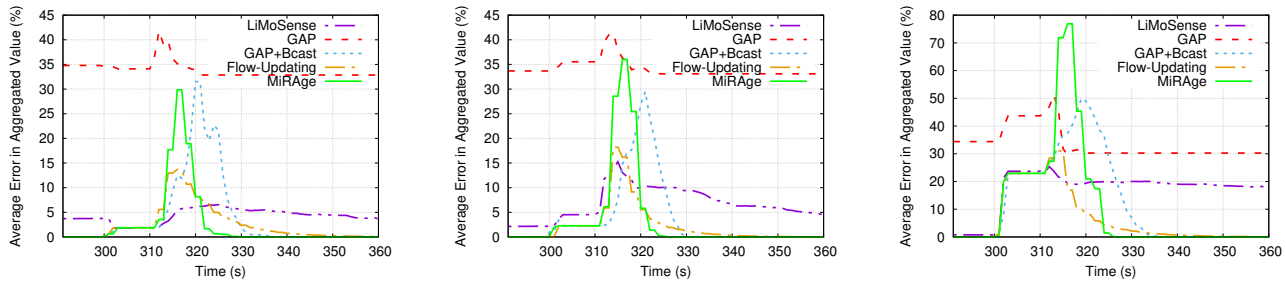
2) Dense Deployment with Overlay: In this setting we have conducted multiple experiences. We note that while collisions in the wireless medium are highly probable, asymmetric communication and fault detection is less likely. We start by examining the behavior of protocols in a fault free environment. Then we explore the effect of three dynamic aspects: i) input value change; ii) node failure; and iii) link failure. All experiments were again conducted for a period of 10 minutes. In experiments where we introduce dynamic aspects, these happen around 5 minutes (300 seconds) in the experiment.

a) Fault-Free Scenario: Figure 2 reports the overlay configuration employed in this scenario. Figure 3 presents the *AvgErr* in a fault free execution for all protocols.



(a) 1 node (b) 12 nodes (c) 24 nodes

Fig. 4. Average error in aggregated value with dynamic input values at different number of nodes



(a) 1 node (b) 6 nodes (c) 12 nodes

Fig. 5. Average error in aggregated value with variable number of node failures

The results show that in this setting Flow-Updating quickly starts to converge towards a perfect aggregate value. MiRAge converges somewhat slower but reaches an *AvgErr* of 0% slightly before. This happens due to the fact that MiRAge uses a deterministic tree topology to achieve convergence, whereas Flow-Updating relies on an iterative (averaging) technique that iteratively approximates towards the correct value. The results of GAP are consistent with those presented above, while GAP+Bcast converges towards the correct value across all nodes, albeit, slightly slowly than MiRAge. LiMoSense in this setting, where asymmetric communication is less likely, is able to converge to a good approximation of the value although, taking much more time.

b) Dynamic Input Values: In these experiments we have introduced variations on the input value of different amounts of nodes in the system. We have conducted experiments where we modify the input value of 1, 12, and 24 nodes concurrently. Results are summarized in Figure 4 and show consistent results for all experiments. LiMoSense is the protocol that is more susceptible to input value variations, whereas MiRAge, Flow-Updating, and GAP+Bcast all present somewhat similar results, being able to converge to the new aggregate result in less than 20 seconds. The reason why only these three protocols are able to cope in a timely fashion with the change of input values is nuanced. These protocols are the only whose computation of the aggregate result directly depends on the (original) input value. This implies that as soon as the input value changes, nodes start propagating aggregate information that reflects completely the change. Therefore, it suffices that messages propagate through the system to ensure that this effect reaches all nodes.

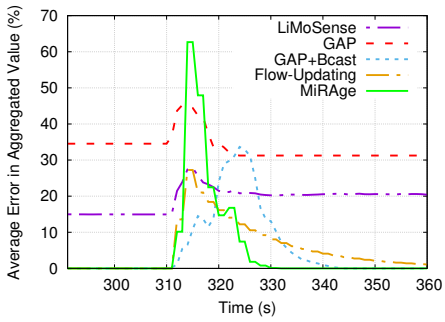


Fig. 6. Mean precision with 12 link fault

c) Node and Link Failures: In these experiments we introduce a variable number of node faults and measure the impact on the *AvgErr*. In the first experiment we introduced concurrently a number of node crashes that vary from 1, 6, and 12 nodes around 300 seconds in the experience. In these we have fixed the root of the trees used by GAP and GAP+Bcast to be node 1, and made sure that this node was not selected to become faulty (as these protocol would not tolerate that fault). Figure 5 depicts the results for these experiments. In the second experiment we introduce 12 concurrent link failures, where 12 pairs of nodes become permanently unable to communicate. This simulates the existence of obstacles or radio pollution in the environment. Figure 6 reports the obtained *AvgErr* in this scenario.

In both faulty scenarios (reported in Figures 5 and 6), all protocols suffer a significant increase in the *AvgErr* after the introduction of faults. With the exception of LiMoSense, all protocols can converge to the new aggregated value. LiMoSense takes significantly more time to converge because,

following the Push-Sum strategy, at each communication step it only exchanges information with a single neighbor. While MiRAge is the protocol that consistently shows a higher *AvgErr* immediately after faults, it is also the protocol that converges faster. This happens due to our mechanism to manage and repair the support tree in the presence of faults, which reconfigured the tree in an expedite way, during this period however, computed results are affected by the (temporary) inconsistency of the tree topology.

VI. Conclusion

In this paper we have studied different protocols for performing continuous aggregation in wireless ad hoc setting. Our work is motivated by the need to support emergent edge applications that delegate some computational components to commodity edge devices, that communicate through the wireless medium. In this context, it is relevant to have robust and efficient distributed aggregation protocols, both for supporting applications, and to build adequate monitoring schemes. Considering existing solutions, we propose a novel distributed continuous aggregation protocol, that we named MiRAge, that contrary to the existing state-of-the art is adequate to compute any arbitrary aggregation function, while achieving high precision in a manner that is fault tolerant to both node and link failures. Central to the design of MiRAge is a mechanism to maintain a support spanning tree without the need to have a pre-defined root or sink node, additional message exchange, or explicit coordination among nodes.

Experimental results that were obtained by running real implementations of our protocol and competing state-of-the art solutions, show that our protocol can achieve faster and precise convergence, while being more robust to faults without additional communication overhead.

As future work we plan to further improve continuous aggregation protocols by minimizing the instantaneous increase in the average error in the presence of dynamic conditions (either input value changes and node/link faults), and lower the communication overhead under stable conditions. We believe that this can be achieved by allowing protocols to infer additional information regarding the stability of their local neighborhood and adapt their behavior accordingly. Additionally, we are aware that any wireless ad hoc system is vulnerable to a multitude of attacks, as such we plan to address this issue in the future and incorporate security measures and countermeasures for denial of service attacks using multiple radio channels.

References

[1] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, 4 2010, pp. 27–33.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 10 2015.

[3] Cisco, "Cisco visual networking index: Global mobile data traffic forecast update," https://tinyurl.com/zzo6766, 2016.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 10 2016.

[5] R. Mehmood, J. Schlingensiepen, L. Akkermans, M. C. Gomes, J. Malasek, L. McCluskey, R. Meier, F. Nemtanu, A. Olaya, and M.-C. Niculescu, "Autonomic systems for personalised mobility services in smart cities," King Khalid University, et. al., Tech. Rep., 2014.

[6] W. Tärneberg, V. Chandrasekaran, and M. Humphrey, "Experiences creating a framework for smart traffic control using aws iot," in *Proc. of the 9th International Conference on Utility and Cloud Computing*, ser. UCC'16. New York, NY, USA: ACM, 2016, pp. 63–69.

[7] Y. Yan, N. H. Tran, and F. S. Bao, "Gossiping along the path: A direction-biased routing scheme for wireless ad hoc networks," in *2015 IEEE Global Communications Conference*, 12 2015, pp. 1–6.

[8] I. F. Akyildiz and X. Wang, "A survey on wireless mesh networks," *IEEE Communications Magazine*, vol. 43, no. 9, pp. S23–S30, 9 2005.

[9] P. Jesus, C. Baquero, and P. S. Almeida, "A survey of distributed data aggregation algorithms," *IEEE Communications Surveys Tutorials*, vol. 17, no. 1, pp. 381–404, 1 2015.

[10] S. Abshoff and F. Meyer auf der Heide, "Continuous aggregation in dynamic ad-hoc networks," in *Structural Information and Communication Complexity*, M. M. Halldórsson, Ed. Cham: Springer International Publishing, 2014, pp. 194–209.

[11] R. Van Renesse, "The importance of aggregation," in *Future Directions in Distributed Computing*. Springer, 2003, pp. 87–92.

[12] O. Kennedy, C. Koch, and A. Demers, "Dynamic approaches to in-network aggregation," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 1331–1334.

[13] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 131–146, Dec. 2002.

[14] S. Motegi, K. Yoshihara, and H. Horiuchi, "Dag based in-network aggregation for sensor network monitoring," in *International Symposium on Applications and the Internet (SAINT'06)*, 1 2006, pp. 8 pp.–299.

[15] J.-Y. Chen, G. Pandurangan, and D. Xu, "Robust computation of aggregates in wireless sensor networks: Distributed randomized algorithms and analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 9, pp. 987–1000, 9 2006.

[16] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh, "Peer counting and sampling in overlay networks: Random walk methods," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '06. Denver, Colorado, USA: ACM, 2006, pp. 123–132.

[17] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani, "Estimating aggregates on a peer-to-peer network," Stanford InfoLab, Technical Report 2003-24, 4 2003. [Online]. Available: http://ilpubs.stanford.edu:8090/586/

[18] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: new aggregation techniques for sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 2004, pp. 239–249.

[19] M. Haridasan and R. van Renesse, "Gossip-based distribution estimation in peer-to-peer networks," in *Proceedings of the 7th International Conference on Peer-to-peer Systems*, ser. IPTPS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 13–13.

[20] C. Baquero, P. S. Almeida, R. Menezes, and P. Jesus, "Extrema propagation: Fast distributed estimation of sums and network sizes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 4, pp. 668–675, 4 2012.

[21] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, 10 2003, pp. 482–491.

[22] I. Eyal, I. Keidar, and R. Rom, "Limosense: live monitoring in dynamic sensor networks," *Distributed computing*, vol. 27, no. 5, pp. 313–328, 2014.

[23] P. Jesus, C. Baquero, and P. S. Almeida, "Fault-tolerant aggregation by flow updating," in *Distributed Applications and Interoperable Systems*, T. Senivongse and R. Oliveira, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–86.

[24] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148.

[25] M. Dam and R. Stadler, "A generic protocol for network state aggregation," *self*, vol. 3, p. 411, 2005.

Towards Enabling Novel Edge-Enabled Applications*

João Leitão, Pedro Ákos Costa, Maria Cecília Gomes, and Nuno Preguiça
jc.leitao@fct.unl.pt, pah.costa@campus.fct.unl.pt, {mcg, nuno.preguica}@fct.unl.pt
NOVA LINCS & DI-FCT-UNL

Abstract

Edge computing has emerged as a distributed computing paradigm to overcome practical scalability limits of cloud computing. The main principle of edge computing is to leverage on computational resources outside of the cloud for performing computations closer to data sources, avoiding unnecessary data transfers to the cloud and enabling faster responses for clients.

While this paradigm has been successfully employed to improve response times in some contexts, mostly by having clients perform pre-processing and/or filtering of data, or by leveraging on distributed caching infrastructures, we argue that the combination of edge and cloud computing has the potential to enable novel applications. However, to do so, some significant research challenges have to be tackled by the computer science community. In this paper, we discuss different edge resources and their potential use, motivated by envisioned use cases. We then discuss concrete research challenges that once overcome, will allow to realize our vision. We also discuss potential benefits than can be obtained by exploiting the hybrid cloud/edge paradigm.

1 Introduction

Since its inception in 2005, cloud computing has deeply impacted how distributed applications are designed, implemented, and deployed. Cloud computing offers the illusion of infinite resources available in data centers, whose usage can be elastically adapted to meet the needs of applications. Furthermore, data centers in different geographical locations enable application operators to provide better quality of service for large numbers of users scattered around the world by leveraging on geo-distribution and geo-replication.

Cloud computing however, is not a panacea for building reliable, available, and efficient distributed systems. In particular, the increasing popularity of Internet of Things (IoT) and Internet of Everything (IoE) applications, combined with an increase in mobile and user-centric applications, has lead to a significant increase in the quantity of data being produced by application clients. Although cloud computing infrastructures are highly scalable, the time required to process such large amounts of data is becoming prohibitively high. Additionally, the network capacity between clients and data centers is now becoming a significant bottleneck for such applications, namely to timely push data and fetch computation results to, and from the cloud.

Due to this, moving computations towards the edge of systems (i.e, closer to the clients that effectively process and consume data) has become an essential endeavor to sustain the growth of such applications. This led to the emergence of *edge computing*. Edge computing can be defined, in very broad terms, as performing computations outside the boundaries of data centers [43]. Many approaches have already leveraged on some form of edge computing to improve the latency perceived by end-users, such as CDNs [54], or tapping into resources of client devices [41, 52], among others.

This has motivated the emergence of proposals for taking advantage of edge computing. In particular Cisco has proposed the model of Fog Computing [5] which aims at improving the overall performance of IoT applications by collocating servers (and network equipment with computing capacity) with sensors that generate large amounts of data. These (Fog) servers can then pre-process data enabling timely reaction to variations on the sensed data, and filter the relevant information that is propagated towards cloud infrastructures for further processing. Mist computing, is an evolution of the Fog computing model, that has been adopted by industry [7] and that, in its essence, proposed to push computation towards sensors in IoT applications, which enables sensors themselves to perform data filtering computations, alleviating the load imposed on Fog and Cloud servers. While these novel architectures exploit the potential of edge computing, they do so in a limited way, requiring specialized hardware and not taking a significant advantage of computational devices that already exist in the edge. Furthermore, and as noted

*The work presented here was partially supported by the Lightkone European H2020 Project (under grant number 732505) and NOVA LINCS (through the FC&T grant UID/CEC/04516/2013).

for instance in [5] and [7] all of these proposed architectures are highly biased towards IoT applications.

In this paper, we argue that edge computing also offers the opportunity to build new *edge-enabled* applications, whose use of edge resources go beyond what has been done in the past, and in particular beyond proposals such as Fog and Mist computing [5, 7]. Previous authors have already presented their visions for the future of Edge computing [43, 48], Fog computing [47, 35, 38], and IoT specific edge challenges [31]. These works however, present their visions with an emphasis on IoT applications. An exception to this is related with Mobile edge computing [34] which devotes itself to the close cooperation of mobile devices to offload pressure from the cloud. Contrary to these, we take a different approach on edge computing and envision a future where general user-centric applications are supported by a myriad of different and already existing edge resources. In particular, we believe that edge computing will enable the creation of significantly more complex distributed applications, both in terms of their capacity to handle client request and processing data, and also in terms of the number of components. Our vision, is that this will empower the design of user-centric applications that promote additional interactivity among users and between users and their (intelligent) environment.

To realize this vision however, it is relevant to fully understand what are the computational and network resources available for edge-enabled applications, their characteristics, and how they can be used (§2). We further materialize our vision on the potential of edge computing by presenting two envisioned case studies (§3). Using large numbers of heterogeneous edge resources to build novel edge-enabled applications is not trivial, and key research challenges must be addressed by the computer systems community, we further relate these with previous research (§4)[1]. Finally, we conclude this paper by summarizing the main research challenges that are in the way to fully realizing the potential of edge computing (§5).

2 Overview of the Edge

To fully realize the potential of edge computing, one should identify which computational resources lie beyond the cloud boundary, and what are the limitations of such devices and their potential benefits for edge-enabled applications. Figure 1 provides a visual representation of the different edge resources that we envision. We represent these edge resources as being organized in different levels starting with level zero that represents cloud data centers. Edge-enabled applications are not however, required to make use of resources in all edge levels.

[1]We recognize that fully tapping into the potential of edge computing requires concerted research efforts from many fields in computer science, however in this paper we focus only on a computer systems perspective.

To better *characterize* the different levels in the edge resource spectrum, we consider three main dimensions: i) **capacity**, which refers to the processing power, storage capacity, and connectivity of the device; ii) **availability**, which refers to the probability of the resource to be reachable (due to being continuously active or faults); and iii) **domain**, which captures if the device supports the operation of an edge-enabled application as a whole (*application domain*) or just the activities of a given user[2] within an edge-enabled application (*user domain*).

We further classify the *potential uses* of the different edge resources considering two main dimensions: i) **storage**, which refers to the ability of an edge resource to store and serve application data. Devices that can provide storage can do so by either storing *full application state*, *partial application state*, or by providing *caching*. The first two enable state to be modified by that resource, and the later only enables reading (of potentially stale) data; and ii) **computation**, which refers to the ability of performing data processing. Here, we consider three different classes of data processing, from the more general to the more restrictive: *generic computations*, *aggregation and summarization*, and *data filtering*.

We start by observing that as we move farther from the cloud (i.e, to higher edge levels), the capacity and availability of each individual resource tends to decrease, while the number of devices increases. We now discuss each of these edge resources in more detail. We further note that resources could be presented with different granularity however, in this paper we focus on a presentation that allows to distinguish computational resources in terms of their properties and potential uses within the scope of future edge-enabled applications.

E0: Cloud Data Centers Cloud data centers offer pools of computational and storage resources that can be dynamically scaled to support the operation of edge-enabled applications. The existence of geo-distributed locations can be used as a first edge computing level, by enabling data and computations to be performed at the data center closest to the client. These resources have *high capacity and availability* and operate at the *application domain*. They offer the possibility for storing *full application state* and perform *generic computations*.

E1: ISP Servers & Private Data Centers This edge resource represents regional private data centers and dedicated servers located in Internet Service Providers (ISPs) or exchange points that can operate over data produced by users in a particular area. These servers operate at the *application domain*, presenting *large capacity* and *high availability*. They offer the possibility to store *(large) partial application state* and perform *generic computations*.

E2: 5G Towers The new advances in mobile networks

[2]We refer to user in broad terms, meaning an entity that uses an edge-enabled application, either an end-user or a company.

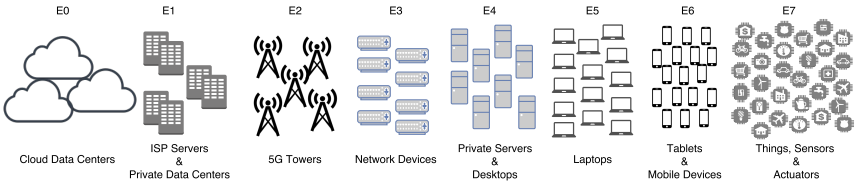


Figure 1: Edge Components Spectrum

		E0 Cloud DCs	E1 ISP Servers & Priv. DCs	E2 5G Towers	E3 Network Devices	E4 Priv. Servers & Desktops	E5 Laptops	E6 Tablets & Mobiles	E7 Things
Characterization	**Capacity**	High	Large	Medium	Low	Medium	Medium	Low	Varied
	Availability	High	High	High	High	Medium	Low	Low	Limited
	Domain	Application	Application	Application	Application	User	User	User	User
Potential uses	**Storage**	Full State	(Large) Partial	(Limited) Partial	None/Caching	(User) Partial	Caching	(User) Caching	(Local) Caching
	Computation	Generic	Generic	Generic	Filtering	Generic	Aggregation	Aggregation	Filtering

Table 1: Edge Devices Per Level Characteristics

will introduce processing and storage power in towers that serve as access points for mobile devices (and tablets) as well as improved connectivity. While we can expect these edge resources to have *medium capacity*, they should have *high availability* operating at the *application domain*. These computational resources can execute *generic computations* over stored *limited partial application state* enabling further interactions among clients (e.g, mobile devices) in close vicinity.

E3: Network Devices Network devices (such as routers, switches, and access points) that have processing power capabilities, offer *low capacity* and *high availability*. From the storage perspective, these offer either none or *caching* capacity. Devices in close vicinity of the user will operate at the *user domain* while equipment closer to servers might operate at the *application domain*. These devices will mostly enable in-network processing for edge-enabled applications in the form of *data filtering* activities over data produced by client devices being shipped towards the center of the system.

E4: Private Servers & Desktops This is the first layer (and more powerful in terms of capacity) of devices operating exclusively in the *user domain*. Private servers and desktop computers can easily operate as logical gateways to support the interaction and perform computations over data produced by levels E5-E7. While individual edge resources have *medium capacity* and *medium availability* they can easily perform more sophisticated computing tasks if the resources of multiple devices are combined together. These edge resources are expected to store *(user-specific) partial application state* while enabling *generic computations* to be performed. Private servers in this context are equivalent to *in-premises servers* frequently referred as part of Fog computing architectures [5].

E5: Laptops User laptops are similar to resources in the E4 level albeit, with *low availability*. Low availability in this context is mostly related with the fact that the uptime of laptops can be low due to the user moving from location to location. Due to this, we expect these devices to be used for performing *aggregation and summarization* computations and eventually provide *(user-specific) caching* of data for components running farther from the cloud. Laptops might act as application interaction portals, enabling users to use such devices to directly interact with edge-enabled applications.

E6: Tablets & Mobile Devices Tablets and Mobile devices are nowadays preferred interaction portals, enabling users to access and interact with applications. We expect this trend to become dominant for new edge-enabled applications since users expect continuous and ubiquitous access to applications. These devices have *low capacity* and *low availability*, the latter is mostly justified by the fact that the battery life of these devices will shorten significantly if the device is used to perform continuous computations. These devices however, can be used as logical gateways for devices in the E7 level in the *user domain* context. They can provide *user-specific caching* storage and perform either *aggregation and summarization* or *data filtering* for data produced by E7 devices in the context of a particular user.

E7: Things, Sensors, & Actuators These are the most limited devices in our edge resource spectrum. These devices will act in edge-enabled applications mostly as data producers and consumers. They have *extremely limited capacity* and *varied availability* (in some cases low due to limited power and weak connectivity). They operate in the *user domain*, and can only provide extremely limited forms of *caching* for edge-enabled applications. Due to their limited processing power they are restricted to perform *data filtering* computations. Devices in the E7 layer with computational capacity are the basis for Mist computing architectures [7].

Table 1 summarizes the different characteristics and potential uses of edge resources at each of the considered levels. We expect application data to flow along the edge resource spectrum although, different data might be processed differently at each level (or skip some entirely).

3 Envisioned Case Studies

We now briefly discuss two envisioned case studies of novel edge-enabled applications, and argue how edge resources in different levels of the edge spectrum can be leveraged to enable or improve these case studies.

Mobile Interactive Multiplayer Game Consider an augmented reality mobile game that allows players to use their mobile devices to interact with augmented reality objects and non-playing characters similar to the popular Pókemon Go game [3]. Such game could enable direct interactions among players, (e.g, to trade game objects or fight against each other) and allow players to interact in-game with (local) third party businesses that have agreements with the company operating the game (e.g, a coffee shop that offers in-game objects to people passing by their physical location).

Pókemon Go does not allow these interactions, with some evidences [6] pointing to one of the main reasons being the inability of cloud-based servers to support such interactions in a timely manner due to large volumes of traffic produced by the application. However, edge computing offers the possibility to enable such interactions, by leveraging on edge resources on some of the levels discusses above. Considering that the game is accessed primarily through mobile phones, one could resort to computational and storage capabilities of *5G Towers (E2)* to mediate direct interactions (e.g, fights) between players. One could also leverage on regional *ISP and Private Data centers (E1)* to manage high throughput of write operations (and inter-player transactions) to enable trading objects. Some trades could actually be achieved by having transaction executed directly between the *Tablets & Mobile Devices (E6)* of players and synchronizing operations towards the *Cloud (E0)* later. Special game features provided by third party businesses could be supported by *Private servers (E4)* being accessed through local networks (supported by *Network Devices (E3)*) located on business premises.

Intelligent Health Care Services Consider an integrated and intelligent medical service that inter-connects patients, physicians (in hospitals and treatment centers), and emergency response services [4], that can leverage on wearable devices (e.g, smart watches or medical sensors), among other IoT devices (e.g, smart pills dispensers), to provide better health care including, handling medical emergencies, and tracking health information in the scope of a city, region, or country.

These systems are not a reality today due to, in our opinion, two main factors. The first is the large amounts of data produced by a large number of health monitors, the second is related with privacy issues regarding the medical data of individual patients. Edge computing and the clever usage of different edge resources located in different levels (as discussed previously) can assist in realizing such application. In particular, *Wearables and medical sensors (E7)* can cooperate among them and interact with users' *Mobile Devices (E6)* and *Laptops (E5)*, which can archive and perform simple analysis over gathered data. The analysis of data in these levels could trigger alerts, to notify the user to take medicine, to report unexpected indicators, or to contact emergency medical services if needed. This data could be further encrypted and uploaded to *Private Servers (E4)* of hospitals, so that physicians could follow their patients' conditions. Additionally, health indicators aggregates could be anonymously uploaded to *Private Data Centers (E1)* for further processing, enabling monitoring at the level of cities, regions, or countries to identify pandemics or to co-related them with environmental aspects.

4 Research Challenges

The presented case studies (§3) rely on the use of multiple edge layers as discussed previously (§2). Other novel edge-enabled applications will have similar requirements. Some of the most challenging aspects of the edge is its high heterogeneity in terms of capacity, and that one has to deal with the increasing number of devices. We identify the following main challenges to fully realize the potential of edge computing, which we also discuss in relation to our envisioned case studies.

Resource management: Resource management solutions are crucial to keep track and manage computational resources across multiple edge levels. Solutions must provide efficient mechanisms to allow the dynamic creation and decommission of application components across multiples resources, allowing these components to interact efficiently. Considering the use case of a *mobile interactive multiplayer game* this translates in two complementary aspects. The first is to track computational resources to enable executing components of the game in cloud platforms, ISP and private data centers, or private servers located in Coffee shops. The second is related with mobile devices in close vicinity finding each other and interacting, either directly or through 5G towers.

Large-scale decentralized resource tracking and management has been previously addressed in the context of decentralized peer-to-peer systems [41]. Overlay networks [26] have been used to enable the tracking and communication among large numbers of resources [28, 27, 51] and also to enable efficient application-level routing [44, 42] and assignment of responsibilities in a decentralized way [22]. Solutions such as Mesosphere [3] and Yarn [49], that are specially tailored for cloud and clus-

[3] `https://www.pokemongo.com/`

[4] A significative evolution of the Denmark Medical System briefly described in [45].

ter environments, can be employed for resource management in edge levels closer to the cloud, potentially complemented with other solutions in different levels. However, most of these solutions assume resources to be homogeneous in terms of capacity and connectivity which makes them unsuitable for edge environments. Since edge resources are located in different levels, hierarchical management of resources could be an interesting approach, feasible at the overlay management protocols level.

Enable computations to move: Edge-enabled applications require computations to be executed across heterogeneous edge resources located in different levels. Computations cannot be executed in arbitrary edge resources (i.e, any edge level), and shipping computations across different edge levels must cope with heterogeneous execution environments (e.g, virtualization, containers, middleware, different operating systems, etc). It is therefore relevant to develop solutions that enable the migration of (generic) computational tasks among different (compatible) edge levels, and also to allow computations to be decomposed into simpler computational tasks, and symmetrically, be recomposed as single processing units dynamically. For instance in the *intelligent heath care services* examples this is relevant to allow computations that perform initial analysis of data gathered by wearable sensors to migrate between patients mobile devices and laptops according to their availability and available battery.

This is not a new proposal, in fact, research from the software systems community on osmotic computing already proposed the migration of microservices that encode fractions of the computational logic of applications to the edge and back to cloud infrastructures according to evolving workloads [50]. Mobile agents [14] and mobile code solutions [20] are proposals that allow having arbitrary code move and execute along a (logical) network of components. However, none of these approaches deal with resource heterogeneity nor the decomposition of computations in simpler tasks (and their coordination). Amazon [1] and Google [2] have enriched their cloud infrastructure to pre-process and redirect HTTP requests to data centers closer to clients. Yet, these only operate at the E0 edge level and extending them towards the edge is an open challenge.

Dynamic and partial state replication: Edge-enabled applications will naturally need to perform computations over application data; however, as the computations can be scattered through multiple edge levels, and to avoid communication overheads with the core of network, application state should be able to move towards the edge. This brings additional challenges, as dynamically spawning replicas of data storage systems supporting edge-enabled applications will inevitably lead to an increase in the overhead produced by replication protocols, requiring also the automatic decommission of (unuseful) replicas. Furthermore, not all edge resources can hold the same amount of application data, which motivates the need for effective partial replication solutions, and replication protocols that, potentially, provide different consistency guarantees at different edge levels. Consider the use case of the *mobile interactive multiplayer game*, in this context 5G towers can provide better quality of service for direct interactions among users if they replicate a fraction of the application state for users in its vicinity. To do this however, it is essential to be able to freely spawn and remove (to minimize operational costs) partial replicas of the application state.

There have been many recent works exploring geo-replication, where replicas are dispersed in remote locations. Some focus on offering strong consistency guarantees [37] while others focus on providing causal+ consistency as to ensure availability [32, 33, 8, 53]. However, few solutions exploit the use of partial replication. The ones that do so, such as Saturn [13] and Kronos [16] are highlly limited in terms of scalability and hence will not be able to cope with large number of replicas managed dynamically. There has also been few works exploring the combination of multiple consistency models. Gemini [30] and Indigo [10] do so per operation type over the data store instead of per replica. Although, none of these approaches entirely address the data management requirements for future edge-enabled applications.

Lightweight and decentralized monitoring: Dynamically migrating computations and storage components along the edge spectrum requires knowledge about available edge resources. This knowledge can be attained by distributed monitoring systems that obtain information regarding applications' operation and the load of edge resources, which can then be employed to perform adequate management decisions. Efficient dissemination of monitoring information will require in-network processing for filter and aggregate data. Again picking up the example of moving the preliminary analysis of sensors data between the mobile devices and laptops of patients in the *intelligent health care services* use case, to decide the best device to conduct computations requires having (somewhat) up-to-date information regarding the status of each device, namely if the device is active, if it has available CPU and RAM to perform the computations efficiently, and its current battery level (or if the device is currently connected to a power source).

There are several previous works that focus on decentralized monitoring of large-scale platforms, typically on cloud and grid infrastructures [9, 46, 29]. Many of these solutions resort to gossip-based dissemination protocols [27, 12] to propagate relevant information towards special sink nodes. While these solutions offer an interesting departing point for devising new monitoring schemes, they do not consider the monitoring of heterogeneous re-

sources organized in hierarchies as captured in our edge resource spectrum. In addition, monitoring tasks and in-network processing, could change the grain of execution dynamically as to adjust to variations in the workload and available resources in the system.

Enable systems to become autonomic: The management of the life cycle and interactions of large numbers of edge components operating at resources scattered throughout multiple edge levels will be unfeasible by hand, particularly when considering the need to timely adapt the operation of the system in reaction to unpredictable failures of components or sudden surges in access patterns (i.e, peak loads). A possible way to overcome this challenge is to design novel mechanisms to enrich edge-enable applications to have autonomic management capabilities. To achieve this, we require the capacity to enable moving, decomposing, and/or replicating computations, novel dynamic partial replication schemes, and lightweight distributed monitoring. Considering the use of 5G towers in the *mobile interactive multiplayer game*, spawning application computational components and partial replicas of the game state is manually unfeasible in a timely way considering the potentially large number of available 5G towers. Such process requires autonomic control that, based on monitoring information, can automatically trigger application management mechanisms.

Autonomic systems have been proposed and studied since 2001 [4] and have been applied to multiple types of systems from web services [19], to management of grid resources and others [24]. Typically autonomic systems are designed around the $MAPE$-K architectural design, where the system has Monitoring, Analysis, Planning, and Execution components that are interconnected by a common Knowledge base. The main challenge in making edge-enabled applications autonomic is due to the typical central nature of the Planning (and potentially Analysis and Execution) components. Large-scale edge-enabled applications combining multiple components scattered across several edge levels will require decentralized planning schemes, capable of operating (and executing) reconfigurations of the system with incomplete and partial knowledge. While some previous works have explored this [25], they are still far from meeting the needs of future edge-enabled applications.

Security & Data protection: Future edge-enabled applications will manipulate sensitive user data. Doing so can compromise users' privacy. Furthermore, executing computations and storing data in hardware controlled by individual users can also compromise data integrity. To overcome these challenges new data protection schemes have to be developed. Considering the use case of the *intelligent health care services* where medical data would be stored in user devices and in private servers in hospitals. This data is highly sensitive as it can easily compromise the privacy of patients, while compromising its integrity could have serious medical implications, for instance, if a patient receives the incorrect treatment.

Homomorphic [21] and partially homomorphic [40, 15] encryption schemes allow computations to be performed in the encrypted domain. However, these solutions have high computational cost and cipher-text expansion therefore, being unsuitable to support edge computations. A more promising alternative is to use schemes based on (efficient) symmetric cryptography, enabling some operations (such as indexing and search) in the encrypted domain. This has been demonstrated for a few data formats [18, 17]. Nevertheless, these schemes do not provide guarantees of data integrity. Protecting data integrity can rely on trusted hardware [36] (e.g,IntelSGX [11]) that provide attestation and verification mechanisms for outsourced computations. Unfortunately, limited resources and complex key management/distribution schemes, makes the use of trusted hardware an open challenge. The blockchain design [39, 23] offers an interesting approach to make edge-enabled applications more robust, but still lack adequate scalability and efficiency.

5 Conclusion

In this paper we provide our own perspective on the future of edge-computing. We categorize edge-enabled applications as applications that can resort to a myriad of computational resources outside of cloud environments, leveraging on their different properties for distinct purposes. We argue that this vision will enable the emergence of a novel class of edge-enabled applications. To motivate this, we presented two envisioned case studies.

While we note that the future of edge computing requires efforts from all computer science fields, from a systems perspective, we identified a set of key research challenges and related them with previous contributions. These include: i) decentralized and scalable resource management schemes; ii) enable the migration, replication, and decomposition of computational tasks along edge resources; iii) develop dynamic and scalable replication schemes, leveraging on partial replication and capable of extending towards the edge; iv) develop lightweight and scalable monitoring schemes; v) enable systems to become autonomic and self-adapt to dynamic resource availability and workloads; and finally, vi) develop novel cryptographic and computational schemes for data privacy and integrity.

References

[1] Aws lambda@edge. https://docs.aws.amazon.com/lambda/latest/dg/

lambda-edge.html. Accessed: 2018-04-24.

[2] Google cloud edge architecture. https://peering.google.com/#/infrastructure. Accessed: 2018-04-24.

[3] Mesosphere. https://mesosphere.com/product/. Accessed: 2018-04-24.

[4] An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[5] Cisco. Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf, 2015. Accessed: 2018-05-16.

[6] Hu Yan (Huawei iLab). Research Report on Pokémon Go's Requirements for Mobile Bearer Networks. http://www.huawei.com/~/media/CORPORATE/PDF/ilab/05-en, 2016. Accessed: 2018-05-16.

[7] Raka Mahesa (IBM). FHow cloud, fog, and mist computing can work together. https://developer.ibm.com/dwblog/2018/cloud-fog-mist-edge-computing-iot/, 2018. Accessed: 2018-05-16.

[8] S. Almeida, J. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[9] L. Baduel and S. Matsuoka. A decentralized, scalable, and autonomous grid monitoring system. In E. Tovar, P. Tsigas, and H. Fouchal, editors, *Principles of Distributed Systems*, pages 1–15, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[10] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[11] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. Foundations of hardware-based attested computation and application to sgx. In *EuroS&P*, pages 245–260. IEEE, 2016.

[12] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[13] M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, New York, NY, USA, 2017. ACM.

[14] R. S. Chowhan and R. Purohit. Study of mobile agent server architectures for homogeneous and heterogeneous distributed systems. *International Journal of Computer Applications*, 156(4):32–37, Dec 2016.

[15] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, Jul 1985.

[16] R. Escriva, A. Dubey, B. Wong, and E. G. Sirer. Kronos: The design and implementation of an event ordering service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 3:1–3:14, New York, NY, USA, 2014. ACM.

[17] B. Ferreira, J. Leitão, and H. Domingos. Multimodal indexable encryption for mobile cloud-based applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 213–224, June 2017.

[18] B. Ferreira, J. Rodrigues, J. Leitao, and H. Domingos. Practical privacy-preserving content-based retrieval in cloud image repositories. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2017.

[19] J. Ferreira, J. Leitão, and L. Rodrigues. A-osgi: A framework to support the construction of autonomic osgi-based applications. In *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems*, page (to appear), Limassol, Cyprus, Sept. 2009.

[20] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[21] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 850–867, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[22] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In M. F. Kaashoek and I. Stoica, editors, *Peer-to-Peer Systems II*, pages 160–169,

Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[23] A. Hari and T. V. Lakshman. The internet blockchain: A distributed, tamper-resistant transaction framework for the internet. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 204–210, New York, NY, USA, 2016. ACM.

[24] M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, Aug. 2008.

[25] J. O. Kephart. Engineering decentralized autonomic computing systems. In *Proceedings of the Second International Workshop on Self-organizing Architectures*, SOAR '10, pages 1–2, New York, NY, USA, 2010. ACM.

[26] J. Leitão. *Topology Management for Unstructured Overlay Networks*. PhD thesis, Technical University of Lisbon, Sept. 2012.

[27] J. Leitao, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 301–310, Oct 2007.

[28] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429, June 2007.

[29] J. Leitão, L. Rosa, and L. Rodrigues. Large-scale peer-to-peer autonomic monitoring. In *GLOBECOM Workshops*, pages 1–5. IEEE, November 2008.

[30] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[31] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal*, 4(5):1125–1142, Oct 2017.

[32] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[33] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

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

[35] R. Mahmud, R. Kotagiri, and R. Buyya. *Fog Computing: A Taxonomy, Survey and Future Directions*, pages 103–130. Springer Singapore, Singapore, 2018.

[36] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[37] H. Moniz, J. Leitão, R. J. Dias, J. Gehrke, N. Preguiça, and R. Rodrigues. Blotter: Low latency transactions for geo-replicated storage. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 263–272, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[38] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos. A comprehensive survey on fog computing: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 20(1):416–464, Firstquarter 2018.

[39] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[40] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[41] R. Rodrigues and P. Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, Oct. 2010.

[42] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In R. Guerraoui, editor,

Middleware 2001, pages 329–350, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[43] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[44] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, Aug. 2001.

[45] G. Tomás, P. Zeller, V. Balegas, D. Akkoorath, A. Bieniusa, J. a. Leitão, and N. Preguiça. Fmke: A real-world benchmark for key-value data stores. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '17, pages 7:1–7:4, New York, NY, USA, 2017. ACM.

[46] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, May 2003.

[47] L. M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, Oct. 2014.

[48] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26, Nov 2016.

[49] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

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

[51] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, Jun 2005.

[52] L. Vu, I. Gupta, K. Nahrstedt, and J. Liang. Understanding overlay characteristics of a large-scale peer-to-peer iptv system. *ACM Trans. Multimedia Comput. Commun. Appl.*, 6(4):31:1–31:24, Nov. 2010.

[53] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, New York, NY, USA, 2015. ACM.

[54] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec. Peer-assisted content distribution in akamai netsession. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 31–42, New York, NY, USA, 2013. ACM.

A Case for Autonomic Microservices for Hybrid Cloud/Edge Applications*

João Leitão, Maria Cecília Gomes, Nuno Preguiça, Pedro Ákos Costa,
Vitor Duarte, David Mealha, André Carrusca, André Lameirinhas
NOVA LINCS & DI-FCT-UNL

Submission Type: Vision

Abstract

Edge computing has emerged with the promise of enabling applications to leverage resources beyond the boundaries of the cloud data center. This paradigm enables applications to tap into computational and storage resources in the vicinity of end users, significantly reducing the network traffic's pressure towards cloud infrastructures, providing users with better quality of service namely, in terms of availability and latency.

To explore these edge resources, programmers currently have to develop, besides the services running in cloud infrastructures, extra software to run at the edge servers and connect both. We envision a different approach, where the execution of applications designed using the increasingly popular microservice design pattern can adapt to users' demands, with microservices moving to or being partially replicated at multiple locations, to provide better quality of service for users. Realizing this vision requires novel mechanisms to monitor, move, and replicate application data and microservices' logic, interchangeably between the resources in the cloud and the edge of the system.

1 Introduction

Cloud computing [2, 21, 24] is an established affordable solution for ubiquitous access that eases the deployment and scaling of applications. It supports the dynamic provisioning of almost unlimited resources (i.e, storage, computation, communication) offering diverse tools/execution environments in the form of virtualization mechanisms and support services. The increase in the number of mobile devices with enhanced computation and communication capabilities accelerated the need to deploy globally accessible applications. The result is an escalation on the load imposed on cloud services, culminating in an increase on user-perceived latency and an overall degradation of Quality of Service (QoS). The eminent explosion of Internet of Things (IoT) devices will further increase the strain on cloud infrastructures, due to the large volume of data that is going to be produced by such devices and that needs to be stored and/or processed in cloud data centers [34].

Edge Computing [20, 32, 33] enriches cloud architectures with computational nodes geographically closer to users (i.e, on the edge of the system). This allows moving particular application computations, such as data analysis with fast response, filtering of data sources, and computing user profiles, to edge nodes. These include base stations, routers, switches, regional data centers, proprietary nodes, mobile devices, among others. In summary, edge computing brings the following potential benefits [3, 35]: i) offload computations towards the edge, reducing user-perceived latency; ii) lessen the data deluge problem, by filtering data sources and lowering network traffic; and iii) minimize the computational load imposed on cloud infrastructures by leveraging on nodes outside the data center perimeter.

The resulting hybrid *Cloud/Edge Infrastructure* is highly heterogeneous with a high number of very diverse edge resources (i.e, nodes), some of which with limited capacity when compared with cloud infrastructures. Due to the large number of edge nodes, cloud/edge systems must be highly dynamic to cope with variable availability of resources (for instance due to failures or resource exhaustion), and with the demand for these resources (due to variable access patterns both across time and space). This paradigm has been identified as being suitable for a wide range of applications that include sensor data filtering, video/sound streaming, social networks/crowd sourcing, and smart cities and smart spaces [32].

Microservices on Cloud/Edge Platforms The highly heterogeneous and dynamic nature of cloud/edge platforms will require applications to be more agile as to adapt themselves to varied execution conditions (edge resource

*The work presented here was partially supported by the Lightkone European H2020 Project (under grant number 732505) and NOVA LINCS (through the FC&T grant UID/CEC/04516/2013).

availability) and sudden changes in workloads. In this paper we present our vision for a possible way to simplify the construction of efficient and robust applications in the cloud/edge setting that is based on *Microservice Architectures* (MSA) [6, 7], a paradigm to develop, deploy, execute, and manage applications that has gained significant traction recently [4, 36]. MSA capitalises on Service Oriented Architecture/Computing (SOA/SOC) [27, 29], defining applications as being composed of a set of small, independent, single purpose services [30]. This enables an easier development and integration of components [9, 13, 22, 25], since each microservice can be developed independently, with different technologies, having their interactions mediated by standard communication protocols, hence avoiding tight coupling between different aspects of the application logic. Individual microservices instances have independent life-cycles, which implies that the failure of an instance should have minimal effects in other application components. Finally, each microservice relies on its own and independent data storage solution to manage its (persistent) state [8].

MSA can simplify the design, implementation, deployment, execution, and management of applications in cloud/edge environments due to the following main aspects: i) microservices can be scaled independently of other application components in a more expedite way and with lower cost; ii) microservices can (efficiently) be moved/replicated to edge nodes where they might have the highest impact on performance; iii) complex functionalities may be built by combining multiple microservices in an incremental way, promoting efficient management of applications by enabling the inclusion of novel microservices or by evolving existing ones.

While these properties are well aligned for applications in cloud/edge environments, the management of applications with large numbers of microservices deployed across large numbers of heterogeneous devices at multiple geographic locations, subjected to varied workloads demanding the dynamic reconfiguration of the application by moving or partially replicating microservices' logic and data is a daunting task. To address this challenge we envision a middleware for creating autonomic Microservices Architectures for cloud/edge hybrid environments that handles most of this complexity by monitoring, moving, and replicating microservices dynamically.

The remainder of this paper is organized as follows. In Section 2 we motivate our vision through a possible use case. Section 3 describes three key aspects for achieving autonomic Microservices Architectures: Cloud-Edge Services (§§3.1), Cloud-Edge Data (§§3.2), and Cloud-Edge Monitoring (§§3.3). Section 4 describes how these three aspects can be integrated to support self-managing microservices applications with emphasis on the requirements to enable these components to cope with the heterogeneous nature of edge computational resources. Finally, Section 5 concludes the paper.

2 Use Case Application

We believe that many existing cloud-based applications could highly benefit from the proposed hybrid cloud/edge architecture based on autonomic microservices. In this section, we exemplify with a concrete use case and explain how our vision could benefit this particular class of applications.

Consider a web application such as Reddit[1], that operates as a message board, where users can freely discuss, comment, and vote multiple topics. Reddit has currently millions of users scattered throughout the world, and it has effectively become the *comments section* of the Internet [26]. There are virtually subreddits (specialized message boards) for every topic, and they continue to appear in reaction to events and trends on the real world. This implies that local events, such as football matches, government elections, and so forth, have their own subreddit. For the period of the event, these subreddits are highly accessed, particularly by users in the event location. Naturally, this can lead to high loads, potentially with a negative impact on the overall performance of the application. Such scenario can be effectively mitigated by relying on a hybrid cloud/edge architecture. In particular, the main application state (user accounts, posts, and votes) and logic components (materialized by a set of cooperating microservices) would reside in cloud infrastructures, potentially replicated at multiple geographic locations. In this setting, a sudden surge of users starting to access a particular subreddit, can be identified by two factors: increased user requests received by the application logic components (at a given data center); and increased database requests for data of a given subreddit.

This *monitoring* information can be used to trigger an autonomic reconfiguration of the system deployment, in particular, it would be possible to identify the geographic location generating the surge of requests and relate it to the subreddit being accessed (from that location). The system could then locate available edge resources in that location and create, on those resources, new replicas of (some) microservices and new database replicas to handle the requests of close users for that subreddit. Database replicas may hold only part of the information for that subreddit, potentially copied in a lazy fashion (i.e, as users access it). Data updated at this location can be synchronized with the data storage services in the cloud in background.

The resulting architecture allows to provide quality of service to the users in that geographic location without disrupting quality of services for other users worldwide.

[1] http://reddit.com

Furthermore, for topics that become popular on larger geographic areas (for instance, the European cup or the US Presidentials), additional database and application logic instances could be created, at different locations. Monitoring accesses to these edge components would also allow the system to identify when they are no longer useful, allowing to decommission these edge components, saving resources.

Additionally, our envisioned architecture could also be beneficial if such a web service grows to include further interactions among users, for instance live streaming of video among closely-located clients or some sort of multi-user game. In particular, by allowing interactions to be mediated by edge application components, we could expect improved latency and reduced load in the servers.

In the following we discuss in more detail how to materialize and implement this vision.

3 Autonomic Microservices Architecture

Enriching Microservice Architectures towards becoming *autonomic* [14, 23, 28] is crucial to our vision of having microservice-based applications adapting dynamically on hybrid cloud/edge infrastructures. To do this, it is essential to build mechanisms that can allow the (autonomic) control of the deployment, life-cycle, and binding of applications' components, considering both the application logic and application data planes.

Understanding the nature of autonomic systems is hence essential to achieve these goals. The fundamental architecture of autonomic systems, as proposed originally by IBM, follows the *MAPE-K* pattern [11]. It defines autonomic systems as systems that own the following components: $i)$ Monitoring, which gathers information regarding the system's operation and the state of its components; $ii)$ Analysis, which performs data analysis and reasons about the information provided by the Monitoring component, identifying situations when the current configuration of the system needs to be changed; $iii)$ Planning, which defines possible strategies to adapt the current configuration of the system (and predicts the benefits and costs of each strategy); and $iv)$ Execution, which coordinates the execution of the actions recommended by Planning to update the system's configuration. All these components are entwined by a shared $v)$ Knowledge base, which maintains information on the system's (current and past) configurations and monitoring data.

A clear challenge in devising an autonomic Microservice Architecture is to identify which aspects of the application configuration can be autonomously managed, the information required to execute such re-configurations, and being able to operate with localized (and possible incomplete) information. Self-managing applications composed of a large number of components (microservices and database instances) requires accessing large volumes of data to guide re-configuration decisions (as well as significant coordination overhead). This implies that management must be achieved in a decentralized fashion relying on partial and localized information.

We argue that to make a microservice application autonomic, one has to be able to dynamically control: $i)$ the application logic plane, which entails controlling the life-cycle of individual instances of microservices, including deciding where new instances should be deployed and how these instances interconnect among each other (i.e, when a microservice has to access another microservice, which instance should it contact); $ii)$ the application data plane, which entails controlling the location of data storage replicas used in the operation of individual microservices. This involves not only controlling the life-cycle of these replicas, but also understanding which fraction of the microservice state is relevant to maintain in each of these replicas, and the consistency guarantees (enforced by replication protocols) provided by different replicas located in different cloud and edge resources; and finally, $iii)$ an adaptive and distributed monitoring mechanism that can efficiently gather relevant information. Besides information related to used and available resources, the monitoring component needs to gather the necessary information to guide the dynamic and partial replication of data and provide this information to all components in the system that make re-configuration decisions.

We now detail the goals and requirements for these three aspects, which we envision in three components named *Cloud-Edge Services*, *Cloud-Edge Data*, and *Cloud-Edge Monitoring*. In Section 4 we discuss how these components interact and how we envision their materialization.

3.1 Cloud-Edge Services

To take advantage of the hybrid cloud/edge infrastructure, microservices should be executed in the most appropriate edge resource. Deciding the ideal location to execute a particular microservice is however, a non-trivial decision since this is affected by multiple factors, including its access pattern (i.e, number of requests and their origins), its resource consumption (CPU/Memory/Bandwidth), its interaction pattern with, and dependencies of, other services (and those services' locations), the location of the data manipulated by the service (in the case of microservices that frequently access data), and the availability of computational resources at different edge locations. Some of these aspects are dynamic, and will change frequently. To address this challenge we need a component in our autonomic Microservice Architecture that can ponder these aspects and dynamically adjust the deployment of the microservices, that compose the application, accordingly.

In its essence, this component is responsible for controlling the deployment of microservices instances across available nodes (both in the cloud and the edge). This control translates itself in three fundamental operations: i) migrate (i.e, move) existing instances of microservices among nodes; ii) create new instances (i.e, replicas) of microservices in nodes with available capacity; and iii) remove existing instances that are no longer useful for the correct and efficient operation of the application.

Since microservices may need to interact with other microservices, this component must also keep track of the location of (different) microservices instances and modify the binding between them, to ensure efficient communication patterns; or that sets of microservices instances interacting frequently are migrated/replicated together. It should also handle the binding of microservices to data storage instances in cooperation with the Cloud-Edge Data component (discussed further below).

Such goals are hard to achieve due to the large number of different deployment decisions that are possible at runtime, the need to consider the dependency graph of microservices (i.e, which microservices communicate among them and how frequently), as well as multiple performance indicators, such as latency to process requests or local resource consumption, and restrictions related with the capacity of individual nodes.

To tackle such challenges we propose to reduce the complexity of managing microservices instances by leveraging on local management decisions, which restrict themselves to consider computational resources and services in close vicinity to the local node. Local decisions significantly reduces the amount of monitoring information required by this component to manage a set of microservices instances, as well as the number of possible re-configurations. However, this brings challenges related with global correctness of a configuration that is defined by multiple and independent decisions, and the potential for a system to be continuously re-configured (lack of stability). Nonetheless, these situations can be locally inferred and, in such cases, localized[2] coordination mechanisms can be employed.

3.2 Cloud-Edge Data

Microservices often resort to data storage systems to maintain their state (potentially shared among multiple instances). Since our Cloud-Edge Service component can migrate and replicate instances of microservices in the application logic plane, the same may be required for data storage systems to ensure the continuous and efficient operation of microservices that frequently manipulate application state. While dynamically managing the application data plane depends on the same set of operational aspects discussed above (from available resources, current load and resource consumption, among others), migrating or replicating data implies additional challenges. Namely, regarding the consistency of the replicated data at multiple locations that extend towards the edge of the system (i.e, closer to end clients).

In a nutshell, this component manages the life-cycle and interactions of the multiple data storage systems of an application. This implies that this component has the capacity to perform the following main operations: i) migrate a database instance; ii) create a new database instance (i.e, a new replica); and iii) decommission an existing database instance. Often, hosting a full copy of a (large) database for every microservice instance is inefficient or even impossible. Thus, the ability of spawning partial replicas of a database is a crucial requirement (i.e, replicas of a database that only contain a fraction of the data stored in the whole database). To do so, it is necessary to infer which data elements in a database are accessed at each location. Contrary to the management of the application logic plane, the management of database replicas is made additionally complex by the need to maintain the data replicated across multiple instances consistent. This entails re-configuring or even running replication protocols that can offer some sensible form of consistency in a scenario using partial replication.

Managing the application data plane presents multiple and significant challenges. The first of which is the fact that different microservices in an application can use different data storage solutions, owning potentially different properties (e.g, relational versus non-relational databases). The result is a potentially high heterogeneity of database systems that have to be managed in an integrated fashion. When considering the creation of partial replicas of databases, it is necessary to infer which data objects are effectively accessed at each location. Additionally, the number of active replicas will have a direct impact on the performance and overhead of replication protocols, particularly those that provide strong consistency. Finally, when replicating different databases, determining the right consistency model to implement is hard, as this is highly dependent on the application (i.e, microservice) logic.

Similarly to the management of microservices, we argue that the management of database instances can be decomposed in, mostly, localized decisions. This implies that decisions to migrate, replicate, partition, or decommission a replica can be performed locally at the location where the replica is deployed, considering only information gathered from a limited number of close nodes and replicas. Furthermore, the replication of database instances can leverage on a hierarchical approach, where replicas of a database are organized in a tree topology.

[2]By localized we mean a process or mechanism which is executed by a node that only requires coordination and information exchange with a very limited (and nearby) set of other nodes.

The root represents a replica (potentially partitioned) containing all data, and children along the tree always replicate a subset of the data hosted by their parents. This allows to design replication protocols that leverage on this topology and that can potentially offer different consistency guarantees across replicas at different levels of the tree topology. Finally, in some cases, we can use read only database replicas (i.e, caches), which have lower impact on replication overhead.

3.3 Cloud-Edge Monitoring

Monitoring is essential to operate any large-scale distributed system. In the context of cloud computing there are multiple tools to collect information about the system's execution, including resource consumption and performance indicators [1, 5]. In fact, monitoring is paramount for Microservice Architectures [10] to enable the enforcement of QoS metrics, such as request processing latency, and more so to build autonomic Microservice Architectures. In our context, we have to extend the monitoring capacity towards the edge of the system, which requires acquiring information across large numbers of resources, potentially located on distant geographical areas. Additionally, to efficiently move or replicate service close to the edge, our monitoring service needs to collect information about which data is necessary for processing requests from a given geographic location.

We envision a monitoring component that executes in a distributed fashion across computational resources located in the cloud and the edge. In each of these resources we are interested in acquiring data related with four groups of metrics: i) available capacity of the resource (CPU, Memory, Bandwidth); ii) resource consumption of microservices and databases executing locally; iii) performance indicators associated with the execution of operations in (local) individual microservices and databases, which include latency for processing requests, number of received requests and their origin, and tracking of errors and faults; and iv) data access patterns that can guide (partial) replication in the edge, by tracing and collecting information about which data is needed to process requests originated in a given geographic location.

This component has additional responsibilities, that include pre-processing and/or co-relating different metrics locally, delivering relevant data to local components responsible for the dynamic management of application logic and application data planes, and propagating monitoring information to other nodes in close vicinity (and symmetrically receive information from other nodes) that can be relevant for the management decisions taken by the (local) Cloud-Edge Services and Cloud-Edge Data components. Finally, this component might need the ability to perform remote monitoring of edge resources that have limited capacity and hence cannot monitor themselves.

Evidently, building such a monitoring solution has its own challenges. Acquiring information from microservices and databases requires using information from existing monitoring APIs and other advanced techniques, such as invocation tracing across multiple components [19], and extend them to run efficiently in the target infrastructure. Monitoring the available capacity in resources can be extracted from the operating system, but this entails supporting different operating systems and execution environments (e.g, virtualization, containers). Monitoring, pre-processing acquired data, and propagating this data to the relevant (distributed) components in the autonomic Microservice Architecture needs to be done efficiently, requiring the monitoring component itself to adapt the number of monitored metrics and the periodicity with which these metrics are obtained, to avoid exhausting the capacity of nodes. Finally, the monitoring component might be required to aggregate acquired data for efficient transmission, while maintaining an adjustable, or known, error bound/deviation on reported values.

Likewise to Cloud-Edge Services and Cloud-Edge Data components, the Cloud-Edge Monitoring component can rely on localized operations to ensure its scalability and efficiency. This implies that acquired monitoring data is only used by (and propagated to) the components running in the local node and nodes in very close vicinity. Additionally, one can leverage on multiple existing monitoring tools and frameworks, both for cloud/cluster environments and edge environments, to simplify the development of this component and its associated probes. This component can also leverage on the configuration information regarding the application logic and application data planes made available by other components, to guide monitoring data dissemination.

4 Integration and Implementation

We now discuss possibilities that pave the way for realizing and implementing the autonomic microservice architecture discussed above. Figure 1 provides an overview of the envisioned architecture. We expect to build a distributed middleware layer that combines the three main components and exposes APIs to simplify the development of applications. This middleware interacts directly with the applications' components being managed (microservice instances in the application logic plane and database instances in the application data plane).

Each component interacts only with the other components within the same middleware instance (green lines), and with the components of the same type in other middleware instances (orange lines). Furthermore, communication among middleware instances is restricted to those that are in close vicinity. This implies that each component operates in a localized fashion, where it exchanges information and coordinates with a limited number of other

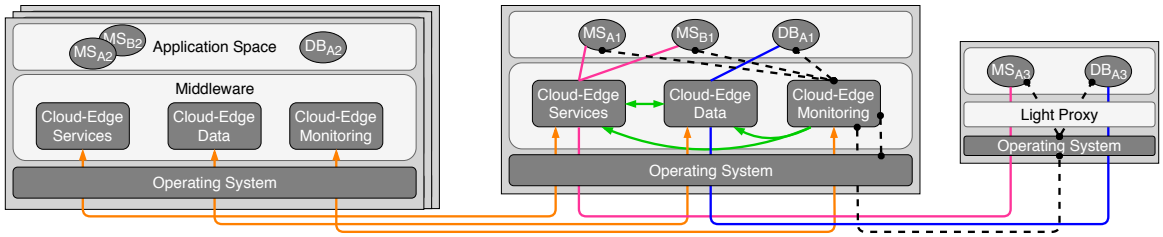


Figure 1: Autonomic Microservice Architecture

entities. Additionally, we consider the use of light proxies for edge nodes with limited capacity, enabling the remote and direct management/monitoring of applications' components by a more powerful (nearby) node.

In each middleware instance the three components interact among them. The Cloud-Edge Monitoring component provides information gathered locally and from close-by nodes, to the Cloud-Edge Services and Cloud-Edge Data. These take into consideration the current configuration of each one to make adaptation decisions for the application logic and application data planes, respectively.

To interconnect different middleware instances across different cloud/edge nodes, we can use a lightweight and robust overlay network, whose topology is defined to take into consideration the proximity between nodes (i.e, in terms of latency or administrative domains), as well as the needs of each of the three components of our architecture [12,15,17]. For instance, the overlay links can map the hierarchical relations between different database instances (i.e, replicas) and/or the flow patterns of monitoring information. This overlay can then be used to efficiently exchange information among the components residing in the middleware instances by leveraging on lightweight and robust gossip mechanisms [16, 17]. Application's configuration and specification can also be updated by operators, leveraging on this overlay to efficiently and reliably disseminate such modifications.

Evidently, not all edge resources support the direct execution of software developed for cloud environments, in particular those that are tightly coupled with software stacks that are specific to some cloud infrastructures. This is particularly true for storage services, and has two implications. The first is that our middleware layer must offer the execution environment for microservices being managed by the Cloud-Edge Services component. This evidently depends on the capacity of the hardware at the edge. In some cases its materialization may resort to containers using for instance Docker, while in other cases microservices must be executed directly on the operative system. The last option might require multiple implementations of the same microservice. The second implication is that not all data storage solutions can be executed in arbitrary edge hardware, creating the need to develop lightweight versions of these storage solutions that can efficiently execute in hardware with limited capacity. These *edge database instances* will only replicate small fractions of the data. Specialized replication protocols to manage the interactions of edge data storage replicas with counterparts executing in cloud infrastructures and other edge nodes, will need to be developed. A potentially interesting venue is to explore replication schemes that offer different consistency guarantees [18] among database replicas executing on different locations.

5 Conclusion

In this paper we argue for the development of an autonomic Microservice Architecture to address the inherent challenges of developing and managing complex edge applications, whose components execute across a wide spectrum of resources, from cloud infrastructures to devices very close to end-users. This will allow edge-enabled applications to adapt their deployments at runtime in response to variations in workloads and available computational resources. Our vision focuses on the combination of three main components that provide autonomic features to the application logic plane (Cloud-Edge Services), autonomic features to the application data plane (Cloud-Edge Data), and a self-adaptive distributed monitoring component that gathers (and distributes) relevant information about the system state to guide the behavior of the other components (Cloud-Edge Monitoring). We discuss how to realize this architecture using a distributed middleware that combines these three components. We note that leveraging on microservices to build adaptable edge applications has been previously proposed in the context of osmotic computing [31, 36] however, our proposal goes a step further by considering both the application logic and application data planes.

Although not addressed in this paper, the proposed architecture should also consider security issues, particularly regarding the access control to edge resources and data privacy and integrity issues. The proposed architecture could also benefit from the use of machine learning mechanisms, potentially operating on cloud infrastructures, to enable analysis of access patterns and predict situations (i.e, peak loads) enabling a more timely adaptation of the application configuration, potentially requiring additional monitoring information.

References

[1] G. Aceto, A. Botta, W. De Donato, and A. Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[3] K. Bilal, O. Khalid, A. Erbad, and S. U. Khan. Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. *Computer Networks*, 130:94 – 120, 2018.

[4] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. *CoRR*, abs/ 1606.04036, 2016.

[5] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, 74(10):2918 – 2933, 2014.

[6] M. Fowler. Microservices resource guide. https://martinfowler.com/microservices/, 2017.

[7] M. Fowler and J. Lewis. Microservices, a definition of this new architectural term. http://martinfowler.com/articles/microservices.html, 2014.

[8] M. Garriga. Towards a taxonomy of microservices architectures. In A. Cerone and M. Roveri, editors, *Software Engineering and Formal Methods*, pages 203–218, Cham, 2018. Springer International Publishing.

[9] E. Haddad. Service-oriented architecture: Scaling the uber engineering codebase as we grow. https://eng.uber.com/soa/, 2015.

[10] S. Haselböck and R. Weinreich. Decision guidance models for microservice monitoring. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pages 54–61. IEEE, 2017.

[11] IBM. An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[12] M. Jelasity, A. Montresor, and O. Babaoglu. T-man: Gossip-based fast overlay topology construction. *Comput. Netw.*, 53(13):2321–2339, Aug. 2009.

[13] M. Jung, S. Mllering, P. Dalbhanjan, P. Chapman, and C. Kassen. Microservices on aws. https://d0.awsstatic.com/whitepapers/microservices-on-aws.pdf, 2017.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[15] J. Leitão, J. P. Marques, J. Pereira, and L. Rodrigues. X-bot: A protocol for resilient optimization of unstructured overlay networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2175–2188, Nov 2012.

[16] J. Leitão, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 301–310, Oct 2007.

[17] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429, June 2007.

[18] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX.

[19] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393, New York, NY, USA, 2015. ACM.

[20] R. Mahmud, R. Kotagiri, and R. Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of Everything*, pages 103–130. Springer, 2018.

[21] D. C. Marinescu. *Cloud Computing: Theory and Practice*. Morgan Kaufmann, 2013.

[22] T. Mauro. Adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, 2015.

[23] R. Mehmood, J. Schlingensiepen, L. Akkermans, M. C. Gomes, J. Malasek, L. McCluskey, R. Meier, F. Nemtanu, A. Olaya, and M.-C. Niculescu. Autonomic systems for personalised mobility services in

smart cities. Technical report, King Khalid University, et. al., 2014. (Work in the context of the ARTS TUD Cost Action TU1102. Under submission).

[24] P. M. Mell and T. Grance. Sp 800-145. the nist definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.

[25] S. Newman. *Building Microservices - Designing Fine-Grained Systems*. O'Reilly Media, Inc, Gravenstein Highway North, Sebastopol, CA, 2015.

[26] J. O'Gara and C. J. Garcia. What is reddit? a guide to the front page of the internet. https://www.digitaltrends.com/social-media/what-is-reddit/, 2017.

[27] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, Nov 2007.

[28] M. Parashar and S. Hariri. Autonomic computing: An overview. In J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *Unconventional Programming Paradigms*, pages 257–269, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[29] R. Perrey and M. Lycett. Service-oriented architecture. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, SAINT-W '03, pages 116–, Washington, DC, USA, 2003. IEEE Computer Society.

[30] M. Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc., 1 edition, 2016.

[31] V. Sharma, K. Srinivasan, D. N. K. Jayakody, O. F. Rana, and R. Kumar. Managing service-heterogeneity using osmotic computing. *CoRR*, abs/1704.04213, 2017.

[32] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[33] K. Toczé and S. Nadjm-Tehrani. Where resources meet at the edge. In *Computer and Information Technology (CIT), 2017 IEEE International Conference on*, pages 302–307. IEEE, 2017.

[34] B. Varghese and R. Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.

[35] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud, SmartCloud 2016, New York, NY, USA, November 18-20, 2016*, pages 20–26, 2016.

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

LiteSense: An Adaptive Sensing Scheme for WSNs

João Marco C. Silva
HASLab, INESC TEC
Universidade do Minho
Email: joao.marco@inesctec.pt

Kalil Araujo Bispo
Departamento de Computação
Universidade Federal de Sergipe
Email: kalil@dcomp.ufs.br

Paulo Carvalho, Solange Rito Lima
Centro Algoritmi
Universidade do Minho
Email: {pmc,solange}@di.uminho.pt

***Abstract*—Adaptability and energy-efficient sensing are essential properties to sustain the easy deployment and lifetime of WSNs. These properties assume a stronger role in autonomous sensing environments where the application objectives or the parameters under measurement vary, and human intervention is not viable. In this context, this paper proposes *LiteSense*, a self-adaptive sampling scheme for WSNs, which aims at capturing accurately the behavior of the physical parameters of interest in each WSN context yet reducing the overhead in terms of sensing events and, consequently, the energy consumption. For this purpose, a set of low-complexity rules auto-regulates the sensing frequency depending on the observed parameter variation. Resorting to real environmental datasets, we provide statistical results showing the ability of LiteSense in reducing sensing activity and power consumption, while keeping the estimation accuracy of sensing events.**

I. Introduction

WSNs may include several types of sensing, from continuous sensing to event detection or triggering of local actuators. This versatility allows their use in a wide spectrum of application fields, for instance, precision agriculture, intrusion detection and security systems, health care, wearables and environmental sensing. In many cases, WSNs operate without human intervention as devices substitution or maintenance may be impracticable, making the presence of self-management and power saving mechanisms strongly recommended [1]. In this way, sensor nodes must waste the minimum of power to accomplish their tasks, trying to maximize lifetime resorting to the utilization of efficient algorithms, lower-power usage components or reconfiguration procedures.

Considering that the energy consumed by the communication module when transmitting a bit across the network can overcome the energy required to process thousands of instructions [2], it is clear that reducing the volume of data transmitted is a key aspect in WSNs operation. Although this can be addressed resorting to data aggregation schemes, the underlying processes may imply data loss. Furthermore, the sensing subsystem is also a significant source of power consumption and, for this reason, sensing data should only be acquired when necessary, avoiding the waste of sensing, processing and transmission capacity.

In this context, this paper proposes *LiteSense*, an adaptive sampling scheme oriented to WSNs aiming at improving the trade-off between capturing data accurately and saving energy to enhance operational sensors' lifetime. The mechanism relies on self-regulation of sensing events in order to reduce the amount of data acquired and transmitted without human intervention. The proof-of-concept demonstrates that adaptive sampling can be a solid approach to reduce significantly the number of sensing events and power consumption, while maintaining an accurate view of WSN activity and behavior.

This paper is organized as follows: related work is discussed in Section II; the sampling scheme design goals and rational are described in Section III; the proof-of-concept and the corresponding evaluation results are discussed in Section IV; and the conclusions are summarized in Section V.

II. Related work

Although adaptive sampling has been successfully used in conventional computer networks [3], [4], the available solutions can hardly be applied in resource-constrained networks such as WSNs. Thus, several authors have devoted efforts in proposing adaptive sampling approaches for WSNs, attending to their inherent computational and energetic limitations.

In [5], it is proposed an optimal sensing scheduling policy for a power harvesting system equipped with a finite battery. This system aims to rule sensing frequency based on the battery power levels of sensor nodes, disregarding the variability of the observed parameter. In [6], it is proposed a recovering framework for big data sets through a small number of sensing events taking into account the space and time correlation properties from previous samples. The analysis resorts to probabilistic relations among variables involved in the compression, transmission and recovering processes. This process lowers the number of transmissions and transmission rate, disregarding the accuracy of the sensing process. Similarly, the authors in [7] establish a framework for collecting data from a WSN based on adaptive compressive sensing that considers the power consumption and the amount of information in sensing data. The study proposes an algorithm to obtain a more precise approximation of the measurements by wasting as less energy as possible. This proposal is also confined to adapt the transmission rate.

Taking into account the above discussion it is clear that most of the studies on adaptive sensing only cover a partial set of variables to optimise, such as the transmission rate, disregarding measurement accuracy. This evinces the need of studying and improving the trade-off between sensing events and accuracy for distinct WSN contexts. The present work intends to be a contribution in this respect.

III. LITESENSE SAMPLING SCHEME

The proposed sampling scheme aims at sensing physical parameters by detecting their values and variations accurately while saving sensor nodes' resources. The main design goals are identified in Figure 1 and explained next.

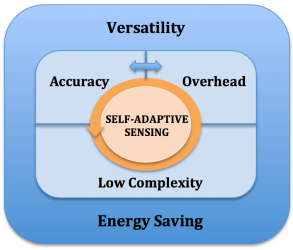


Fig. 1. LiteSense adaptive sampling - design goals

Versatility - the sampling scheme should be able to operate over distinct type of WSN measuring scenarios and parameters. This can be obtained endowing the sampling scheme with an intrinsic self-adaptability property in order to auto-regulate the data sensing events according to each parameter behavior. It involves detecting variations on the measures and increasing/decreasing the sampling frequency accordingly.

Accuracy - the sampling scheme should be able to capture the correct behavior and values of the physical parameters under measurement. This involves identifying either the continuous behavior of a parameter being sensed or detecting critical and sporadic events.

Low overhead - the sampling process should reduce the overhead of sensing events, without compromising accuracy. This can be achieved taking advantage of parameter stability to decrease sampling frequency, reacting to variations with reduced impact on the overall performance.

Low complexity - the self-adaptive nature of the sampling scheme should be driven by simplicity of implementation and low consumption of resources. Adaptiveness should rely on a simple arithmetic process, e.g., inspired on TCP RTT estimation mechanism [8].

Energy saving - the sampling scheme should be able to monitor the parameters of interest based on energy-aware sensing, processing and communication subsystems. In this way, (i) reducing the number of sensing events needed to capture the parameter behavior; (ii) adapting the sampling frequency through low-cost algorithms; and (iii) reducing the number of transmissions to other sensors/actuators, are steps to impact positively on the energy consumption.

A. Self-adaptive sampling

As mentioned before, attending to the heterogeneity of application scenarios WSNs may have, self-adaptiveness is a mandatory property to assure. In this context, LiteSense adaptive sampling scheme uses the temporal variation in the observed scalar physical quantities in order to self-adjust the interval between two consecutive sensing events.

Basically, when the sampled values of the observed parameter do not vary significantly, the interval between two sensing events is increased, reducing its frequency, which leads to less computational effort and consequent less energy consumption [2]. Conversely, if a significant variation in the sampled parameter is observed, the time scheduling for the next sensing event is decreased in order to improve the accuracy in identifying its temporal fluctuation. Thus, when a sensing event is performed, the mean of the observed variable is calculated using the moving average $\bar{X}_i = (1-\alpha)\bar{X}_{i-1} + \alpha S$, where $\bar{X}_{i-1}$ is the mean calculated in the previous event and α is the weight of new observed value S.

Although the standard deviation σ is the conventional choice for estimating the variance, it imposes costly operations to constrained devices. Alternatively, as discussed in [8], the mean deviation is a good approximation to σ, being easier to compute. Therefore, using the current mean value $\bar{X}_i$, the variation in the observed values is calculated resorting to the mean deviation $\bar{V}_i = (1-\beta)\bar{V}_{i-1} + \beta|\bar{X}_i - S|$, where, $\bar{V}_{i-1}$ is the mean deviation identified in the last sensing event and β determines the weight of the current deviation $|\bar{X}_i - S|$.

As presented in Table I, the adaptive mechanism compares the current estimated mean variation ($\bar{V}_i$) with the variation calculated in the previous sensing event ($\bar{V}_{i-1}$) in order to identify if the observed parameter has changed significantly, and then, the time interval used to schedule the next sensing event is adjusted accordingly. If $\bar{V}_i$ is lower or equal to $\bar{V}_{i-1}$, the observed parameter did not change significantly from the last sensing events, which allows to reduce the sensing frequency by increasing ΔT. Otherwise, the observed parameter changed significantly since the last observation, which requires more frequent sensing events, obtained by reducing ΔT. An additional constraint is used to prevent ΔT from growing indefinitely (ΔT_{max}), thus guaranteeing a minimum number of samples per time unit. Conversely, the maximum frequency of sampling (ΔT_{min}) is also limited so that the interval between sensing events does not tend to zero, which would result in an overwhelming resource consumption. The factor ϵ in Table I is a filter expressing the scheme reactivity. This value may be set as a constant, indexed to the variation level of the observed parameter, or a function related to the battery level.

TABLE I
RULES TO SCHEDULE THE NEXT SENSING EVENT

$\bar{X}_{i-1} \xleftarrow{\bar{V}_{i-1}} \bar{X}_i \xleftarrow{\bar{V}_i} S$	ΔT_{i+1}
$\bar{V}_i \leq \bar{V}_{i-1}$	$min(\Delta T_i + (\Delta T_i \times \epsilon), \Delta T_{max})$
$\bar{V}_i > \bar{V}_{i-1}$	$max(\Delta T_i - (\Delta T_i \times \epsilon), \Delta T_{min})$

IV. EVALUATION RESULTS

This section provides the proof-of-concept for LiteSense, including a set of evaluation results regarding the ability of adaptive sampling in: (i) reducing the number of sensing events; (ii) identifying the temporal variation of the observed parameters accurately; (iii) self-adjusting the sensing

frequency in accordance with the variability of environmental parameters; and (iv) reducing the power consumption of sensing and communication subsystems.

The analysis compares the performance of LiteSense adaptive sampling scheme to a deterministic scheme, widely used in monitoring scalar physical quantities. The publicly available dataset used in the tests was previously collected during approximately six hours, at intervals of five seconds, in a WSN using TelosB motes [9]. The WSN consists of eight sensors deployed in indoor and outdoor environments, sensing temperature (indoor: ST1,ST2; outdoor: ST3,ST4) and humidity (indoor: SH1,SH2; outdoor: SH3,SH4). Aiming at evaluating the accuracy in identifying sudden changes in the observed parameters, sensors (ST1, ST4; SH1, SH4) were exposed to a steam of hot water to spark humidity and temperature levels.

The adaptive scheme was experimentally set with $\alpha = \beta = 0.7$ in order to stress the weight of the latest sensed value and corresponding variation. The rules in Table I were set with $\epsilon = 0.15$, meaning that the interval between samples (ΔT) is increased by a factor of 15% when the observed parameter remains stable, and reduced by the same factor when it changes significantly. As discussed in Section III-A, this adjustment may also be indexed to the observed variation or to the current battery level. In this proof-of-concept, ΔT_{max} and ΔT_{min} were set to 30 and 5 seconds, respectively.

The performance of LiteSense is evaluated relating the number of sensing events along the monitoring period with the accuracy in identifying the temporal variation of the observed parameters (temperature and humidity). As deterministic and adaptive sampling mechanisms yield to distinct number of samples, the statistical analysis considers the mean estimated value per second from the resultant time series. The accuracy is then estimated resorting both to the Mean Squared Error (MSE) and to the correlation between the time series.

A. Sensing events and estimation accuracy

Regarding the ability to reduce the number of sensing events, Figure 2 shows that, for all nodes and parameters, the adaptive scheme reduces in around 80% the number of events observed through the deterministic scheme. Attending to the lightweight algorithm ruling LiteSense, this significant reduction will also contribute to reduce energy consumption in the sensors, as discussed in Section IV-B.

However, despite of reducing the number of sensing events significantly, proving the efficiency of the proposed scheme requires verifying its ability to capture the real distribution of the observed parameters. This is accomplished by estimating the statistical representativeness of adaptive time series against the deterministic behavior. In this way, Figure 3 shows the distribution of sampled values along the test period, for indoor and outdoor sensors. The almost complete overlap of adaptive and deterministic resultant series, highlighted during both smooth and unstable periods, demonstrates that, even reducing the sensing events in around 80%, the adaptive scheme in LiteSense has the capability to catch up the real parameter pattern. Note that, even in presence of sudden environmental changes

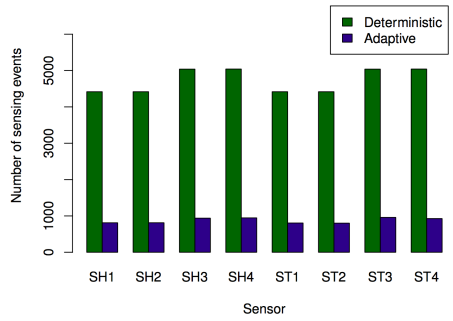


Fig. 2. Number of sensing events: Temperature (ST); Humidity (SH).

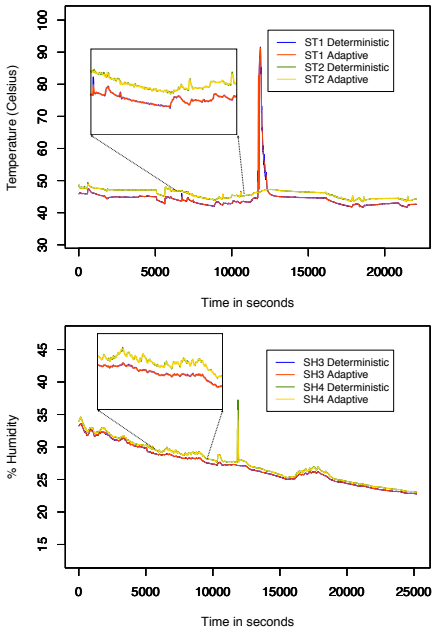


Fig. 3. Indoor observations (ST); Outdoor observations (SH).

affecting sensors ST1 and SH4, the adaptive scheme adjusts the sensing schedule correctly, confirming the versatility of LiteSense. These observations are corroborated through the high correlation (0.98 for all series) between the distributions of estimated parameters, and the low MSE (approximately 0.5 for temperature and 0.03 for humidity). This means that, for the considered scenarios, the estimation of humidity levels was more accurate, however, as depicted in Figure 3, the temperature was also accurately estimated.

The study of how the variation of ST1 impacts on the adaptiveness of ΔT along the time is illustrated in Figure 4. As shown, in presence of stable temperature readings (e.g., around t=15000 seconds), ΔT assumes a steady behavior in its highest values, meaning that the sampling frequency is kept low. Conversely, upon more unstable periods of the sensed parameter (as in the anomaly peak mentioned previously), the mechanism reacts promptly reducing ΔT, which is reflected in a sampling frequency increase.

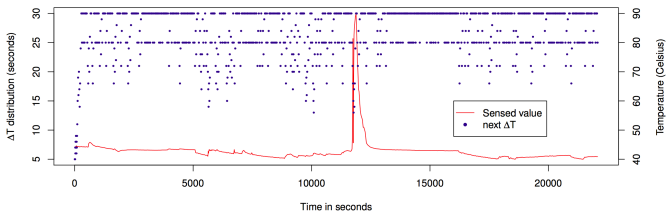


Fig. 4. ST1 - Distribution of ΔT regarding temperature variation.

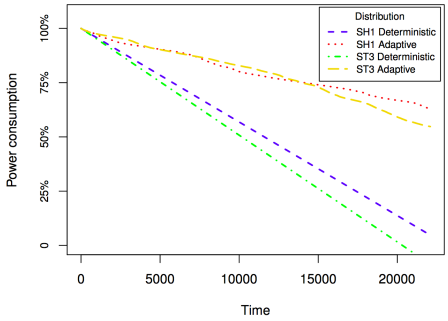


Fig. 5. Power Consumption: Indoor sensor (SH); Outdoor sensor (ST)

B. Power Consumption

Aiming at assessing the enhancement in power consumption promoted by LiteSense, the main operational states in the sensing and communication subsystems were identified and measured individually for each sensing event in a single-hop topology. The power consumption for different operation states in TelosB sensors is presented in Table II. The total consumption (T_P) is reflected by the sum of partial average times in which the sensor was involved in *transmitting*, *receiving*, *listening*, *sensing* and *sleeping* events, i.e.,:

$$T_P = T_{transmitting} \times n_{tx} + T_{receiving} \times n_{rx} + \\ + T_{sleeping} \times n_{sl} + T_{listening} \times n_l + T_{sensing} \times n_s \quad (1)$$

TABLE II
POWER CONSUMPTION FOR TELOSB OPERATIONAL STATES

Operation state	Power consumption (W)
$Transmitting$	$5.9E-5$
$Receiving$	$2.86E-5$
$Sleeping$	$1.0E-7$
$Listening$	$1.0E-6$
$Sensing$	$0.5E-6$

Figure 5 shows the evolution of power consumption for sensors SH1 and ST3 along the performed tests. At the beginning, all sensors had batteries fully charged (*i.e.,* 19160 W). As depicted, the progression of power consumed by the deterministic scheme is linear and leads to a faster battery drain when compared with the adaptive scheme. In this way, by using LiteSense, the power consumption is reduced in about 60% for all scenarios considered, which demonstrates that the ability in self-adapting sensing events has a major impact on reducing power consumption without affecting the accuracy of physical parameter measurements.

V. CONCLUSIONS

Attending to the inherent resource constraints of WSNs, this paper has proposed LiteSense, a self-adaptive sampling scheme aiming at capturing the behavior of multiple physical parameters accurately while reducing the overhead of sensing events and, consequently, the levels of energy consumption. The ability to adapt the sampling frequency through a low-cost algorithm was also defined as a design goal, due to the well known low-processing capabilities of sensor nodes. The proof-of-concept has provided initial results attesting the proposal ability to measure environmental parameters accurately, while reducing in 80% the number of sensing events comparing to deterministic sampling. As future work, the tests will consider a wider range of sensing parameters, and the tuning of the proposed sampling scheme (e.g., exploring how ϵ can be related to both the variation observed in sensed values and the battery level) aiming at further increasing its versatility.

Acknowledgments - The research leading to these results has received funding from European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505. This work has also been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT within the Project Scope: UID/CEC/00319/2013.

REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: A Survey," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 38, no. 4, pp. 393–422, 2002.

[2] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella, "Energy conservation in wireless sensor networks: A survey," *Ad Hoc Networks*, vol. 7, no. 3, pp. 537–568, 2009.

[3] E. A. Hernandez, M. C. Chidester, and A. D. George, "Adaptive Sampling for Network Management," *Journal of Network and Systems Management*, vol. 9, no. 4, pp. 409–434, 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1012980307500

[4] J. M. C. Silva, P. Carvalho, and S. Rito Lima, "A multiadaptive sampling technique for cost-effective network measurements," *Computer Networks*, vol. 57, no. 17, pp. 3357–3369, Dec. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2013.07.023http://linkinghub.elsevier.com/retrieve/pii/S1389128613002491

[5] J. Yang, X. Wu, and J. Wu, "Adaptive sensing scheduling for energy harvesting sensors with finite battery," in *2015 IEEE International Conference on Communications (ICC)*, June 2015, pp. 98–103.

[6] G. Quer, R. Masiero, G. Pillonetto, M. Rossi, and M. Zorzi, "Sensing, Compression, and Recovery for WSNs: Sparse Signal Modeling and Monitoring Framework," *IEEE Transactions on Wireless Communications*, vol. 11, no. 10, pp. 3447–3461, October 2012.

[7] C. T. Chou, R. Rana, and W. Hu, "Energy efficient information collection in wireless sensor networks using adaptive compressive sensing," in *2009 IEEE 34th Conference on Local Computer Networks*, Oct 2009, pp. 443–450.

[8] V. Jacobson, "Congestion Avoidance and Control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 314–329. [Online]. Available: http://doi.acm.org/10.1145/52324.52356

[9] S. Suthaharan, M. Alzahrani, S. Rajasegarar, C. Leckie, and M. Palaniswami, "Labelled data collection for anomaly detection in wireless sensor networks," in *Intelligent sensors, sensor networks and information processing (ISSNIP), 2010 sixth international conference on*. IEEE, 2010, pp. 269–274.

Flexible WSN Data Gathering through Energy-aware Adaptive Sensing

João Marco C. Silva[◇]
HASLab, INESC TEC
Universidade do Minho
Email: joaomarco@di.uminho.pt

Kalil Araujo Bispo
Departamento de Computação
Universidade Federal de Sergipe
Email: kalil@dcomp.ufs.br

Paulo Carvalho[*], Solange Rito Lima[*]
Centro Algoritmi
Universidade do Minho
Email: {pmc,solange}@di.uminho.pt

***Abstract*—The multitude of Wireless Sensor Networks (WSNs) environments, being typically resource-constrained, clearly benefit from properties such as adaptiveness and energy-awareness, in particular, in presence of demanding data gathering applications. This paper proposes a self-adaptive, energy-aware sensing scheme for WSNs (e-LiteSense), which aims at self-adjusting the data gathering process to each specific WSN context, capturing accurately the behaviour of physical parameters of interest yet reducing the sensing overhead. The adaptive scheme relies on a set of low-complexity rules capable of auto-regulate the sensing frequency according to the parameters variability and energy levels. The proof-of-concept resorts to real-world datasets to provide evidence of e-LiteSense ability to optimise the data gathering process according to energy levels, improving the trade-off between accuracy and WSN lifetime.**

I. Introduction

Wireless Sensor Networks (WSNs) have earned much attention in recent years due to research advances in the field of computation, robotics, microelectronics, nanotechnology and wireless communications, making possible the development of multifunctional low-cost wireless sensors, mostly fuelled by the Internet of Things (IoT) context [1]–[3].

The global performance and lifetime of this type of networks are typically dependent on the applications being supported and on the constraints affecting the infrastructure resources. The energy consumption is a critical aspect in WSNs as sensor nodes must optimally waste the least amount of energy possible to perform their tasks in order to maximise their lifetime. Research on this topic has identified the communication subsystem as the most demanding regarding to energy consumption [4]. This is true even for devices capable of energy harvesting. Other subsystems impacting on energy are sensing and processing.

The typical approach used to reduce energy consumption consists in aggregating data acquired regularly in order to reduce transmission events. However, the sensing subsystem also impacts on energy consumption, as several works have indicated that acquiring and processing data could even overcome the communication subsystem for some operational scenarios [5] [6]. This suggests that data should only be acquired when necessary, avoiding redundant information, even in presence of data aggregation.

In this context, the main objective of this work is to enhance the scope and efficiency of WSNs data gathering processes, through an energy-aware adaptive sensing scheme able to balance accuracy in data acquisition with energy conservation. The proposed e-LiteSense scheme is self-adjustable, being able to adapt sensing to specific WSN data gathering requirements, while extending sensor's lifetime and, consequently, the network utility. Through low-complexity rules, devised to optimise the *sensing, processing, communicating* vectors (see Figure 1), e-LiteSense adjusts the sensing frequency based on the variation in scalar physical quantities being measured and the device's residual battery level. The obtained results, using real datasets, evince the effectiveness of the proposed rules in distinct WSNs scenarios.

This paper is organised as follows: related work is discussed in Section II; the sensing scheme design goals and rational are described in Section III; the proof-of-concept and the corresponding evaluation results are discussed in Section IV; and the conclusions are summarised in Section V.

II. Related work

The definition of WSNs lifetime, although intrinsically related with battery lifetime, may vary depending on the application's functionality and behaviour, on the objective function and on the network topology in use. A comprehensive overview of the most relevant WSN lifetime maximisation techniques in the literature is provided in [7], where the underlying optimisation algorithms are classified as resource allocation, opportunistic transmissions, sleep-wake scheduling, routing, clustering, mobile relays and sinks, coverage and connectivity, optimal deployment, data gathering, network coding, data correlation, energy harvesting and beamforming, depending on their focus.

In particular, data correlation techniques take advantage of temporal-spatial data correlation characteristics to reduce the overall transmitted traffic and, by reducing/removing redundancy, they benefit energy conservation and network lifetime. When these techniques are combined with self-adaptive sensing schemes, data gathering and transmission may be substantially reduced, improving the versatility and lifetime of WSNs in heterogeneous environments.

Several adaptive sensing approaches have been proposed for WSNs taking into account the inherent computational and energetic limitations of these networks. In [8], an optimal sensing scheduling policy is proposed for a power harvesting system

equipped with a finite battery. The goal is to select sensing periods strategically so that the average sensing performance is optimised. In that system, the authors assume that performance depends on time duration between two consecutive sensing periods, and develop an algorithm where the sensing rates depend on the battery power levels of sensor nodes. However, this analytical study only considers the battery level of sensors disregarding the variability of the observed parameter.

In [9], an analytical study for sensing, compression and recovering signals from large datasets is carried out. The analysis resorts to probabilistic relations among variables involved in those processes to capture the space and time correlations of the signals of interest. The proposal lowers the number of transmissions and, consequently, the power consumption in sensor nodes. However, this study is mainly focused on reducing the transmission rate, disregarding the accuracy of the sensing process.

In fact, the complex relation between data analysis and data acquisition in adaptive sensing paradigms can be extremely powerful, as it allows a reliable estimation and signal detection in situations where non-adaptive detection fails [10]. For this reason, [10] investigates the signals' estimation over the adaptive detection problem. The authors present a general estimation procedure which may be satisfactory for a variety of cases, however, the authors recognise that the prior knowledge of some parameters might not be available in a real-life setting.

In [11], the authors establish a framework for collecting data from a WSN based on adaptive compression sensing (as in [9]), aiming to reduce the power consumption and the amount of information in sensing data. The paper proposes an algorithm for obtaining a more precise approximation of the measurements using the lowest possible energy. This proposal is also confined to adapt the transmission rate.

In [6], the authors worked in a prediction model for estimating battery lifetime. The methodology followed relied on the decomposition of the hardware operations by time periods, using datasheets of sensor nodes as reference.

In [5], the authors propose a system to calculate in runtime energy consumption of WSN platforms. The authors based the power consumption evaluation on separate pieces of hardware such as leds, data transmission, transmission power, microcontroller and CPU instructions.

The above review shows that most of the studies on adaptive sensing either: (i) assume a theoretical approach limiting their applicability to real WSN scenarios; or (ii) only cover a partial set of variables to optimise, such as the transmission rate, disregarding accuracy. This evinces the need of studying and improving the trade-off between sensing events and measurements accuracy for distinct WSN contexts.

III. Energy-aware adaptive sensing

This section identifies the main goals driving the design of the energy-aware adaptive proposal, followed by a detailed explanation of the underlying algorithms.

A. e-LiteSense architecture

This section includes an initial discussion clarifying how self-adaptive sensing may fit within WSN context. The envisioned architecture integrating e-LiteSense sensor nodes comprises three planes, as presented in Figure 1, and detailed below:

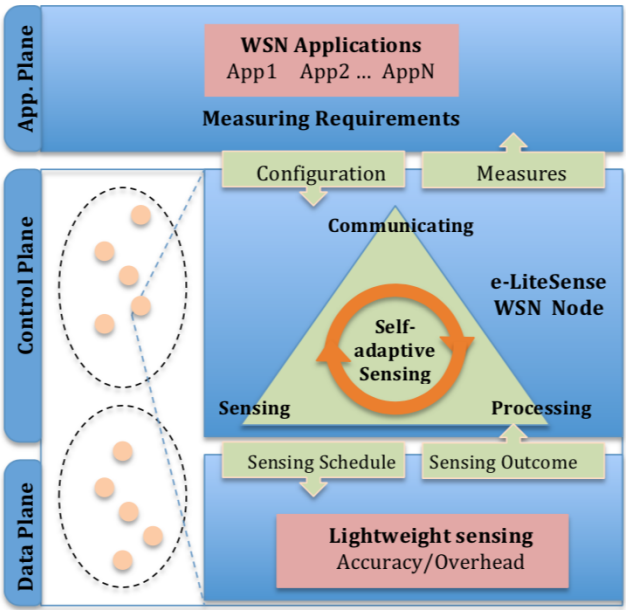


Fig. 1. e-LiteSense architecture

Application plane: WSNs sustain a plethora of application areas broadly categorised as: environmental monitoring; transportation and logistics; medical and health; security and emergencies; smart world; entertainment; retail, industrial control and automation; and military related applications [7]. Based on the requirements of each application context, specific measurement needs determine the configuration of sensor nodes. The obtained measures allow to keep track of crucial aspects of WSNs operation, such as the temporal evolution of the parameters under evaluation. The application plane is also responsible for processing measurement results reported by the control plane and for providing a visualisation component, when applicable. The required processing may involve results from single or multiple sensor measurements.

Control plane: The control plane, deployed in WSN sensor nodes, implements the e-LiteSense strategy. As detailed below, e-LiteSense allows an autonomous adjustment of sensing events, through lightweight algorithms which consider the main vectors involved in energy consumption: sensing; processing; communicating. These self-adaptive algorithms are devised to obtain accurate measures of the WSNs parameters under observation while reducing sensing overhead. Considering this trade-off, sensing schedule is adaptively configured, according to short/medium-term variations on the measured parameters. Conversely, the sensing outcome received from the data plane is processed (e.g., aggregated) to fulfil the application measurement needs.

Note that, the control plane may be deployed in sensor nodes directly (as represented in Figure 1) or in external coordination entities, according to SDNs and Edge Computing architectures/principles. In this case, depending on the application measurement requirements, a SDN controller implementing e-LiteSense will interact with edge nodes through southbound interfaces, being the edge nodes responsible for configuring the WSN sensor devices according to e-LiteSense algorithms dynamics.

Data plane: At data plane, the measures from sensors are obtained following the defined schedule and passed to the control plane. The use of a flexible sensing scheme will allow to configure sensing events according to the application measurement purpose and dynamics.

B. e-LiteSense sampling algorithms

Ground truth shows that, independently of the application type, a high volume of data acquired leads to high accuracy in measurements. However, this also implies a high-energy consumption, not only for sensing but also for processing and transmitting subsystems.

Attending to the interrelation among the above subsystems, the e-LiteSense self-adaptive sensing scheme uses the variation observed in scalar physical quantities being measured and the residual battery level of the device in order to adjust the sensing frequency targeting the best trade-off between accuracy and sensor lifetime.

Basically, when the data being acquired does not vary significantly, the time interval until the next sensing event (ΔT_{next}) is increased, reducing its frequency and, consequently, the overall energy consumed. This needs to be accomplished without noticeable impact on measurements accuracy. Conversely, if a significant variation is observed in the sensed values, the time scheduling for the next sensing event is decreased aiming to improve the ability in identifying and capturing the variable fluctuation. In both scenarios, the rules defining the time interval until the next sensing event are defined considering the current power level of the sensor's battery.

The proposed sensing scheme, relying on simple arithmetic processes, was devised having low overhead, simplicity of implementation and low resource consumption as main design goals. The algorithms 1 and 2 illustrate the expected behaviour through a high-level code description, and Table I describes their main elements.

Inspired on TCP RTT estimation mechanism [12], defining the parameter scalar variation resorts to the moving average of observed values ($\bar{X}_i$) and mean deviation ($\bar{V}_i$) as a simple way to compute an approximation for the standard deviation (see lines 6 and 7 in Algorithm 1).

On this basis, after a sensing event S, the algorithm compares how the measures evolve ($\bar{X}_{i-1} \xleftarrow{\bar{V}_{i-1}} \bar{X}_i \xleftarrow{\bar{V}_i} S$) in order to define the time interval until the next event (ΔT_{next}). If the current variation ($\bar{V}_i$) is higher than the last variation ($\bar{V}_{i-1}$), the measured value is changing significantly, so the time interval to the next sensing event is reduced aiming at getting an accurate view on the new pattern. Conversely, if the current variation is lower than the previous one, the observed physical quantity is stable, which allows using a higher interval until the next sensing event, conserving overall device's energy (Algorithm 2 performs this comparison and returns the corresponding adjustment value).

```
main()
1  X̄_{i-1} ← 0
2  V̄_{i-1} ← 0
3  repeat
4      S ← getSense()
5      ΔT_i ← timestamp of S
6      X̄_i ← (1 − α)X̄_{i-1} + αS
7      V̄_i ← (1 − β)V̄_{i-1} + β|X̄_i − S|
8      ΔT_next ← ΔT_i + (ΔT_i × getScale(V̄_i, V̄_{i-1}))
9      ΔT_next ← max(min(ΔT_max, ΔT_next), ΔT_min)
10     V̄_{i-1} ← V̄_i
11     wait(ΔT_next)
12 until end execution;
```

Algorithm 1: Setting ΔT_{next}

```
getScale(V̄_i, V̄_{i-1})
1  if (V̄_i > V̄_{i-1}) then
2      return −((getBattery() × (ε_max − ε_min))/100 + ε_min)
3  else
4      return ((100 − getBattery()) × (ε_max − ε_min))/100 + ε_min
```

Algorithm 2: Defining scheme reactivity

The algorithm reactivity (*i.e.,* ΔT changing scale) is defined according to the residual energy in node's battery. The main rational relies on: (*i*) a fast reduce and slow increase of ΔT in presence of high energy levels and (*ii*) a slow reduce and fast increase of ΔT in presence of low energy levels. Such reactivity is bounded by limit values (ϵ_{min} and ϵ_{max}) adjusted according to the application and type of measured phenomena. Algorithm 2 presents such operations.

An additional constraint is used to prevent ΔT from growing indefinitely (ΔT_{max}), thus guaranteeing a minimum number of samples per time unit. Conversely, the maximum frequency of sampling (ΔT_{min}) is also limited so that the interval between sensing events does not tend to zero, which would result in an overwhelming resource consumption. They might be set according to the application scenario.

TABLE I
DESCRIPTION OF ALGORITHM 1 AND 2 ELEMENTS

$\bar{X}_i$	- moving average of the observed parameter.
$\bar{X}_{i-1}$	- moving average calculated in the previous event.
S	- scalar value observed in the current sensing event.
α	- weight of S in the moving average $\bar{X}_i$.
$\bar{V}_i$	- current mean deviation of observed values.
$\bar{V}_{i-1}$	- mean deviation in the last sensing event.
β	- the weight of the current deviation $\|\bar{X}_i - S\|$.
ΔT_{next}	- the time interval until the next sensing event.
$\Delta T_{min} \| \Delta T_{max}$	- upper and lower ΔT bounds.
$getScale()$	- function ruling the scheme's reactiveness.
$\epsilon_{min} \| \epsilon_{max}$	- scale reactiveness boundaries.

Although not represented in Algorithms 1 and 2, e-LiteSense also provides a critical operation mode which is triggered if a threshold value is observed in the measurements. In this operation mode, sensing events are performed at the highest frequency regardless the sensor's battery level. Monitoring human vital signs or sensitive environments are examples in which such feature is essential, even though redundant batteries are in place.

IV. Evaluation Results

This section presents the proof-of-concept for e-LiteSense scheme by assessing its ability in reducing energy consumption while providing accurate measurements about the observed phenomena. In addition, the analysis addresses the impact of different algorithm settings on measurement accuracy and energy burden, in particular, the impact of reactivity (ϵ_{min} and ϵ_{max}) and time boundaries (ΔT_{min} and ΔT_{max}).

A. Methodology of tests

In order to evaluate the performance of e-LiteSense, a software-based prototype built according to TelosB motes' specifications [5] was implemented, allowing to assess a fine-grain power profile of commercial sensors running the proposed scheme.

Measurements accuracy and energy consumption are then evaluated resorting to real datasets previously collected and publicly available[1,2]. These datasets consist of: (*i*) an outdoor sensor measuring environmental temperature along six hours; (*ii*) an outdoor sensor measuring environmental humidity along six hours; and (*iii*) an indoor sensor measuring environmental temperature along four years, located in Luxembourg City.

In the first two test scenarios, the sensors were exposed to a steam of hot water to spark humidity and temperature levels in order to assess e-LiteSense's accuracy in identifying sudden changes in the observed parameters.

The accuracy evaluation is performed through the estimation of the *correlation coefficient* and the *Mean Squared Error* (MSE) between the measured values and the real phenomena dynamics. These metrics are commonly used to evaluate the accuracy of sampling-based estimators.

The parameterisation of tests, discussed in Section IV-B, considers $\Delta T_{min} = 5$ and $\Delta T_{max} = 45$, meaning that, at least, one sensing event occurs every 40 seconds for scenarios (*i*) and (*ii*) and every 40 minutes for scenario (*iii*). The reactivity factor is defined as $\epsilon_{min} = 0.05$ and $\epsilon_{max} = 0.30$, meaning that ΔT_{next} will vary between 5% and 30% in each cycle. The overall impact of adjusting these parameters is further discussed in Section IV-C. In addition, the weight of the last observations defining $\bar{X}_{i-1}$ and $\bar{V}_{i-1}$ are $\alpha = 0.7$ and $\beta = 0.7$, respectively.

[1]http://issnip.unimelb.edu.au/research_program/downloads
[2]https://doi.org/10.4121/uuid:284697ee-9dbe-4f63-b4fb-d5e8e3bc4606

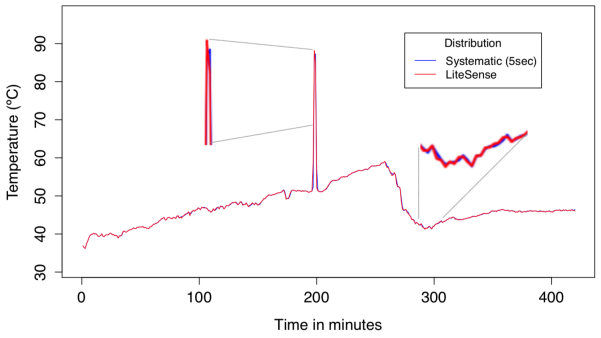


Fig. 2. Measurements for the outdoor temperature sensor

TABLE II
Measurement accuracy for different scenarios

Luxembourg		Temperature		Humidity	
Correlation	Error	Correlation	Error	Correlation	Error
0.99669	0.00653	0.99968	0.02078	0.99992	0.00142

B. Evaluating e-LiteSense performance

Regarding the ability of producing accurate measurements in diverse application scenarios, Figure 2 presents the distribution of sensed values along the tests for the temperature sensor. The almost complete overlap of e-LiteSense and original dataset (systematically sensed every 5 seconds), even when the data series exhibit sudden variability, is corroborated by the high correlation coefficient and low MSE presented in Table II. It is important to highlight that such high accuracy is achieved reducing the number of sensing events in 54% (from 5041 in the original dataset to 2800) through e-LiteSense. Similar performance is achieved for all datasets, which demonstrates the effectiveness of e-LiteSense in self-adjusting the sensing frequency accordingly towards the observed parameter dynamics.

Designed with the aim of extending the sensor's lifetime compared to the predominantly deployed scheme, *i.e.*, systematic sensing, the self-adaptiveness nature of e-LiteSense also achieves high efficiency while optimising energy consumption. As illustrated in Figure 3, for the Luxembourg scenario, e-LiteSense has extended the sensor operational lifetime in approximately 5 times when compared with the systematic approach currently under operation in this environment, namely 3 minutes. Even compared with a more conservative systematic scenario, *i.e.*, one sensing event every 10 minutes, a sensor running an e-LiteSense instance would extend its operation in around 1 year after a sensor with systematic sensing completely depleted its battery.

As detailed in Section III-B, e-LiteSense adjusts the reactivity of the sensing frequency according to the sensor's residual battery level. In this way, Figure 4 presents the impact of such behaviour on the measurements accuracy along the battery consumption stages. It represents the dispersion of temperature per minute measured through the proposed scheme in com-

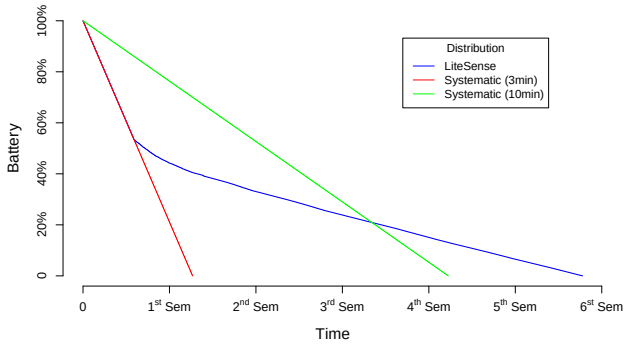
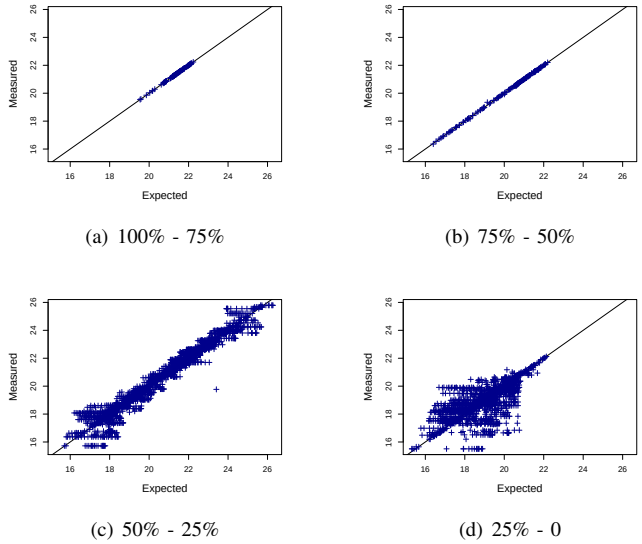


Fig. 3. Power consumption for Luxembourg sensor



(a) 100% - 75%

(b) 75% - 50%

(c) 50% - 25%

(d) 25% - 0

Fig. 4. Dispersion of observed values for distinct battery levels

parison to the original dataset for the Luxembourg scenario. According to this analysis, as the battery is depleted, measurement error increases. Therefore, balancing the extension of the sensor lifetime with the tolerated error for a specific application is relevant for configuring the proposed scheme, *i.e.*, reactivity and time boundaries, discussed in Section IV-C.

C. Analysing the algorithm reactivity and time boundaries

Following the design goals discussed in Section III, e-LiteSense scheme provides the flexibility of adjusting the algorithm reactivity and ΔT boundaries in order to foster its deployment in multiple scenarios and applications. This parameterization might be defined beforehand for a specific scenario or adjusted dynamically by an external controller (in response to the underlying application requirements or to the values collected in different sensors).

In this way, Table III presents the impact of adjusting the scheme reactivity, *i.e.*, ϵ_{min} and ϵ_{max} for all datasets. These tests consider $\Delta T_{min} = 5$ and $\Delta T_{min} = 45$ as fixed values. As observed, although varying slightly, the overall accuracy is kept high for the considered scenarios. In these cases, the major impact observed is the energy consumed, which vary approximately 7% for all scenarios, in average.

TABLE III
IMPACT OF ADJUSTING THE ALGORITHM REACTIVITY

	Luxembourg			Temperature			Humidity		
ϵ_{min}:ϵ_{max}	Correlation	Error	Battery	Correlation	Error	Battery	Correlation	Error	Battery
0.05:0.3	0.998	0.011	78%	0.999	0.025	55%	0.999	0.003	54%
0.05:0.5	0.999	0.005	79%	0.999	0.060	57%	0.999	0.002	58%
0.05:0.7	0.999	0.003	81%	0.998	0.085	59%	0.999	0.003	59%
0.05:1.0	0.999	0.002	84%	0.999	0.020	61%	0.999	0.001	62%

Similarly, Table IV presents the impact of adjusting e-LiteSense's time boundaries on measurement accuracy and energy consumption. In this case, the most conservative scenario, (*i.e.*, $\Delta T_{min} = 5$ and $\Delta T_{max} = 60$) means that, at least one sensing event occurs every minute[3]. On the other hand, the most eager scenario, (*i.e.*, $\Delta T_{min} = 5$ and $\Delta T_{max} = 15$), means that, at least four sensing events occur every minute. For all setups, the maximum frequency is eleven sensing events per minute and the reactivity is configured as $\epsilon_{min} = 0.05$ and $\epsilon_{max} = 0.3$

As observed for the reactivity adjustments, in the considered scenarios, tuning the time boundaries does not affect the high accuracy of measurements significantly. Again, the energy consumption is the main affected aspect, varying around 20% for the temperature and humidity datasets.

TABLE IV
THE IMPACT OF ADJUSTING TIME BOUNDARIES

	Luxembourg			Temperature			Humidity		
ΔT_{min}:ΔT_{max}	Correlation	Error	Battery	Correlation	Error	Battery	Correlation	Error	Battery
5:15	0.999	0.0001	100%	0.999	0.010	71%	0.999	0.0008	73%
5:30	0.999	0.0002	95%	0.999	0.012	62%	0.999	0.0009	61%
5:45	0.999	0.0009	81%	0.999	0.020	55%	0.999	0.001	57%
5:60	0.999	0.0032	74%	0.999	0.025	55%	0.999	0.003	54%

V. CONCLUSIONS

The deployment and operation of WSNs, very dependent on the type of applications and user services being supported, is commonly conditioned by the amount of available resources (energy, processing and transmission capacity, etc.). To mitigate these problems, particularly relevant for demanding data gathering processes, this paper has proposed e-LiteSense - a self-adaptive, energy-aware sensing scheme for WSNs. The present study evinces that this scheme is able to self-adjust data gathering to a specific WSN context, capturing accurately the behaviour of physical parameters of interest. Relying on low-complexity rules, e-LiteSense is able to auto-regulate the sensing frequency according to the parameters' variability and energy levels. Based on real datasets, the results showed that e-LiteSense was able to measure accurately the parameters under observation while reducing significantly the number of sensing events, evincing its ability to improve the trade-off between accuracy and WSN lifetime.

[3]For the Luxembourg scenario, it means, at least, one sensing event every hour. The same time scale is applied for all tests in such dataset.

Acknowledgments - ♢ This research work has received funding from European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505. * This work has also been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT within the Project Scope: UID/CEC/00319/2013.

REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: A Survey," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 38, no. 4, pp. 393–422, 2002.

[2] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, oct 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1389128610001568

[3] E. P. K. Gilbert, B. Kaliaperumal, and E. B. Rajsingh, "Research Issues in Wireless Sensor Network Applications: A Survey," *International Journal of Information and Electronics Engineering*, vol. 2, no. 5, pp. 702–706, 2012.

[4] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella, "Energy conservation in wireless sensor networks: A survey," *Ad Hoc Networks*, vol. 7, no. 3, pp. 537–568, may 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870508000954

[5] C. Antonopoulos, A. Prayati, T. Stoyanova, C. Koulamas, and G. Papadopoulos, "Experimental evaluation of a WSN platform power consumption," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, may 2009, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5161185

[6] F. Kerasiotis, A. Prayati, C. Antonopoulos, C. Koulamas, and G. Papadopoulos, "Battery Lifetime Prediction Model for a WSN Platform," in *2010 Fourth International Conference on Sensor Technologies and Applications*. IEEE, jul 2010, pp. 525–530. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5558125http://ieeexplore.ieee.org/document/5558125/

[7] H. Yetgin, K. T. K. Cheung, M. El-Hajjar, and L. H. Hanzo, "A survey of network lifetime maximization techniques in wireless sensor networks," *IEEE Communications Surveys Tutorials*, vol. 19, no. 2, pp. 828–854, Secondquarter 2017.

[8] J. Yang, X. Wu, and J. Wu, "Adaptive sensing scheduling for energy harvesting sensors with finite battery," in *2015 IEEE International Conference on Communications (ICC)*, jun 2015, pp. 98–103.

[9] G. Quer, R. Masiero, G. Pillonetto, M. Rossi, and M. Zorzi, "Sensing, Compression, and Recovery for WSNs: Sparse Signal Modeling and Monitoring Framework," *IEEE Transactions on Wireless Communications*, vol. 11, no. 10, pp. 3447–3461, oct 2012.

[10] R. M. Castro and E. Tanczos, "Adaptive Sensing for Estimation of Structured Sparse Signals," *IEEE Transactions on Information Theory*, vol. 61, no. 4, pp. 2060–2080, apr 2015.

[11] C. T. Chou, R. Rana, and W. Hu, "Energy efficient information collection in wireless sensor networks using adaptive compressive sensing," in *2009 IEEE 34th Conference on Local Computer Networks*, oct 2009, pp. 443–450.

[12] V. Jacobson, "Congestion Avoidance and Control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 314–329. [Online]. Available: http://doi.acm.org/10.1145/52324.52356