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:	January 15, 2019
Actual submission date:	January 15, 2019
Lead contributor:	NOVA
Revision:	2.0
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
01/03/2018	0.0	File creation	NOVA
06/07/2018	1.0	First submitted version	NOVA
01/11/2018	1.1	Generated new version to be edited for re-submission	NOVA
13/01/2019	2.0	Revision is complete and ready for submission	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.

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. This deliverable builds upon the results reported in Deliverable 5.1 (D5.1) [12], and presents the progress achieved in the last five months of the project.

The main results reported in this deliverable are as follows:

- We present a preliminary study regarding the different types of devices that form the edge spectrum. We further explore how these devices differ among them and how can they be better leveraged to support edge-enabled applications. We complement this effort by considering examples of applications that can benefit from different devices along the edge spectrum, particularly focusing on user-centric applications.

- We report on the evolution of the software support for the GRiSP platform, which includes better support for executing Erlang applications, and improved drivers that not only offer additional support for (relevant) sensors, but also increases the overall stability of the platform.

- We report on the evolution of the Yggdrasil framework, which includes: a new interface to design protocols that allows programmers to only focus on event handlers; a new execution model that enables multiple protocols to share a single execution thread; and tools to assist in the experimental evaluation of protocols and applications using the framework on realistic settings (which are required to conduct experimental assessments in the context of WP7).

- We present MiRAge, a new aggregation protocol for wireless AdHoc environments. This protocol is suitable for commodity devices, and tackles the more complex variant of aggregation where input values being aggregated can change independently across different devices in unpredictable fashion.

Software Artefacts

Similarly to Deliverable 5.1, we report on software artefacts 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 artefacts 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[1].

Completion of WP5 Goals and Project Milestones

This deliverable reports the work conducted in the context of WP5 towards achieving the following goals of the work package:

Efficient support of aggregation-based computations in light edge scenarios. This report present MiRAge, a novel distributed protocol to solve the continuous aggregation problem in the context of wireless AdHoc networks. As discussed further ahead MiRAge enriches the current state of the art by tacking a variant of the aggregation problem that naturally handles the independent variation of input values across nodes. Additionally, MiRAge relies on a fault-tolerant tree-based topology that is naturally managed by the evolution of the aggregation process. This concludes the primary efforts of the Lightkone consortium on providing efficient support for aggregation-based computations.

Efficient support of generic computations in light edge scenarios. This goal has been pursued by studying how different devices in the edge spectrum can support edge-enabled applications. It has also been pursued while evolving the design and implementation of both the Yggdrasil framework and the software stack of the GRiSP board. Considering the Lasp and Legion frameworks previously presented in D5.1, we consider that this goal has been mostly achieved in the context of the project although, additional effort will be invested in integrating different frameworks, and also on expanding the Yggdrasil framework to operate on wired IP networks.

In relation to milestone ***MS3: Light edge applications are successful*** on month 18 of the project, which entails the development of support for the industrial use cases that focus on the light edge, the fundamental support to develop these applications has been achieved by the project through the set of frameworks and innovations presented in this deliverable, and previously on D5.1. We note that there are demonstrators built on top of all frameworks, that validate their design and implementation. In order to fully support industrial use cases we have to enrich the Yggdrasil framework with support for operation on wired networks (an effort that is currently ongoing) and devise mechanisms to enable the inter-operation among some of these frameworks. We consider the milestone to be 80% complete.

Summary of Deliverable Revision

This deliverable has been revised since its original submission to incorporate comments and modifications requested by the European Commission Reviewers. The main changes made to the deliverable are as follows:

- The structure of the deliverable was completely revised, in particular we have: *i*) discussed the state of the art and how new results of the work package improve on

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

that state of the art; and *ii*) dedicated a section for dissemination activities carried in the context of this work package.

- We have provided some clarification on how the presented solutions work together to provide infrastructure support for the light edge.
- We have contextualized the results reported in this deliverable in relation to the Lightkone Reference Architecture (LiRA).
- We have extended the state of the art discussion to refer to additional relevant works, and to clarify the novelty of the work presented here.
- We have contextualized the results presented here in regards to the goals of the work package and Lightkone project milestones (and quatified progress of the work package work).
- We have provided preliminary data on the resource consumption of Yggdrasil.
- We have completely rewritten the exploitation plan for the (revised) Gluk use case.

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)* [47], *fog computing* [38], and *mist computing* [4]. 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* [2, 4][2]. 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 devices, and how they can actively support different forms of computation at the edge.

To take full 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 an execution environment and adequate packaging of such components. General purpose computations delegated to edge resources may potentially need additional data sources, other than the ones that already exist locally at 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 artefacts and systems in the edge. To this end, we continue 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 AdHoc 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 D5.1 [12]) that aims to simplify the development of distributed protocols and applications in this extreme edge scenario. In the last few

[2]We further discuss this forms of computation in Section 3.2 (c)

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 Units (CPUs)) 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. Finally, we have conducted a preliminary assessment of resource consumption by Yggdrasil, that we briefly mention here for completeness (additional details can be found in D7.1, produced by WP7).

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 [12]), particularly in its support software stack. Improvements performed in the platform recently improve its robustness while also providing new functionality.

Additionally, we have started a line of work to combine the results that were previously reported in [12]. 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 [12] 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 AdHoc environments. The design of this protocol exploits some design principles of *hybrid gossip* [19, 32] 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, were crucial for the development of MiRAge.

2.1 Structure of the Deliverable

The remainder of the report is structured as follows:

Section 3 discusses the work plan and goals of WP5 of the Lightkone project, which is dedicated to address the challenges of light edge scenarios, and presents the main results achieved by the Lightkone consortium between months 14 and 18 of the project, explaining their relevance in the LiRA. We further elaborate on future planned activities for the work package and quantify the current progress achieved so far.

Section 4 summarizes the software artefacts that are an integral part of this deliverable.

Section 5 expands the discussion on the state of the art in D5.1, by considering the results reported here.

Section 6 discusses additional exploratory research work that is currently being pursued by the Lightkone consortium in the context of WP5. While these exploratory works might not integrate directly with the LiRA, they are essential research and development efforts that align with the overall goals of WP5.

Section 7 lists the publications produced in the context of WP5 for the reported period and discusses dissemination activities carried out by the project consortium related with the activities of WP5.

Section 8 discusses the relationship of the results produced by WP5 with the work being conducted in the context of other work packages.

Section 9 reports on current exploitation plans to apply results achieved in the context of WP5 to the use cases of industrial partners.

Section 10 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 [12]. Some of these results directly build upon previous results reported there.

3 Progress and Plan

3.1 Plan

Work package 5 of the Lightkone project is focused on providing adequate infrastructure support (in the form of frameworks and distributed protocols) to address the inherent challenges of edge-enabled applications that leverage on computational resources in the light edge.

As discussed in Deliverable 5.1 [12], the activities of this work package are organized around three main research and development tasks, which can be summarized as: *i*) infrastructure support for aggregation in edge computing; *ii*) generic edge computing; and *iii*) self management and security in edge computing. All of this tasks naturally have their emphasis on the light edge, which is characterized by large numbers of application components interacting among themselves, in a mostly independent fashion of components in the heavy edge (e.g., data centers). Naturally, as in any research and innovation activities, the work conducted in this work package is not fully contained on the goals of these tasks. Instead, the presented work advances the state of the art towards achieving these goals.

The results presented in this deliverable represent the final results of the Lightkone consortium on supporting data aggregation, particularly focused on a challenging edge scenarios of large number of devices operating over wireless AdHoc networks. This work also explores mechanisms that can be leveraged to have self management properties in the light edge. The work presented here also advances on the capacity for supporting generic edge computing in the light edge, by studying the capabilities of different edge devices and exploring how different devices can be exploited, and combined, to support novel edge-enabled applications. We further consider new applications domains where edge computing can be beneficial.

As reported in D5.1, we have continued the efforts in developing some of the frameworks that have resulted from research and development efforts of WP5. In particular, we have refactored and further enriched the Yggdrasil framework with a new execution model, that will allow the framework to have more stable behavior in devices with limited CPU capacity; and designed and implemented the first version of a tool that simplifies the use of Yggdrasil for experimental assessment of distributed protocols' performance. The software stack of the GRiSP platform was enriched with additional drivers, to support more sensors, and the software was improved for additional stability.

The remaining work to be tackled in the context of this work package is to devise mechanisms to allow the inter-operation among the different innovations produced by WP5 and the remaining technical work packages (Work Package 3, Work Package 4, and Work Package 6). In addition, we will start the implementation work toward the construction of the use-case demonstrators that have a stronger emphasis on the light edge: Gluk's *Self Sufficient precision agriculture management for Irrigation*, UPC's *Guifi.net Network monitoring*, and Peer Stritzinger's *RFID-powered conveyor*.

3.2 Progress

In this Section we report the progress made by WP5 in relation to building support for edge-enabled applications in light edge scenarios. We will start by discussing the work plan of WP5. We note that a complete discussion of the relevant state of the art and innovation achieved by the results is presented further ahead in Section 5.

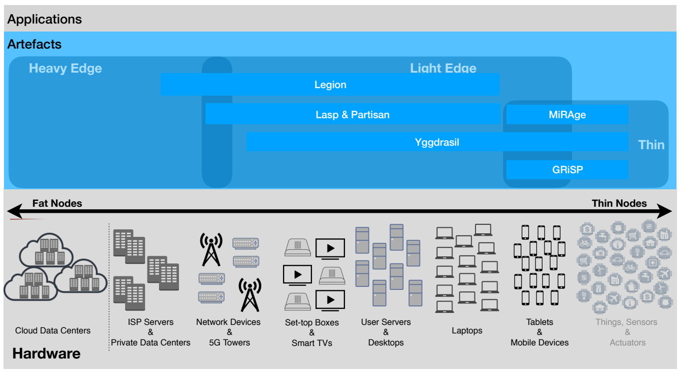

Figure 3.1: Relation of Results with Lightkone Reference Architecture

(a) Overview and Relation between Results

There are a large number of device types that can materialize the light edge of a system, by supporting the execution of (different) application components. The large range of devices is visually captured in Figure 3.1, and detailed further ahead in the text.

The set of results and innovations produced by work package 5, provide fundamental abstractions and essential infrastructure support for building applications that take advantage of such computational and storage resources. Many of these results have been previously introduced in D5.1 [12] and some of them have been meanwhile evolved and improved. We now provide an overview of the results and how they relate to each other in supporting edge-enabled applications[3].

Moving from the end of the spectrum dominated by small sensors, actuators, and things (following the nomenclature of IoT), we have developed the GRiSP platform, which is composed of a board (that was already in development at the start of the Lightkone project) and an accompanying software stack based on RTEMS, support for the Erlang VM, and drivers to allow the operation of multiple sensors. This device allows us to have a specialized embedded system whose capacity (i.e., CPU and memory) fall between that of small sensors and micro-computers, allowing us to delegate to them some edge-computations that are very close to the location producing (and potentially consuming) data, with lower cost. Additionally, the support to run Erlang directly on the bare metal, opens the way to enable executing some of the other innovations produced by Lightkone (e.g., Lasp and Antidote) at this point of the edge spectrum.

We note that a common computational task that can be executed at the very edge of a system, and significantly contribute to local decisions and alleviate the pressure produced

[3]For completeness, we also briefly discuss results that are only referred in [12].

by the data deluge phenomenon on heavy edge infrastructures, is data aggregation. To provide an effective way to aggregate data that is generated, and changes over time, at the edge of the system, particularly among devices that interact through wireless AdHoc networks, we have designed and implemented MiRAge, a novel aggregation distributed protocol. MiRAge relies on a fault-tolerant tree-based topology that is self managed by the protocol. This protocol can be easily integrated within solutions that require precise data aggregation in such scenarios.

Yggdrasil allows the construction of efficient distributed protocols and applications. Yggdrasil was written in C as a way to allow it to be executed in several different devices with varied capabilities. This can range from small integrated systems (such as GRiSP) and micro-computers (such as Raspberry Pi) to other user devices, such as laptops and desktops, and potentially switches/routers. Yggdrasil allows fast prototyping and evaluation of distributed protocols, offering a programming API that is similar to the usual strategies to specify such protocols. We also expect to be able to run a small subset of the Yggdrasil framework to support data acquisition and filtering on devices with less capacity than GRiSP, which we are currently trying to validate. Additionally, Yggdrasil currently only supports communication through WiFi (at the machine layer). We are now generalizing Yggdrasil to support communication at the IP layer and for wired networks. We can further extend Yggdrasil to support other wireless communications standards such as Zigbee [45] and 6LoWPAN [42] if necessary for implementing demonstrators of the industrial use-cases.

Lasp offers a computational model based on computations over CRDTs. Lasp, and its companion membership module Partisan, can operate across multiple user devices, including laptops, desktops, and even some network devices that have the capability of running Erlang applications. Furthermore, Lasp allows easy integration with heavy edge components, by simply executing a Lasp instance in one (or a small set of) such components.

Legion allows to imbue web applications, that represent a significant fraction of popular applications nowadays, with edge computing capabilities. This is achieved by enabling clients, in a mostly transparent way when considering some web application backends, to replicate portions of the applications state locally (in the form of CRDTs). Legion allows clients to operate over their local replicas and to directly synchronize such replicas directly, without the direct intervention of a heavy edge centralized component. Notwithstanding, Legion integrates naturally with heavy edge components. We note that we have successfully run mobile web clients using Legion, however, this framework operates at the application level, that is independent of Mobile Edge Computing (MEC) architectures [48, 53].

(b) Relation with Lightkone Reference Architecture

The LiRA presents the vision of the Lightkone consortium on how to build applications that take advantage of the edge computing paradigm, considering a wide spectrum of devices. We note that the full discussion on the LiRA is presented in deliverables D3.1 and D3.2 produced in the context of WP3. The work conducted in the context of WP5 is both an integral part of the LiRA and also contributes to a better understanding of the edge spectrum model.

In particular, the relations of the artefacts produced by WP5 to the LiRA are represented visually in Figure 3.1. The diagram demonstrates how the different innovations provide support for edge-enabled applications across different points of the edge spectrum.

The main component of Legion executes in browsers as a JavaScript library that uses WebRTC. It can operate on Tablets & Mobile Devices, Laptops, and Users Desktops. It also contains components that operate in the heavy edge (more centralized components) and can integrate with backends such as Google Real Time API.

Lasp & Partisan are Erlang artefacts that can run in almost every device of the light edge (maybe with the exception of Mobile and Tablets). It can also have instances executing on the heavy edge, which allows for easier integration between components in both of these contexts.

Yggdrasil is a framework written in C that can operate across many different devices (this includes devices such as GRiSP through the use of RTEMS, something in which we are currently working). Yggdrasil supports the execution of distributed protocols and applications in such devices. Currently the main limitation of Yggdrasil is that it only supports communication through wireless AdHoc networks. This is an artificial limitation, and we are currently adding support for IP (and wired) networks. This make Yggdrasil a very lightweight support for executing distributed algorithms across a wide range of the edge spectrum.

MiRAge is a novel aggregation protocol for wireless AdHoc networks that is specialized to operate at the extreme of the edge spectrum. While GRiSP, is an integrated system that provides us the flexibility to be able to execute artefacts, written both in Erlang and in low level languages such as C, very close to the small things that produce (and potentially consume) data.

Finally, to better understand how to leverage different points of the edge spectrum to support different forms of computations of edge-enabled applications, we have been trying to characterize the different types of devices that compose the edge spectrum. This provides us interesting insights that can be leveraged in building the demonstrators of use cases in the following months of the project.

We now provide a complete description of the work conducted in the context of WP5 between the months 13 and 18 of the project.

(c) Generic Edge Computing Vision

As discussed in the previous deliverable of this work package [12], edge computing can take many forms. However, the support infrastructure for edge computing is yet to be clearly defined. Fog computing [10, 38, 62], 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 [2], the cloud, the fog servers, and the IoT devices. Mist computing, an evolution of the fog computing model that has been adopted by industry [4], 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

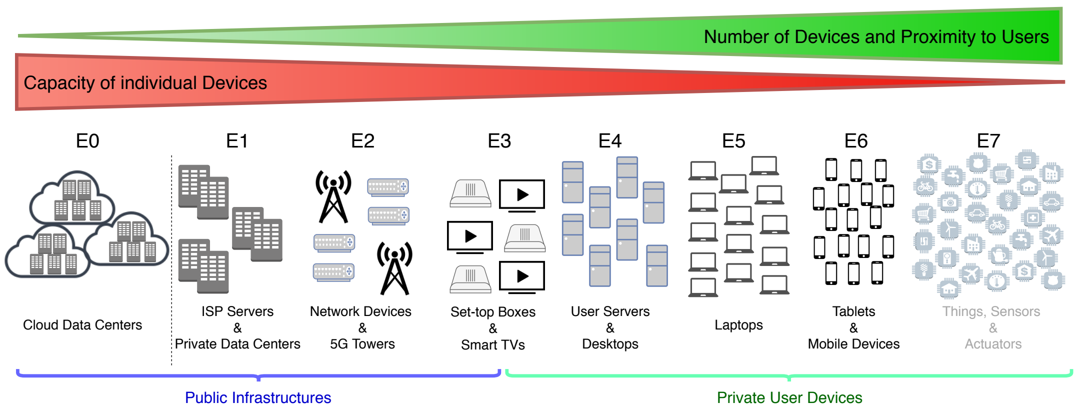
Number of Devices and Proximity to Users
Capacity of individual Devices
E0
E1
E2
E3
E4
E5
E6
E7
Cloud Data Centers
ISP Servers & Private Data Centers
Network Devices & 5G Towers
Set-top Boxes & Smart TVs
User Servers & Desktops
Laptops
Tablets & Mobile Devices
Things, Sensors & Actuators
Public Infrastructures
Private User Devices

Figure 3.2: Edge Components Spectrum

of computational devices that already exist in the edge. Furthermore, and as noted for instance in [2] and [4], 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 [2, 4]. 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 will empower the design of user-centric applications that promote additional and enriched interactivity among users and between users and their (intelligent) environment(s).

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 3.2 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[4] within an edge-enabled application (*user domain*).

[4]We refer to user in broad terms, meaning an entity that uses an edge-enabled application, either an

		E0 Cloud DCs	**E1** ISP Servers & Priv. DCs	**E2** Network Devices & 5G Towers	**E3** Set-top Boxes & Smart TVs
Characterization	**Capacity**	High	Large	Medium	Low
	Availability	High	High	High	Medium
	Domain	Application	Application	Application	User
Potential uses	**Storage**	Full State	(Large) Partial	(Limited) Partial	Caching
	Computation	Generic	Generic	Generic	Aggregation

		E4 Priv. Servers & Desktops	**E5** Laptops	**E6** Tablets & Mobiles	**E7** Things
Characterization	**Capacity**	Medium	Medium	Low	Varied
	Availability	Medium	Low	Medium	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

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 dimension 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, 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

end-user or a company.

perform *generic computations*.

E2: Network Devices & 5G Towers Network devices (such as routers, switches, and access points) that have processing power capabilities, offer the possibility to store some data and perform computations in networks. Additionally, 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 varied capacity between *low* and *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 in close vicinity.

E3: Set-top Boxes & Smart TVs Set-top boxes and the increasingly popular Smart TVs increasingly serve as an access portal to (distributed) applications and hence can be leveraged in the context of edge computing. Such devices have *low capacity* and *medium availability*, the later due to they being frequently shutdown by users. From the storage perspective, they offer *caching* capacity. These devices are in close vicinity of users, but not all of them are controlled by the user. In fact set-up boxes are usually under the control of providers, while Smart TVs are partially controlled by the user. However, for simplicity, we consider that they will operate at the *user domain*, as most of the computations are to benefit the end-user. We expect these devices, particularly set-top boxes, to be able to conduct aggregation of data for users in close vicinity or using the same operator. We note that for the particular case of the Lightkone project, these types of edge resources are not of special interest, and will not be used in the remainder of the project. We refer to them here for completeness.

E4: Private Servers & Desktops This is the first edge level of devices operating exclusively in the *user domain* that have higher capacity. 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 [2].

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. 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 *medium 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 [4].

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

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. We note that these case studies are not directly related with the use cases put forward by the industrial partners of the Lightkone project. However, these case studies serve as a reassurance that the vision described here is general enough and avoids it to be biased towards the Lightkone use cases.

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 [5]. Such game could enable direct interactions among players, (e.g., to trade in-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 and fights (among users that are registered friends in the application only) and does not support the remaining discussed interactions, with some evidence [3] 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 *Network Devices & 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

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

supported by *Private servers (E4)* being accessed through local networks (supported by *Network Devices & 5G Towers (E2)*) 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 [6], 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 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 indicator 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.

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 WP5 goals of fully exploiting 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 for edge-enabled 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 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 discuss further ahead in the deliverable (while also being presented in [12], is focused on

[6]A significative evolution of the Denmark Medical System briefly described in [55].

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.

(d) Improvements to the GRiSP Platform

The GRiSP platform has seen many improvements since the last deliverable of the work package [12].

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[17] and the newest version, 21[16]. They present many improvements to Erlang, including but not limited to performance improvements and features such as a new distribution Application Programming Interface (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.

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.

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 support).

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 Section 6.2, we have started to conduct efforts in integrating existing tools produced by the consortium into this platform.

(e) 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 earlier in this deliverable). This forces processes that run on these devices to rely on wireless AdHoc networks as a means of communication. However, and as we discussed in the previous deliverable [12], there is a significant lack of tools and support to design, implement, and test wireless AdHoc distributed protocols and applications.

In the period of six months reported here, 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.

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.

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 of 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 regarding the aggregation process).

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).

Execution Model of Protocols/Applications In this version of Yggdrasil we have added a new protocol that is automatically initialized along with the Dispatcher and Timer protocols by the framework. 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.

New Protocols in Yggdrasil

Distributed Aggregation Protocols. Since the writing of Deliverable 5.1 [12], we have leveraged Yggdrasil to implement some aggregation protocols found in the literature. These include Push Sum [28]; LiMoSense [18]; Distributed Random Grouping (DRG) [11]; Flow Updating [26]; and Generic Aggregation Protocol (GAP) [14]. 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.

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. [27]. 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[7]. 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.

Yggdrasil Control Process Besides serving as a framework to implement and execute distributed protocols and applications in wireless AdHoc 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. We ourselves were faced with this problem when conducting experimental assessment of the MiRAge aggregation protocol (described further ahead in Section (f)).

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

[7]The node that created the announce.

24 Raspberry Pi 3 - Model B [1]) 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 [12]. 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 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 [32] 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

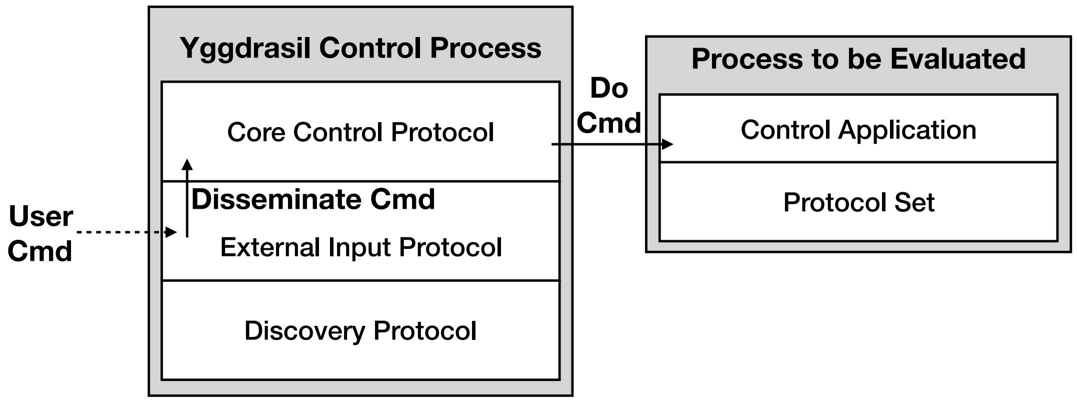

Figure 3.3: Execution mode where independent processes are spawned

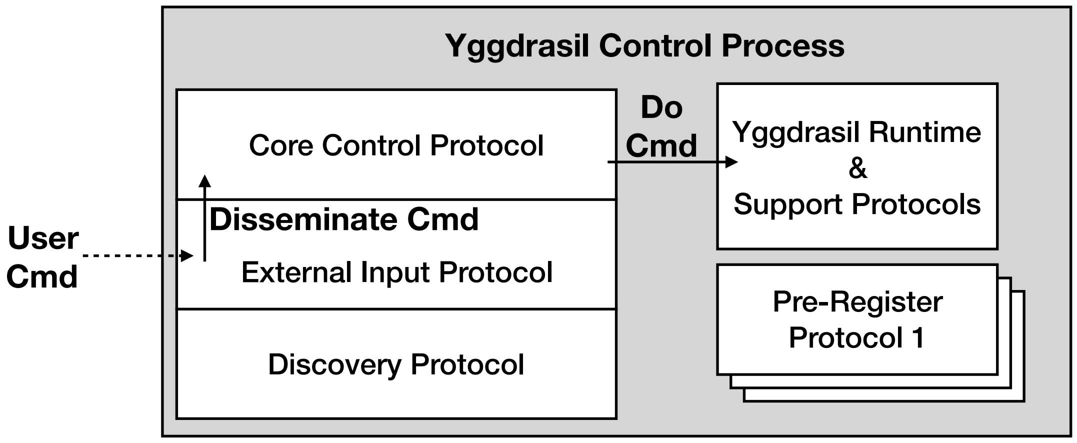

Figure 3.4: Execution mode where protocols are executed in the context of the Yggdrasil Executor (single process)

stable in the current prototype, and will be improved in the future. 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.

The first mode (illustrated in Figure 3.3) 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*[8]. 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 3.4) 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

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

crashes. This however, requires protocols to be written under the new programming interface presented previously in this document.

Summary of Resource Consumption Since Yggdrasil aims at supporting limited resource devices, we have started to conduct experimental assessment of the resource consumption of the framework. Detailed results (that use both Raspberry Pis and the GRiSP platform) are presented in D7.1 produced by WP7. To ensure that this deliverable is self contained we report the key numbers here. We start to node that due to the fact that these preliminary experiments were conducted using Raspberry Pis and the GRiSP platform, we have not measured energy consumption, since none of these platforms offered an easy way to conduct that measurement.

We have conducted measurements both on Raspberry Pis and the GRiSP platform that a Yggdrasil process, executing a user level protocol and one application in addition to the Yggdrasil core protocols. The applications sends a message every second. In terms of memory consumption, our experiment have shown it to be approximately 700 KiB[9] in both devices. Which implies that the memory footprint was below 1Mb. We note that for our experiments with GRiSP we have not taken into account the memory footprint of the RTEMS.

In terms of CPU consumption, and considering the a process with the same properties as pointed above, we have noticed that the CPU was mostly idle both when using Raspberry Pis or the GRiSP platform. In more detail, for an execution of slightly below 3 hours, the average CPU consumption in Raspberry Pis for the Yggdrasil process was 0.001%. We note that we used Raspberry Pis 3, that have a CPU with four cores, with a clock rate of $1.2Ghz$. In the case of the GRiSP board, that has CPU with a clock rate of 300Mhz, the CPU consumption was approximately 1%.

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 are integrating Yggdrasil with the GRiSP platform. In fact, we have a preliminary prototype of this integration that was achieved using the packaged version of RTEMS that is part of the software stack of GRiSP and the drivers of GRiSP. We expect to complete this integration very soon.

- 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.

[9] 1 KiB = 1,024 bytes.

- 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 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.

(f) 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 wireless AdHoc 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 infrastructure network. This complements the work previously presented in [12] regarding the study on aggregation protocols for wireless AdHoc networks, by proposing a novel protocol that is more adequate for relevant applications (such as monitoring) in the context of edge computing. MiRAge was published at the International Symposium on Reliable and Distributed Systems (SRDS) [13].

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

Motivation and Context Edge computing implies performing computations outside of the data center boundary, on devices located closer to clients of a system [51]. 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 [40] as well as IoT [7]. 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 [54], 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 [61]. In such a network, devices would interconnect forming a multi-hop network. This network can have routing capabilities (creating a mesh network [6]), 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 [25].

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* [5] 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.

Aggregation Overview Aggregation is an essential building block in many distributed systems [56]. 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 [5] 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 [29]. 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.

The Design of MiRAge We now discuss the design of MiRAge. We start by noting that our protocol is inspired in the design of GAP [14], 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 AdHoc mode. Each node is pre-configured to join a single AdHoc 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 [14], LiMoSense [18], Flow-Updating [26], 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, the management of the support spanning tree in MiRAge does not require any additional exchange of messages.

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: $Tress[T_{id}] \to$ 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 ($\Delta T$)

20: Upon Beacon do: //every $\Delta T$
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 $\setminus(id, \_, stat, \_, \_) \cup (id, val, stat, tid, tlvl)$
```

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, 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

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 $\cup$ ( $id, val, stat, tid, tlvl$ )
```

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.

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 wireless AdHoc 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 infrastructure 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 implemented 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.

3.3 Future Planning

Considering the results already achieved by the Lightkone consortium in the context of WP5, the next steps to be taken by the project are as follows:

- Improve the current design and implementation of the Yggdrasil framework, namely by adding support for IP wired networks and large messages (the last two are relevant for the application of Yggdrasil to the monitoring use case of the UPC partner).
- Conclude the integration between Yggdrasil and GRiSP.
- Explore additional integration of the different innovations developed within the context of WP5 and other work packages of the Lightkone project, motivated by the needs of the industrial use cases.
- Explore self-management mechanisms for managing both the placement of computational components and data across the edge spectrum.
- Address additional security challenges as motivated by needs of the industrial use cases.
- Implement demonstrators of use cases discussed in WP2 that are mostly focused on the light edge.

3.4 Quantification of Progress

In the following we provide quantification of the progress achieved by WP5 on the first 18 months of the project regarding its three main technical tasks, and the relevant project milestone.

Task 5.1: Infrastructure support for aggregation in edge computing The design of MiRAge has completed this task. We note that the design of MiRAge was made possible by leveraging on other results also generated in WP5, namely the Yggdrasil framework. Completion is fundamentally 100%.

Task 5.2: Generic edge computing There was some progress in this task, particularly through the study of different devices that compose the light edge. This has allowed us to understand how different devices can support different forms of computations. Combined with the results reported on D5.1, we consider this task to be completed up to 70%.

Task 5.3: Self management and Security in edge computing Regarding self-management, we have enriched the Yggdrasil framework with mechanisms to dynamically enable and disable protocols. This forms a basis for building additional self-managing properties into Yggdrasil. Considering the progress previously reported in D5.1, we consider this task to be completed up to 50%.

Relevant Project Milestones The project has a milestone that is fully dependent on the results and innovation being produced by WP5. This is a milestone for month 18, denoted *MS3: Light edge applications are successful.* This milestone requires the existence of adequate infrastructure support (in the form of frameworks and distributed protocols) that can be leveraged to build prototypes of the industrial use cases focused on light edge scenarios.

The progress of WP5 already has lead to the creation of fundamental support for the industrial use cases. Additionally, all frameworks and innovations produced in this context have been shown to be practical through a collection of demonstrators and prototypes. Considering the requirements of use cases, the work presented here covers 80% of the necessary efforts to fully achieve this milestone, lacking additional support at the level of the Yggdrasil framework (reported above) and additional effort on the integration among innovations. We consider however that the milestone has been successfully achieved, from a practical stand point.

4 Software Deliverables

This deliverable reports on the evolution of some of the software artefacts presented in Deliverable 5.1. These artefacts include: *i*) the new version of the Yggdrasil framework that includes, not only the modifications reported previously, but also implementations of a new set of distributed protocols, including an implementation of MiRAge and a prototype control protocol to perform experimental evaluations in wireless AdHoc networks; and *ii*) the new version of the software associated with the GRiSP platform.

The software artefacts are publicly available:

Yggdrasil framework: https://github.com/LightKone/Yggdrasil.git.

GRiSP platform: The related software for GRiSP is divided in two git repositories:

1. GRiSP Erlang Runtime Library: https://github.com/grisp/grisp.git.
2. Rebar plug-in for GRiSP: https://github.com/grisp/rebar3_grisp.git.

5 State of the Art Revision

In the following we discuss the state of the art related with each of the new results produced by WP5 and reported in this deliverable. We also identify and briefly discuss the main novelty and innovations introduced by our work in relation to the state of the art. We note that the state of the art for Yggdrasil and GRiSP is already reported in Deliverable 5.1 [12] and refer the reader to that document for that discussion.

5.1 State of the Art: Generic Edge Computing Vision

Edge computing can be defined, in very broad terms, as performing computations outside the boundaries of data centers [51]. Many approaches have already leveraged on some form of edge computing to improve the latency perceived by end-users, such as Content

Distribution Networks (CDNs) [63], or tapping into resources of client devices [47, 60], 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 [2] 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 [4] 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 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 [2] and [4] all of these proposed architectures are highly biased towards IoT applications. The proposed vision on edge computing differs significantly by considering devices that are already readily available at the edge. Furthermore, the vision put forward by the Lightkone consortium aims at finding strategies to leverage these different devices to extract different benefits for edge-enabled applications that go beyond the IoT domain.

Previous authors have already presented their visions for the future of Edge computing [51, 58], Fog computing [38, 44, 57], and IoT specific edge challenges [34]. These works however, present their visions with an emphasis on IoT applications. An exception to this is related with Mobile edge computing [36] 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 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.

A recent proposal, named osmotic computing [50, 59], has explored how to allow application components to migrate between cloud computing infrastructures and edge computing environments. To this end the authors exploit the use of microservice architectures [20, 21, 46] to allow individual components of applications (that exist as a microservice) to be dynamically migrated from cloud to the edge (and vice-versa) in reaction to varying operational conditions. While this approach is more generally applicable than the previously discussed ones, it treats computational and storage resources in the edge in a mostly uniform way, assuming that they have enough capability to execute a microservice (this is natural, since the authors were considering a edge model composed of localized servers). Instead, we focus on a more broad notion of edge computing by considering also user devices and network infrastructure. Additionally, we take a more fine grained approach, since we consider that different devices in the edge spectrum, can be leveraged for different purposes to improve edge-enabled applications.

5.2 State of the Art: MiRAge

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 aggregate function over a set of input values, where each input value is owned by a node in the system. Typical aggregate functions include *count*, *sum*, *average*, *minimum*, and *maximum*.

Not all of these aggregate 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 result is computed.

However, while computing the aggregate of a set of input values, one has to consider the possibility of values changing over time. To cope with this aspect we focus on protocols that can perform *continuous aggregation*. Continuous aggregation was initially coined in [5] 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 [29]. 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. Additionally, a continuous aggregation protocol should allow all nodes in the system to access the result while being fault-tolerant. To the best of our knowledge, MiRAge is the first protocol to solve the continuous aggregation problem meeting all of these criteria.

Distributed aggregation algorithms have been widely studied in the past, particularly in the context of sensor networks [11, 37, 43] and peer-to-peer systems [39]. In the context of sensor networks, most solutions strive to propagate partial aggregate 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 result, the supported aggregate functions, and how they deal with duplicated input values.

Sampling Techniques: 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 [39] as well as Randomized Reports [9]. Unfortunately, the precision of these solutions is highly sensitive to the distribution of input values among the nodes of the system, a phenomena that does not affects the correction of MiRAge.

Specialized Data Structures: Other distributed aggregation algorithms resort to somewhat complex data structures in order to collect more information from the system than simply computing an aggregate function. For instance, Q-Digest [52] and Equi-Depth [22] 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. Hence, inhibiting the computation of precise aggregate results from the computed distribution. Additionally, Q-Digest can only compute the distribution at a special sink node. Extrema Propagation [8], on the other hand, transforms the problem of computing an average in computing a minimum vector among all nodes in the system. Nodes iteratively exchange information, which is not sensitive to input duplicates however, the solution can only compute an approximate result.

Contrary to these aggregation schemes, MiRAge does not requires complex data structures, and is able to obtain high precision.

Iterative Approaches: Many solutions rely on having nodes continually exchange information among them to compute estimates of the aggregate result, that becomes increasingly precise with the number of iterations. In DRG [11] nodes iteratively form groups with a leader. The leader gathers the current estimates of the group members, computes a new estimate from those contributions and its own estimate, and propagates the new estimate to all elements of the group.

Push-Sum [28] 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 splits its local value and mass, transfers one half to a random peer, that incorporate them into its local value and mass, respectively. At any moment, a node can compute its local estimation of the aggregate result by dividing its current value by the locally stored mass, being only able to compute the average or sum/count aggregates. 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 [18] employs the same principles of Push-Sum, but stores the value and mass received, and sent to each peer, being then able to restore both in case of failure (through a compensation mechanism). Additionally in LiMoSense, 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. Unfortunately, imprecise and asymmetric fault-detection leads the protocol to apply unilateral corrections, which leads the protocol to compute completely incorrect aggregated values.

Flow-Updating [26] is another iterative approach where, contrary to Push-Sum and variants, nodes exchange and maintain flows to all their neighbors. Flows encode the difference between the local estimation of the aggregate result of a node and that of its neighbor. These 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 results of the average aggregate function, being unclear how to generalize this approach to other aggregate functions.

In relation to all these iterative approaches, MiRAge can obtain precise aggregate values at all nodes, and is able to naturally deal with faults, even with asymmetric and imprecise fault detection.

Tree Topologies: Finally, some aggregation protocols leverage on tree topologies to enable efficient aggregation, while avoiding the duplication of input values. The most well known of these solutions is the Tiny AGgregation (TAG) protocol [37], that was developed as part of TinyOs [33] 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. These, retransmit the message and set as their parent in the tree the node from whom they received 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 constrained sensors). Aggregation happens by having nodes report to their tree parent the result of a partial aggregate with their own input value and the partial aggregate results of all their 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 (DAG) [43] protocol, further improves the fault tolerance of TAG by building a multi-path tree rooted in the sink node. This is achieved by assigning a grandparent node to every node, and leveraging on these grandparent nodes to effectively compute partial aggregate values. This allows computations to proceed even if some nodes fail during the propagation of input values and partial aggregates towards the sink node.

The GAP [14] 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 is initially 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 received or sent). This set contains, for each other node, the current level of the node, 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 its local perception of the (current) tree topology (e.g., 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 aggregate values of all its children in the tree and its own input value. Unfortunately, GAP cannot tolerate the failure of the tree root.

All the solutions based on trees discussed above require the existence of a pre-defined and static 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 aggregate 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, which leads to the consumption of additional resources. MiRAge does not suffers from these effects, as it does not require a pre-defined sink node and all nodes naturally compute the aggregates as a result of the protocol execution.

6 Exploratory Work

In this Section we report on exploratory work that is being conducted by the Lightkone consortium and that is aligned with the overarching goals of Work Package 5 (WP5). The results and innovations presented here are not yet integrated within the LiRA, however they are important research efforts of the project that aim at further pushing the state of the art.

6.1 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.

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 Ph.D. student Pedro Ákos Costa, alongside multiple M.Sc. students. 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 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. This work can be seen as an evolution of osmotic computing [50, 59] albeit, further separating application logic from application state, and considering distributed schemes to manage application deployments at runtime.

Overview Our proposal is based on building an autonomic Microservice Architectures (MSA) that covers cloud and edge infrastructures. Application components, materialized in the form of individual microservices (and, potentially, their own storage

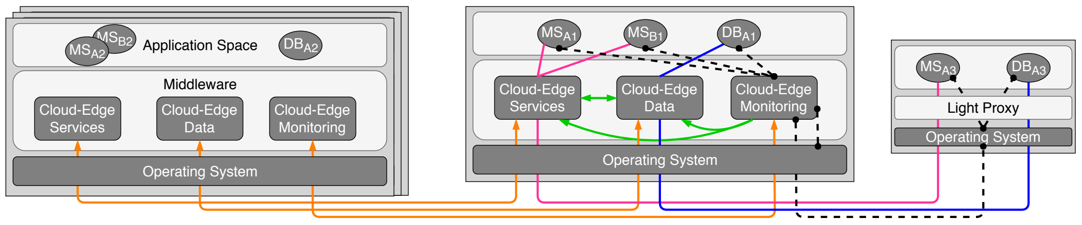

Figure 6.1: Autonomic Microservice Architecture

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 MSA is to identify which aspects of the application configuration can be autonomously managed, the information required to execute such reconfiguration, 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 different 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 6.1 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 [24, 30, 31]. 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 [31, 32]. 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 having 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.

6.2 Lasp Applications for Wireless Edge with GRiSP

As explained in 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 year 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 this deliverable, the software stack of GRiSP has been improved recently.

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

The goals of this work are three fold: *i*) to build a platform for edge computation; *ii*) to write applications on this platform; and *iii*) 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.

(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 demonstrator, Lasp is used both as a key/value store and to support computations. 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: *i*) as local computations done on each node (standard Erlang computations); or *ii*) as Lasp computations performed over data encoded in the form of CRDTs [49] 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.

6.3 Adaptive Sensing in Wireless Sensor Networks with LiteSense

(a) Context & Motivation

The multitude of Wireless Sensor Networks (WSNs) 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 3.2 (c)).

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 the 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 becomes, not only dependent on the observed variation in scalar physical quantities measured, but also on the perceived devices residual battery level over time.

7 Publications and Dissemination

7.1 Publications

Some of the results reported in this deliverable have been made public through the following publications. 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.

- 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. Technical Report.

- João Marco Silva, and Kalil Bispo. Flexible WSN Data Gathering through Energy-aware Adaptive Sensing. Proceedings of the IEEE International Conference on Smart Communications in Network Technologies (SaCoNeT), 2018.

7.2 Dissemination Activities

GRiSP Our showcase of GRiSP and its activities in the Lightkone project have made its rounds at various conferences and events. The dissemination efforts in 2018 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 [64].

At LambdaDays 2018, the platform was presented in two separated talks. One talk by Peer Stritzinger and Kilian Holzinger that focused on a prototype for functional reactive programming and talked about the process to approach hard real-time using Erlang/OTP and the GRiSP platform [23]. 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 [15].

Later in March 2018 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 [41] when developing home automation and robotics projects. At the sister conference CodeBEAM STO in Stockholm, Sweden. In the end of May 2018, Peer Stritzinger and Adam Lindberg presented the 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 [35].

Finally, we went back to Kraków in June 2018 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.

8 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 [12], 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 *Self Sufficient precision agriculture management for Irrigation* presented by GLUK. We discuss these applications further ahead in this document.

WP3: This work package focuses on devising the Lightkone Reference Architecture (LiRA) and 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. Naturally, the innovations exposed in the deliverable are integral part of the LiRA as documented in Deliverable 3.1 and 3.2.

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 on 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 Deliverable 4.2.

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).

9 Exploitation of Results by Industrial Partners

This Section briefly present existing plans for exploitation of the results generated so far by WP5 for some of the 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).

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

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

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 Yggdrasil framework developed within LightKone (and reported in this document and Deliverable 5.1 [12]) 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 adapting Yggdrasil to operate at the IP level (and wired networks) and then deploy 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.

9.2 GLUK

(a) Self Sufficient precision agriculture management for Irrigation

The self sufficient precision agriculture management for irrigation use case presented by Gluk described in deliverable D2.2, discusses the use of a set of devices to control, in a fully distributed way, and without resorting to control components in the heavy edge, the irrigation process of a farm (in particular, a citrus production farm). This system will be composed of devices, enriched with sensors and actuators, that will manage a set of irrigation paths that depart from a well (where water is extracted using a controllable electric pump).

The irrigation process will be controlled autonomously and in a (localized) coordinated fashion, as ensuring that only some zones of the farm are irrigated requires managing multiple actuators between the zone to be irrigated and the pump that extracts water. Additionally, to minimize installation costs, devices should rely on *zero touch configuration* where devices are simply deployed (or replaced individually in case of fault) and the system is able to determine the relative position of the device and start coordinating with neighboring nodes. Solutions for this use case must naturally cope with transient failures of both devices and communications links. Devices can be powered by solar panels, but this can still lead some devices to fail for a period of time. Optionally, these devices might be connected to a sink node, that can export some control information (for instance record activity) of individual devices.

This use case departs from normal sensor networks since it requires direct communication among nodes while at the same time, ensuring that the devices can operate correctly without any form of specialized configuration or management. Additionally, data being gathered by sensors regarding the humidity of the soil, which is used to take decisions regarding the control of the irrigation, should be spread among nodes in close vicinity such that errors in individual sensors can be compensated, and to ensure fault-tolerance.

(b) Exploitation of Yggdrasil and Lasp

There are multiple technologies and results from WP5 that can be exploited to build this use case. Particularly, the Yggdrasil framework and the MiRAge aggregation protocol are suitable solutions to materialize aspects related with the monitoring of humidity in the soil. MiRAge (or a simple variation of this protocol) can be leveraged to perform the aggregation of data across neighboring sensors. Additionally, Yggdrasil already offers features that can simplify the development of protocols for AdHoc networks that enable individual devices to sense the environment and take self-management configurations. Developing variations of distributed protocols developed in WP5 to implement this use case will be highly simplified by leveraging on the programming model of Yggdrasil.

This use case might also need to store some control information across nodes, and to perform additional forms of computation, this appears naturally as some actuators should be triggered with some amount of coordination. This forms of computations can be achieved through the use of Lasp. Lasp will provide simultaneously the capacity for storing configuration data (and potentially historical data sensed by sensors) and at the same time allow to replicate computations across multiple nodes.

Finally, to materialize the use case one can leverage on devices with the properties of GRiSP, where multiple sensors and actuators can be easily plugged and used. This as an additional advantage that we already have efforts (and preliminary results) on integrating Lasp with GRiSP and Yggdrasil with GRiSP, which are the fundamental innovations produced by the project that will be used in this use case.

The next steps to be taken in this context are: *i*) finish the integration of Yggdrasil, Lasp, and GRiSP; *ii*) leverage on the lessons learn during the development of MiRAge, and other protocols for wireless AdHoc networks to design a fundamental mechanism to enable the zero touch configuration of devices for this use case, and conduct measurements among neighboring devices; *iii*) build the decision control mechanisms for the use case, based on the Lasp framework. We will then conduct validation of the initial design in alignment with the evaluation activities conducted by WP7 and then use preliminary evaluation data to feedback to the development of the use case demonstrator.

10 Final Remarks and Future Directions

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 towards 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, more precisely 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 [12] 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.

The future work to be conducted in WP5 will focus on improving the integration between the innovations produced by this work package (and with other work packages) and start the implementation of the industrial use cases demonstrators that focus on the light edge. These efforts will require adapting and further evolving results presented in this deliverable and in D5.1 [12], as well as build variations of these results. Security issues, such as improving data privacy and data integrity as well as devising additional mechanisms to promote self management will also be tackled in the Future (also as part of the efforts towards the completion of Task 5.3, as discussed in D5.1). These efforts will mostly be guided by specific requirements of the industrial use cases (as discussed in D2.1 and D2.2). Implementations of the demonstrators will be fed to WP7 for early evaluation, and preliminary results will feedback to WP5 as to further improve those demonstrators as we approximate the end of the Lightkone project.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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*, 23(4):668–675, 4 2012.

[9] Mayank Bawa, Hector Garcia-Molina, Aristides Gionis, and Rajeev Motwani. Estimating aggregates on a peer-to-peer network. Technical Report 2003-24, Stanford InfoLab, 4 2003.

[10] 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.

[11] 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.

[12] LightKone Consortium. D5.1: infrastructure support for aggregation in edge computing (revised). https://github.com/LightKone/wp5-public/blob/master/deliverables/D5.1.pdf, Jan 2019.

[13] Pedro Ákos Costa and João Leitão. Practical continuous aggregation in wireless edge environments. In *37th IEEE International Symposium on Reliable Distributed Systems (SRDS 2018)*. IEEE, Oct 2018.

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

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

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

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

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

[19] 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.

[20] Martin Fowler. Microservices resource guide. https://martinfowler.com/microservices/, 2017.

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

[22] Maya Haridasan and Robbert van Renesse. Gossip-based distribution estimation in peer-to-peer networks. In *Proceedings of the 7th International Conference on Peer-to-peer Systems*, IPTPS'08, pages 13–13, Berkeley, CA, USA, 2008. USENIX Association.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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*, pages 115–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[34] 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.

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

[36] 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.

[37] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, December 2002.

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

[39] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi 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*, PODC '06, pages 123–132, Denver, Colorado, USA, 2006. ACM.

[40] 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.

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

[42] Gabriel Montenegro, Christian Schumacher, and Nandakishore Kushalnagar. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919, August 2007.

[43] 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)*, pages 8 pp.–299, 1 2006.

[44] 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.

[45] C. M. Ramya, M. Shanmugaraj, and R. Prabakaran. Study on zigbee technology. In *2011 3rd International Conference on Electronics Computer Technology*, volume 6, pages 297–301, 4 2011.

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

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

[48] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust. Mobile-edge computing architecture: The role of mec in the internet of things. *IEEE Consumer Electronics Magazine*, 5(4):84–91, Oct 2016.

[49] 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.

[50] Vishal Sharma, Kathiravan Srinivasan, Dushantha Nalin K. Jayakody, Omer F. Rana, and Ravinder Kumar. Managing service-heterogeneity using osmotic computing. *CoRR*, abs/1704.04213, 2017.

[51] 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.

[52] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249. ACM, 2004.

[53] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys Tutorials*, 19(3):1657–1681, thirdquarter 2017.

[54] 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.

[55] 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.

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

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

[58] 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.

[59] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer F. Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[60] Long Vu, Indranil Gupta, Klara Nahrstedt, and Jin 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, November 2010.

[61] 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.

[62] 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.

[63] Mingchen Zhao, Paarijaat Aditya, Ang Chen, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, Bill Wishon, and Miroslav 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.

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

A Description of Aggregation Protocols implemented in Yggdrasil

- Push Sum [28] is a well know aggregation protocol where nodes continuously exchange their internal state. The internal state of a node is composed of two parts: 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 (local) 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 [18] 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 [11] 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 [26] 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 [14] 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 AdHoc 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

This appendix briefly discusses the 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 3.2 (e)), 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 it 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 blocked 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 previous command. This allows the control topology to be recovered, for instance after the departure of join of a device.

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*
CDN *Content Distribution Network*
CPU *Central Processing Unit*
DAG *Directed Acyclic Graph*
D2.1 *Deliverable 2.1*
D2.2 *Deliverable 2.2*
D3.1 *Deliverable 3.1*
D3.2 *Deliverable 3.2*
D5.1 *Deliverable 5.1*
D7.1 *Deliverable 7.1*
DHCP *Dynamic Host Configuration Protocol*
DRG *Distributed Random Grouping*

GAP	*Generic Aggregation Protocol*
IoE	*Internet of Everything*
IoT	*Internet of Things*
IP	*Internet Protocol*
ISP	*Internet Service Provider*
LiRA	*Lightkone Reference Architecture*
MAC	*Media Access Control*
MEC	*Mobile Edge Computing*
MSA	*Microservice Architectures*
OS	*Operating System*
P2P	*Peer-to-Peer*
RTEMS	*Real-Time Executive for Multiprocessor Systems*
SSH	*Secure Shell*
SSL	*Secure Socket Layer*
TAG	*Tiny AGgregation*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
VM	*Virtual Machine*
WP2	*Work Package 2*
WP3	*Work Package 3*
WP4	*Work Package 4*
WP5	*Work Package 5*
WP6	*Work Package 6*
WP7	*Work Package 7*
WSN	*Wireless Sensor Network*