LIGHTKONE

Lightweight computation for networks at the edge

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

D5.1: Infrastructure Support for Aggregation in Edge Computing

Deliverable no.:	D5.1
Title:	Infrastructure Support for Aggregation in Edge Computing
Due date of deliverable:	January 15, 2019
Actual submission date:	January 15, 2019
Lead contributor:	NOVA
Revision:	2.0
Dissemination level:	PU
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
01/12/2017	0.1	1st draft with outline and ToC	NOVA
29/01/2018	0.5	1st complete draft of the deliverable	NOVA
09/02/2018	1.0	Deliverable is complete and ready for submission	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

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
Vasileios Karagiannis	NOVA
Peter Van Roy	UCL
Christopher Meiklejohn	UCL & IST/INESC-ID
Roger Pueyo Centelles	UPC
Carlos Baquero	INESC TEC
Giorgos Kostopoulos	GLUK

Contents

1 Executive summary **1**

2 Introduction **4**
2.1 Structure of the Deliverable 6

3 Progress and Plan **6**
3.1 Plan 7
3.2 Progress 8
(a) Overview and Relation between Results 8
(b) Relation with Lightkone Reference Architecture 10
(c) Yggdrasil framework and Aggregation in Wireless Settings 10
(d) Data Computation at the Edge with the Legion Framework 20
(e) Data Computation on Lasp 29
(f) GRiSP platform and hardware 33
3.3 Future Planning 36
3.4 Quantification of Progress 36

4 Software Deliverables **37**

5 State of the Art Revision **38**
5.1 State of the Art: Yggdrasil 38
5.2 State of the Art: Legion 39
5.3 State of the Art: Lasp 41
5.4 State of the Art: GRiSP 41

6 Exploratory Work **42**
6.1 Non Uniform CRDTs 42
(a) System Model & Relevant Concepts 42
(b) Non-Uniform Operation-based CRDTs 44
(c) Additionnal State of the Art Discussion 48
(d) Discussion 50
6.2 Enriching Consistency at the Edge 50
(a) System Model 51
(b) Architecture 51
(c) Enriching Consistency 52
(d) Additional State of the Art Discussion 54
(e) Discussion 54

7 Publications and Dissemination **55**
7.1 Publications 55
7.2 Dissemination Activities 55

8 Relationship of Results with Industrial Use Cases **57**
(a) Self Sufficient precision agriculture management for Irrigation . . 57
(b) Monitoring systems of the Guifi.net Community 58

9 Relationship with Results from other Work Packages 58

10 Final Remarks and Future Directions 59

A Pseudo-Code for the Aggregation Protocols 69

1 Executive summary

Moving computations towards the edge of distributed systems is an essential endeavor that ensures the continued growth and sustainability of large-scale distributed applications. Doing so, also enables the emergence of novel applications that leverage on the edge computing paradigm offering better and improved services to end users. We refer to such applications as edge-enabled applications.

The Lightkone consortium posits itself as a leading force in the development of new techniques and support to enable this new generation of edge-enabled applications. We expect edge-enabled applications to have components executing over a broad spectrum of edge scenarios. These scenarios range from data centers and nodes located in their vicinity (what we refer to as *heavy edge*), to locations closer to end users, or even in their devices (what we refer to as *light edge*). One of the end goals of the Lightkone project is to develop new methodologies, solutions, and tools that enable application developers to fully tap into the potential of the edge computing paradigm. Hence, new solutions will have to be developed to tackle challenges that arise at multiple points of this spectrum.

This report discusses the first steps of the Lightkone consortium in addressing the multitude of challenges related to the materialization of edge computing solutions for the light edge-end of the spectrum. In particular, challenges related to the inherent complexities of developing and executing available and correct edge-enabled applications with few dependencies on the core of the system (i.e., heavy edge).

The main results presented in this deliverable tackle two primary goals of WP5: *i*) supporting efficient aggregation computations on light edge scenarios, and *ii*) paving the way to support general-purpose computations on edge computing. Furthermore, the results presented here form key components to materialize the *Lightkone Reference Architecture* (LiRA) for edge-enabled applications as reported on deliverable 3.1, regarding the light edge and thin (i.e., IoT and sensors) layers of the considered edge spectrum. The key results presented in this deliverable can be summarized as follows:

- The Yggdrasil framework, that was designed to support the development of protocols and applications operating on wireless AdHoc networks. We have also designed and implemented of a set of exploratory aggregation protocols for this setting.
- The Legion framework, that aims at enriching web applications with edge computing features, namely through the replication and direct synchronization of relevant application state directly among clients.
- The Lasp framework, that allows to specify robust computations across large number of devices. These computations are defined in terms of operators that combine state encoded within CRDT data structures.
- The GRiSP Platform, an embedded device that offers computational and memory capacity between small sensors and micro-computers. GRiSP features a set of software packages that allow the execution of the Erlang VM directly on the bare metal.

Software Artefacts

In addition to the description of the main results achieved by the Lightkone consortium in the context of the WP5, we also report on software artefacts that were produced. These artefacts materialize some of these results, paving the way for building edge-enabled applications.

The software artefacts presented as part of this deliverable are the following: *i*) the Yggdrasil framework; *ii*) the Legion framework; *iii*) the Lasp framework; and finally, *iv*) the software to support the development of applications on the GRiSP board.

All software artefacts are available currently through the git repository of the work package.

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. We have tackled this goal in the context of Yggdrasil, where we have developed multiple (and simple) distributed aggregation protocols for wireless AdHoc networks. This allowed us to identify limitations in current state of the art, which will be explored in the future in the context of WP5.

Efficient support of generic computations in light edge scenarios. This goal has been pursued by the design and development of different frameworks that allow to specify different types of edge computations for different points of the edge spectrum (as discussed in D3.1 produced by WP3). Furthermore, the GRiSP development presented here will allow to perform additional computations very close to the thin layer of sensor and IoT devices at the end of the edge spectrum.

These goals are requirements to achieve 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. To achieve this milestone, we require: *i*) adequate infrastructure support for edge devices to be used in the implementation of these use cases; *ii*) distributed protocols to support the operation of applications in those contexts; *iii*) mechanisms to interact with applications components on the heavy edge; and *iv*) support for minimizing human management of these applications. Considering this, and the work reported in this deliverable, we argue that we have completed between 30% and 40% of the work necessary to achieve milestone *MS3*.

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*) explained the work package plan; *ii*) discusses the state of the art and how results of the work package improve on that state of the art; and *iii*) dedicated a section for dissemination activities carried in the context of this work package.

- We have provided some clarification on how the presented frameworks and 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.
- 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.

2 Introduction

Since its inception in 2005, the cloud computing paradigm has deeply impacted the design and implementation of user-facing distributed systems [81]. Nowadays, most services resort to cloud infrastructures to store both application and user data, perform complex computations over data, and provide services for users that are scattered throughout the world. This paradigm has lead to a rise in popularity of social network applications and mobile applications, among others. In the particular case of mobile applications, the cloud is able to circumvent the resource limitations of mobile devices, enabling the mobile devices to only execute a thin cache and presentation layer to users, while all application logic and data are handled by cloud infrastructures [36].

The cloud computing paradigm however, is not a panacea for building and executing reliable and efficient distributed applications. In fact, the use of cloud-based infrastructures presents significant downsides [35]. In particular, the use of cloud infrastructures has monetary costs for application operators, that not only grow with the effective amount of processing and stored data, but also with the amount of data transferred between (client) applications and the cloud infrastructure; furthermore, even cloud infrastructures have limits to their scalability and, under heavy usage, the latency experienced by users when interacting with applications can become too high. Finally, outsourcing data storage to the cloud comes with a significative risk for data privacy: on one hand, storing user data in the cloud implies that the user is releasing the control of its own data and, on the other hand, the cloud is a particularly attractive target for internet hackers due the potential high return for a successful attack, as demonstrated by incidents in the past [46, 58].

The rise in popularity of user-centric applications, such as the above mentioned social networks and mobile applications, associated with the rapid growth in popularity of the Internet-of-Things (IoT) applications has lead to a shift in the behavior and requirements of distributed applications. The most obvious of these changes is that data is now produced and consumed by clients whereas, in the past, many distributed applications had clients consuming the data that was mostly produced at the center of the system. Additionally, the volume of data produced by client devices, particularly in the context of IoT applications, has increased significantly, and is expected to continue to do so, as denoted by recent reports [28, 81]. This puts a significant burden on cloud infrastructures, that must be able to receive and store huge quantities of data generated by the clients and process it to provide useful information back to those clients, a burden that will become impossible to support in the near future [81]. This shift also exacerbates the downsides of the cloud computing paradigm discussed previously. Latency experienced by users and applications will tend to rise due to the time required to transfer large quantities of data. At the same time, in some application domains such as medical care IoT applications, the private nature of the data being manipulated by applications becomes more and more pronounced.

This has lead to the emergence of a new computing paradigm, usually called *edge computing*, where some, or most, data computations over user-generated data is moved beyond the data center boundaries towards the edge of the system, closer to the end user devices that both generate the data and consume the (processed) data. Such approach reduces the load on both cloud infrastructures and networks, while allowing data to be produced for end user consumption in a more expedite way. Furthermore, such an ap-

proach has the additional advantage of avoiding the transmission of private user data to an external infrastructure that lies beyond the user control.

The practicality of edge computing however, requires a complete rethinking on how to build and execute distributed applications. Applications have to be flexible enough to deal with a large number of heterogeneous devices that support the execution of different components of these applications. Devices themselves might need to evolve as to provide the adequate hardware (and associated software stacks) required to support general purpose edge computing applications. Furthermore, deciding where data should be sent and processed has to be handled at runtime, in order to take into consideration the operational environment of applications.

Additionally, computations themselves have to be specified by application programmers in a different way, enabling computations to operate over incomplete data sets, and avoiding strong synchronization between the devices that generate data, and those that support the execution of such computations, as to avoid the creation of bottlenecks in the system and to promote fault-tolerance and availability. Finally, data has to be protected, as to avoid its manipulation by unauthorized entities and protect the privacy of sensitive data.

When speaking about edge computing, one can envision multiple edge scenarios. These scenarios can range from the use of small data centers and dedicated infrastructure that are located beyond the data center boundary, and that can be responsible for managing data generated at a national or regional level. This is a particular edge scenario commonly referred by Fog Computing [23, 97]. Going farther away from the core of the network, one can find federation of resources (i.e., servers, services, sensors, and actuators) at the scale of a city (a scenario that falls within the scope of smart cities and sensor networks) or gridlets and cloudlets of small computational devices very close to end users. Finally, one can rely on end users' devices, such as laptops, desktops, or mobile devices to perform computations (a field usually called peer-to-peer computing or mobile edge computing for the particular case of mobile devices).

One of the end goals of the Lightkone project is to develop new methodologies, solutions, and tools that enable application developers to fully tap into the potential of the edge computing paradigm. To build effective and robust applications that fully leverage the potential of the edge (i.e., edge-enabled applications), one has to be able to manipulate data in all scenarios discussed above, with a high degree of independence from the particularities of the hardware or the execution environment where these computations are performed. The most fundamental data computations that can be tackled in this context are data aggregation primitives.

Data aggregation primitives, such as average, min, max, sum, and the generation of application-dependent data-collection summaries, are an essential step to pave the way for supporting general purpose computations in the edge. This is justified by two complementary observations. First, transmitting the whole data produced at the edge among devices, even if these devices are in close proximity, will result in a performance bottleneck, as connections between devices might be limited (consider for instance devices communicating through a wireless medium in an area with significant interference). Second, data aggregation are essential primitives to build distributed monitoring services, that can extract relevant metrics concerning the availability of devices and their free resources at runtime (e.g., computational, storage, energy, etc.). Moreover, these techniques can also be used to infer details regarding the current workload of running applications. This in-

formation is essential to empower edge-enabled computing platforms to make the correct runtime planning and decisions on how to orchestrate different components and computational tasks of an edge-enabled application.

2.1 Structure of the Deliverable

The remainder of this deliverable is structured as follows:

Section 3 discusses the work plan and goals of work package 5 (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 in the first 12 months of the project, explaining their relevance to instantiate the Lightkone Reference Architecture (LiRA) for edge computing applications and systems. We further elaborate on future planned activities for the work package and quantify the current progress achieved so far.

Section 4 provides guidelines on how to access and execute the software artifacts associated with the results presented in this deliverable.

Section 5 discusses the relation of the key results produced in the context of WP5 with the state of the art.

Section 6 discusses additional exploratory research work that has been produced by the Lightkone consortium in the context of WP5. While these exploratory works might not integrate directly with the LiRA, they are essential efforts to understand how to better allow the interactions between system components in the light edge and the heavy edge of applications.

Section 7 reports the scientific publications and dissemination activities produced in the context of WP5.

Section 8 discusses the relationship and applicability of the main results reported in this deliverable with the use case applications reported by the Lightkone consortium in deliverable D2.1.

Section 9 discusses the relationship of the contributions reported here with the work being conducted by the Lightkone consortium in the context of other work packages.

Section 10 concludes this report discussing the main achievements obtained by the project consortium in the context of WP5 and their relationship with relevant goals of the work package and milestones of the project. It further elaborates on future steps to be taken in the context of WP5 during the remaining time of the project.

3 Progress and Plan

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.

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. The work package is organized as three complementary research and innovation tasks, which for clarity of presentation are organized in sequential yearly activities.

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.

In the following we discuss the three (main) tasks of WP5, and then relate the relevance of these tasks with milestones of the project as a whole.

Task 5.1 (Year 1): *Infrastructure support for aggregation in edge computing.* This first task addresses the inherent challenges of distributed computing that aim at aggregating information concerning a (large) collection of values that are produced across a large number of resource constrained edge nodes (e.g., computing averages, maximum, or minimum values across a collection of values produced by a large number of sensors). To complete this task it is essential to devise a set of building blocks that enable computing such aggregates with minimal coordination among large collections of nodes, that can be resource constrained, and potentially communicating through unreliable communication links (such as is the case of wireless AdHoc networks).

Task 5.2 (Year 2): *Generic edge computing.* The second task extends the previously proposed set of distributed protocols and building blocks to support more generic edge computations across large resource constrained collections of edge nodes. Generic edge computations go beyond data aggregation and can include complex data mining tasks over large and dynamic collections of data produced at the edge, enabling to react to conditions such as alarms or faults in a timely and effective fashion. To complete this task it is essential to build abstractions and protocols that, while being efficient, are fault-tolerant providing these properties to applications that leverage them in their operation.

Task 5.3 (Year 3): *Self management and Security in edge computing.* Finally, the third task will extend tools (i.e., frameworks) and distributed protocols developed in the previous tasks of WP5 with self-management and security features. Self-management is an essential aspect to ensure the scalability and correct operation in highly dynamic scenarios in the light edge, where the capacity and the set of devices available can vary over time. Such management can include adjusting runtime parameters of protocols, automatic management of system affiliation, or event runtime replacement of distributed protocols being used, to better cope with varying runtime conditions. From the security perspective, the focus will lie on data integrity manipulated in light edge scenarios as well as minimizing the effects of denial of service (DoS) attacks in some scenarios, such as wireless AdHoc networks.

A fourth task that is not discussed in this report is related with packaging results of WP5 for exploitation in the context of a startup to be created in the later phase of

the project. As it will become apparent in the following sections, most of the results produced in WP5 that will integrate the Lightkone Reference Architecture (LiRA) can hence, be exploited in the context of a Lightkone startup and continually be maintained as independent and self-contained software artefacts.

Finally, we note that WP5 will be responsible for coordinating the prototyping of use cases presented by industrial partners that focus on light edge scenarios. This will allow to transport the results generated in this work package, and integrated with the LiRA to industry, demonstrating the practicality of the main innovations generated in this work package.

3.2 Progress

This deliverable reports on the progress of WP5 in building support to develop edge-enabled applications focused on the light edge. The work produced by the Lightkone consortium in the first 12 months of the project has generated 4 main results, which are detailed below. We start however by explaining the motivations for the work produced, and how they tackle different challenges in providing infrastructure support for edge-enabled applications. We then discuss how the results fit within the Lightkone Reference Architecture.

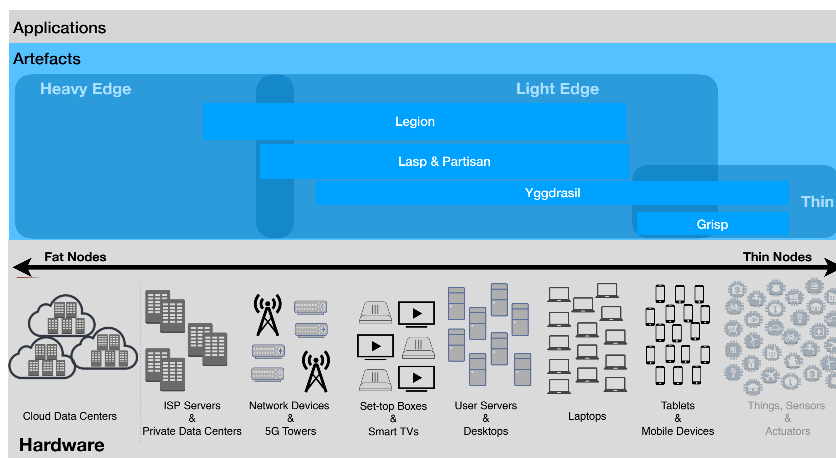

Figure 3.1: Relation of Results with Lightkone Reference Architecture

(a) Overview and Relation between Results

As illustrated in Figure 3.1, the light edge covers several different devices that are either user devices or data acquisition/actuators devices. The work conducted in the context of WP5 provides support to take advantage of such devices to build edge-enabled applications for addressing different and relevant applications domains.

One of the use cases presented deliverable D2.1, and put forward by the Gluk Advice BV partner, requires data aggregation and computing to be performed in scenarios that resemble sensor networks. One of the type of computations relevant in this scenario is distributed data aggregation. We have decided to explore distributed data aggregation protocols that can operate in an efficient and robust way among devices that interact

through wireless communication channels and without infrastructure. However, implementing this type of protocols and applications has significant overhead due to the lack of effective tools and support for these scenarios. To overcome this challenge we have worked towards building a framework that would allow us to implement and test, in an efficient way and using commodity devices, distributed protocols and applications that operate over wireless AdHoc networks. This has lead us to build the Yggdrasil framework. Using this framework we have explored existing distributed aggregation protocols to identify their limitations and suitability for use cases such as the one presented by Gluk Advice BV (see D2.1 for more details on the use case). These efforts were essential to achieve the goals of Task 5.1, which focus on supporting efficient aggregation, and Task 5.2, which aims at supporting general purpose computations. Yggdrasil was designed already considering the possibility of extending it to support other communication mediums, such that distributed protocols and applications built on top of it can be executed effectively on more types of devices along the light edge fraction of the spectrum.

Legion is a different framework whose goal was to allow enriching web applications - a very relevant application domain now-a-days - with edge computing functionality. Particularly, by enabling web client applications to transparently replicate and synchronize among them fractions of the web application state that are manipulated by them. Legion also explores how to allow clients, that exist in multiple user devices in the light edge scenario, to efficiently interact with components in the heavy edge, particularly to ensure durability of the web application state. We note that Legion was designed to have adapters that allow the framework to seamlessly interact with different back-end storage services. This allows Legion to, in the future, be able to interact with components of the LiRA that operate in the heavy edge domain. Additionally, Legion explores mechanisms to protect data privacy in scenarios where data is replicated and manipulated directly on user devices, hence contributing to the goals of Task 5.3, which aims at exploring questions related with data security (i.e., data privacy and integrity) in the context of the light edge, where devices are not trusted.

To allow more general purpose computations to be executed in the light edge, one needs infrastructure support to allow these computations to be specified over a specific data representation model. The light edge is characterized by having unreliable communication mediums and large number of independent devices, meaning that such computations should be executed in a way that must naturally cope with disconnection among nodes. A viable data representation model that provides these properties are CRDTs. Lasp is a framework that explores this idea, allowing to define computations that operate over CRDTs among large numbers of different devices. Lasp can operate on different devices in the light edge, which also being allowed to execute some of its nodes in the heavy edge, offering the possibility for the results of general purpose computations executed in the light edge to migrate towards components on the heavy edge and vice versa. Lasp contributes to achieving the goals of Task 5.2, on supporting general purpose computations in the light edge.

Finally, cyber-physical environments such as the ones in the use cases put forward by Gluk Advice BV, that involves smart agriculture, and Peer Stritzinger GmbH, that involves industrial factory setting and control of machinery, requires devices that must be small and cheap, while offering enough resources to perform computations in those settings. Furthermore, the Lightkone consortium requires such devices to effectively build demonstrators of the technology and innovation produced in the context of the

Project. This motivated the creation of GRiSP, an integrated system built and managed by Peer Stritzinger GmbH, and accompanying software packages that enable for instance, the execution of the Erlang VM directly on the board, which brings benefits since some of the project results have been developed in Erlang. This allows to enrich the very extreme of the light edge spectrum with a device that offers additional computational capacity, and that can sit in between very limited sensors and actuators, only gathering data or sending instructions to them, while coordinating computations at this level. This was an important step to address the final goal of WP5 in building demonstrator implementations of the use cases put forward by industrial partners of the Lightkone consortium that have strong emphasis on the light edge.

(b) Relation with Lightkone Reference Architecture

The Lightkone Reference Architecture (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 deliverable D3.1. Naturally, the work conducted in WP5 contributes directly to instantiate components to the LiRA particularly towards supporting applications taking benefit of the light edge. In the first 12 months of the Lightkone project four main artefacts have been developed for integration in the LiRA. Figure 3.1 presents a schematic view these artefacts within the overall LiRA.

The Yggdrasil framework allows to build protocols and applications that operate on devices that communicate through wireless networks, particularly using AdHoc networks. This framework allowed us to explore distributed aggregation protocols, which are one of the goals of WP5 (Section (c)).

The Legion framework, contrary to Yggdrasil, focuses on an edge scenario closely related to peer-to-peer systems. In particular, the focus of Legion is on supporting edge-enabled web applications running in end-users browsers. Legion also supports the interaction with server-side components that operate in the heavy edge (Section (d)).

Lasp is a programming abstraction and runtime support for edge applications that operate over CRDTs in fog edge computing scenarios. This framework also contains Partisan, a stand-alone module that allows to manage the membership of large-scale systems using varied overlay network topologies (Section (e)).

Finally, the GRiSP platform and associated software stack, provides an embedded system that has the capability of running native Erlang applications. GRiSP offers a new test-bed for the design and testing of edge-enabled applications running on specialized hardware, particularly it allows to run computations very close to tiny sensors with very limited computing capabilities (Section (f)).

In the following we present the work conducted by the Lightkone consortium in developing these artefacts.

(c) Yggdrasil framework and Aggregation in Wireless Settings

A particularly challenging edge scenario can be found in systems that operate at the very edge of the system, with very limited access to infrastructure network (i.e., Internet and cloud resources), where devices have to interact through wireless communication directly among them.

These edge scenarios include sensor networks, medium to large-scale Internet of Things (IoT) deployments, and some aspects of smart cities. In all of these settings, we expect to have a significant number of resource constrained nodes, that acquire large volumes of data. These devices, while being cheap to produce, and consuming low amounts of energy, have limited computational power and memory, and might be complex to program from the standpoint of application developers. For instance, many sensors have a single execution environment, where the operating system is a library linked to the application, and only a single application executes in the device.

To overcome these limitations, and to pave the way for performing general purpose computations in such edge scenarios we propose to leverage on more general purpose devices that: *i*) have more computational power and memory; *ii*) have an execution environment that is more familiar for developers; and *iii*) are affordable. One such device is the Raspberry Pi [7] whose latest model (Raspberry Pi 3 Model B+) can be purchased by approximately 30 to 40 euros. This model has a Quad Core 1.4GHz Broadcom BCM2837B0 64bit CPU, wireless capabilities built-in, and 1Gb of RAM.

Using these devices we can enrich applications, by having some devices in the system that own additional resources. Sensors and IoT devices can connect to one of these devices to report acquired data, and these devices can locally perform computations and exchange information among them. Furthermore, these devices can also control actuators that act on their deployment environment based on the results of local computations.

To ease the deployment of such devices and to ensure scalable management, applications should be able to operate with minimal human interaction. This requires such devices to self-manage and to interact without the need of a per-device configuration. In particular these devices should be able to be added into a deployment using a plug-and-play approach, enabling systems to grow organically as required by application operators. To this end, we have decided to exploit AdHoc networking, that while avoiding the overhead of Mesh networks, enable devices to discover each other and interact through wireless communication with no configuration.

Unfortunately, programming distributed applications in this setting is non-trivial. There are two main reasons for this; first, AdHoc Networking requires WiFi radios of devices to be configured appropriately. Furthermore, in some cases, as for instance high density scenarios, it could be useful to enable devices to switch (at runtime) among multiple AdHoc networks, operating under different frequencies, as to avoid frequent collisions in the wireless medium, which would significantly degrade the operation of applications, or render them completely ineffective. The second reason, is that there are few frameworks that support the construction of distributed protocols in AdHoc scenarios. Such frameworks should provide programmers with abstractions to send and receive messages through one hop broadcast directly at the MAC layer, and fundamental abstractions that are used frequently in the construction of distributed protocols and applications (such as Timer management, inter-protocol communication, and so on).

To address these challenges we have designed and implemented a prototype of a framework named Yggdrasil. Yggdrasil supports the development and execution of distributed protocols and applications in wireless AdHoc networks. In the following we discuss the design and implementation of Yggdrasil, as well as our work in exploring aggregation protocols in wireless AdHoc networks using the framework.

System Model and Requirements Yggdrasil operates at the MAC layer. It was designed for devices with, at least, one WiFi radio, capable of operating in AdHoc mode (sometimes called IBSS). It further assumes that devices have a limited amount of RAM and some processing power and can execute multi-threaded applications.

Yggdrasil further assumes that the operating system is based on linux, and in particular it assumes the availability of an implementation of the Netlink Protocol Library Suite (libnl) [4]. Library libnl exposes a set of API calls that allow to configure the wireless radio interfaces of devices. Decoupling our current design and implementation of the libnl is in our long term plans. Such decoupling will allow to execute Yggdrasil in other types of devices, such as embedded boards with wireless capabilities, as the GRiSP board described further ahead in this deliverable (§ (f)).

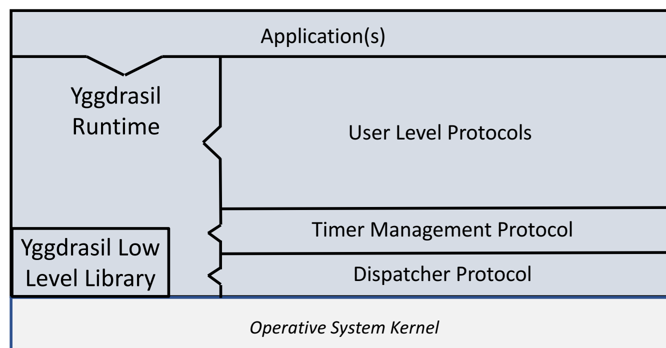

Figure 3.2: Yggdrasil Architecture Overview

Yggdrasil Architecture We now present and discuss the Yggdrasil architecture that is illustrated in Figure 3.2. Yggdrasil allows for multiple applications to execute in a single node, resorting to a single instance of Yggdrasil. In fact, we promote a single process model where both the Yggdrasil Runtime, protocols, and applications are compiled into a single binary and hence, are executed within the context of a single process. The motivation for this is to simplify, in the future, the support of the execution of Yggdrasil applications in devices that have a single process execution model such as the GRiSP board, and to allow the execution of small data acquisition applications written using Yggdrasil to operate on multiple commercially available sensors and motes [42].

In Yggdrasil, each component (with the exception of the Yggdrasil Low Level Library and the Runtime) has its own execution thread. This is valid for each of the essential support protocols (Timer Management and Dispatcher Protocol) and well as each user Protocol and Application. While this might lead to a significant number of threads for applications that resort to many protocols, we made this decision to promote parallel and independent execution of protocols, as to minimize latency and avoid starvation of execution when complex computations are executed, for instance, by some client application (a feature that does not exists in most existing frameworks for protocol development/implementation/execution).

Yggdrasil promotes an event driven execution environment for both protocols and applications. This is an execution model where both protocols and applications state evolve locally through the processing of asynchronous events. Events can be the reception of

(network) messages, the expiration of a timer, or the reception of an event, request, or reply issued by another protocol/application in the local process. This execution model has been used in other frameworks that were designed to support the implementation and execution of distributed protocols, such as Appia [66], as it simplifies the reasoning on the operation of a distributed protocol. Overall, this model leads to a more structured design of protocols. The interested reader can find more details regarding this model in [47].

In short, the main components in the Yggdrasil framework are as follows:

Yggdrasil Low Level Library: This component interacts with the kernel through the libnl, ioctl, and sockets interfaces to manage the radio interface (or interfaces), configure relevant network properties, and providing low level primitives for receiving and sending messages at the MAC layer. This component exposes its functionality through a *Channel* abstraction, that allows other components to configure, at runtime, the relevant properties of the AdHoc network being used, such as its name, radio frequency, among others. It further exposes mechanisms to enable an application to verify the networks that are enabled in the vicinity of a device, as well as mechanisms to gather information related with the wireless interfaces of the device.

Another important feature of this library, is that it allows the user to define a message pattern to filter messages received from the network at the kernel level. In the context of AdHoc networking, one typically exchange messages at the MAC level (which allows for instance to perform one hop broadcast [89]). However, this implies that whenever a device transmits a message in the radio frequency being employed by our AdHoc network, Yggdrasil would receive such message. This mechanism, allows all messages sent by applications and protocols running within our framework to be tagged with a particular pattern of bytes in the header[1], enabling the framework to ignore all messages that do not exhibit such pattern.

We have implemented the Yggdrasil Low Level Library as a fully isolated component for two primary reasons: *i*) it allows the reuse of this library in other contexts other than the Yggdrasil framework, for instance, by having this library as an isolated component one can wrap its implementation to allow its usage in other programming environments such as Erlang; and *ii*) it simplifies the replacement of this library by another implementation (exposing an equivalent interface) that can correctly operate in non linux environments, such as embedded systems or sensors.

Yggdrasil Runtime: This is the core component of our framework. Its main responsibilities are as follows: *i*) it manages the life cycle of all components in the framework, and in particular it manages the execution threads associated with each protocol and application that are loaded; *ii*) it exposes a set of communication mechanisms that can be used by applications and protocols to interact with each other. These mechanisms are, in a nutshell, an interface for protocols/applications to emit and register their interest in specific *events*, which resembles a publish-subscribe interface [38] and an interface for a protocol/application to issue a *request* for another

[1] In our prototype, we rely on the bytes corresponding to the ASCII code of "LKP", which stands for LightKone Protocol, in the 14th byte of each message.

specific protocol or applications, and for a *reply* to be sent back; *iii*) it exposes an API to allow applications to specify the protocols required for its operations and to specify the requirements of the application in terms of the wireless network to be used; and *iv*) it exposes interfaces to interact with the essential support protocols used by the framework, which we detail below.

Essential Support Protocols: Since Yggdrasil adopts an event driven execution model, a common necessity when implementing applications and protocols in Yggdrasil (and in general) is to send/receive messages and manage local timers (either to capture a timeout condition, or to allow the execution of a periodic task). To simplify these tasks, Yggdrasil features two essential support protocols. An API to interact with these protocols is provided by the Yggdrasil Runtime.

- Dispatcher Protocol: This protocol is responsible for managing the direct interactions with the channel abstraction provided by the Yggdrasil Low Level Library. In particular, this protocol is responsible for sending and receiving messages through the radio interface. This protocol also features a mechanism that enables a protocol or application to block (and unblock) a given device identified by its radio MAC address. When a MAC address is blocked, all messages that are received from that device are discarded by the Dispatcher. This feature is useful for testing purposes, such as emulating a multi-hop network when all devices are physically collocated or to block nodes that exhibit link flapping behavior [71].
- Timer Protocol: This protocol is responsible for managing the life cycle of all timers used by any protocol or application. The protocol supports timers that are triggered a single time or that are triggered periodically. It features mechanisms for other protocols and applications to create new timers and cancel existing timers. When a timer expires, the protocol is responsible for notifying (resorting to the Yggdrasil Runtime) the appropriate protocol or application as to execute the correspondent handler for that timer.

Our implementations of the Yggdrasil support protocols are similar to any other user level Yggdrasil protocol, and in fact, their execution is managed by the Yggdrasil Runtime in a similar fashion to any other user protocol. While this can add some overhead compared with providing such functionality directly as part of the Yggdrasil Runtime, we believe that this is a more sensible design choice, as it empowers developers to write their own variants of these protocols to use in concrete applications. We envision this as being a useful feature in particular, for performing debugging (enabling the logging of all messages and timers generated by the framework) as well as to perform experimental assessment of the performance of protocols and applications, by using instrumented variants of these protocols that extract performance metrics.

User Level Protocols: User level protocols can provide any form of functionality required by an application. Protocols in Yggdrasil are identified by a unique global numerical identifier, which is defined when the protocol is registered by an application. When a protocol is initialized, it can receive a set of runtime parameters, that are defined by the application requesting the execution of the protocol. The

two properties described above, allow multiple instances of the same distributed protocol to be executed within the context of an Yggdrasil process if different applications require it, or for multiple applications to share a single instance of a given protocol.

Protocol implementation is simple when using Yggdrasil. Protocols are defined by a main control function that is executed by a thread controlled by the Yggdrasil Runtime. This function, usually features a control loop, where the protocol waits for an event to be delivered to it. Upon the reception of an event, the protocol will verify the type of event and execute its handler for that event type. Such handlers can generate new events (e.g., register new timers, send messages, send notifications, issue requests or replies).

Events are delivered to protocols (and applications) through a queue, that is also managed by the Yggdrasil Runtime. This queue features an API that allows a protocol to block (and for its thread to yield the processor) waiting for the next event indefinitely or for a limited amount of time.

For validating the design and implementation of Yggdrasil, we have designed and implemented a set of simple distributed protocols that offer common features that are useful for the design and implementation of distributed applications. In more detail, we have designed and implemented the following protocols:

- Neighbor Discovery Protocols: This is a class of protocols that offer to the local process a local view of the other processes in the system, considering all or a sub-set of the direct neighbors of that process. In this context, we consider a direct neighbor of process a to be another process b such that messages can be directly exchanged from b to a. We have implemented three different protocols for this class: *i*) a complete neighbor discovery protocol, that exposes a list with all processes reachable from the local process; *ii*) a partial neighbor discovery protocol, that exposes a list with at most K randomly selected processes reachable from the local process; and *iii*) a dynamic neighbor discovery that behaves in a similar fashion to the total neighbor discovery protocol, except that it resorts to a fault-detection protocol (described below) to infer the arrival and departure (i.e., due to failure or shut-down) of neighbors.
- Fault-Detection Protocols: This class of protocols is responsible for suspecting that processes have failed and for producing notifications to other protocols or applications about these events. Optionally, this class of protocols can also operate as a neighbor discovery protocol. We have implemented two protocols that fall within this class: *i*) a simple failure detection protocol, that resorts to piggyback messages to monitor the continuous activity of neighboring processes. When a process does not issue a message for a configurable amount of time T, this protocol issues a suspect notification to all interested protocols or applications (through the event notification interface exposed by the Yggdrasil Runtime); and *ii*) a reactive failure detection protocol that behaves similarly to the previously discussed protocol, but that also detects events typically denoted as *link flapping* where, due to poor reachability, a process is continuously suspected and observed as active. In this case,

this protocol resorts to the feature of the Dispatcher protocol that allows to blacklist a device, by adding the MAC address of such node to the black list for a configurable amount of time.

- Membership Protocols: This class of protocols provide processes with either complete or partial views of the system membership that are not restricted by the capacity of processes to communicate directly. We have implemented a single global membership protocol that provides each process with a complete view of the system membership. This protocol relies on one of the fault-detection protocols to allow the membership to react to the departure, or failure, of a process.
- Application Level Broadcast Protocols: A common requirement of distributed applications is to disseminate information across all processes of the system. To this end, one usually resorts to an application level broadcast. We have implemented such a protocol based on an eager-push gossip protocol[56]. The protocol offers probabilistic guarantees of delivery, and resorts to random delays in re-transmissions to minimize the risk of message collisions in the wireless medium.

Application(s): In Yggdrasil, applications have the same structure of protocols. Applications also have a queue that allows them to receive events and can also generate events as any protocol. As discussed previously, Yggdrasil allows for multiple applications to execute within the context of a single Yggdrasil process.

In addition to the components described above, we also have implemented multiple test applications for the features discussed above, and simple demo applications that showcase how to use our framework.

The prototype of the framework was also used in a test pilot in the context of an advanced course taught by Professor João Leitão at the Faculdade de Ciências e Tecnologia of the NOVA University of Lisbon. The course, entitled *Algorithms and Distributed Systems*, is an advanced course of the Integrated Master's Program in Computer Science, and teaches the fundamental aspects on the design, implementation, and correctness of distributed algorithms. The framework was used by a group of two students to develop a global membership protocol and an aggregation protocol on top of the framework.

In fact, the framework described here can easily be used for teaching advanced courses in distributed algorithms, by allowing students to design and experiment with distributed protocols operating in wireless AdHoc networks, and in particular protocols that resort to MAC level communication, using commodity hardware, such as Raspberry Pis or even student's laptops running linux. We plan to further explore this venue in the future.

Aggregation Protocols As a way to showcase the benefits of our framework for supporting the development and execution of distributed protocols, and also to pave the way for supporting general-purpose computations in this edge environment, we have developed three different aggregation protocols. All our aggregation protocols operate over numerical data, that we assume is distributed across all nodes in the system[2]. Our algo-

[2] Such data will usually be acquired from sensors that are in the vicinity of each edge device, where each process executes. For testing these algorithms we have provided each process with a static and distinct numerical input.

rithms support the following four aggregation operators: maximum, minimum, sum, and average.

All our protocols are reactive, in the sense that the protocol only computes the intended value when an application, or another protocol, explicitly requests the computation. Background aggregation protocols, that continuously maintain a long-running computation to keep up-to-date aggregations of values that continually change in each independent edge device are also interesting, and will be addressed as part of future work.

The three aggregation protocols that we implemented are (the pseudo-code for these aggregation protocols is presented in Appendix A for completeness):

Dissemination Aggregation: This aggregation algorithm operates by having each process perform a one hop broadcast of all contributions (i.e, individual values tagged with the unique identifier of the source process) that the process knows locally. Initially a process simply disseminates its own local value. Upon reception of this message a process adds the received value to a local set of values (that is always initialized with the local process value) and applies the requested aggregation function, computing a partial result. After some time, the process disseminates this set to all its direct neighbours. After initializing the protocol (either by an explicit request of a local protocol or applications, or through the reception of the values from another process) each process will continue to disseminate its own set periodically every T time units (T is a configurable parameter of the protocol). Upon every reception of a set, a process first extracts the contributions that are not present in its own local set. After this, it adds each contribution to its local set and applies the requested aggregation function.

The protocol terminates in the following way. After a process disseminates its local set of known contributions C times (C is another parameter of the protocol) without receiving any message from another process that leads the local known set of contributions to grow, the process considers that there are no other contributions to acquire. This leads the algorithm to terminate and the protocol to issue a reply to the protocol or application that requested the aggregation result (evidently, this last part only takes place in the process where the request was originally issued).

Bloom Aggregation: This aggregation algorithm is very similar to the previous one, with some modifications that address a scalability limitation of the Dissemination Aggregation algorithm, that is, for a large number of processes, the size of the set propagated among processes might become excessively large. To avoid this, the Bloom Aggregation algorithm instead propagates among processes the following information: its own contribution, the partial-result obtained through the application of the target aggregation function over the received information, the number of different processes that contributed for that partial-result, and a bloom filter [22] (of configurable size) containing the identifiers of all processes that have contributed to the disseminated partial-result. Furthermore, each process stores the contributions of their direct neighbours.

In this algorithm however, the processing of received messages is somewhat more complex and, at a high level, proceeds with the following steps:

1. If the received bloom filter is exactly the same as the locally stored bloom filter (notice that each process begins the execution of the algorithm with a bloom filter containing its own unique identifier), then the process does not modify its state.
2. If the received bloom filter is completely divergent from the locally stored bloom filter (i.e, there is no bit set to one in both bloom filters) then the process merges the two bloom filters (with a binary OR operation) and re-computes its local partial-result based on the number of locally known contributions and previous partial-result and the partial-result and number of contributions in the received message.
3. If the local stored bloom filter does not contain the identifier of the process that sent the message, then the locally stored partial-result and bloom filter are updated to take into account the individual contribution of the process that sent the message.
4. If the received bloom filter does not contain the identifier of the local process or the identifier of the locally stored contributions (i.e, from direct neighbors), the received partial-result and bloom filter are updated to include the missing contributions, present in the local node.
5. In the two previous cases, the local process stores the partial-result (and associated bloom filter) that contains contributions from more processes. In the case of a tie (and in the presence of different partial-results) an heuristic is used to select one of those (currently we pick one at random).

The termination of the algorithm is similar to the one described for the Dissemination Aggregation algorithm above, with the key difference that the local condition for termination depends on the locally stored bloom filter remaining unchanged for N periodic transmission steps.

This algorithm fundamentally trades additional processing power and potentially a larger time until the computation of the average terminates to enable the transmission of smaller messages. This is only useful in deployments with large numbers of processes.

Single Tree Aggregation: This algorithm is an implementation of an algorithm originally proposed in [17] adapted to take advantage of the one hop broadcast primitive.

The algorithm operates as follows. Upon receiving a request for performing an aggregation operation by a protocol or application, this algorithm resorts to a neighbor discovery protocol to obtain the list of all direct neighboring processes. After this, the process broadcasts a request to execute the target aggregation operation. When a process receives such a message for the first time, it replies to the originator of the message with a YES message, otherwise, if the request had already been received from another process, a NO message is sent back. When a node replies with a YES message it also stores the identifier of process from which it received the aggregation request as being its *parent* and re-transmits that request.

Upon receiving the YES message, a process moves the identifier of the sender of that message from its neighbor list to its *children* list. When a process receives a NO message, it simply removes that process from the neighbor list.

After this initial step, a tree is established across the paths over which a YES message was sent. Then the aggregation step is conducted. In this step, processes that have received no YES messages from their neighbors (that don't have any children) send their individual value. After receiving a message from one of its *children*, a process first computes a partial results, given the reported value of his *children* and the locally stored value, and then proceeds to remove the *children* from its *children* set. Once there are no *children* left in the set, the process reports the partially computed result to its *parent*. This process continues up-stream through the tree, until reaching the root (i.e, the process that started the aggregation). This process can then compute the final result and report it to the protocol or application that requested it.

This protocol however is not tolerant to failures while the aggregation is happening. If a fault detector protocol notifies a process that one of its children is suspected as failed, then two options are available: *i*) we abort the aggregation operation by sending a failure notification up-stream through the tree; or *ii*) the process removes the suspected process from its *children* set and continue executing the algorithm regularly. The last option can lead to the computation of an incorrect result for the aggregation, as the result will disregard not only the contribution of the process that failed, but also of any process that was below in the tree. Our current implementation relies on the later strategy.

The protocols presented above are only used to illustrate how one can leverage Yggdrasil to build effective aggregation protocols. In the future we plan to implement more sophisticated protocols, and also explore how to build effective protocols that can continually run aggregation computations in the background across all nodes.

The experimental comparison on the performance of these and other aggregation protocols for AdHoc network scenarios, will be conducted in the context of Work Package 7 and initial results will be reported in deliverable D7.1.

Discussion The current design of Yggdrasil focuses on the fundamental requirements for supporting the execution of distributed protocols and applications in wireless AdHoc networks, which is a particularly challenging domain in the context of edge computation. It provides abstractions for building and executing distributed protocols that execute on laptops/micro-computers or things, sensors, and actuators (see D3.1 for details regarding these devices and the Lightkone Reference Architecture).

We believe that Yggdrasil already offers an interesting setting of features and protocols that enable the use of the framework to conduct real experimentation and evaluation, in commodity hardware, of distributed algorithms for AdHoc environments. This is a significant benefit, since, to the best of our knowledge, most algorithms found in the literature for this domain where only experimentally validated through simulation.

Additionally, the current version of Yggdrasil and accompanying protocols, form an interesting tool-kit for use in pedagogical activities, particularly in advanced courses related with wireless AdHoc protocols and applications, offering a set of abstractions that

allow students to develop and experiment with protocols in this domain in commodity hardware.

Notwithstanding, there is significant space for Yggdrasil to grow and offer additional features. In particular we consider the following venues interesting directions to enrich Yggdrasil:

- Offering support for other runtime execution environments, such as single-process operating systems (e.g., RTEMS) as the ones used in the GRiSP board and other IoT devices.
- Extend its design and implementation to support other types of devices in the edge spectrum (see D3.1).
- Integration of support for allowing applications and protocols in Yggdrasil to interact with sensor/actuators devices, typically used in sensor networks, as to enable processes running Yggdrasil applications to gather information and issue commands over these devices. This implies enriching Yggdrasil with support for Channels operating over the ZigBee [76] and 6LowPan [68] protocols.
- Offering support for self-management of protocols and applications through the integration of distributed monitoring schemes.

(d) Data Computation at the Edge with the Legion Framework

Legion addresses another edge computation scenario, in particular that of peer-to-peer computations performed in end-user devices. While a lot of work as been conducted in developing support for the construction and deployment of peer-to-peer applications [20, 43, 43, 53–56, 77, 83, 93, 95] Legion addresses a different challenge: transparent support for web applications. This is an important application domain for which very few efforts have been invested by both the industry and research communities.

A large number of web applications mediate interactions among users. Examples are plentiful, from collaborative applications, to social networks, and multi-user games. These applications manage a set of shared objects, the application state, and each user reads and writes on a subset of these objects. For example, in a collaborative text editor, users share the document being edited, while in a multi-user game the users access and modify a shared game state. In these cases, user experience is highly tied with how fast interactions among users occur. Legion explores a different area of the Lightkone Reference Architecture, and paves the way for both supporting more general purpose computations in the edge while also enabling interactions with applications components on the heavy edge.

These applications are typically implemented using a centralized infrastructure that maintains the shared state and mediates all interactions among users. This approach has several drawbacks. First, servers become a scalability bottleneck, as all interactions have to be managed by them. The work performed by servers has polynomial growth with the number of clients, as not only there are more clients producing contributions but also each contribution must be disseminated to a larger number of clients. Second, when servers become unavailable, clients become unable to interact, and in many cases, they cannot even access the application. Finally, the latency of interaction among nearby users is

unnecessarily high since operations are always routed through servers. This might not be noticeable for applications with low interaction rates such as social networks. However, user experience in games and collaborative applications relies on interactive response times below 50ms [48].

One alternative to overcome these drawbacks is to leverage on a edge computing approach making the system less dependent on the centralized infrastructure. Besides avoiding the scalability bottleneck and availability issues of typical web applications, leveraging on edge computing can also bring lower latency of interactions among clients. Additionally, it has the extra benefit of lowering the cost of dedicated centralized infrastructure.

At its core, Legion leverages on recent advances in browser technology, such as the new interfaces and functionalities offered by HTML5 and the capability of browsers of establishing direct communication channels through WebRTC [9], to enable the clients of web applications to interact directly. To support these interactions, each client maintains a local data store with replicas of a subset of the shared application objects. These data objects are replicated under an eventual consistency model, allowing each client to locally modify its replica without coordination. Updates are propagated asynchronously to other replicas by the Legion framework. To guarantee that all replicas converge to the same state despite concurrent updates, Legion relies on Conflict-free Replicated Data Types (CRDTs) [80].

Below we discuss the system model and assumptions underlying the Legion design and overview its architecture and the design of each component. We also discuss two demonstrators that we have constructed to showcase publicly the framework.

System Model Legion is a framework for data sharing and communication among web clients. It allows programmers to design web applications where clients access a set of shared objects replicated at the client machines. Web clients can synchronize local replicas directly with each other. For ensuring durability of the application data as well as to assist in other relevant aspects of the systems operation (discussed further ahead), Legion resorts to a set of centralized services (operating in the heavy edge). We designed Legion so that different Internet services (or a combination of Internet services and Legion's own support servers) can be employed. These services are accessed uniformly by Legion through a set of *adapters* with well defined interfaces.

We assume that the application state is organized in *containers*, that aggregate multiple data objects. Containers are replicated by clients completely. While Legion provides causal consistency guarantees over the state observed by clients over their local replicas, this is only provided for each container (i.e., causality is not enforced among different containers).

Legion current design and prototype assumes that applications have tens to few hundreds of users accessing a sharing the same state (i.e., the same data container).

Overview and Architecture By replicating objects in web clients and synchronizing in a peer-to-peer fashion, Legion reduces both dependency and load on the centralized component (as the centralized component is no longer responsible for propagating updates to all clients), and minimizes latency to propagate updates (as they are distributed directly among clients). Furthermore, it allows clients (already running) to continue interacting

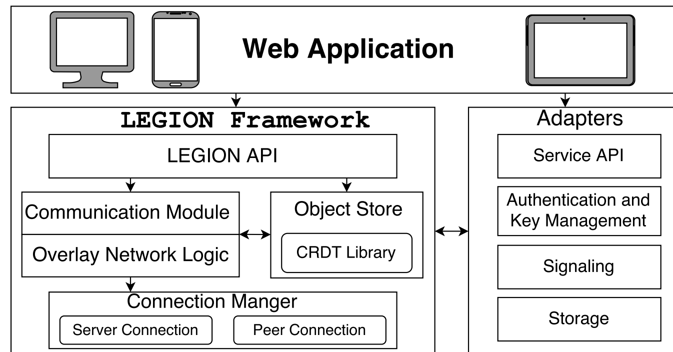

Figure 3.3: Legion Architecture Overview

when connectivity to servers is lost (for instance, due to a transitive network failure such as a partition).

Figure 3.3 illustrates the client-side architecture of Legion with the main components and their dependencies/interactions. We now discuss the goals of these components and in the following section discuss the design and implementation that we have pursued when developing Legion.

Legion API: This layer exposes the API through which applications interact with our framework.

Communication Module: The communication module exposes two secure communication primitives: point-to-point and point-to-multipoint. Although these primitives are available to the application, we expect applications to mostly interact using shared objects stored in the object store.

Object Store: This module maintains replicas of objects shared among clients, which are grouped in *containers* of related objects. These objects are encoded as CRDTs [80] from a pre-defined (and extensible) library including lists, maps, strings, among others. Web clients use the communication module to propagate and receive updates to keep replicas up-to-date.

Overlay Network Logic: This module establishes a logical network among clients that replicate some (shared) container. This network defines a topology that restricts interactions among clients. Therefore, only overlay neighbors maintain (direct) WebRTC connections among them and exchange information directly.

Connection Manager: This module manages connections established by a client. To support direct interactions, clients maintain a set of WebRTC connections among them. (Some) Clients also maintain connections to the central component, as discussed further ahead.

Legion uses two additional components that reside outside of the client domain:

- one or more *centralized infrastructures* accessed through adapters for: (i) user authentication and key management (*authentication and key management* adapter); (ii) durability of the application state and support for interaction with legacy clients

(*storage* adapter); (iii) exposing an API similar to the server API, thus simplifying porting applications to our system (*service API* adapter); (iv) assisting clients to initially join the system (*signaling* adapter);

- a set of STUN [65] servers, used to circumvent firewalls and NAT boxes when establishing connections among clients.

While we have also developed particular adapters for the Google Drive Real Time API (GDriveRT) and for a simple Node.js [50] server implemented by us, in this document we do not detail the design and implementation of adaptors for GDriveRT. The interested reader can refer to [91] for additional details.

Legion Design We now detail the main design and implementation decisions for each of the main components in the Legion framework.

Communication Module The communication module exposes an interface with point-to-point and multicast primitives, allowing a client to send a message to another client or to a group of clients. In Legion, each container has an associated multicast group that clients join when they start replicating an object from the container. Updates to objects in a container are propagated to all clients replicating that container.

Messages are propagated through the overlay network(s) provided by the *Overlay Network Logic* module. The multicast primitive is implemented using a push-gossip protocol (similar to the one presented in [56]).

Messages exchanged among clients are protected using a symmetric cryptographic algorithm, using a key (that is associated with each container) which is shared among all clients and obtained through the centralized component. Clients need to authenticate towards the centralized component to obtain this key, ensuring that only authorized (and authenticated) clients are able to observe and manipulate the objects of a container.

Overlay Network Logic Legion maintains an independent overlay for each container, defining the communication patterns among clients (i.e., which clients communicate directly). The overlay is used to support the multicast group associated with that container.

Our overlay design is inspired by HyParView [56]. It has a random topology composed by symmetric links. Each client maintains a set of $\mathtt{K}$ neighbors (where $\mathtt{K}$ is a system parameter with values typically below 10 for medium scale systems). Overlay links only change in reaction to external events (clients joining or leaving/failing).

In contrast to HyParView, we have designed our overlay to promote low latency links. As such, each client connects to $\mathtt{K}$ peers, with $\mathtt{K} = \mathtt{K}_n + \mathtt{K}_d$, where $\mathtt{K}_n$ denotes the number of *nearby* neighbors and $\mathtt{K}_d$ denotes the number of *distant* neighbors.

As shown by previous research [54], each client must maintain a small number of distant neighbors when biasing a random overlay topology to ensure global overlay connectivity and yield better dissemination latency while retaining the robustness of gossip-based broadcast mechanisms.

This requires clients to classify potential neighbors as being either nearby or distant. A common mechanism to determine whether a potential neighbor is nearby or distant is to measure the round-trip-time (RTT) to that node [54]. However, in Legion, since clients are typically running in browsers, it is impossible to efficiently measure round

trip times between them, since a full WebRTC connection would have to be setup, which has non-negligible overhead due to the associated signaling protocol.

To circumvent this issue we rely on the following strategy that avoids clients to perform active measurements of RTT to other nodes. When a client starts, it measures its RTT to a set of W well-known web servers through the use of an HTTP HEAD request (the web servers employed in this context are given as a configuration parameter of the deployment). The obtained values are then encoded in an ordered tuple which is appended to the identifier of each client. These tuples are then used as coordinates in a virtual Cartesian space of W dimensions. This enables each client to compute a distance function between itself and any other client c given only the identifier of c.

Connection Manager This module manages all communication channels used by Legion, namely *server connections* to the centralized infrastructure, and *peer connections* to other clients. We now briefly discuss the management of these connections.

Server Connections: A server connection offers a way for Legion clients to interact with the centralized infrastructure. We have defined an abstract connection that must be instantiated by the adapters that provide access to the centralized services. Independently of the employed centralized component, server connections are only kept open by a small fraction of clients.

Peer Connections: A peer connection implements a direct WebRTC connection between two clients [3]. To create these connections, clients have to be able to exchange – out of band – some initial information concerning the type of connection that each end-point aims to establish and their capacity to do so, which also includes information necessary to circumvent firewalls or NAT boxes using STUN/TURN servers. This initial exchange is known, in the context of WebRTC, as *signaling*.

Legion uses the centralized infrastructure for supporting the execution of the signaling protocol between a client joining the system and its initial overlay neighbors (that have to be active clients i.e., with active server connections). After a client establishes its initial peer connections, it starts to use its overlay neighbors to find new peers. In this case, the signaling protocol required to establish these new peer connections is executed through the overlay network directly.

Object Store The object store maintains local replicas of shared objects, with related objects grouped in containers. Client applications interact by modifying these shared objects. Legion offers an API that enables an application to create and access objects.

CRDT Library Legion provides an extensible library of data types, which are internally encoded as CRDTs [80]. Objects are exposed to the application through (transparent) object handlers that hide the internal CRDT representation.

The CRDT library supports the following data types: Counters, Strings, Lists, Sets, and Maps. Our library uses Δ-based CRDTs [92], which are very flexible, allowing replicas to synchronize by using deltas with the effects of one or more operations, or the full state.

Causal Propagation: This module uses the multicast primitive of the *Communication module* to propagate and receive deltas that encode modifications to the state of local

[3] Our experiments have shown that WebRTC connections can be established even among mobile devices using 3G/4G connectivity when devices use the same carrier.

replicas in a way that respects causal order (of operations encoded in these deltas). To achieve this, we use the following approach.

For each container, each client maintains a list of received deltas. The order of deltas in this list respects causal order. A client propagates, to every client it connects to, the deltas in this list respecting their order. The channels established between two clients are FIFO, i.e., deltas are received in the same order they have been sent.

When a client receives a delta from some other client, two cases can occur. First, the delta has been previously received, which can be detected by the fact that the delta timestamp is already reflected in the version vector of the container. In this case, the delta is discarded. Second, the delta is received for the first time. In this case, besides integrating the delta, the delta is added to the end of the lists of deltas to be propagated to other peers.

The actual implementation of Legion only keeps a suffix of the list of deltas received. Note that, at the start of every synchronization step, clients exchange their current vector clocks, which allow them, in the general case where their suffix list of deltas is large enough to include the logical time of their peer replicas, to generate deltas for propagation that contain only operations that are not yet reflected in that peer's state.

However, when two clients connect for the first time (or re-connect after a long period of disconnection), it might be impossible (or, at least, inefficient) to compute the adequate delta to send to its peer. In this case the two clients will synchronize their replicas by using the efficient initial synchronization mechanism supported by Δ-based CRDTs. In this case, if only a delta has been received, it is added to the list of deltas for propagation to other nodes. If it was necessary to synchronize using the full state, then the client needs to execute the same process to synchronize with other clients it is connected to.

Security Mechanisms Allowing clients to replicate and synchronize among them a subset of the application state offers the possibility to improve latency and lower the load on central components. However, it also leads to concerns from the perspective of security, in particular regarding data privacy and integrity. In more detail, *privacy* might be compromised by allowing unauthorized users to circumvent the central system component to obtain copies of data objects from other clients; additionally, *integrity* can be compromised by having unauthorized users manipulate application state by propagating their operations to authorized clients.

We assume that an access control list is associated with each data container, and that clients either have full access to a container (being allowed to read and modify all data objects in the container) or no access at all. While more fine-grained access-control policies could easily be established, we find that this discussion is orthogonal to main design challenges of Legion. We also assume that the centralized infrastructure is trusted and provides an authentication mechanism that ensures that only authorized clients can observe and modify data in each container. Finally, we do not address situations where authorized clients perform malicious actions.

Considering these assumptions, Legion resorts to a simple but effective mechanism that operates as follows. The centralized infrastructure generates and maintains, for each container C, a persistent symmetric key K_C within that same container. Due to the authentication mechanism of the centralized infrastructure, only clients with access to a container C can obtain K_C. Every Legion client has to access the infrastructure upon bootstrap, which is required to exchange control information required to establish direct

connections to other clients. During this process, clients also obtain the key K_C for the accessed container C.

K_C is used by all Legion clients to encrypt the contents of all messages exchanged directly among clients for container C (with the exception of the version number of K_C). This ensures that only clients that have access to the corresponding container (and have authenticated themselves towards the centralized component) can observe the contents and operations issued over that container, addressing data privacy related challenges.

Whenever the access control list of a container is modified to remove some user, the associated symmetric key is invalidated, and a new key for that container is generated by the centralized infrastructure (we associate an increasing version number to each key associated with a given container).

To enable clients to detect when the key is updated in a timely fashion, the centralized component periodically generates a cryptographically signed message (using the asymmetric keys associated with the certificate of the server, used to support SSL connections) containing the current version of the key, and a nonce. This message is sent by the server to active peers, that disseminate the message (without being encrypted) throughout the overlay network.

If a client receives a message encrypted with a different key from the one it knows, either the client or its peer have an old key. When the client has an old key (with a version number smaller than the version number of the key used to encrypt the message), the client contacts the centralized infrastructure to obtain the new key. Otherwise, the issuer of the message has an old key and the client discards the received message and notifies the peer that it is using an old key. This will lead the sender of the message to connect to the central infrastructure (going again through authentication) to update the key before re-transmitting the message.

Note that clients that have lost their rights to access a container are unable to obtain the new key and hence, unable to modify the state of the application directly on the centralized component, send valid updates to their peers, or decrypt new updates.

Demonstrators

Multi-user Pacman: We adapted a JavaScript version of the popular arcade game Pacman [5] to operate under the GDriveRT API with a multi-player mode. We also added support for multiple passive observers that can watch a game in real time. In our adaptation up to 5 players can play at the same time, one player controlling Pacman (the hero) and the remaining controlling each of the four Ghosts (enemies).

The Pacman client is responsible for computing, and updating the adequate data structures, with the *official* position of each entity. Clients that control Ghosts only manipulate the information regarding the direction in which they are moving. If no player controls a Ghost, its direction is determined by the the original game's AI, running in the client controlling Pacman.

In this game, we employed the following data types provided by the GDriveRT API: (i) a map with 5 entries, one for Pacman and the remaining for each Ghost, where each entry contains the identifier (ID) of the player controlling the character (each user generates its own random ID); (ii) a list of events, that is used as a log for relevant game events, which include players joining/leaving the game, a Ghost being eaten, Pacman being captured, etc. (iii) a list representing the game map, used to maintain a synchronized

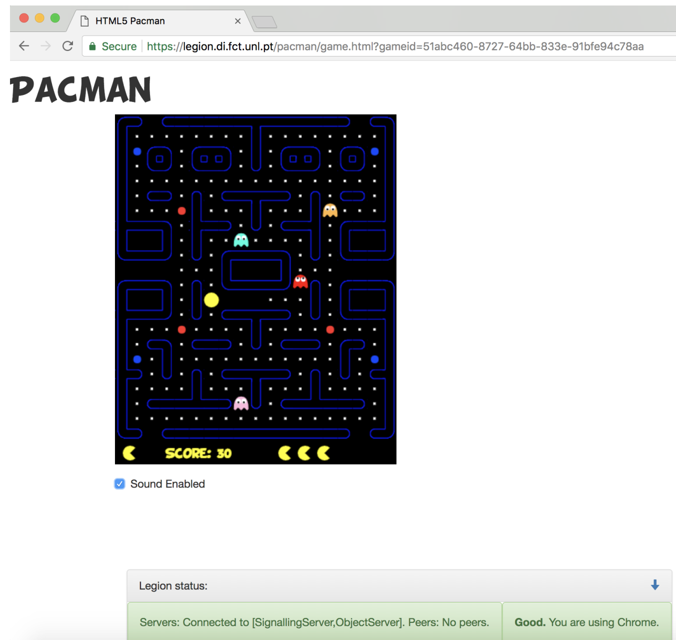

Figure 3.4: Legion Pacman Running in the Browser

view of the map between all players. This list is modified, for instance, whenever a *pill* is eaten by Pacman; (iv) a map with 2 entries, one representing the width and the other the height of the map. This information is used to interpret the list that is used to encode the map; (v) a map with 2 entries, one used to represent the state of the game (paused, playing, finished) and the other used to store the previous state (used to find out which state to restore to when taking the game out of pause); finally, (vi) 5 maps, one for each playable character, with the information about each of these entities, for maintaining a synchronized view of their positions (this is only altered when the corresponding entity changes direction, not at every step), directions, and if a ghost is in a vulnerable state.

After concluding the implementation of the game to use the GDriveRT API, modifying the game to leverage on Legion (using our GDriveRT adapters) required to change only two lines of code. Figure 3.4 illustrates this demonstrator.

Figure 3.5: Legion Shooter Running in the Browser

Legion Shooter To showcase the flexibility of the Legion framework, we also developed a second game that does not resort to the Object Store component of Legion, and instead operates by having clients disseminating messages directly among them. This is a very simple Shooter, where each player controls a small *square* in a battlefield. Each player can shoot bullets in any of the 4 main directions (up, down, left, and right). The game shows associated with the *square* of each player the username of the player and its score (how many times he destroyed an enemy and how many times he destroyed an enemy since spawning).

The game is modelled though a set of native JavaScript objects that are loaded together with the web page: (i) a map that is represented by a matrix of numbers that encode what is in each position of the map (free space, a wall, or a position where a player should not be spawned) and two integers that define the height and length of the map; (ii) a string with the username selected by the player; (iii) two integers representing the position of the player; (iv) a map with the player id, username, and current position of each player in the game; and (v) a list containing the list of bullets in the field (which includes information for the player that shot that bullet, its current position, and its speed).

The web client is responsible for computing the effects of fog of war (hiding from the player the locations of the map for which he does not have a direct line of sight), computing the positions of bullets, updating the score of the local player, and gathering the input from the user. Figure 3.5 shows the shooter application, from the perspective of a non-player user (left) that observes the whole game and from the perspective of a player that is conditioned by the fog of war.

The distributed logic of the game is achieved by having each client disseminate a message whenever: (i) a player spawns in the map, to notify all other clients of the new location of the client; (ii) a player modify its movement, to update the new location and direction of movement for the local player; (iii) a player shoots, to create the new bullet with the associated relevant information; (iv) a player kills an enemy, to notify the killed player of the event, and update the score of both players.

While the game was extremely easy to implement, it has been a success in public demonstrations of the framework, as detailed in Section 7.2.

Discussion Legion is a framework to support the design of user centered web applications, that leverage on edge computing by allowing web clients to directly interact with each other, through the replication of relevant fractions of the application state. Legion focus on particular edge setting of peer-to-peer systems leveraging on user devices (Desktops and Laptops). It further propose an interaction mechanism with heavy edge components through data replication based on CRDTs (see D3.1 for additional details on CRDTs).

Legion paves the way for a new generation of edge-enable web applications which are increasingly relevant in our society and economy. It also allows small and medium companies to be more competitive in this environment, by minimizing the investment required on centralized (i.e., cloud) infrastructures, while enabling applications that become suddenly popular to gather additional resources from the edge of the system in an organic fashion.

We plan to extend Legion in two fundamental aspects:

- Develop new adapters, that simplify the task of designing edge-enabled distributed applications which have their components executing in other edge scenarios.
- Extend the Legion architecture and design principles to take better advantage of mobile edge computing scenarios, a market that is continually growing now-a-days, hence expanding Legion toward a additional point of the edge spectrum captured in the Lightkone Reference Architecture (see D3.1).

(e) Data Computation on Lasp

Lasp is a framework that empowers programmers to perform computations over CRDTs [80] in a synchronization free way. Lasp is tailored for running close to the center of edge systems, in either data centers or fog computing environments (such as regional data centers or clusters closer to the edge of the system). In the context of the Lightkone Reference Architecture, it can operate at a wide range of different points of the edge spectrum, namely Private Servers& Desktops, Laptops, and Tablets & Mobile Devices, and it can seamlessly run on heavy edge environments. Compared with Legion, it provides similar abstractions for all devices in these points of the edge spectrum. Furthermore, it support more high level computations to be performed over CRDTs, enabling computations to be specified combining information from different CRDTs.

At its core, Lasp provides all the support for expressing (distributed) computations over CRDTs, where CRDTs are replicated across large numbers of nodes. Lasp propagates operations executed over CRDTs and hence, computations in Lasp are themselves replicated across nodes in a deployment. To achieve this, the Lasp framework combines a set of abstractions, including a programming API, a library of CRDTs, membership and communication abstractions (encapsulated in the Partisan component), and a storage service.

Lasp original design was motivated to support mobile applications, particularly mobile applications that operate over fractions of the applications state, encoded as CRDTs in a disconnected fashion. When these mobile applications can contact the Lasp infrastructure running in data centers, operations executed offline are propagated and synchronized with the infrastructure, leading to the progress of computations and the update of views over CRDTs being maintained and executed by Lasp dynamically.

In the following we present the system model assumed by Lasp, and then provide an overview of Lasp architecture and main components.

System Model Lasp is a framework that combines a distributed programming model, replicated CRDT data store, and distribution layer for Erlang. It's presented as a library, and available for use in new or existing Erlang and Elixir applications. Developers can therefore, leverage Lasp to augment portions of their distributed applications, rather than building their entire applications over Lasp.

Lasp makes no assumptions on the underlying infrastructure. While a default deployment of Lasp assumes that nodes are connected via Distributed Erlang, Lasp provides an alternative distribution layer, named Partisan, that can be used to support the operation of Lasp across different Distributed Erlang clusters. This allows these clusters to grow to larger sizes, or adopt different topologies, than the ones provided by normal Distributed Erlang clusters (i.e., fully connected topologies).

Furthermore, Lasp makes no assumptions on the underlying communications framework. Lasp provides an implementation of an anti-entropy protocol that ensures eventual delivery of all messages in a Lasp system, providing more guarantees to applications operating on top of a variety of different network protocols, such as TCP, UDP, and other frameworks such as publish/subscribe.

Following from this, Lasp assumes eventual consistency at its core, ensuring the correctness and convergence of computations regardless of update order or visibility. This is ensured by the programming model offered by Lasp, which prohibits expressing computations that would violate this property.

However, Lasp requires a key-value store for the storage of application state. Since Lasp assumes eventual consistency, any storage engine can be plugged in. As such, Lasp automatically takes care of data synchronization and ensures all updates are propagated to all nodes belonging to the system.

Architecture The Lasp architecture is composed by the following main components:

- **Lasp API:** Lasp API provides the top-level API for adding and removing CRDTs from the scope of an application. It also provides basic data transformations and aggregation operations using CRDTs within the application, allowing to create views over CRDTs.
- **Partisan:** Partisan provides the communication layer used by Lasp. Partisan provides a topology-agnostic programming model for cluster maintenance and point-to-point communication between nodes in the application.
- **Lasp KV:** Lasp KV provides the underlying key-value store inside Lasp which is used for efficient synchronization of data objects (CRDTs) used in a Lasp application.
- **Sprinter:** Sprinter is the deployment system for Lasp applications that eases the task of application deployment across a variety of cloud providers (e.g., Google Cloud Platform, Apache Mesos, and Docker Compose).
- **Types:** Types is a library of CRDTs used within the Lasp KV storage engine.

Lasp Design We now detail the design and implementation of each of the architectural components of a Lasp system.

Lasp API: Lasp API exposes the programming model for Conflict-Free Replicated Data Types offered by the framework. Lasp allows developers to write application that operate and manipulate CRDT objects, namely by exposing views that aggregate the state of multiple CRDTs. As the source objects of the computation are CRDTs, the functions are either monotone or homomorphisms, and the output objects are CRDTs. This ensures two important properties: *i*) compositionality, where the outputs of computations can be used as the inputs for other computations; and *ii*) convergence, in that, regardless of message ordering or message duplication, results converge to the same value across all nodes that are executing the (distributed) computation.

Developers using Lasp rely on the provided API to create and update CRDT objects in their applications, which are stored in the underlying key-value store (Lasp KV) and fully replicated across all nodes within the system. Computations are specified by using a set of combinators, such as `map`, `filter`, and `fold` to derive new CRDTs, which are also stored in the key-value store.

Partisan: Partisan is a topology-agnostic programming model and distribution layer for Erlang that is meant to be used to either: *i*) interconnect independent Distributed Erlang clusters; or *ii*) as a full replacement for Distributed Erlang.

Partisan provides a unified API over all of it's supported topologies, which are: *i*) full mesh; *ii*) client-server; *iii*) peer-to-peer; *iv*) static; and *v*) publish-subscribe. The API provides functions for asynchronous message delivery and cluster membership operations such as joining and removing nodes from the cluster.

Developers using Partisan can implement their own topologies as well, by writing a module that implements the Partisan peer service behavior, similar to an interface in the Java programming language. Lasp uses Partisan to support node-to-node communication across a cluster of nodes.

Lasp KV: Lasp KV is the underlying storage engine used by Lasp to store CRDT state and propagate it to other nodes in the system. Lasp KV supports two strategies of CRDT dissemination: state-based and delta-based, both configurable at runtime. As Lasp KV is a fully replicated data store, state is asynchronously propagated to all other nodes across the system, using the Partisan distribution layer.

Lasp KV also provides a small layer on top of an existing data store, allowing any existing data store to be used. Hence, Lasp natively supports Redis, Riak, Erlang Term Storage (ETS) and Durable Erlang Term Storage (DETS).

Moreover, Lasp KV interacts with a Lasp system by recomputing views when the source objects that contribute to those views change. This behaviour can be specified through the Lasp API. By default, Lasp KV will fully replicate both the source objects and the derived objects, but this can be configured in the application code, as well.

Lasp KV can, and has, also been used as a standalone storage system, where existing applications use only the get/put API of Lasp KV and its background node synchronization functionality.

Sprinter: Sprinter is a deployment system for Lasp applications that facilitates the operation of large-scale Lasp clusters on Apache Mesos (deployed on any of the existing cloud providers), Kubernetes (deployed locally or via Google Cloud Platform, or Microsoft's Azure), and Docker's Compose. Sprinter only works with Partisan-enabled systems, because it is closely tied to Partisan's membership and clustering API.

Types: Types is an Erlang library of CRDTs that is used by the Lasp components. This library contains two sets of implementations: one for state-based CRDTs (extended with delta-mutator support) and pure-operation based CRDTs. Currently, Lasp only supports the use of state-based and delta-based CRDTs.

Edge Computing over Lasp By assuming eventual consistency, computations are safe under disconnection and reconnection, message duplication, and message re-ordering. Therefore, Lasp is well suited to an environment where very few guarantees can be made. Lasp was also designed with one specific edge case in mind, providing support for mobile computing where clients replicate state and operate locally, with asynchronous propagation of state changes.

Every component of Lasp can be configured at runtime and was designed to be pluggable. Due to this, Lasp can likewise be used to run experiments where applications can be tested by alternating between different topologies, data structures, and storage engines at runtime, allowing analysis over which combinations yields the best performance under different application scenarios.

Discussion The Lasp framework offers support for performing computations in a distributed way over CRDTs without requiring explicit synchronization among replicas to enable progress. This provides an important step on supporting the computation of evolving data aggregation is a distributed fashion (leveraging CRDTs) and on supporting general purpose computations in complex edge-enabled applications. While Lasp core is tailored for running in the core of the system and smaller data centers it can support edge applications running in user devices, such as mobile phones, that operate, in a disconnected way, over fractions of the application state.

Moving forward, we plan to evolve Lasp, and in particular to enrich it with multiple features, that will allow its use in more edge scenarios. In particular, some of the interesting research and development venues for Lasp are:

- **Partial replication:** Lasp currently assumes full replication. However, when considering large number of devices and large amounts of data being produced by edge enabled applications, full replication can easily become a limiting factor for scalability. To address this, we plan to explore how to adapt the computation model of Lasp to take into account partial replication.

- **Causality:** Causality has been shown to simplify the management of application invariants, particularly when there are dependencies among operations that manipulate different fractions of the state of an application. Currently the communication abstractions and replications schemes provided by Lasp do not provide causality. How to efficiently provide this additional guarantee without compromising the computational model of Lasp is going to be addressed in future work.

- **Transactions:** While causality can assist in protecting some application invariants, others require the execution of sets of operations over the application state in an atomic way. Transactions are the common abstraction to enforce this guarantee. Enriching the interfaces provided by Lasp to allow the operation over multiple CRDTs in the context of transactions can, therefore, be useful to many edge enabled applications.

- **Expressivity:** Finally, the Lasp API that exists today is very simplistic. The current API only supports very simple transformations and aggregation operators. Exposing a richer interface that allows to execute more complex computations is, thus a necessity. Additionally, the API will have to be enhanced to expose abstractions

for programmers that capture complex aspects of distributed computations such as causality and transactions.

(f) GRiSP platform and hardware

Traditional Internet of Things (IoT) development is usually done with custom hardware. Sometimes development uses off-the-shelf components soldered onto custom boards. However, custom hardware makes prototyping and changes in later iterations more expensive. Additionally, errors present in the hardware are also more costly to address if the hardware has to be re-designed and re-built.

Embedded software is often developed using low-level languages. Such languages are closer to the hardware and can be more powerful for hardware access. They also make development more time consuming and error prone. Furthermore, software development in low-level languages is less productive than higher level languages[90].

We have developed a platform called GRiSP to work around these limitations. The goal is to support rapid prototyping and development of IoT edge applications. It consists of a reference base hardware board (whose designed had been conducted outside the scope of Lightkone) and a companion software stack to assist in the use of the hardware to build novel edge-enabled applications. The hardware has connectors for many common interfaces, while the software stack lets you develop applications using a high-level language: Erlang. Erlang is a highly productive functional language, that is also used in other artefacts produced by the Lightkone consortium, namely the AntidoteDB (see D6.1). It comes with a soft real-time virtual machine suited for networked edge devices.

GRiSP addresses two concrete edge scenarios, the IoT scenario which is the main focus of GRiSP, while potentially being also useful for application in sensor networks scenarios, particularly for hosting sensors and potentially actuators that can interact with the environment. This shows that in relation to the LiRA, GRiSP positions itself as a viable hardware platform to prototype and develop applications in the very extreme edge scenarios where large numbers of extremely small sensors and actuators are used. This can enable the construction of two-tier sensor-network and IoT applications where computations can be executed very close to data acquisition and actuation over the system on a few GRiSP boards that can potentially interact among them (for instance by leveraging Yggdrasil as presented before). The development and dissemination of GRiSP is being conducted by Peer Stritzinger GmbH, one of the members of the Lightkone consortium.

Hardware Design Part of the GRiSP project is a reference platform hardware board that we call GRiSP base. It is a USB powered micro computer.

CPU and Memory The board has a 32-bit System on a Chip (SoC) 300 Mhz ARM Cortex M7 central processing unit (CPU). It has a single and double precision floating point unit (FPU). It also comes with on-board digital signal processing (DSP) extensions. The CPU itself has 384 KiB static random-access memory. This we have enhanced with 64 MiB synchronous dynamic random-access memory (SDRAM).

Storage The board has storage in the form of 2048 KiB flash memory used for the boot loader. A Secure Digital (SD) card slot provides extensible storage for applications. We have also added 2 KiB electrically erasable programmable read-only memory

(EEPROM). Developers can store configuration and other data here which will survive power loss.

Pluggable Hardware The board has several different connectors for pluggable hardware. This makes it easy to prototype IoT edge devices.

- **GPIO:** Two 6-pin General Purpose Input/Output (GPIO) ports with configurable pins.
- **UART:** One 6-pin Universal Asynchronous Receiver/Transmitter (UART) port. UART provides asynchronous serial communication. It has generic bit streams with configurable data format and transmission speed.
- **SPI:** Two Serial Peripheral Interface (SPI) ports, one 6-pin and one 12-pin. SPI provides a synchronous serial communication bus which many chips support. It is fast with speeds up to 20 Mbit/s.
- **I_2C:** One Inter-Integrated Circuit (I_2C) bus connector. I_2C is a primary/secondary protocol where one primary controls secondary devices. Primary and secondaries on the bus communicates with addressable data packets. The protocol is slow with speeds up to 0.4 Mbit/s. It is also unreliable over longer distances. This is why it is almost exclusively used for board-local purposes.
- **1-Wire:** One Dallas 1-Wire bus connector. 1-Wire consists of only 1 wire plus a ground wire, hence the name. Devices consumes data and power from the same wire. They power themselves using small capacitors when no data is being sent. The bus is an addressable micro local area network (LAN) with unique 64-bit addresses. It is similar to I_2C, but slower. The protocol supports communication over longer distances. This makes it useful for external sensors not local to the board.

The GPIO, UART, and SPI ports follows the Peripheral Module (PMOD) form factor. You can connect plug-and-play devices without soldering. Many vendors supply PMOD devices implementing many different hardware functions. Examples include network interfaces, analog-digital converters, displays or servor motor connectors. Others provide sensors such as accelerometers, gyroscopes, and global positioning system (GPS) receivers. Users can also develop your own PMOD by following the specification.

Network and Connectivity Besides providing power the USB supplies serial and debugging connections. It has an integrated serial interface and a Joint Test Action Group (JTAG) interface. The serial interface accesses applications running on the board. Developers can use the JTAG interface to connect a debugger. The debugger will have full external access to the CPU. This allows for stepping through CPU instructions on an hardware level. There is also an external 2x10 pin JTAG connector on the board itself. Here you can attach special JTAG hardware debuggers.

The board has built-in Wi-Fi. The chip provides standard 802.11n wireless access for the 2.4 Ghz band with speeds of up to 150 Mbps. With its power saving mechanisms it is useful for temporary ad-hoc mesh networking.

Users and developers can also attach external hardware providing more connectivity options. Examples include ethernet, bluetooth, or custom radio PMODs.

Software Stack We have selected Erlang as the main development language and runtime environment. The Erlang virtual machine (VM) comes with a standard library and a set of design principles, OTP. It provides a distributed, fault-tolerant soft real-time environment.

Erlang itself is a high-level functional programming language. It has immutable data and pattern matching suited to advanced application development. Erlang has many useful primitives for implementing networking protocols. You can also interface with C making it a good fit for embedded development.

Our goal is to run Erlang on real bare metal hardware. That means without a separate operating system in-between the hardware and the application. We have opted to use The Real-Time Executive for Multiprocessor Systems (RTEMS). It is a Real Time Operating System (RTOS) as a library. RTEMS implements common operating system application programming interfaces (API). One such API is Portable Operating System Interface (POSIX) which Erlang uses. When compiling Erlang together with RTEMS, you get one executable. The boot loader then executes that executable as the OS itself. It has everything needed to interface with the hardware built-in. For application developers it looks like a normal Erlang VM.

One definition of GRiSP is thus the combination of Erlang together with RTEMS. This makes it possible to run the VM on various hardware boards. The GRiSP base is one such board.

Discussion GRiSP provides a trade-off between simpler smaller devices and larger more powerful devices. It is powerful enough to run advanced applications implemented in Erlang. It is also small enough to serve as a compact base for embedded IoT devices.

The possibility to use a high-level language and plug-and-play hardware makes it attractive. Applications can be rapidly prototyped in Erlang, combined with off-the-shelf components. They come in the form of PMODs and other peripheral hardware.

Due to the pluggable nature of the GRiSP board, it is not always suited as a target for final embedded devices. It is always cheaper to manufacture larger amounts of devices with custom hardware. Once you settle on the requirements and design many optimizations are possible. Manufacturing custom hardware makes it possible reduce the size of the device. You can also make it more efficient and lower power consumption with custom hardware (and by removing any hardware that is not required for a concrete use case).

In the future we want to continue the development the software stack. To get new features, we want to keep up with the latest versions of Erlang. We also want to add support for the crypto libraries that come with Erlang. We can then build secure IoT applications and other edge-enabled applications, and support protocols with built-in security.

Because other Lightkone partners use GRiSP, we expect to continue to make changes. Either new Erlang features or developing low-level libraries to access certain hardware. We also aim to target different hardware platforms other that GRiSP base. This will need more support from our build tools. Another interesting venue for future development would be to integrate Yggdrasil into the GRiSP ecosystem. This can be achieved by either extending the Yggdrasil low level library for it to become compatible with the interface for radio devices exposed by RTEMS. Another approach, would be by wrapping the current prototype of Yggdrasil with an Erlang API, enabling edge applications written in Erlang to take advantage of the mechanisms exposed by Yggdrasil, namely some of the

implementations of distributed membership, dissemination, and aggregation protocols currently featured in the framework.

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:

- Better understand what are the differences in terms of capacity and the potential uses of different devices in the edge spectrum.
- Improve the current design and implementation of the Yggdrasil framework, namely by simplifying its interface, and integrate 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 Guifi.net partner).
- Continue improving the software stack of the GRiSP board to simplify its use to address the challenges of concrete use cases.
- Develop a novel distributed aggregation protocol for wireless AdHoc networks, than can easily cope with the failure of devices and communication links, while providing accurate aggregation results to all nodes involved in the process.
- Dedicate efforts for the integration of the different innovations developed within the context of WP5 and other work packages of the Lightkone project.
- Explore self-management mechanisms for managing both the placement of computational components and data across the edge spectrum.
- Implement demonstrators based on the 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 12 months of the project regarding its three main technical tasks, and the relevant project milestones.

Task 5.1: Infrastructure support for aggregation in edge computing The work conducted in the context of Yggdrasil has built the fundamental infrastructure support for performing aggregation computations on wireless AdHoc networks, a setting where this form of computation will tend to be more useful. We have demonstrated this through exploring three different distributed aggregation schemes using Yggdrasil, however, these simple protocols have shown the need for developing a novel solution for aggregation in this setting. This work will be conducted in the next few months of the project.

Task 5.2: Generic edge computing Lasp and Legion are two frameworks that pave the way to support more general purpose computations in the edge in different edge scenarios and for different applications domains. Both of these alternatives rely on CRDTs to support such computations beyond the scope of the heavy edge.

Task 5.3: Self management and Security in edge computing Regarding self-management, Yggdrasil already includes features that allow a system to automatically discover nodes and perform aggregation computations among them without the need of configuration. This was achieved by combining neighbor discovery and fault detectors protocols, that operate in a localized fashion. Legion has explored how to leverage on application components that execute on the heavy edge to provide some measure of data integrity/privacy for computations performed in the light edge of the system, in particular on user devices. This technique can be further extended for integration with other innovations produced by the project, and also to lower its overhead.

In summary, we believe that the work reported in this document allow us to state that the completion of the three main technical tasks of WP5 have achieved a degree of completion close to 70%, 50%, and 30% respectively, for each of the Tasks 5.1, 5.2, and 5.3.

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 work reported in the deliverable already offers significant innovations to partially support these use cases. Evidently, future work will further evolve the presented frameworks and build new distributed protocols to address the needs of use cases. Considering the requirements of use cases, the work presented here covers 30% to 40% of the necessary efforts to fully achieve this milestone.

4 Software Deliverables

We now briefly present the four main software artefacts that are part of this deliverable. All software artefacts are publicly accessible through the Lightkone Work Package 5 public git repository at: https://github.com/LightKone/wp5-public.git. Each of the software artefacts have their own repository, which can be accessed through the url: https://github.com/LightKone/wp5-public/tree/master/software

Yggdrasil Framework and Protocols: The current version of the Yggdrasil framework, and relevant membership and aggregation protocols presented and discussed in this deliverable can be found in the following git repository: https://github.com/LightKone/Yggdrasil-devel.git.

Legion Framework: The Legion framework is available at: https://github.com/albertlinde/Legion.git.

Lasp Framework: The Lasp software is divided in three git repositories:

1. Main Lasp repository: https://github.com/lasp-lang/lasp.git.
2. Lasp's CRDT library: https://github.com/lasp-lang/types.git.
3. Partisan: https://github.com/lasp-lang/partisan.git.

GRiSP Software: The GRiSP software 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 results produced by WP5 in the first 12 months of the project. We also identify and briefly discuss the main novelty and innovations introduced by our work in relation to the state of the art.

5.1 State of the Art: Yggdrasil

There have been other frameworks and toolkits proposed to support the development of distributed protocols/systems and their execution. However, as we discuss below and to the best of our knowledge, there is no such tool specific for wireless AdHoc networking. Some frameworks were designed for group communication, or to provide fault tolerance mechanisms. Even though these frameworks might share some design choices with Yggdrasil, they do not consider the additional challenges and opportunities of AdHoc networks. Other frameworks, that specialize in wireless networks, where built to support the development and execution of application in specialized hardware (e.g., sensors) however, due to this they tend to be highly restrictive and not suitable for executing on top of commodity hardware and general-purpose operating systems.

ISIS [19, 21] and Appia [66] are examples of frameworks that were developed to support group communication protocols, ensuring the quality of service of a communication channel between processes that form a distributed application. The most recent of these, Appia, while being developed in Java, presents an execution model that is similar to the one used by Yggdrasil, being based on events that can transverse multiple protocols. Contrary to Yggdrasil however, Appia specifies a strict stack of protocols for each application, where the relative position of protocols in the stack has semantic implications on the guarantees provided for the communication channel. Furthermore, Appia does not support multi-threaded execution, and all protocols event handling (and hence protocol progress) are managed by a single thread. This makes the execution less efficient, particularly if some protocols have costly event handlers.

TinyOs [57] is a lightweight operating system to develop applications for wireless sensor networks. As such, TinyOs is fully designed to take advantage of wireless communication abstractions. Similar to Yggdrasil, applications and protocols developed in TinyOS leverage on an event driven execution model, having considerable support provided by TinyOs for this purpose. However, TinyOs was developed assuming a simple process execution model, meaning that each device will only be able to execute a single application. In contrast, Yggdrasil is more flexible, enabling the execution of multiple applications and protocols in the context of a single process.

Impala [60] is the framework that shares more common aspects with Yggdrasil, being a middleware that acts as a lightweight operating system that supports the implementation and execution of sensor network applications and associated protocols. However, the focus of Impala is on the remote update of applications and protocols and not the ease

of development, debug/evaluation, and execution. To support remote updating Impala, similarly to Yggdrasil, requires applications and protocols to be develop in a modular way. Contrary to Yggdrasil, in Impala multiple applications and protocols can be instantiated, but only one can be executed at a time, wheres Yggdrasil allows for multiple applications and protocols to execute concurrently, a feature that is not present in any of its predecessors (which are restricted to a single execution thread shared among all components).

Other frameworks and solutions to support the development of applications for fog and edge computing scenarios have recently been proposed, mostly in the field of Internet of Things (IoT). These include solutions and systems such as Stack4Things [63], and efforts from the OpenFog Consortium (https://www.openfogconsortium.org/research/) such as the ENORM framework [96]. However, these solutions do not focus in wireless AdHoc networks, nor on building the fundamental support to leverage those networks to perform complex tasks cooperatively. We argue however, that Yggdrasil could be used to build application components that execute at the edge of the system, leveraging AdHoc networks, that could interact and cooperate with other components supported by such frameworks.

5.2 State of the Art: Legion

The Legion framework is a contribution that touches on multiple fields of research of computer systems. In particular we identify four main relevant fields, and briefly discuss how existing solutions in each of these fields relate with Legion, and how Legion improves on the state of the art.

Internet services: Internet services often run in cloud infrastructures composed by multiple data centers, and rely on a geo-replicated storage system [12, 29, 31, 33, 61] to store application data. Some of these storage systems provide variants of weak consistency, such as eventual consistency [33] and causal consistency [12, 61], where different clients can update different replicas concurrently and without coordination. Similar to Google Drive Realtime, Legion adopts an eventual consistency model where updates to each object are applied in causal order. Contrary to common Internet Services however, Legion enables clients to synchronize their operations directly among themselves, meaning that clients (in this case Web client applications) can operate and interact without needing the cooperation of a centralized (i.e., heavy edge) component, lowering latency and making client applications less dependent on permanent connectivity to such centralized components.

Replication at the client: While many web applications are stateless, fetching data from servers whenever necessary, a number of applications cache data on the client for providing fast response times and support disconnected operation. For example, Google Docs and Google Maps can be used in offline mode; Facebook also supports offline feed access [39].

Several systems that replicate data in client machines have been proposed in the past. In the context of mobile computing [86], systems such as Coda [51] and Rover [49] support disconnected operation relying on weak consistency models. Parse [6], SwiftCloud [99] and Simba [74] are recent systems that allow applications to access and mod-

ify data during periods of disconnection. While Parse provides only an eventual consistency model, SwiftCloud additionally supports highly available transactions [14] and enforces causality. Simba allows applications to select the level of observed consistency: eventual, causal, or serializability. In contrast to these systems, Legion allows clients to synchronize directly with each other, thus reducing the latency of update propagation and allowing collaboration when disconnected from servers.

Bayou [85] and Cimbiosys [75] are systems where clients hold data replicas and that exploit decentralized synchronization strategies (either among clients [75] or servers [85]). Although Legion shares some of the goals and design decisions with these systems, its focus lies on the integration with existing Internet services. This poses new challenges regarding the techniques that can be used to manage replicated data and the interaction with legacy clients, as most of these services can only act as storage layers (i.e., they do not support performing arbitrary computations). Legion goes beyond the current state of the art by allowing enriching existing applications with transparent support for direct interactions while cooping with legacy clients (that do not use Legion).

Collaborative applications: Several applications and frameworks support collaboration across the Internet by maintaining replicas of shared data in client machines. Etherpad [37] allows clients to collaboratively edit documents. ShareJS [44] and Google Drive Realtime [45] are generic frameworks that manage data sharing among multiple clients. All these systems use a centralized infrastructure to mediate interactions among clients and rely on operational transformation for guaranteeing eventual convergence of replicas [72, 84]. In contrast, our work relies on CRDTs [80] for guaranteeing eventual convergence while allowing clients to synchronize directly among them. Collab [27] uses browser plugins to allow clients in the same area network to replicate objects using a peer-to-peer interaction model. Legion, contrary to existing solutions, uses standard techniques for supporting collaboration over the Internet, requiring no installation by the end user, while also enabling interactions with existing Internet services, while at the same time reducing the number of clients that are required to directly interact with such systems to ensure state persistence, hence lowering the overhead imposed on such services.

Peer-to-Peer systems: Extensive research on decentralized unstructured overlay networks [43, 56, 95] and gossip-based multicast protocols [20, 26, 56] have been produced in the past. Legion leverages and adapts some of the approaches proposed to implement its peer-to-peer model of direct communication among clients. Although our design for supporting peer-to-peer communication among clients builds on the HyParView overlay network [56], it differs from this system in the way it promotes low latency links among clients and leverages the centralized infrastructure for handling faults efficiently. Additionally, the overlay network used in the design of Legion, was designed to cope with the signaling mechanism that is required by WebRTC connections. The signaling mechanisms makes it impossible for any process to freely establish a connection with another process in the system by knowing only the IP and port of that process, a common assumption is the design of most peer-to-peer overlay networks. Hence, Legion also features, to the best of our knowledge, the first overlay network designed to operate over WebRTC connections.

5.3 State of the Art: Lasp

Lasp is a fault-tolerant dataflow system. Existing systems such as Apache Hadoop, Apache Spark [98], Apache Flink, Apache Storm, and Google Dataflow are all distributed dataflow systems, however, with different fault-tolerance properties. Designs like Spark provide fault-tolerance through tracking the lineage of computations, and use backup replicas to replace, or recompute, missing or lost data (i.e., partial results) when failures occur. In Spark, computations occur across a series of nodes, and rely on a scheduling strategy for optimal completion of a query with minimal data reorganization. However, most of these systems operate on immutable data. Naiad [69] is a dataflow system that supports fixed-point computations, but is not fault-tolerant, therefore requiring computations be restarted in the event of a single process failure. Naiad is also designed for immutable data. Lasp overcomes these challenges by combining the use of CRDTs as the fundamental unit of its data model, and replication. Ensuring that partial state of computations executed by Lasp is not lost despite individual process failures.

Lasp's execution model is closer to materialized view maintenance, where views are constantly refreshed as the source data changes. However, Lasp differs from traditional materialized view maintenance in modern database systems because views are themselves CRDTs: therefore views are mergeable, and views have an ordering relation defined on them which is compatible with their semi-lattice order. To the best of our knowledge, Lasp is the first system to explore the construction of materialized views that depend on multiple independent data sources with this property.

5.4 State of the Art: GRiSP

There are many hardware projects with accompanying development boards in the market . The most popular, with a very well-established community, is the Arduino project [1] that focuses on resource constrained prototyping boards. Other alternatives include the BeagleBoard, Raspberry Pi or Raspberry Pi Zero [8], among others. Commonly, resource constrained devices, that are usually programmed using C-like languages, lack the capacities to implement a full network stack for interactions with Internet services. On the other hand, the boards that are able to implementing network interfaces, usually run an instance of a general-purpose Operating System and are not energy-efficient.

Nerves [3] provides an Erlang VM based environment suited to embedded development. It is targeting larger devices that run Linux as the base OS. Supported platforms include BeagleBone [2] and Raspberry Pi. GRiSP sets itself in a different point of the design space occupied by existing state of the art, between very resource limited boards, that offer very little abstractions to develop interesting edge-enabled applications, and complete micro computers that operate using commodity (and expensive from the point of view of resource consumption) operating systems. From the perspective of the Lightkone consortium, the GRiSP board and its associated software packages, will enable easily porting of solutions (or small subsets of these solutions) originally developed in Erlang for heavy edge devices (e.g., Antidote as discussed in D6.1) to light edge scenarios.

6 Exploratory Work

In this Section we report on exploratory work that was conducted by the Lightkone consortium and that is aligned with the overarching goals of WP5. The results and innovations presented here are not yet integrated within the Lightkone Reference Architecture (LiRA), however they are important as they are results produced by research efforts of the project that aim at understanding how to better integrate applications components that operate within the light edge and the heavy edge.

These results pave the way for tackling challenges in the future, and might be incorporated within the LiRA if necessary. In this deliverable we report on two exploratory works, one focused on evolving the design of CRDTs to support computation of operators that do not require the replication of all the internal state of CRDTs among different replicas, and the second that explores a form to enrich the observable consistency properties of heavy edge storage systems from the client side (i.e., light edge).

6.1 Non Uniform CRDTs

A key aspect in building the necessary tools and support for enabling a new generation of edge-enabled applications is to allow the inter-operation of different system components running in highly distinct edge environments. As discussed in deliverable D3.1, the Lightkone consortium envisions two key aspects for achieving this purpose. The first is to define adequate data models that can ease the transference of application state and computations among different components of an edge-enabled distributed application; the second is related to the definition of clear and standard APIs and simple support services that can ease the integration and communication between the different components of the application.

CRDTs leveraging Non-Uniform Replication schemes is a first and decisive step in devising an adequate data model for edge-enabled large-scale distributed applications, as these data types allow for relevant application state to be encapsulated within a well defined data type, with clear semantics. This allows to perform efficient data replication and data transference, by avoiding to transfer large quantities of data. In particular, only data that is relevant for performing a given step of a large distributed computation or task. Furthermore, in the particular case of replication, a key technique in the design of efficient and reliable distributed systems, is to allow replicas to synchronize under the eventual consistency model, hence avoiding costly (both in terms of latency and processing power) synchronization steps in the critical path of user operations.

While this result is also discussed in D3.1, in this document, for completeness, we provide a brief overview of it, and discuss how these novel data types can be used for instance, to improve the performance of data aggregation algorithms across multiple edge scenarios. We note that this work was originally published in [25].

(a) System Model & Relevant Concepts

There are two key (novel) concepts for understanding the design and implementation of Non-Uniform operation-based CRDTs: *i*) Non-Uniform Replication and *ii*) Non-Uniform Eventual Consistency. Here we provide a condensed overview of these concepts as introduced in [25].

Non-Uniform Replication We consider an asynchronous distributed system composed by n nodes. Without loss of generality, we assume that the system replicates a single object. The object has an interface composed by a set of read-only operations, $\mathcal{Q}$, and a set of update operations, $\mathcal{U}$. Let $\mathcal{S}$ be the set of all possible object states, the state that results from executing operation o in state $s \in \mathcal{S}$ is denoted as $s \bullet o$. For a read-only operation, $q \in \mathcal{Q}$, $s \bullet q = s$. The result of operation $o \in \mathcal{Q} \cup \mathcal{U}$ in state $s \in \mathcal{S}$ is denoted as $o(s)$ (we assume that an update operation, besides modifying the state, can also return some result).

We denote the state of the replicated system as a tuple $(s_1, s_2, \ldots, s_n)$, with s_i the state of the replica i. The state of the replicas is synchronized by a replication protocol that exchanges messages among the nodes of the system and updates the state of the replicas.

A system is in a quiescent state for a given set of executed operations if the replication protocol has propagated all messages necessary to synchronize all replicas.

Definition 1 (Equivalent state) *Two states, s_i and s_j, are* equivalent, *$s_i \equiv s_j$, iff the results of the execution of any sequence of operations in both states are equal, i.e., $\forall o_1, \ldots, o_n \in \mathcal{Q} \cup \mathcal{U}, o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$.*

Note that this property does not require the internal state of the replicas to be the same, but only that the replicas always return the same results for any executed sequence of operations.

We propose to relax this property by requiring only that the execution of read-only operations return the same value. This property is named *observable equivalence* and is defined as:

Definition 2 (Observable equivalent state) *Two states, s_i and s_j, are* observable equivalent, *$s_i \overset{o}{\equiv} s_j$, iff the result of executing every read-only operation in both states is equal, i.e., $\forall o \in \mathcal{Q}, o(s_i) = o(s_j)$.*

As read-only operations do not affect the state of a replica, the results of the execution of any sequence of read-only operations in two observable equivalent states will also be the same. We now define a non-uniform replication system as one that guarantees only that replicas converge to an observable equivalent state.

Definition 3 (Non-uniform replicated system) *We say that a replicated system is non-uniform if the replication protocol guarantees that in a quiescent state, the state of any two replicas is observable equivalent, i.e., in the quiescent state $(s_1, \ldots, s_n)$, we have $s_i \overset{o}{\equiv} s_j, \forall s_i, s_j \in \{s_1, \ldots, s_n\}$.*

Non-Uniform Eventual Consistency For any given execution, with O the operations of the execution, we say a replicated system provides *eventual consistency* iff in a quiescent state: (i) every replica executed all operations of O; and (ii) the state of any pair of replicas is equivalent.

For any given execution, with O the operations of the execution, we say a replicated system provides *non-uniform eventual consistency* iff in a quiescent state, the state of any replica is observable equivalent to the state obtained by executing some serialization of O. As a consequence, the state of any pair of replicas is also observable equivalent.

For a given set of operations in an execution O, we say that $O_{core} \subseteq O$ is a set of core operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet O_{core}$. We define the set of operations that are irrelevant to the final state of the replicas as follows: $O_{masked} \subseteq O$ is a set of masked operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet (O \setminus O_{masked})$.

(b) Non-Uniform Operation-based CRDTs

Algorithm 1 Replication algorithm for non-uniform eventual consistency

```
1:  S : state: initial s^0                                          ▷ Object state
2:  log_recv : set of operations: initial {}
3:  log_local : set of operations: initial {}                       ▷ Local operations not propagated
4:
5:  EXECOP(op): void                                                ▷ New operation generated locally
6:    log_local = log_local ∪ {op}
7:    S = S • op
8:
9:  OPSTOPROPAGATE(): set of operations   ▷ Computes the local operations that need to be propagated
10:   ops = maskedForever(log_local, S, log_recv)
11:   log_local = log_local \ ops
12:   opsImpact = hasObservableImpact(log_local, S, log_recv)
13:   opsPotImpact = mayHaveObservableImpact(log_local, S, log_recv)
14:   return opsImpact ∪ opsPotImpact
15:
16: SYNC(): void                                    ▷ Propagates local operations to remote replicas
17:   ops = opsToPropagate()
18:   compactedOps = compact(ops)                                   ▷ Compacts the set of operations
19:   mcast(compactedOps)
20:   log_coreLocal = {}
21:   log_local = log_local \ ops
22:   log_recv = log_recv ∪ ops
23:
24: ON RECEIVE(ops): void                                           ▷ Process remote operations
25:   log_recv = log_recv ∪ ops
26:   S = S • ops
```

CRDTs [80] are data-types that can be replicated, modified concurrently without coordination, and guarantee the eventual consistency of replicas given that all updates propagate to all replicas. We now present the design of two useful operation-based CRDTs [80] that adopt the non-uniform replication model. Unlike most operation-based CRDT designs, we do not assume that the system propagates operations in a causal order. These designs were inspired by the state-based computational CRDTs proposed by Navalho *et al.* [70], which also allow replicas to diverge in their quiescent state. These CRDTs can be leveraged to support specific data aggregation operators in an distributed and efficient way in the edge.

Our design follows a general replication strategy that strives to minimize the amount of operations that have to be replicated. The algorithm is depicted in Algorithm 1. The key idea is to avoid propagating operations that are part of a masked set. The challenge is to achieve this by using only local information, which includes only a subset of the executed operations.

This is fundamentally captured by the function *sync*, which is called to propagate local operations to remote replicas. It uses the function *opsToPropagate*, that addresses

the key challenge of deciding which operations need to be propagated to other replicas. To this end, we divide the operations in four groups.

First, the *forever masked* operations, which are operations that will remain in the set of masked operations independently of the operations that might be executed in the future.

Second, the *core* operations ($opsImpact$, line 12), as computed locally. These operations need to be propagated, as they will (typically) impact the observable state at every replica.

Third, the operations that might impact the observable state when considered in combination with other non-core operations that might have been executed in other replicas.

Fourth, the remaining operations that might impact the observable state in the future, depending on the evolution of the observable state.

A detailed explanation of this algorithm can be found in [25]. In the following we detail the design of two Non-Uniform CRDTs. The design of these new data types is presented by materializing the generic protocol discussed here.

Top-K with removals NuCRDT In this section we present the design of a non-uniform top-K CRDT. The data type allows access to the top-K elements added and can be used, for example, for maintaining the leaderboard in online games. The proposed design could be adapted to define any CRDT that filters elements based on a deterministic function by replacing the *topK* function by another filter function.

The semantics of the operations defined in the top-K CRDT is the following. The *add(el,val)* operation adds a new pair to the object. The *rmv(el)* operation removes any pair of *el* that was added by an operation that happened-before the *rmv* (note that this includes non-core add operations that have not been propagated to the source replica of the remove). This leads to an *add-wins* policy [80], where a remove has no impact on concurrent adds. The *get()* operation returns the top-K pairs in the object, as defined by the function *topK* used in the algorithm.

Algorithm 2 presents a design that implements this semantics. The prepare-update *add* operation generates an effect-update *add* that has an additional parameter consisting in a timestamp $(replicaid, val)$, with *val* a monotonically increasing integer. The prepare-update *rmv* operation generates an effect-update *rmv* that includes an additional parameter consisting in a vector clock that summarizes add operations that happened before the remove operation. To this end, the object maintains a vector clock that is updated when a new add is generated or executed locally. Additionally, this vector clock should be updated whenever a replica receives a message from a remote replica (to summarize also the adds known in the sender that have not been propagated to this replica).

Besides this vector clock, *vc*, each object replica maintains: (i) a set, $elems$, with the elements added by all *add* operations known locally (and that have not been removed yet); and (ii) a map, *removes*, that maps each element *id* to a vector clock with a summary of the add operations that happened before all removes of *id* (for simplifying the presentation of the algorithm, we assume that a key absent from the map has associated a default vector clock consisting of zeros for every replica).

The execution of an *add* consists in adding the element to the set of $elems$ if the add has not happened before a previously received remove for the same element – this can happen as operations are not necessarily propagated in causal order. The execution of a

Algorithm 2 Top-K NuCRDT with removals

1: $elems$: set of $\langle id, score, ts\rangle$: initial $\{\}$
2: $removes$: map $id \mapsto vectorClock$: initial $[]$
3: vc : $vectorClock$: initial $[]$
4:
5: GET() : set
6: **return** $\{\langle id, score\rangle : \langle id, score, ts\rangle \in topK(elems)\}$
7:
8: **prepare** ADD($id, score$)
9: **generate** $add(id, score, \langle getReplicaId(), ++vc[getReplicaId()]\rangle)$
10:
11: **effect** ADD($id, score, ts$)
12: **if** $removes[id][ts.siteId] < ts.val$ **then**
13: $elems = elems \cup \{\langle id, score, ts\rangle\}$
14: $vc[ts.siteId] = max(vc[ts.siteId], ts.val)$
15:
16: **prepare** RMV(id)
17: **generate** $rmv(id,vc)$
18:
19: **effect** RMV(id, vc_{rmv})
20: $removes[id] = pointwiseMax(removes[id], vc_{rmv})$
21: $elems = elems \setminus \{\langle id_0, score, ts\rangle \in elem : id = id_0 \wedge ts.val \leq vc_{rmv}[ts.siteId]\}$
22:
23: MASKEDFOREVER($log_{local}, S, log_{recv}$): set of operations
24: $adds = \{add(id_1, score_1, ts_1) \in log_{local}$:
25: $(\exists add(id_2, score_2, ts_2) \in log_{local} : id_1 = id_2 \wedge score_1 < score_2 \wedge ts_1.val < ts_2.val) \vee$
26: $(\exists rmv(id_3, vc_{rmv}) \in (log_{recv} \cup log_{local}) : id_1 = id_3 \wedge ts_1.val \leq vc_{rmv}[ts_1.siteId]\}$
27: $rmvs = \{rmv(id_1, vc_1) \in log_{local} : \exists rmv(id_2, vc_2) \in (log_{local} \cup log_{recv}) : id_1 = id_2 \wedge vc_1 < vc_2\}$
28: **return** $adds \cup rmvs$
29:
30: MAYHAVEOBSERVABLEIMPACT($log_{local}, S, log_{recv}$): set of operations
31: **return** $\{\}$ ▷ This case never happens for this data type
32:
33: HASOBSERVABLEIMPACT($log_{local}, S, log_{recv}$): set of operations
34: $adds = \{add(id_1, score_1, ts_1) \in log_{local} : \langle id_1, score_1, ts_1\rangle \in topK(S.elems)\}$
35: $rmvs = \{rmv(id_1, vc_1) \in log_{local} : (\exists add(id_2, score_2, ts_2) \in (log_{local} \cup log_{recv})$:
36: $\langle id_2, score_2, ts_2\rangle \in topK(S.elems \cup \{\langle id_2, score_2, ts_2\rangle\}) \wedge id_1 = id_2 \wedge ts_2.val \leq vc_1[ts_2.siteId])\}$
37: **return** $adds \cup rmvs$
38:
39: COMPACT(ops): set of operations
40: **return** ops ▷ This data type does not require compaction

rmv consists in updating *removes* and deleting from *elems* the information for adds of the element that happened before the remove.

We now analyze the code of these functions.

Function MASKEDFOREVER computes: the local adds that become masked by other local adds (those for the same element with a lower value) and removes (those for the same element that happened before the remove); the local removes that become masked by other removes (those for the same element that have a smaller vector clock). In the latter case, it is immediate that a remove with a smaller vector clock becomes irrelevant after executing the one with a larger vector clock. In the former case, a local add for an element is masked by a more recent local add for the same element but with a larger value as it is not possible to remove only the effects of the later add without removing

the effect of the older one. A local add also becomes permanently masked by a local or remote remove that happened after the add.

Function MAYHAVEOBSERVABLEIMPACT returns the empty set, as for having impact on any observable state, an operation also has to have impact on the local observable state by itself.

Function HASOBSERVABLEIMPACT computes the local operations that are relevant for computing the top-K. An add is relevant if the added value is in the top; a remove is relevant if it removes an add that would be otherwise in the top.

Top Sum NuCRDT We now present the design of a non-uniform CRDT, Top Sum, that maintains the top-K elements added to the object, where the value of each element is the sum of the values added for the element. This data type can be used for maintaining a leaderboard in an online game where every time a player completes some challenge it is awarded some number of points, with the current score of the player being the sum of all points awarded. It could also be used for maintaining a top of the best selling products in an (online) store (or the top customers, etc).

The semantics of the operations defined in the Top Sum object is the following. The *add(id, n)* update operation increments the value associated with *id* by *n*. The *get()* read-only operation returns the top-K mappings, $id \rightarrow value$, as defined by the *topK* function (similar to the Top-K NuCRDT).

This design is challenging, as it is hard to know which operations may have impact in the observable state. For example, consider a scenario with two replicas, where the value of the last element in the top is 100. If the known score of an element is 90, an add of 5 received in one replica may have impact in the observable state if the other replica has also received an add of 5 or more. One approach would be to propagate these operations, but this would lead to propagating all operations.

To try to minimize the number of operations propagated we use the following heuristic inspired by the demarcation protocol and escrow transactions [16, 73]. For each id that does not belong to the top, we compute the difference between the smallest value in the top and the value of the id computed by operations known in every replica – this is how much must be added to the id to make it to the top: let *d* be this value. If the sum of local adds for the id does not exceed $\frac{d}{num.replicas}$ in any replica, the value of id when considering adds executed in all replicas is smaller than the smallest element in the top. Thus, it is not necessary to propagate add operations in this case, as they will not affect the top.

Algorithm 3 presents a design that implements this approach. The state of the object is a single variable, *state*, that maps identifiers to their current values. The only prepare-update operation, *add*, generates an effect-update *add* with the same parameters. The execution of an effect-update *add(id, n)* simply increments the value of *id* by *n*.

Function MASKEDFOREVER returns the empty set, as operations in this design can never be forever masked.

Function MAYHAVEOBSERVABLEIMPACT computes the set of *add* operations that can potentially have an impact on the observable state, using the approach previously explained.

Function HASOBSERVABLEIMPACT computes the set of *add* operations that have their corresponding *id* present in the top-K. This guarantees that the values of the elements in the top are kept up-to-date, reflecting all executed operations.

Algorithm 3 Top Sum NuCRDT

1: $state$: map $id \mapsto sum$: initial []
2:
3: GET() : map
4: **return** *topK(state)*
5:
6: **prepare** ADD(id, n)
7: **generate** *add(id, n)*
8:
9: **effect** ADD(id, n)
10: $state[id] = state[id] + n$
11:
12: MASKEDFOREVER$(log_{local}, S, log_{recv})$: set of operations
13: **return** {} ▷ This case never happens for this data type
14:
15: MAYHAVEOBSERVABLEIMPACT$(log_{local}, S, log_{recv})$: set of operations
16: $top = topK(S.state)$
17: $adds = \{add(id, _) \in log_{local} : s = sum_{val}(\{add(i, n) \in log_{local} : i = id\})$
18: $\wedge\ s \geq ((min(sum(top)) - (S.state[id] - s))\ /\ getNumReplicas())\}$
19: **return** *adds*
20:
21: HASOBSERVABLEIMPACT$(log_{local}, S, log_{recv})$: set of operations
22: $top = topK(S.state)$
23: $adds = \{add(id, _) \in log_{local} : id \in ids(top)\}$
24: **return** *adds*
25:
26: COMPACT(ops): set of operations
27: $adds = \{add(id, n) : id \in \{i : add(i, _) \in ops\} \wedge n = sum(\{k : add(id_1, k) \in ops : id_1 = id\})\}$
28: **return** *adds*

Function COMPACT takes a set of *add* operations and compacts the *add* operations that affect the same identifier into a single operation. This reduces the size of the messages sent through the network and is similar to the optimization obtained in delta-based CRDTs [11].

(c) Additionnal State of the Art Discussion

Replication: A large number of replication protocols have been proposed in the last decades [12, 34, 61, 62, 64, 78, 94]. Regarding the contents of the replicas, these protocols can be divided in those providing full replication, where each replica maintains the full database state, and partial replication, where each replica maintains only a subset of the database state.

Full replication strategies allow operations to concurrently modify all replicas of a system and, assuming that replicas are mutually consistent, improves availability since clients may query any replica in the system and obtain an immediate response. While this improves the performance of read operations, update operations now negatively affect the performance of the system since they must modify every replica which severely affects middle-scale to large-scale systems in geo-distributed settings. This model also has the disadvantage of limiting the system's total capacity to the capacity of the node with fewest resources.

Partial replication [13, 32, 79, 88] addresses the shortcomings of full replication by

having each replica store only part of the data (which continues being replicated in more than one node). This improves the scalability of the system but since each replica maintains only a part of the data, it can only locally process a subset of queries. This adds complexity to the query processing, with some queries requiring contacting multiple replicas to compute their result. In our work we address these limitations by proposing a model where each replica maintains only part of the data but can reply to any query.

Despite of adopting full or partial replication, replication protocols enforce strong consistency [30, 64, 67], weak consistency [12, 33, 61, 62, 94] or a mix of these consistency models [59, 82]. In this exploratory work we push the current state of the art to show how to combine non-uniform replication with eventual consistency. An important aspect in systems that adopt eventual consistency is how the system handles concurrent operations. CRDTs have been proposed as a technique for addressing such challenge.

CRDTs: Conflict-free Replicated Data Types [80] are data types designed to be replicated at multiple replicas without requiring coordination for executing operations. CRDTs encode merge policies used to guarantee that all replicas converge to the same value after all updates are propagated to every replica. This allows an operation to execute immediately on any replica, with replicas synchronizing asynchronously. Thus, a system that uses CRDTs can provide low latency and high availability, despite faults and network latency. With these guarantees, CRDTs are a key building block for providing eventual consistency with well defined semantics, making it easier for programmers to reason about the system evolution.

When considering the synchronization process, two main types of CRDTs have been proposed: state-based CRDT, where replicas synchronize pairwise, by periodically exchanging the state of the replicas; and operation-based CRDTs, where all operations need to be propagated to all replicas.

Delta-based CRDTs [10] improve upon state-based CRDTs by reducing the dissemination cost of updates, sending only a delta of the modified state. This is achieved by using *delta-mutators*, which are functions that encode a delta of the state. Linde et. al [92] propose an improvement to delta-based CRDTs that further reduce the data that need to be propagated when a replica first synchronizes with some other replica. This is particularly interesting in peer-to-peer settings, where the synchronization partners of each replica change frequently. Although delta-based CRDTs reduce the network bandwidth used for synchronization, they continue to maintain a full replication strategy where the state of quiescent replicas is equivalent. This work has show how to reduce the amount of information that has to be synchronized among replicas while ensuring the correctness of the outputs generated.

Computational CRDTs [70] are an extension of state-based CRDTs where the state of the object is the result of a computation (e.g. the average, the top-K elements) over the executed updates. As with the model we propose in this paper, replicas do not need to have equivalent states. This work enriches the state of the art and the initial ideas proposed in computational CRDTs in several aspects, including the definition of the non-uniform replication model, its application to operation-based eventual consistency and the new data type designs.

(d) Discussion

The current design of Non-Uniform CRDTs paves the way towards an adequate data representation model for supporting the operation of complex edge-enabled applications. This is achieved by minimizing data transference and hence, synchronization costs, among replicas, while providing support to perform efficient operations over potentially large collections of data.

The current prototype implementation of Non-Uniform CRDTs only addresses very specific aggregation functions, however this demonstrates the feasibility of building specific Non-Uniform CRDTs to support other aggregation operators. Another relevant aspect that we plan to address in the context of Non-Uniform CRDTs is related with the support of multiple operations (i.e, queries) over the same data collection. The current prototype design strives to find the minimal data that has to be synchronized among replicas in order to ensure that a particular query achieves non-uniform eventual consistency. Generalizing this design will allow to move towards the support of general purpose computations with low synchronization and data transference overhead. Aggregation operators are an essential primitive for achieving general purpose computations. However, to achieve this, there must be ways to compose these computations into more complex ones. The current design of Non-Uniform CRDTs do not offer a simple API to compose multiple computations over multiple CRDTs. Additional effort has to be conducted to find the most adequate way to offer such support.

We plan to address these research and development questions in future work, as well as to integrate these new types of CRDTs into some of the tools described in this report (Yggdrasil, Legion, and Lasp).

6.2 Enriching Consistency at the Edge

Services running in the heavy edge, such as distributed and geo-replicated storage systems [12, 52] and online services such as Facebook or Twitter, have public APIs to enable an easy integration with (other) applications. These applications usually run on clients and can be part of sophisticated edge architectures, with components scattered throughout many different points of the edge spectrum. However, the developers who design these applications may be required to handle specific anomalies allowed by the consistency models exposed by these services. This can be a complex task, particularly in the case of online services that provide little or no information about the provided consistency guarantees. This forces application developers to reason on how to enforce the semantics of their applications when centralized components of the system offers little consistency guarantees.

To overcome this challenge, and to pave the way for easier integration of different components in edge-enabled applications, we have developed a transparent middleware that can operate between the edge component of an application and the service that executes at the core of the system (or even at another layer of the edge closer to the center of the network). Our solution enables a fine-grained control over the session guarantees that comprise the consistency semantics provided by such systems, without having to gain access or modify the implementation of those services. To demonstrate the feasibility of our approach we have applied it for the Facebook public API and the Redis datastore, allowing to have fine-grained control over the consistency semantics observed by clients

with a small local storage and modest latency overhead.

While this result focus on enriching the session guarantees from the standpoint of a single client session, this technique is particularly useful to allow the integration of heavy edge storage system and third party services into edge-enabled applications. Due to this, additional details on this contribution can be found in the deliverable D6.1 produced by Work Package 6. This work was original published in [41].

(a) System Model

Our goal is to enrich consistency guarantees provided by third-party applications, allowing easier integration in more complex (edge-enabled) applications. In particular, the application developer may choose to have individual session guarantees (read your write, monotonic reads, monotonic writes, and writes follows reads) as well as combinations of these properties (in particular, all four session guarantees corresponds to causality [24]). To achieve this, we provide a library that can be easily attached to a client application (operating as a middleware layer), allowing to enrich the semantics exposed through the system public API. We focus on services that expose a data model based on key-value stores, where data objects can be accessed through a key, and that associate a list of objects to each key. We observe that this data model is prevalent in online social network services, particularly since they share concepts, such as user feeds and comment lists. In particular, we focus on services where the API provides two fundamental operations to manipulate the list of objects associated with a given key: an insert operation to append a new object to the first position of the list, and a get operation that exposes the first N elements of the list (i.e., the most recent N elements).

Since we access these services through their public APIs, we need to view the service implementation as a black box, meaning that no assumptions are made regarding their internal operation. Furthermore, we design our protocols without making any assumption regarding the consistency guarantees provided through the public service API. Previous work has shown a high prevalence of session guarantees violations in services exposing public APIs [40].

Our middleware solution leverages on a set of algorithms that require storing metadata alongside the data, which can be difficult to do when accessing services as black boxes, namely when the service has no support for including user managed metadata. In this case, we need to encode this meta-data as part of the data itself.

In order to arbitrate an ordering among operations issued by the local client and other remote clients, our Middleware has the need to have an approximate estimate of the current time. To achieve this, two options are available. If the service has a specific call in its public API that exposes the time in the server, such call can directly be used by our system. Otherwise, if the service exposes a REST API (which is typical in many services) a simple REST call can be performed to the service, and the server time can be extracted from a standard HTTP response header (called *Date*). Note that, even though it is desirable that this estimate is synchronized across clients, we do not require either clock or clock rate synchronization for correctness.

(b) Architecture

Our solution is materialized in a library implementing a middleware layer. The architecture of this middleware solution is depicted in Figure 6.1. Our system consists of a thin

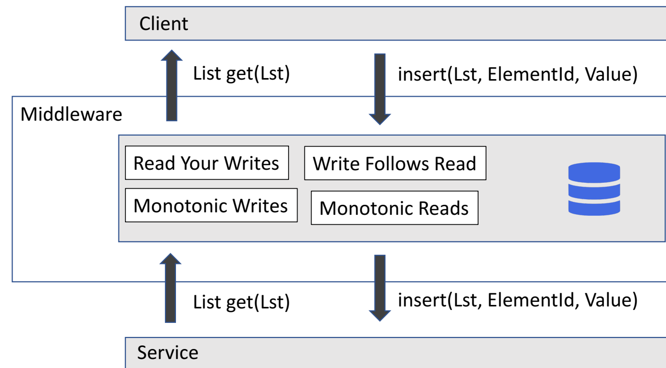

Figure 6.1: Middleware Architecture

layer that runs on the client side and intercepts every call made by the client application over the service, mediating access to the service. In particular, our layer is responsible for contacting the service on behalf of the client application, process the responses returned by the service and generate responses to the client applications according with the session guarantees being enforced.

Our system can be configured by the client application developer to enforce any combination of the individual session guarantees (as defined by Terry et. al [87]), namely: *i*) read your writes, *ii*) monotonic reads, *iii*) monotonic writes, and *iv*) writes follows reads. In order to enforce these guarantees, our system is required to maintain information regarding previous operations executed by the client application, namely previous writes that were issued or previous values that were observed by the client.

(c) Enriching Consistency

As previously mentioned, our system intercepts each request performed by the client application, executes the request in the service, and then processes the answer generated by the service to provide a (potentially different) answer to the client application. This answer is computed based on a combination of the internal state that records the previous operations that were run by that particular client, and the actual response that was returned by the service.

Tracking application activity. In order to keep track of user activity, our system maintains in memory a set of data structures for each part of the service state that is accessed by the application. These data structures are updated according to the activity of the applications (i.e., the operations that were invoked) and the state that is returned by the service. These data structures are: *i*) the *insertSet*, which stores the elements inserted by the client and *ii*) the *localView*, which stores the elements returned to the client.

Enforcing session guarantees. Enforcing session guarantees entails achieving two complementary aspects. First, and depending on the session guarantees being enforced, some additional metadata must be added when inserting operations. As mentioned, this metadata can be either added to a specialized metadata field (if the API exposed by the service allows this) or directly encoded within the body of the element being added to the list. Such metadata has to be extracted by our library when retrieving the elements of a list, thus ensuring transparency towards client applications. Second, our system might be required to either remove or add elements to the list that is returned by the service when

the application issues an operation to obtain the current service state, in order to ensure that the intended session guarantees are not violated.

We now discuss the concrete guarantees that can be enforced by our solution. The interested reader is referred to [40] or to the deliverable D6.1 to obtain the full details regarding the algorithms employed to achieve this. We also note, that these individual guarantees can be combined (i.e, the middleware can provide more than one of these guarantees simultaneously).

Read Your Writes The Read Your Writes (RYW) session guarantee requires that, in a session, any read observes all writes previously executed by the same client. More precisely, for every set of insert operations W made by a client c over a list L in a given session, and set S of elements from list L returned by a subsequent get operation of c over L, we say that RYW is violated if and only if $\exists x \in W : x \notin S$.

This definition, however, does not consider the case where only the N most recent elements of a list are returned by a get operation. In this case, some writes of a given client may not be present in the result if more than N other insert operations have been performed (by client c or any other client). Considering that the list must hold the most recent writes, a RYW anomaly happens when a get operation returns an older write performed by the client but misses a more recent one. More formally, given two writes x, y over list L executed in the same client session, where x was executed before y, an anomaly of RYW happens in a get that returns S when $\exists x, y \in W : x \prec y \wedge y \notin S \wedge x \in S$.

Monotonic Reads This session guarantee requires that all writes reflected in a read are also reflected in all subsequent reads performed by the same client. To define this in our scenario where a truncated list of N recent elements is returned, we say that Monotonic Reads (MR) is violated when a client c issues two read operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists x, y \in S_1 : x \prec y \text{ } in \text{ } S_1 \wedge y \notin S_2 \wedge x \in S_2$, where $x \prec y$ means that element x appears in S_1 before y.

Monotonic Writes This session guarantee requires that writes issued by a given client are observed in the order in which they were issued by clients. More precisely, if W is a sequence of write operations issued by client c up to a given instant, and S is a sequence of write operations returned in a read operation by any client, a Monotonic Writes (MW) anomaly happens when the following property holds, where $W(x) \prec W(y)$ denotes x precedes y in sequence W: $\exists x, y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$.

However, this definition needs to be adapted for the case where only N elements of a list are returned by a get operation. In this case, some session sequences may be incomplete, because older elements of the sequence may be left out of the truncated list of N returned elements. Thus, we consider that older elements are eligible to be dropped from the output, provided that we ensure that there are no gaps in the session subsequences and that the write order is respected, before returning to the client. Formally, we can redefine MW anomalies as follows, given a sequence of writes W in the same session, and a sequence S returned by a read: $(\exists x, y, z \in W : W(x) \prec W(y) \prec W(z) \wedge x \in S \wedge y \notin S \wedge z \in S) \vee (\exists x, y \in W : W(x) \prec W(y) \wedge S(y) \prec S(x))$.

Write Follows Read This session guarantee requires that the effects of a write observed in a read by a given client always precede the writes that the same client subsequently performs (Note that although this anomaly has been used to exemplify causality violations [12, 61], any of the previous anomalies represent a different form of a causality violation [87]). To formalize this definition, and considering that the service only returns at most N elements in a list, if S_1 is a sequence returned by a read invoked by client c, w a write performed by c after observing S_1, and S_2 is a sequence returned by a read issued by any client in the system; a violation of the Write Follows Read (WFR) anomaly happens when: $w \in S_2 \wedge \exists x, y \in S_1 : x \prec y \; in \; S_1 \wedge y \notin S_2 \wedge x \in S_2$.

(d) Additional State of the Art Discussion

The current state of the art in improving consistency guarantees from the client side can be found in recent proposals that also leverage a middleware layer that can mediate access to a storage system in order to upgrade their respective consistency guarantees [15, 18].

In particular, Bailis et al. [15] proposed a system called "bolt-on causal consistency" to offer causal consistency on top of eventually consistent data stores. There are two key aspects that makes our work go beyond this work. First, we provide a fine-grained choice of which session guarantees the programmer intends the system to provide, and only pay a performance penalty that is associated with enforcing those guarantees. Second, they assume the underlying system offers a general read/write storage interface, which gives significant more flexibility in terms of the system design than the model assumed in the work presented here.

The other closely related system is the one proposed by Bermbach et al. [18], which is also based on a generic storage interface, such as the ones exported by S3, DynamoDB, or SimpleDB, in contrast to our focus on high level service APIs. While they also provide fine-grained session guarantees chosen by the programmer, they limit these to Monotonic Reads and Read Your Writes. Out work goes beyond the current state of the art by allowing flexibility in the choice of any number of the four session guarantees, while considering a more realistic service API.

(e) Discussion

This work enables the client of an external service (either a distributed data storage system or some service accessed through a public API) to observe states of the application that respect some or a subset (eventually all) of the session guarantees. This is achieved through the use of a middleware layer that intercepts the calls performed by the client to the service and interacts with the service by the client. This middleware is then responsible for transparently transforming the reply of the service that is returned to the client to enforce the required session guarantees.

This is a powerful abstraction that can simplify the integration of independent services within a complex application, namely an edge application, enabling developers to design their applications without being concerned with potential session guarantees that are not enforced by that system consistency model.

We note however that currently our approach is focused on the interfaces that are commonly exposed by social network applications, namely that of a list where different users can add (i.e, append) elements to the list and read the N most recent entries in

the list. Expanding this approach to deal with more general data models is an open challenge. We also note that this approach is fundamentally useful for assisting in the integration of existing services to edge-enabled applications. Services that already offer causal consistency guarantees (such as AntidoteDB described in D6.1) or that expose data models based on CRDTs (see D3.1) will not require the use of this approach, and hence, avoid the storage and latency overheads introduced by it.

7 Publications and Dissemination

7.1 Publications

Some of the results reported in this document have been published in the following publications.

- Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In Proceedings of the 26th International Conference on World Wide Web (WWW '17). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, pp. 283-292. DOI: https://doi.org/10.1145/3038912.3052673

- Christopher S. Meiklejohn, Vitor Enes, Junghun Yoo, Carlos Baquero, Peter Van Roy, and Annette Bieniusa. 2017. Practical evaluation of the Lasp programming model at large scale: an experience report. In Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP '17). ACM, New York, NY, USA, pp. 109-114. DOI: https://doi.org/10.1145/3131851.3131862

- Gonçalo Cabrita and Nuno Preguiça. Non-uniform Replication. In Proceedings of the 21st International Conference on Principles of Distributed Systems (OPODIS'17). 18-20 December 2017, Lisboa, Portugal.

- F. Freitas, J. Leitão, N. Preguia and R. Rodrigues, Fine-Grained Consistency Upgrades for Online Services. 2017. IEEE 36th Symposium on Reliable Distributed Systems (SRDS), Hong Kong, 2017, pp. 1-10. DOI: https://doi.prg/10.1109/SRDS.2017.9

7.2 Dissemination Activities

Activities with Legion and Demonstrators We have used Legion as a demonstrator of the LightKone project results and also of the research activities conducted by NOVA in three public events. We now briefly report on these dissemination activities.

ExpoFCT (April 2017): The ExpoFCT is a public event hosted at Faculdade de Ciências e Tecnologia of the NOVA University of Lisbon where hundred of high school students and their professors (as well as some general public) visit the school and each department offers a set of activities that showcase the research and development activities conducted in the department. In the 2017 edition, the Legion

framework was one of the activities put forward by the department, where a room with 15 desktop computers was made available for visitors to experiment with the Legion demonstrators described above.

During this day, we had more than 200 users trying our demonstrators, most of them were high school students. We have explained to them what made that new, and the benefits that could be extracted from the use of the Legion framework. Overall the reaction of the users were highly positive. They were very happy playing the games (particularly the shooter game, that allowed all players in the room to play against each other) and were also impressed with the new technology, and the fact that this could easily be used from their homes.

Science 2017 National Meeting (July 2017): The Science 2017 National meeting, is a national event organized by the Portuguese government and directed at the general public to showcase highlights of the research results achieved by Portuguese research centers and Universities. In 2017, Legion was one of the demo highlights put forward by the NOVA LINCS laboratory in one of the days, where the usefulness of the framework was shown through the use of mobile devices (smart phones and tablets) using the demonstrators reported previously. The underlying technology was explained to the attendees that interacted with the demos.

The event allowed to expose these research results to many dozens of people. Overall, and similar to the other events that leveraged on our demonstrators to showcase the benefits of Legion, attendees where impressed that this new technology could immediately be used by them by simply accessing a web page that was leveraging on the Legion framework.

The web page of the event can be found in http://encontrociencia.pt/home/.

Closing Session of the CodeMove PT (December 2017): The CodeMove PT is a national movement in Portugal, sponsored by the Portuguese Government (through the Ministry for Science, Technology, and Higher Education; the Ministry for Education; the Ministry for Work, Solidarity, and Social Security; and the Ministry for Economy) that hosts a final public event during the weekend that occurs for a whole day. Since NOVA is also an active partner of this movement, we were invited to this event, and one of the presentations that we had for the general public was the Legion framework through the use of the same demonstrators, in four computers that were made available at the event location.

During the event, we had close to 100 users trying our demonstrators. Again the reactions were extremely positive, and most of the attendees, when we explained the underlying technology, were impressed by the fact that this was new technology that could be readily available at their homes.

The web page of the event can be found in https://www.codemove.pt.

Overall, these dissemination activities were useful to both promote the results of the project to the general public, and also to reinforce the idea that simple demonstrators and the fact that new edge technology can easily be brought to the everyday life of the general public, increasing the impact of achieved results.

Lasp Adoption Both Lasp, and it's membership module Partisan, have had adoption in industry and are available as open-source on GitHub at: https://github.com/lasp-lang

Industry adoption of Lasp has been focused on the use of Lasp as a replicated data storage system tailored for CRDT objects, and also of the Partisan module, which is leveraged to support point-to-point communication in clusters running Erlang software.

GRiSP Other partners in the Lightkone project will use the platform as a base for prototyping, particularly for prototype of use-cases.

Members of the Lightkone consortium made several talks at various conferences presenting the GRiSP project. We are showing why it is beneficial for IoT applications. We have also been giving hands-on tutorials on using the GRiSP board.

The project itself is open source. It is also non-profit in the sense that Peer Stritzinger GMBH develops, manufactures, and sells the hardware at production cost. The motivation for this is in expanding the community taking advantage of GRiSP for prototyping projects, both by the Lightkone partners and by hobbyists. This will increase the uptake of the platform. In turn, it increases knowledge of advances in edge computing and the Lightkone project itself.

The Lightkone consortium also believes that GRiSP can become a useful product in the context of teaching activities. In particular for advanced courses related with embedded systems and, potentially in combination with the Yggdrasil framework discussed earlier in this document, in the context of advanced networking and distributed systems courses.

8 Relationship of Results with Industrial Use Cases

We now discuss the relationship with the Industrial Use Cases previously reported in D2.1. We restrict our discussion to the use cases that can at least partially be tackled by the results reported in this document. We refer the interested reader to the D2.1 report for obtaining full details on these use cases.

The main results reported in this deliverable that can be explored by the Lightkone consortium are the GRiSP platform and the Yggdrasil framework. These two contributions both pave the way for a new generation of edge-enabled applications. This is achieved by enabling the development of novel hardware prototypes and experimentation in an expedite way and also, by offering different programming abstractions to develop distributed protocols and applications leveraging on large numbers of small devices. These two main results can be leveraged in the following use cases.

(a) Self Sufficient precision agriculture management for Irrigation

The combination of GRiSP and Yggdrasil will allow us to tackle some of the challenges that arise in the use case *Self Sufficient precision agriculture management for Irrigation*[4] (provided by Gluk). This use case will benefit from having small devices with wireless antennas, that coordinate the monitoring and processing of data directly on the edge. The GRiSP platform will enable to prototype new devices for this end, while the programming

[4] The description of this use case can be found in the revised version of deliverable D2.2.

environment offered by the GRiSP software stack will allow to develop applications in this context (and others). In fact, Peer Strinzinter GMBH offers GRiSP for this end.

This use case will also require a set of distributed protocols to support its efficient operation, namely membership protocols, data aggregation protocols, and potentially, data dissemination protocols (among others). In this context, the Yggdrasil Runtime and associated protocols might simplify the task of developing (and executing) the stack of distributed protocols to support applications and computations in the use case.

(b) Monitoring systems of the Guifi.net Community

The Yggdrasil framework will also allow to further explore the performance/overhead and precision (and the associated trade-off between these two aspects) of decentralized aggregation protocols in practice. While this will bring benefits to the use case discussed above, this will be employed to devise new aggregation strategies. Such strategies will potentially be useful in the context of the *monitoring systems of the Guifi.net Community* (provided by UPC), as aggregation is a good strategy to improve the operation of distributed monitoring mechanisms.

In particular, the monitoring schemes used in the Guifi.net could be improved by allowing relevant information to be gathered with lower communication overhead, and by aggregating this information directly at the edge, one could avoid the transmission of all monitoring information to a single point at the core of the network.

Using decentralized protocols is well suited for all use cases discussed above, where we have a potentially large numbers of devices interacting with each other while collecting and processing data. This is reinforced by previous research and experimentation conducted in the Guifi.net context [5], where it was observed that gossip protocol-based communication provided better results than other alternatives for supporting distributed directories of services provided through the Guifi.net platform.

9 Relationship with Results from other Work Packages

We now briefly discuss the relationship between the results presented in this deliverable and the work being conducted in the other Work Packages of the Lightkone project.

WP1: This work package is concerned with data protection and privacy, and it articulates with all other work packages including work package 5. At this point of the work however, we are still concerned with devising fundamental techniques and mechanisms to ease the development of edge-enabled applications. In the following months, we will start thinking about these aspects in the context of relevant demonstrators to be built. We note however, that the Legion framework already presents mechanisms to ensure data privacy for users that leverage the framework.

WP2: The results and tools developed in the context of WP5 will assist in supporting the implementation of the use cases proposed previously in the context of this work package, particularly those with strong emphasis on light edge (Gluk and Guifi.Net). We discussed these uses in the previous Section.

[5] http://wiki.clommunity-project.eu/pilots:distributed_announcement_and_discovery

WP3: This work package presents both the Lightkone Reference Architecture (LiRA) where the main results of WP5 are integrated, and also proposes fundamental tools and abstractions for allowing applications to leverage on components tailored for the heavy and light edge.

WP4: This work package is focused on devising semantics and programming abstractions for supporting edge applications, whose components might exist in any point of the heavy to light edge spectrum. Lasp, presented here, also exposes a programming interface that is fully detailed in D4.1.

WP6: While WP5 focuses on the light edge end of the spectrum, this work package focuses on the complementary heavy edge end of the spectrum. Naturally, we expect future edge-enabled applications to be build of components that operate on multiple points of this spectrum. The Legion framework already demonstrates how an edge-enabled application leverages on components that execute both at the core of the system and on user devices. Furthermore, Legion shows a way to integrate these two components, although it still lacks adapters for integration with AntidoteDB (reported in D6.1).

WP7: The WP7 work package is focused on the evaluation of solutions and systems produced by the Lightkone consortium. While the efforts of this work package started recently, there are multiple results presented here that will be the target of evaluation by this work package.

In particular, the GRiSP board will be used to devise prototypes of embedded devices that can support novel edge computing strategies being developed by the consortium. The frameworks presented here, will be evaluated in the context of WP7. Finally, we plan on leveraging the Yggdrasil framework to conduct a practical experimental evaluation of distributed data aggregation algorithms (among others) in the context of wireless AdHoc networks, as most existing work found in the literature has been experimentally validated and evaluated through simulation only.

10 Final Remarks and Future Directions

This deliverable reports the main and secondary contributions of the Lightkone consortium on developing new technology and approaches to support a new generation of edge-enabled applications. In particular, this report covers the work conducted in the context of Work Package 5, that focuses on the light edge. This entails devising adequate support for applications that are both available and correct while operating in a mostly independent way from components in the heavy edge (i.e., components executing within data centers).

The report covers four main results and two exploratory works. The main contributions focus on the support for edge-enabled applications, namely through a set of frameworks to support the development and execution of edge applications in concrete edge scenarios. Yggdrasil focuses on edge computations performed among resource constrained devices that are inter-connected through AdHoc networks. Legion focuses on enabling edge interactions for web applications by supporting direct browser-to-browser

communication. Lasp focuses on edge scenarios closer to the core of the network, and allows to specify decentralized computations performed over CRDTs. Finally, GRiSP is a novel and general purpose embedded system and software component to allow the fast prototyping and testing of edge devices for scenarios related to Internet of Things and sensor networks.

We also report on some of the relationships of the presented contributions with the use cases that have been identified by the Lightkone consortium previously. Additionally, we also discuss the relationship of these contributions with the work being conducted by the consortium in the context of other work packages.

As part of this deliverable we also present a set of software artefacts that are now publicly available through git repositories.

In the future, we shall continue developing the existing solutions to improve their support for edge-enabled applications in additional edge scenarios. We will continue the development of some of the frameworks presented here, and also design and implement a novel aggregation protocols for wireless AdHoc networks. Finally, we will dedicate some effort in developing additional demonstrators of our ideas and results.

References

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

[2] Beagleboard.org. http://beagleboard.org.

[3] Craft and deploy bulletproof embedded software in Elixir Nerves Project. http://nerves-project.org/.

[4] libnl - Netlink Protocol Library Suite. https://www.infradead.org/ tgr/libnl/.

[5] Original implementation of pacman for browsers. http://github.com/daleharvey/pacman.

[6] Parse. http://parse.com.

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

[8] Raspberry Pi Zero. https://www.raspberrypi.org/products/raspberry-pi-zero/.

[9] WebRTC. http://www.webrtc.org/.

[10] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by delta-mutation. In *Proc. of NETYS'15*, Morocco, 2015.

[11] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018.

[12] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proc. 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013.

[13] Gustavo Alonso. Partial database replication and group communication primitives. In *Proc. European Research Seminar on Advances in Distributed Systems*, 1997.

[14] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. of VLDB Endow.*, 7(3), November 2013.

[15] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[16] Daniel Barbará-Millá and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal*, 3(3):325–353, July 1994.

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

[18] David Bermbach, Jorn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, IC2E '13, pages 114–123, Washington, DC, USA, 2013. IEEE Computer Society.

[19] Kenneth Birman and Robert Cooper. The isis project: Real experience with a fault tolerant programming system. In *Proceedings of the 4th Workshop on ACM SIGOPS European Workshop*, EW 4, pages 1–5, Bologna, Italy, 1990. ACM.

[20] Kenneth P Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM TOCS*, 17(2), 1999.

[21] Kenneth P. Birman and Robert V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[22] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[23] 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.

[24] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 152–158, Feb 2004.

[25] Gonçalo Cabrita and Nuno Preguiça. Non-uniform Replication. In *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS 2017)*, 2017.

[26] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent structure in unstructured epidemic multicast. In *Proc. of DSN'07*, UK, June 2007.

[27] Santiago Castiñeira and Annette Bieniusa. Collaborative offline web applications using conflict-free replicated data types. In *Proc. of PaPoC '15 Workshop*, 2015.

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

[29] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. of VLDB Endow.*, 1(2), 2008.

[30] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In

Proc. 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, 2012.

[31] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3), 2013.

[32] Tyler Crain and Marc Shapiro. Designing a Causally Consistent Protocol for Geo-distributed Partial Replication. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015.

[33] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007.

[34] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, 1987.

[35] T. Dillon, C. Wu, and E. Chang. Cloud computing: Issues and challenges. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 27–33, April 2010.

[36] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, 2013.

[37] EtherpadFoundation. Etherpad. etherpad.org.

[38] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[39] Facebook Inc. Continuing to build news feed for all types of connections. goo.gl/Q06CaL, December 2015.

[40] F. Freitas, J. Leitao, N. Preguia, and R. Rodrigues. Characterizing the Consistency of Online Services (Practical Experience Report). In *Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 638–645. IEEE, June 2016.

[41] F. Freitas, J. Leito, N. Preguia, and R. Rodrigues. Fine-grained consistency upgrades for online services. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 1–10, Sep. 2017.

[42] S. Gajjar, N. Choksi, M. Sarkar, and K. Dasgupta. Comparative analysis of wireless sensor network motes. In *2014 International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 426–431, Feb 2014.

[43] Ayalvadi Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Net. Group Comm.* 2001.

[44] Joseph Gentle. ShareJS API. github.com/share/ShareJS.

[45] Google Inc. Google Drive Realtime API. developers.google.com/google-apps/realtime/overview.

[46] G. Greenwald and E. MacAskill. Nsa prism program taps in to user data of apple, google and others. http://tinyurl.com/oea3g8t, 2013.

[47] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[48] Caroline Jay, Mashhuda Glencross, and Roger Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact.*, 14(2), August 2007.

[49] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. SOSP'95*, 1995.

[50] Joyent Inc. Node. js, 2014.

[51] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM TOCS*, 10(1), February 1992.

[52] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[53] J. Leitão. Topology management for unstructured overlay networks, sep 2012.

[54] 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), Nov 2012.

[55] João Leitão, José Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *Proc. of SRDS'07*. IEEE, 2007.

[56] João Leitão, José Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Proc. of DSN'07*. IEEE, 2007.

[57] 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.

[58] D. Lewis. icloud data breach: Hacking and celebrity photos. http://tinyurl.com/nohznmr, 2014.

[59] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, 2012.

[60] Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. *SIGPLAN Not.*, 38(10):107–118, June 2003.

[61] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[62] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

[63] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito. Stack4things: An openstack-based framework for iot. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 204–211, Aug 2015.

[64] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 6(9), July 2013.

[65] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), RFC 5766. Technical report, IETF, April 2010.

[66] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 707–710, Apr 2001.

[67] Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Blotter: Low Latency Transactions for Geo-Replicated Storage. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 263–272, 2017.

[68] 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.

[69] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[70] David Navalho, Sérgio Duarte, and Nuno Preguiça. A Study of CRDTs That Do Computations. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015.

[71] P. C. Ng and S. C. Liew. Throughput analysis of ieee802.11 multi-hop ad hoc networks. *IEEE/ACM Transactions on Networking*, 15(2):309–322, April 2007.

[72] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proc. UIST'95*, 1995.

[73] Patrick E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986.

[74] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V. Madhyastha, and Cristian Ungureanu. Simba: Tunable End-to-end Data Consistency for Mobile Apps. In *Proc. of EuroSys '15*, 2015.

[75] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. of NSDI'09*, 2009.

[76] 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, April 2011.

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

[78] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1), March 2005.

[79] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine Partial Replication in Wide Area Networks. In *Proc. 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, 2010.

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

[81] 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.

[82] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, 2011.

[83] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4), 2001.

[84] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of Comp. supported cooperative work*, 1998.

[85] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP'95*, 1995.

[86] Douglas B. Terry. *Replicated Data Management for Mobile Computing*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2008.

[87] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[88] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proc. 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[89] Marc Torrent-Moreno, Steven Corroy, Felix Schmidt-Eisenlohr, and Hannes Hartenstein. Ieee 802.11-based one-hop broadcast communications: Understanding transmission success and failure under different radio propagation environments. In *Proceedings of the 9th ACM International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '06, pages 68–77, New York, NY, USA, 2006. ACM.

[90] Philip W. Trinder. Comparing C++ and ERLANG for motorola telecoms software. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, page 51. ACM, 2006.

[91] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching internet services with peer-to-peer interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 283–292, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[92] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-crdts: Making δ-crdts delta-based. In *Proc. of the PaPoC'16 Workshop*, United Kingdom, 2016. ACM.

[93] Robbert van Renesse, Kenneth Birman, Dan Dumitriu, and Werner Vogels. Scalable management and data mining using astrolabe*. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pages 280–294, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[94] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1), January 2009.

[95] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. of Net. & Sys. Manag.*, 13(2), 2005.

[96] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. Enorm: A framework for edge node resource management. *IEEE Transactions on Services Computing*, pages 1–1, 2017.

[97] 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.

[98] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[99] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. In *Proc. of Middleware'15*. ACM/IFIP/Usenix, December 2015.

A Pseudo-Code for the Aggregation Protocols

In this appendix the interested reader can find the pseudo-code for the aggregation algorithms discussed previously in § (c).

Algorithm 4 Broadcast Aggregation

```
1:  State:
2:    contributions                          ▷ set of contributions (process, value)
3:    reset                                  ▷ times the timer was reset
4:    threshold                              ▷ maximum number of reset times
5:    period                                 ▷ announce period
6:    aggValue                               ▷ aggregated value
7:    aggFunction                            ▷ aggregation function
8:
9:  Upon Init(t, period) do:
10:   reset ⟵ 0
11:   threshold ⟵ t
12:   period ⟵ period
13:
14: Upon RecvAggRequest(val, f) do:
15:   aggFunction ⟵ f
16:   aggValue ⟵ val
17:   contributions ⟵ (p, val)              ▷ where p is the process id
18:   Setup Timer Announce(period + RND)     ▷ where RND is a random small number
19:
20: Upon Announce() do:
21:   reset ⟵ reset + 1
22:   if (reset < threshold) then
23:     Trigger Send(SAGG_ANNOUNCEMENT, BROADCASTADDR, contributions)
24:     Setup Timer Announce(period + RND)
25:   else
26:     Trigger ReportValue(aggValue)
27:   end if
28:
29: Upon Receive(SAGG_ANNOUNCEMENT, c) do:
30:   newCon ⟵ c − contributions
31:   if newCon ≠ {} then
32:     reset ⟵ 0
33:     for all (p, v) ∈ newCon do
34:       aggValue ⟵ aggFunction(v, aggValue)
35:       contributions ⟵ contributions ∪ (p, v)
36:     end for
37:   end if
38:
```

Algorithm 5 Bloom Aggregation

```
1: State:
2:   contributions        ▷ bloomfilter containing the process IDs that have already contributed to the
     aggregated value
3:   nContributions       ▷ number of contributions in the bloomfilter
4:   myContribution       ▷ the contribution of the process (process, value)
5:   neighbours           ▷ the set of contributions of the neighbour processes (process, value)
6:   reset                ▷ times the timer was reset
7:   threshold            ▷ maximum number of reset times
8:   period               ▷ announce period
9:   aggValue             ▷ aggregated value
10:  aggFunction          ▷ aggregation function
11:
12: Upon Init(t, period) do:
13:   reset ⟵ 0
14:   threshold ⟵ t
15:   period ⟵ period
16:
17: Upon RecvAggRequest(val, f) do:
18:   aggFunction ⟵ f
19:   aggValue ⟵ val
20:   myContribution ⟵ (p, val)
21:   neighbours ⟵ (p, val)                 ▷ where p is the process ID
22:   contributions ⟵ {}
23:   contributions ⟵ contributions ∪ p
24:   Setup Timer Announce(period + RND)    ▷ where RND is a random small number
25:
26: Upon Announce() do:
27:   reset ⟵ reset + 1
28:   if (reset < threshold) then
29:     Trigger Send(BLOOM_ANNOUNCEMENT, BROADCASTADDR, myContribution, aggValue,
      nContributions, contributions)
30:     Setup Timer Announce(period + RND)
31:   else
32:     Trigger ReportValue(aggValue)
33:   end if
34:
```

Bloom Aggregation (continuation)

```
35: Upon Receive(BLOOM_ANNOUNCEMENT, (p,v), val, n, bloom) do:
36:   if contributions = bloom then
37:     return
38:   else if (contribution ∧ bloom) = {} then
39:     contribution ⟵ contribution ∪ bloom
40:     aggValue ⟵ aggFunction(aggValue, val)
41:     nContributions ⟵ nContributions + n
42:     if c ∉ neighbours then
43:       neighbours ⟵ neighbours ∪ (p,v)
44:     end if
45:     reset ⟵ 0
46:     return
47:   else
48:     nc ⟵ nContributions
49:     if p ∉ contributions then
50:       contributions ⟵ contributions ∪ p
51:       aggValue ⟵ aggFunction(v, aggValue)
52:       nContributions ⟵ nContributions + 1
53:       if (p,v) ∉ neighbours then
54:         neighbours ⟵ neighbours ∪ (p,v)
55:       end if
56:     end if
57:     for all (p,v) ∈ neighbours do
58:       if p ∉ bloom then
59:         bloom ⟵ bloom ∪ p
60:         val ⟵ aggFunction(v, val)
61:         n ⟵ n + 1
62:       end if
63:     end for
64:     if n > nContributions then
65:       aggValue ⟵ val
66:       nContributions ⟵ n
67:       contributions ⟵ bloom
68:     else if n = nContributions then
69:       if heuristic then          ▷ pick the information determined by an heuristic e.g, val > aggvalue
70:         aggValue ⟵ val
71:         contributions ⟵ bloom
72:       end if
73:     end if
74:     if nc < nContributions then
75:       reset ⟵ 0
76:     end if
77:   end if
78:
```

Algorithm 6 Single Tree Aggregation

```
1: State:
2:   isRoot                      ▷ whether we are the root node
3:   parent                      ▷ parent node in the tree
4:   children                    ▷ set of children nodes
5:   toBeAccounted               ▷ set of known nodes which have not answered to the node yet
6:   aggFunction                 ▷ aggregation function to perform
7:   aggValue                    ▷ current value of the node
8:
9: Upon Init(neighbourhood, function) do:
10:   isRoot ⟵ False
11:   parent ⟵ ⊥
12:   children ⟵ {}
13:   toBeAccounted ⟵ neighbourhood
14:   aggFunction ⟵ function
15:   aggValue ⟵ ⊥
16:
17: Upon RecvAggRequest() do:
18:   isRoot ⟵ True
19:   aggValue ⟵ GetLocalValue()           ▷ retrieves the local value of the node
20:   Trigger Send(STREE_QUERY, BROADCASTADDR, aggFunction)
21:
22: Upon Receive(STREE_QUERY, s, function) do:
23:   if parent = ⊥ ∧ ¬isRoot then
24:     isRoot ⟵ False
25:     parent ⟵ s
26:     aggFunction ⟵ function
27:     aggValue ⟵ GetLocalValue()         ▷ retrieves the local value of the node
28:     Trigger Send(STREE_CHILD, s, YES)
29:     Trigger Send(STREE_QUERY, BROADCASTADDR, aggFunction)
30:   else
31:     Trigger Send(STREE_CHILD, s, NO)
32:   end if
33:
34: Upon Receive(STREE_CHILD, s, answer) do:
35:   if answer = YES then
36:     children ⟵ children ∪ s
37:   end if
38:   toBeAccounted ⟵ toBeAccounted \ s
39:   Trigger ReportAggregate()
40:
```

Single Tree Aggregation (continuation)

41: **Upon** Receive(STREE_VALUE, s, $receivedValue$) **do:**
42: aggValue $\longleftarrow aggFunction(aggValue, receivedValue)$
43: $children \longleftarrow children \setminus s$
44: **Trigger** ReportAggregate()
45:
46: **Upon** ReportAggregate() **do:**
47: **if** $toBeAccounted \neq \{\} \vee children \neq \{\}$ **then return**
48: **end if**
49: **if** $isRoot$ **then**
50: **Trigger** ReportValue($aggValue$)
51: **else**
52: **Trigger** Send(STREE_VALUE, $parent$, $aggValue$)
53: **end if**
54:
55: **Upon** FaultDetected($node$) **do:**
56: $children \longleftarrow children \setminus node$
57: $toBeAccounted \longleftarrow toBeAccounted \setminus node$
58: **Trigger** ReportAggregate()
59: