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

D7.1: Evaluation

Deliverable no.:	D7.1
Title:	Evaluation
Due date of deliverable:	July 6, 2018
Actual submission date:	January 15, 2019
Lead contributor:	Scality
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
06/07/2018	1.0	Final Version	Scality
15/01/2019	2.0	Final Revised Version	Scality

The complete `git`-based changes tracking details for this document are available at the https://github.com/LightKone/WP7 private repository.

Contributors:

Contributor	Institution
Bradley King	Scality
Dimitrios Vasilas	Scality
Georges Younes	INESC TEC
Manuel Bravo	UCL
Christopher Meiklejohn	UCL & IST/INESC-ID
Annette Bieniusa	TUKL
Igor Zavalyshyn	IST/INESC-ID
Paulo Sérgio Almeida	INESC TEC
Sébastien Merle	STRITZINGER
Peer Stritzinger	STRITZINGER
Stefan Timm	STRITZINGER
João Leitão	NOVA
Pedro Ákos Costa	NOVA
Carla Ferreira	NOVA
Nuno Preguiça	NOVA
Roger Pueyo Centelles	UPC
Felix Freitag	UPC
Leandro Navarro	UPC
Roc Messeguer	UPC
Giorgos Kostopoulos	GLUK
Kostis Kounadis	GLUK

Contents

Executive Summary

This revised version of deliverable WP 7.1 presents the status of the evaluation phase of the LightKone project including certain changes since the initial submission; the list of changes have been noted in the introduction. Evaluation of the work performed is a key requirement of any successful project. While the project is only at the half-way point, and a number of the components are not yet available for evaluation, several evaluations have already been performed, notably on the academic research components. Some preliminary outputs of the formalization phase will also be discussed here. This report will primarily outline the plans in place for the evaluation of each of the use cases. Finally, plans for security evaluation will be discussed briefly.

Evaluation of the use cases

Following the formal model checking phase, actual implementations will be evaluated. The first step is to determine that the methods developed give the correct or expected results. Next the applicability of the methods in real world situations must be evaluated. The evaluations of the defined use cases are currently mostly at the planning stage since their implementations are still underway.

After this phase, we will pursue the determination of the scalability and applicability of the solutions. The Gluk, Guifi, and Stritzinger use-cases will concentrate more on feasibility and reliability while the Scality use-case will more on scalability and use of resources. One of the key motivations behind distributed and edge solutions is the ability to manage very large pools of users, and correspondingly very large collections of data or devices. The scalability of the system is thus a key element of a successful design, but it is often difficult, costly or simply impossible to test many systems at scale, except by deploying them in the real world. For the purpose of this evaluation, efforts will be made to determine the scalability of the platform, principally by using an increasing number of distributed servers and workloads to identify scaling limits in the models chosen.

Evaluation of formal models

Distributed systems can be extremely complex, and their behavior in the presence of partitions and different types of errors often become too difficult to predict using simple reasoning methods. In order to investigate the correctness of the models that have been chosen, we implemented formal modeling techniques. This type of approach is increasingly relevant as distributed systems become more and more common and mission critical. These models include tools such as TLA+ as well as abstract execution formulations. The use of these methods in guiding the evaluations is covered briefly in this report. As will be discussed, the models have provided useful insights for the implementation phase.

Evaluations in an Academic Context

A number of tools are being development in the context of the Lightkone project and will be used in the different use cases. The tools have been evaluated in realistic situations. These include tests of Legion, AntidoteDB with non-uniform replication, Yggdrasil and Mirage.

Evaluation of security at the edge

The scaling of the proposed solutions is important, but verification of the security is also critical. Effort has been invested to apply industry security practices and to study and ascertain the privacy impacts for the different use cases. The identification of security vectors and vulnerabilities have been discussed in WP3.2 and further actions will be taken as resources allow during the evaluation phase.

Chapter 1

Introduction

With the immense volumes of data generated and computed at data centers and cloud servers, it can be useful to relocate some storage and computation towards the edge of the network. Some benefits are: reduction of the load on the data centers, reduction of the network traffic, and improved availability and responsiveness. However, the edge computing approach brings its own challenges, notably: scalability, heterogeneity, resilience, and diverse security concerns. In WP7, we evaluate the methods to determine viability in real world situations. Here we report the evaluation that has been done and future plans. WP7's primary goal is to test and evaluate the work of the project.

1.0.1 Summary of Deliverable Revisions

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 document has been partially restructured to emphasise the key areas of concern, notably by moving the material regarding the experimental evaluations of the use-cases earlier in the document.
- We have added detail regarding targets for the experimental evaluations of the use cases to better define success criteria.
- The revised Gluk use case is presented briefly, with targets for the revised experimental evaluations.
- The Stritzinger use case has been clarified due to new inputs from a potential customer; the evaluation plans have been rewritten.
- Additional evaluations on Yggdrasil resource overheads have been performed and are reported.
- We have added additional explanations regarding the topic of security and which evaluations are feasible within the project timelines.

1.1 Relation to other WPs

As WP7 seeks to evaluate the work done in the other packages, so the relations between the diverse work packages is fairly obvious. The efforts of WP3 to develop a general purpose runtime will be tested and evaluated as the components of the runtime will be specifically evaluated within the framework of the use cases of WP2. Further, we will incorporate findings from the security analysis presented in D3.1. to complement the viability and performance evaluation. Within WP2, the application of formal methods to specific use cases for the evaluation of correctness have been pursued. Efforts in this direction are presented in the deliverable for WP 2.2. The somewhat more theoretical WP4, which seeks to provide formal semantics for the verification and implementation of efficient distributed and edge computing models, will be evaluated to verify to what extent the formalization does indeed cover the challenges at hand. Additionally, the model checking approaches pursued within the framework of WP4 are in fact components of the evaluation to the extent that they are able to formally validate the approaches taken in other work packages. WP5 and WP6, which represent the efforts specifically directed to the light and heavy edge use cases, are notably the focus of the field evaluations, with the field applications being defined by the WP2. One of the additional evaluation efforts is the investigation of issues of security; this feeds specifically into the outputs of WP1. Finally, the evaluations which measure the success or failure of the methods developed will largely impact the possibilities of WP8 where efforts are made to draw commercial benefit from the project. To the extent that the evaluations demonstrate advantages over existing or classical methods, the project will prove more or less interesting commercially. This emphasis will be noted in the choices made to perform the evaluations, to assure that the evaluations test pertinent criteria to determine the commercial viability of the work.

1.2 Topics discussed in this report

This report briefly discusses the general methodologies that are planned for WP7. Then the planning and initial feedback of the actual physical evaluations will be presented for each of the different use cases presented in WP2. This is the key effort of the evaluation phase of the project and in many cases awaits advancement of WP3 where runtime elements must be provided for the actual testing to take place. In these cases, the planning of the tests is presented. The Gluk evaluations resemble what is most commonly considered an IoT usage where sensor collect data and a centralized or cloud platform uses collected data for decision and policy making. The ability to push more of the intelligence into the network in order to improve responsiveness, reliability and ease of installation will be investigated as compared to the more classical centralized data collection approach. The Guifi evaluation is particularly interesting, because it will occur very close to a real operating environment, with a variety of edge devices to determine the applicability of AntidoteDB instances running with limited resources and heterogenous network capabilities. The Scality evaluation seeks to compare the innovative edge based approaches to more traditional centralized architectures for indexing. The more distributed indexing scenarios represent opportunities for more diverse deployments and better liveliness of queries across geographically distributed storage systems. The exact scenario that the Stitzinger evaluations will evaluate is not currently decided. It is intended that a customer specific usage will drive the final choice. While this is more complex from a planning perspective, it greatly increases the chances that the outcomes of the evaluation will be productized and produce commercial benefits soon after the pilot phase. The most likely scenarios are presented here with the work that is in development of the tools for the evaluations.

This is followed with details of the evaluations that have already been performed, mostly within the academic community in ongoing research in the context of the project. The ongoing formal model checking investigations will be discussed. They are explained in more technical detail in deliverables for D2.2 and D4.2, but within the framework of evaluation they are interesting because they permit the viability of certain approaches to be tested without requiring complex physical test setups. One of the key advantages of a formal evaluation is the ability to investigate a much greater part of the state space than would be possible in physical testing with the obvious limits of time and resources.

Finally, we refer to the ongoing security evaluations, both in terms of the high-level evaluations that have thus far taken place along with the plans being made for more complete security tests in certain of the use cases from WP2.

1.3 Methodologies

Multiple approaches to evaluations can be imagined and within the project, a number of different types of evaluations will be performed. In some of the use cases, notably Guifi, the systems are already in production and field testing can be done in a fairly straightforward manner. In others, the viability of future projects could potentially be brought into question depending on the outcome of initial tests, so the tests must be considered carefully and verification made during the testing phases that the tests represent the current or future reality.

As has been mentioned, the key areas under evaluation will be the following:

- Correctness of the pilot implementation in ideal conditions.
- Robustness of the solution to expected families of failures.
- Comparison of the solution to more traditional methods.
- Indications of the viability of the solution.

Evaluation of scalability

An important objective for distributed and edge solutions is to effectively manage very large pools of users and, hence, very large collections of data or devices. However it is often difficult to test the systems at sufficient scale. Resources have been requested and set aside for testing at scale, efforts will be made to make efficient use of the available resources.

The principal evaluation methods can be summarised as follows:

- Increasing the number of clients while monitoring the response times to determines the linearity of the system's scalability. Ideally, the system resource usage increases linearly while maintaining responsiveness.
- In addition to verifying linearity, it is important to determine limits of scale if this type of test is practically possible. Many systems will see limits in scale that were not initially expected. It is likely unrealistic that tests within the framework of the project will fully confirm the ultimate scalability of the chosen approaches.
- Evaluating and comparing resource usage and latency between methods to measure overall efficiency.

The outcomes of these various use case evaluations will be presented in future deliverables, notably D7.2 as well as feedback that should be provided to the work-packages 5 and 6 for light and heavy edge. Depending on the level of success of certain of the evaluations they could feed into the final outputs of the work-package 8 to motivate the execution of a concrete business plan.

Chapter 2

Evaluation plans for the Use Cases

2.1 Gluk Specific Evaluations

In this section, we provide specifics of the revised Gluk use-case evaluation plan.

2.1.1 Use case overview

We present a sensor array for precision agriculture with actuators to achieve management goals for irrigation. The core management ability must be completely autonomous (no need for PC or cloud control) and as low-cost as possible (again, no need for PC or cloud connectivity, which can be too expensive for realistic deployments). For this reason the management system should run on the sensor array itself. The deployment of the sensor nodes should be performed by the farmer, so the network must have zero-touch-configuration capabilities. The basic management should be done by the sensor array itself. Higher-level management goals can be added by external systems, such as PCs or cloud tools, but such external systems cannot be guaranteed to be connected to the sensor array. In order to achieve this, the requirements on the sensor array are that there should be (1) basic computation ability in the sensor nodes, and (2) basic communication ability between sensor nodes (for example, Wifi or Zigbee), with normal reliability of these nodes as provided by off-the-shelf hardware. Given these requirements, the software we develop using LightKone technology should be able to perform 24/7 reliable basic management despite problems in the sensor array, such as nodes going down or unreliable communications.

2.1.2 Evaluation Objectives

The aim of the experimental evaluation will be to validate the following performance objectives:

- **Nodes Discovery and zero-touch-configuration:** After the deployment of nodes by the farmer, an autonomous network setup is required. It requires that the nodes coordinate in discovering each other and creating a network topology. This is one of the most important and challenging targets for evaluation that we have set.

- **Network Management and communication:** During network operations, network management and maintenance is required. Topology requirements and shape may change due to failures in nodes. We seek to evaluate how effective the network management and communication using the LightKone technologies can be. This requirement is also one of the major and challenging evaluation targets.
- **Load balancing:** We seek to evaluate if load balancing would be possible using the Lightone proposed technology in order to extend the lifetime of the network.
- **Information dissemination:** We want to examine the effectiveness of the information pushed to the controllers (actuators). We will evaluate the energy conservation and nodes' tolerance to link failures.

2.1.3 Experiment Design

It is generally considered that sensor/actuator edge networks are too unreliable to do their own management, so that gateway nodes (PCs) or a cloud connection is necessary. In this use case we will test and present a platform that increases the resilience of sensor/actuator edge networks so that they are able to reliably execute basic management tasks directly on the edge nodes themselves. The platform provides reliable decentralized communication, storage, and computation abilities, by leveraging CRDTs (Conflict-Free Replicated Data Types) and hybrid gossip algorithms. This lowers cost, reduces dependencies, and simplifies maintenance.

Our system has no single point of failure. It consists of three parts: Grisp embedded system boards, Lasp CRDT-based key/value store, and the Partisan hybrid gossip-based communication library. We choose Grisp because it directly implements Erlang on the hardware, which simplifies system development, and because it directly supports Pmod sensors and actuators and has built-in wireless connectivity. Computation and storage are limited, but adequate for many management tasks. The system being tested will be a prototype that is able to run applications on networks of Grisp boards. With this system we are going to start evaluating the proof-of-concept application for autonomous irrigation for precision agriculture.

In order to evaluate our system, and due to the fact that we have limited time before the end of the project due to the restructuring of the use case, we will deploy a sensor array in the lab where we will test the aforementioned parameters. Following, and in coordination with the academic partner we are going to use emulators that are being used in Universities (e.g. WSNet) to test the network in a larger scale and in order to show the performance under various networks scenarios based on several performance criteria.

(a) Timeline

We target to have the evaluation procedure finished by the end of October 2019.

(b) Experiment description

We will deploy Grisp nodes in a mesh network topology in the lab. These nodes will be equipped with soil moisture sensors and actuators in order to emulate reality as closely as possible. The nodes will be running the LightKone artifact software described above. We will perform experiments using the real setup and using the network emulator in order to measure:

- New node discovery in the network. Success threshold: 90%.
- Nodes breakdown. When a node of the network breaks down then the sensor array should be reconfigured automatically and continue operating. Success threshold: 80%.
- Network lifetime. We will try different setups in order to extend the energy life of the network. This experiment would be mainly evaluated through simulators. As a threshold, the lifetime of the nodes should be at least one irrigation season: approximately 6 months.
- Information effectiveness. The actuators must be triggered as soon as they actuation threshold is achieved. Success threshold 90%.
- Sensing data accuracy. The node should detect the case that the sensing data (e.g. malfunction of the sensor) are out of the norms and discarded. Success threshold: 90%

2.2 UPC's specific evaluations

In this section, we provide the details for the evaluation of UPC's use case, to be performed in the context of the Guifi.net Community Network (CN).

To recap the use case described in D2.1,Guifi.net is a Community Network where the network infrastructure is crowd-sourced by the different participants (individuals, collectives, enterprises, etc.). The deployment, maintenance and operation of this decentralised network are shared among the diverse participants of the different geographical areas connected. The current project consists of efforts to improve two comoponents: the first, the dynamic assignment of the server - router assignment (case 1), and second, the monitoring data storage (case 2).

2.2.1 Performance evaluation objectives

The proposed use cases aim to use AntidoteDB in an edge network environment. Compared to the networks in computing clusters of data centers, the Guifi.net environment can present significant variations in the dynamics of the network properties, affecting latency and bandwidth, even including eventual disruption of the connection between different AntidoteDB instances. It is also typical, for edge network environments, the need to cope with devices of different computing capabilities and availabilities, while in a data center the resources' attributes can be specified and guaranteed to be more homogeneous. Therefore, the challenges for the performance of AntidoteDB are set by the conditions at the network edge, and by a use case integrated in the Guifi.net infrastructure which needs to operate under realistic conditions.

Among the performance objectives, we have initially identified:

- **Objective 1 (O1): Understanding, for edge networks, the functional performance of AntidoteDB in different settings and application scenarios**
  These settings can be chosen from different options to form AntidoteDB data centers and interconnect them, with the interest to understand how each option performs under different network and available resources scenarios. Different application scenarios can be obtained from the operation of AntidoteDB in the use case: in the first phase the usage consists in storing the *monitoring server* $\Leftrightarrow$ *network devices* mapping, while in a second phase, we also plan to store data from the devices monitoring. The expected outcome is to obtain AntidoteDB usage experiences in edge environments, to be shared with the users and developers community.

- **Objective 2 (O2): Understanding the suitability of AntidoteDB for a shared edge device**
  The targeted scenario considers AntidoteDB service provision in the microclouds of Guifi.net and aims to derive recommendations. Since edge computing devices can also offer end-user services, maintaining an acceptable quality of experience in this context is important. The outcome is to obtain insights of the options to provide the AntidoteDB-based monitoring service to allow performing tasks in a sustainable way, not breaking other services' provisioning.

The measurements obtained from the experiments aim to characterize the different application scenarios and the storage service performance AntidoteDB provides, as well as measure technical metrics related to the cost of using AntidoteDB (e.g., related to resources consumption).

(a) Metrics and criteria

Table 2.2.1 shows the metrics and experiments in order to respond to Objective 1 (i.e., understanding for edge networks, the functional performance of AntidoteDB in different settings and application scenarios).

Metric	**Experiment**	**Evaluation criteria (min // normal // outstanding)**
Stability	long-term (several days) operation	one // part // all DC operational
Robustness	AD DCs operation under a set of changing conditions (workloads, network, node capacity)	one // part // all DC operational
Heterogeneity	run AD on different HW	x86 // rc-x86 // ARM and x86 // ARM and rc-x86
Flexibility	experiment case 1 and case 2	- // case 1 // cases 1 and 2 supported

Table 2.2.1: Metrics and experiments for performance objective O1. "rc-x86" stands for *resource-constraint x86*.

Table 2.2.2 shows the metrics and experiments in order to respond to objective O2, i.e. understanding the suitability of AntidoteDB for a shared edge device.

Metric	**Experiment**	**Evaluation criteria (min // normal // outstanding)**
Multi-tenancy (QoS)	Operate AD DCs along other applications on the same node	- // resource usage with regards to edge environments / -
User experience (UX) in community env.	Dynamic join/leave of AD DCs. Setup tools. Configuration facility. Monitoring and logging facilities.	manual expert configuration // user-configurable // self-configuring
Data protection	Authentication possibilities. Data access protection.	Configurable by expert // available feature // out-of-the box configuration

Table 2.2.2: Metrics and experiments for performance objective O2.

The key success criteria for the two objectives are the following:

- AntidoteDB storage layer is functional in environment (O1).
- AntidoteDB storage layer is suitable for environment (O2).

The performance goals can be quantified by the following measures:

- The *resource consumption* vs *available resources under different workloads and conditions* **ratio**.
- The number of **features** consisting in the *provision of improved features compared to the current (centralized) monitoring solution.*

The deployments that will be done for the experimental evaluation aim to demonstrate that the desired features of the new distributed monitoring system implementation are achieved, while at the same time, insights on the cost of using the new system will be gained.

The minimal planned outcome is that the new monitoring service runs permanently for long-term study and evaluation in Guifi.net along with the existing one.

While the existing monitoring system is operational and used, its limitations of having a single point of failure and lacking a storage service are visible daily in the traffic monitoring of the network and in the obstacles that its design represents for further improvements. A comparison of features of the old and new monitoring solution from different perspectives (e.g., technical, social, economic) is targeted to identify and qualify the benefits achieved by the new solution.

The timing of the experiments until the end of the project is organized in two cycles of six months. In the first cycle, from 01/2019 (M25) to 06/2019 (M30), case 1 will be experimented in order to have consolidated results by 04/2019 (M28), which may lead to an experience paper to be submitted for publication. The second cycle covers the period from 07/2019 (M31) until 12/2019 (M36) and addresses both cases 1 and 2. We expect that, for case 1, a new round of experiments may be needed to deeper investigate determined issues identified in the previous results. At the same time, these experiments can be combined with case 2 in order to assess the integrated functions of the distributed storage service. The gained results are expected to be elaborated in a performance evaluation paper.

2.2.2 Edge node hardware selection

The hardware of the computing devices used in the Guifi.net environment is diverse and heterogeneous. Devices usually range form low-end tiny SBCs to mini-PCs; occasionally, more powerful decommisioned desktop computers or servers are recycled and repurposed as computing devices, shared among a group of users.

To give a rough idea, SBCs are commonly based on single/dual-core ARM processors with 1 GB of RAM, while mini-PCs typically use low-power dual/quad-core x86 CPUs devices with 2-4 GB of RAM.

(a) x86-based SBCs and mini-PCs

In the context of Guifi.net, different generations of x86-based SBCs and embedded computing devices have been widely used over time to perform different network-related tasks. Initially, in the 2000s, repurposed desktop computers were used as 24/7-available tiny servers hosting local contents and network-related services. However, because of their much lower energy footpring, industrial-grade system boards based on 32 bits AMD Geode LX CPUs, like the various ALIX[1] models, were popular to provide these services. Later, these boards were superseded by the newer APU and APU2[2], based on 64 bits AMD CPUs with more memory and power.

In the recent years, commodity mini-PCs based on the Intel Atom CPU family, such as the MINIX NEO Z64[3] and Z83-4[4] devices have become very popular among the community of users because of both their low price and small energy footprint and their capability to run virtual machines and containerized applications. The aforementioned devices are pictured in Fig. 2.2.1, as well as similar ones also used.

For testing and evaluating the use case applications, we have deployed a testbed with 10 MINIX NEO Z83-4 devices equipped with 4 GB of RAM and 32 GB of MMC disk, running Debian Stretch[5] 64 bits. We used the AntidoteDB Docker images from Docker Hub[6] to run AntidoteDB instances in the mini-PCs. The AntidoteDB containers were instantiated correctly on the x86-based 64 bits mini-PCs and SBCs. The small size of the devices allows moving them easily between different locations with varying interconnection capabilities (wired/wireless, data center-like/lossy, etc.) in order to test realistic network environments. Fig. 2.2.3 shows five of the MINIX NEO Z83-4 devices connected to the same wired network, running a [geo-replicated] AntidoteDB cluster, while the other group of five devices runs another replica of the cluster. The two clusters connect between them by means of two wireless mesh network hops.

[1]PC Engines ALIX system boards: https://www.pcengines.ch/alix.htm
[2]PC Engines APU2 system boards: https://www.pcengines.ch/apu2.htm
[3]MINIX NEO Z64: http://www.minix.us/products/NEOZ64.html
[4]MINIX NEO Z83-4: http://minix.com.hk/products/neo-z83-4
[5]Debian Stretch: https://wiki.debian.org/DebianStretch
[6]AntidoteDB Docker image: https://hub.docker.com/r/mweber/antidote/

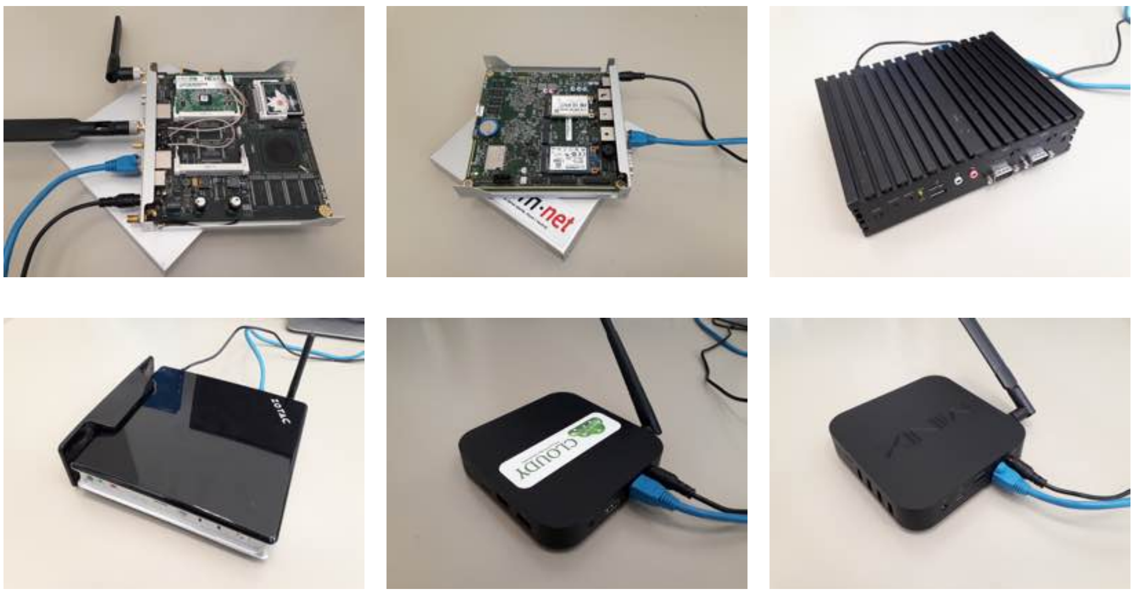


Figure 2.2.1: x86-based embedded system boards and mini-PCs that are commonly found in Guifi.net are part of the testbed used for testing and evaluating the UPC use case application.

(b) ARM-based SBCs

ARM-based SBCs may be considered not yet powerful enough as to efficiently provide enterprise-grade services to a large number of users. However, their computing capacity and memory size are growing rapidly to the point that current devices, such as the Raspberry Pi 3[7], have a performance comparable to a low/mid range laptop of the early 2010s.[8]. In addition to their reduced price and energy footprint, their popularity among end users as a platform for a large amount of do-it-yourself (DIY) projects make them very interesting hardware candidates for computation at the edge [2]. We explored the possibility that, besides the x86 mini-PCs, the use case application and, in particular, AntidoteDB[9], could be compiled and run as-is on ARM-based SBCs like the ones shown in Fig. 2.2.2.

Tables 2.2.3 and 2.2.4 show the results of compiling the frozen (i.e. stable) and trunk versions of AntidoteDB and their dependencies on several popular ARM-based SBCs with different characteristics (32 or 64 bits kernel, different operating system versions, 1 or 2 GB of RAM, etc.). We could observe a number of compilation errors, most of them directly or indirectly related to `eleveldb`[10], a dependence of AntidoteDB. These errors referred in Tables 2.2.3 and 2.2.4 are described as follows:

- Error 1: `eleveldb` fails to compile on 32 bits (both ARM armv7l and x86 i586) architectures. A possible solution could be to update the dependency rules to a newer `eleveldb` version.

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

[8]Roy Longbottom Raspberry Pi benchmarks: http://www.roylongbottom.org.uk/RaspberryPiBenchmarks.htm

[9]AntidoteDB source code at GitHub: https://github.com/SyncFree/antidote

[10]eleveldb - Erlang bindings to LevelDB datastore: https://github.com/basho/eleveldb

Device	**Raspberry Pi 2**	**Raspberry Pi 3**	**Raspberry Pi 3**
CPU	BCM2835	BCM2837	BCM2837
RAM	1 GB	1 GB	1 GB
OS	Debian Stretch (raspbian)	Debian Stretch (raspbian)	Debian Stretch (bamarni/pi64-kernel)
ARCH	armv7l	armv7l	aarch64
KERNEL	Linux 4.14.34-v7+	Linux 4.14.34-v7+	Linux 4.11.12-pi64+
AntidoteDB (frozen)	Error 1 (not solved)	Error 1 (not solved)	Error 2 (solved)
AntidoteDB (trunk)	N/A	N/A	Error 3 (not solved)
ElevelDB	OK	Error 3 (not solved)	OK
Snappy	OK	OK	OK
Google LevelDB	OK	OK	OK
IP	10.228.207.25	10.1.24.150	10.228.207.15

Table 2.2.3: Evaluation of AntidoteDB compilation on different RaspberryPi SBCs.

- Error 2: `eleveldb` fails to compile on 64 bits ARM aarch64 due to the architecture being unknown to one of its dependencies, `snappy-1.0.4`. A possible solution could be to update the `config.guess` and `config.sub` files from the embedded `gettetxt`[11] package.

- Error 3: `eleveldb` fails to compile on 64 bits ARM aarch64 due to its dependency `snappy-1.0.4` not supporting the architecture. A possible solution could be to update to `snappy-1.1.7` before compiling `eleveldb`, which ships an up-to-date `gettext` package.

- Error 4: `eleveldb` fails to compile on aarch64 due to its dependency `LevelDB`[12] not supporting atomic pointers on this CPU architecture. A possible solution could be to update to the newer `LevelDB` v1.1.18-2 which supports aarch64.

We conclude that, on 32-bits ARM devices, the chosen version of the `eleveldb` dependency can not be compiled. However, newer versions can be compiled successfully. On 64-bits ARM devices, the dependency `eleveldb` cannot be compiled because of two of its dependencies (`snappy` and `LevelDB`) failing to compile on this architecture. Newer `snappy` versions can be compiled, while the latest `eleveldb` version most likely will not compile, as it depends in turn on `Basho's LevelDB` which does not support aarch64 and seems not to be maintained anymore. However, the original project where the package was forked from, `Google's LevelDB`, could possibly be used, as it compiles successfully.

[11]GNU gettext: https://www.gnu.org/software/gettext/
[12]leveldb: https://github.com/basho/leveldb

Device	CubieBoard 2	CubieTruck	Pine64	Alix3
CPU	Allwinner A20 Dual-Core Cortex-A7	Allwinner A20 Dual-Core Cortex-A7	Allwinner A20 Dual-Core Cortex-A7	Geode LX 800
RAM	1 GB	2 GB	2 GB	512 MB
OS	Debian Stretch	Debian Stretch	Debian Stretch (armbian)	Debian Jessie (back-ports)
ARCH	armv7l	armv7l	aarch64	i586
KERNEL	Linux 4.9.0-6-armmp-lpae	Linux 4.9.0-6-armmp-lpae	Linux 4.14.36-sunxi64	Linux 4.9.0-0.bpo.5-686
AntidoteDB (frozen)	Error 1 (not solved)	Error 1 (not solved)	Error 2 (solved)	Error 1 (not solved)
AntidoteDB (latest)			Error 3 (not solved)	
ElevelDB	OK	OK	Error 3 (not solved)	OK
Snappy	OK	OK	OK	OK
Basho LevelDB	OK	OK	Error 4 (not solved)	OK
Google LevelDB	OK	OK	OK	N/A
IP	10.228.207.24	10.228.207.16	10.228.207.26	10.1.33.36

Table 2.2.4: Evaluation of AntidoteDB compilation on diverse SBCs.

Interesting pointers can be found on how to use the system's `snappy` library[13] and on LevelDB supporting the aarch64 architecture[14]. Therefore, despite not being able to compile AntidoteDB on any of the ARM boards we have evaluated, it seems that updating the dependencies (namely eleveldb, snappy, leveldb) to more recent versions would make AntidoteDB work (or, at least, compile). These dependencies, however, are managed by external organizations and it might be difficult to update them.

2.2.3 Testbed for experimental evaluation

Our testbed aims to support the experimental evaluation of the use cases in the Guifi.net environment, presented by UPC in WP2. For this purpose, we have installed and configured a set of devices and machines connected to the Guifi.net network on which to deploy the use case applications.

Monitoring servers in Guifi.net are often installed as a service on a diversity of hardware, which can range from resource-constraint SBCs to desktop servers. Devices are

[13]eleveldb pull request #244: https://github.com/basho/eleveldb/pull/244
[14]LevelDB issue #268: https://github.com/Level/leveldown/issues/268

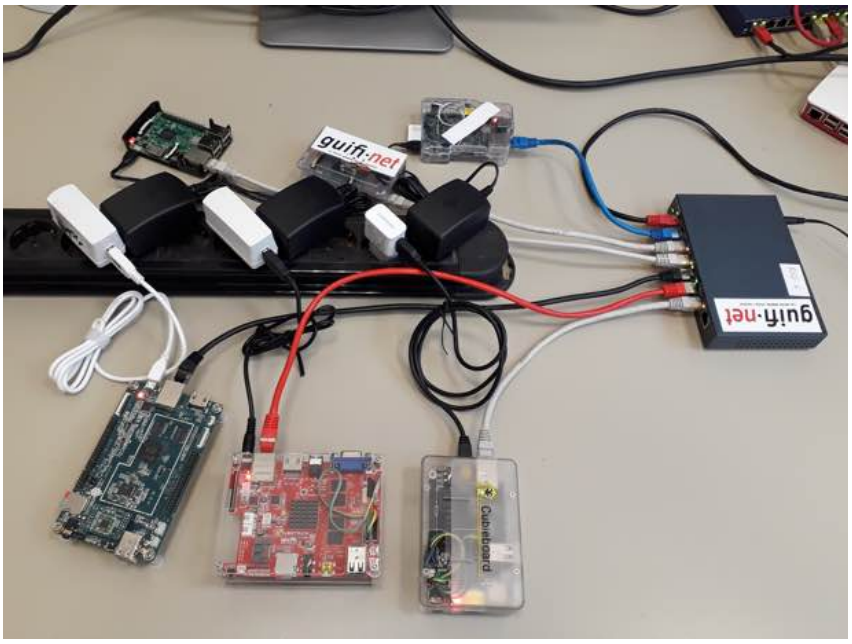


Figure 2.2.2: A number of different ARM-based SBC during the evaluation process.

often purposed not only for monitoring, but they run several other services, be them related to the network as to user applications. This scenario fits into the provision of microclouds in Guifi.net, which are a tool for the community to share services within the network, and that are based on heterogeneous computing devices contributed by independent owners. The monitoring service can become part of the services provided by devices in the microcloud, hence our testbed will be oriented at this scenario.

For the initial evaluation, we have chosen to build a testbed with a set of resource-constraint physical nodes and virtual machines. The physical nodes are edge devices which host AntidoteDB and consist of five MINIX Z83-4 mini-PCs (Figure 2.2.3). We have installed the Cloudy platform on these devices, plus the packages needed to run AntidoteDB instances as Docker containers. Other physical nodes to become part of the testbed are a number of RaspberryPi SBCs that, while not being used to host AntidoteDB, they are part of the microcloud and may host distributed components of the monitoring application which act as clients for AntidoteDB. Finally, the virtual machines are instantiated on several Proxmox servers.

Combining both physical and virtual machines, AntidoteDB instances are deployed and clustered in a way that they create geo-replicated data centers. The different data centers are connected with each other through the Guifi.net network. The fact that the instances are on different geographic locations causes heterogeneity in the network characteristics of bandwidth and latency between the different replicas.

2.3 Scality Specific Evaluations

In this section, we provide specifics of the Scality use-case evaluation plan.

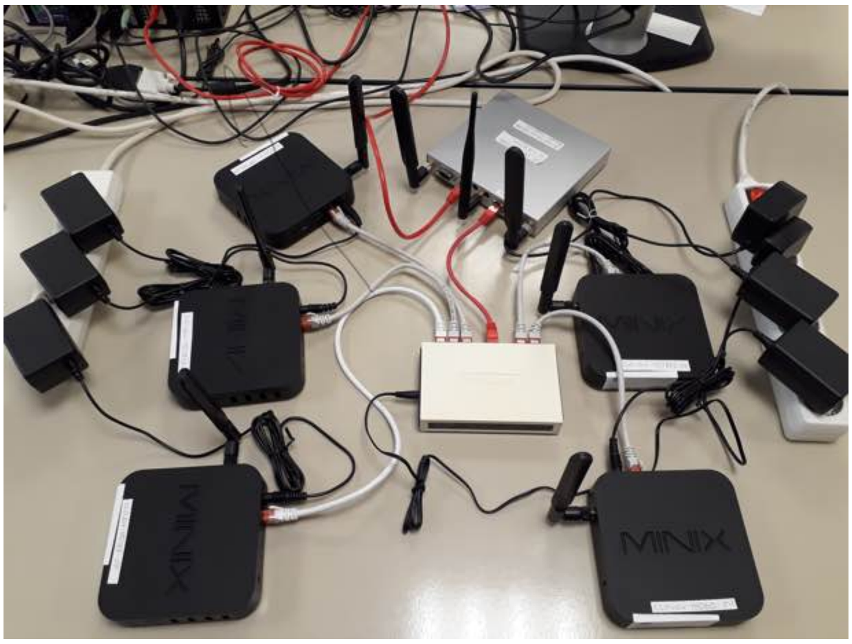


Figure 2.2.3: A group of five MINIX NEO Z83-4 devices forming a cluster.

2.3.1 Use case overview

Scality's core technology is a scalable object storage platform called the Scality Ring, with file-system interfaces and the Amazon Web Services (AWS) Simple Storage Service (S3) interface, made popular in that last decade with the growth of public cloud platforms. The storage system provides customers the option of deploying large scale object storage systems on premises in much the same way they are deployed on public clouds. With an increasing interest in deploying storage and applications across both private and multiple public clouds, Scality has introduced the Zenko Multi-Cloud Controller, an open-source project that provides a unified storage interface across clouds. Zenko provides an abstraction layer that allows developers to use multiple clouds transparently, by providing a single unifying interface (using the Amazon S3 API) while supporting multi-cloud backend storage systems. Backend storage systems support both on-premises and other public cloud services, including: Amazon S3, Microsoft Azure and Google Cloud Platform (GCP).

Zenko provides a federated metadata search capability across all cloud name spaces. This enables applications to attach metadata attributes on each object, and perform queries to retrieve objects based on attribute matching criteria independent of the data location.

Zenko supports both in-band updates, performed directly through Zenko, and out-of-band updates, where applications communicate directly with the backend storage systems. In the case where application create or update objects directly through Zenko (in-band), it becomes aware of objects and tracks changes to metadata so that it can provide federated search. Out-of-band updates are captured using trigger mechanisms offered by backend storage systems. When an update is performed in a backend storage system, Zenko is (eventually) notified about the change.

Currently, Zenko supports federated metadata search by capturing and storing object metadata in a distributed database deployed on a single data center, and maintaining indexes on metadata attributes. The goal of this use case is to improve the search system's data and computation placement flexibility.

To achieve this we will use Proteus, a geo-distributed framework for analytics computations on federated data stores. Proteus maintains materialized views and performs stateful data-flow computations. The framework is designed to enable flexible state and computation placement according to SLA considerations. More specifically, for this use case we will use Proteus as a geo-distributed query processing framework, by instantiating materialized views as secondary indexes and search result caches. For a detailed description of Proteus, we refer to deliverable D6.2.

The query processing sub-system added to Zenko is composed of a modular geo-distributed hierarchical network of microservices, termed Query Processing Units (QPUs). QPUs act as stream operators performing bidirectional data-flow computations, and maintain internal state used for processing queries. We will use three types of QPUs:

- Indexing QPUs that maintain secondary indexing structures.
- Caching QPUs that store the responses to selected queries.
- Filtering QPUs that read the underlying data store and filter objects matching a given query.

Proteus is by design modular and flexible. It allows administrators to deploy varying QPU network configurations by using different types of QPUs, network structures, and data/computation placement strategies. These different system designs make various trade-offs and are suitable for different application requirements.

We will experiment with multiple configurations for the proposed query processing system. A starting point will be a query processing system design implementing a geo-distributed cross-cloud index for metadata attributes. For this design we will use indexing QPUs. The index will be distributed across multiple cloud storage systems; a part of the index will be placed locally on the location (data center) of each storage system, and be responsible for indexing data stored in that particular system. Each of these parts will be additionally partitioned for scalability. Each index partition will be implemented as an indexing QPU. This approach is particularly promising for the out-of-band update case, where indexes can be generated locally and updated together with the data.

2.3.2 Evaluation Objectives

The aim of the experimental evaluation will be to validate the following:

- **Evaluate the flexibility of the metadata search system.** The current search system implementation make certain assumptions about the cloud applications behavior and making the corresponding trade-offs. However, cloud applications have varying characteristics and requirements. For example, different applications may have write- or search-dominated workloads, some may require low search latency while others always consistent search results. The expected outcome is to demonstrate that in some extend the implementation of Zenko's search system using Proteus can be adjusted to target different application requirements and characteristics, by allowing administrators to flexibility on state and computation placement.

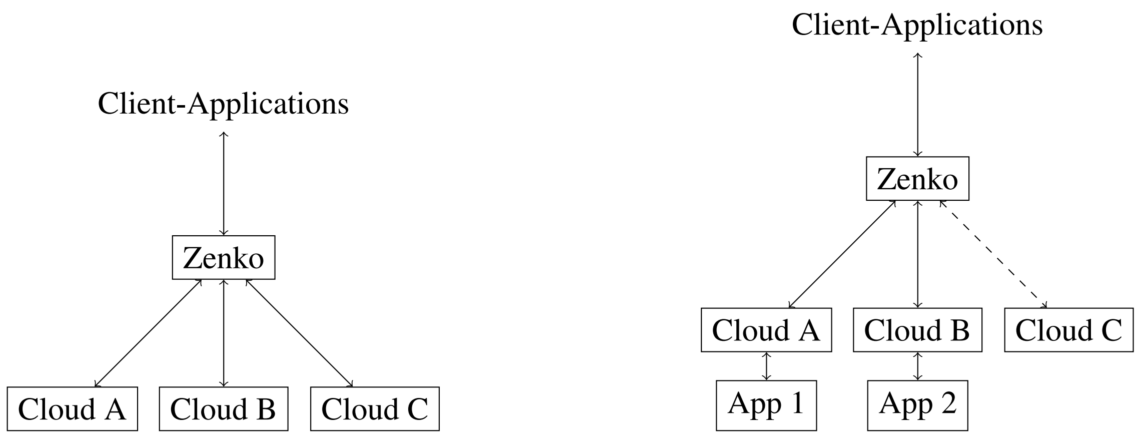


Figure 2.3.1: Deployment scenarios for Zenko with in-band and out-of-band updates.

- **Evaluate that the system can provide scalable metadata search.** We will gradually increase the scale of the evaluation workload by increasing the number of clients, the number of stored objects, and the number of metadata attributes per object. The purpose is to show that the system can be used as a practical search service for multi-cloud applications.

- **Understand the effectiveness of edge-based partial index methods in reducing query staleness and load of the central system.** Proteus enables the search system's data and computations to be flexibly placed across a geo-distributed system architecture, including closer to the clients at the edge. Placing parts of a geo-distributed index closer to client devices can potentially improve search result freshness in the case of out-of-band updates, and reduce the load of the data center. The expected outcome is to expand the design space of distributed query processing with new points in which data and computations are placed closed to the edge of the system.

- **Evaluate the costs associated with providing a solution.** In the case of an on-premises deployment of the solution, the costs are related to the number of servers and their load as is required to provide a reliable service. In the case that the infrastructure is deployed in a public cloud environment using a Kubernetes based service infrastructure such as Elastic Container Service for Kubernetes (EKS) at AWS or on the Kubernetes Engine on the GCP, for instance the direct cost of operating the service can be studied. Since there can be important financial implications for data flows in or out of the platforms or for specific services, the financial impact of certain design choices can be significant.

Additionally, we aim to investigate mechanisms for performing queries with resource consumption and cost constraints. Our goal is to enable users to perform "best-effort" queries by specifying limits on parameters such as runtime, energy consumption, or cost. Query processing will run until the specified bounds are reached, and the resulting query results will be returned. Our goal is to research mechanisms for obtaining the best possible query results given specific resource consumption constraints.

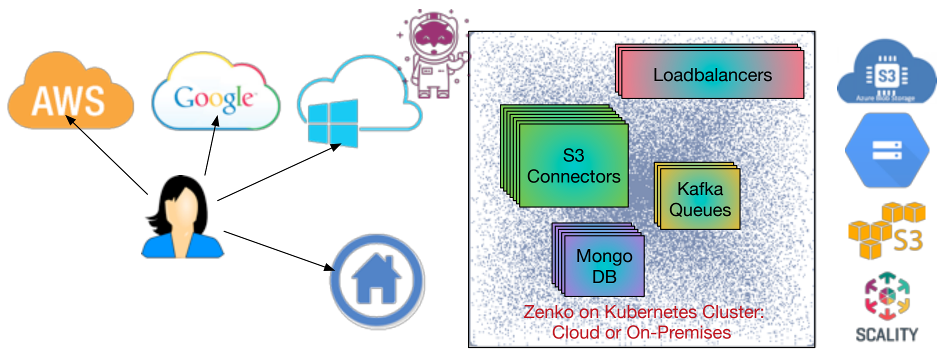


Figure 2.3.2: The Zenko Basic Architecture

However, this work is still in an early stage. With the components deployed in cloud environments using virtualized and shared components, the true resource consumption can be difficult to determine.

2.3.3 Experiment Design

(a) Architecture

Zenko is deployed in a Kubernetes [15] cluster of 5 servers, either on premises or in a public cloud, in order to provide high availability. The system can be managed from a public portal at zenko.io. This will allow it to be deployed in a straightforward fashion for the evaluations, on one or many different clouds.

The evaluation will take advantage of the geographically distributed nature of public cloud infrastructure. The software solution will be deployed across a number of distant sites with a centralized query interface as a reference. The number of satellite sites is expected to be in the range of 3-5. The geographically dispersed nature of global cloud infrastructures can provide latencies of over 300msec between certain sites. This allows tests to be performed which tests the liveliness and efficiency of the models across a range of conditions.

(b) Experiment Description and metrics

The primary objective of our evaluation is to demonstrate that the proposed system has the ability to optimize different metrics as required for a specific usage. To confirm the capability of the proposed solution to accomplish this, we will evaluate different scenarios, each scenario requiring the optimization of different metrics.

For each scenario we will deploy a search system implemented using Proteus and adjusted for targeting the requirements of the specific scenario. Data will be ingested and/or deleted using both in- and out-of-band operations, and the target metric will be measured. We will also measure the target metric for the metadata search system currently implemented in Zenko, and use this as the baseline for comparison.

[15] https://kubernetes.io/

Metric	Explanation
Query latency	The elapsed time required to complete a fully distributed query
Query liveliness	Minimize the delay before remote updates appear in queries
Cross site bandwidth	Minimize the delay before remote updates appear in queries
Storage overhead	Minimize the quantity and cost of storage required to deploy the chosen architecture

The expected outcome of this approach is the optimization of the desired metric in all scenarios by the proposed design, but the negative impact on other metrics will be considered as well.

It is expected that loads will likely have a smaller effect on the results than the chosen architecture. The public cloud infrastructures that will be used for the testing are generally sufficiently sized to support significant loads and bandwidth. The query liveliness is expected to be the most adversely affected by high loads. The ability to demonstrate the ability to significantly affect the metrics that are being optimized is a key goal. By having a composable design of the search architecture, different performance and cost criteria can be met by redistributing the Proteus components.

(c) Tools

Benchmarking tools for object storage platforms exist with the most well known tool being *Cosbench* developed by Intel [16]. This tool will most likely be used extensively for generating object creation and deletion traffic during the evaluations. Object-storage based search queries are currently not supported on most public cloud platforms and so query performance testing tools will likely need to be developed specifically for these evaluations. There are standard tools for the generation of HTTP traffic such as *httperf* [17]. Search queries on the Scality platform are REST based and so these tools should be able to be adapted to generate the search loads.

2.4 Stritzinger Specific Evaluations

In this section, we provide specifics of the Stritzinger use case evaluations

2.4.1 Use case overview

().1 Distributed RFID Cache Stritzinger has implemented an Erlang-based implementation of the RFID protocol and a local cache for the data on embedded nodes which are running a predecessor of the current GRiSP platform for its customer Bosch-Rexroth

[16] https://github.com/intel-cloud/cosbench
[17] https://github.com/httperf/httperf

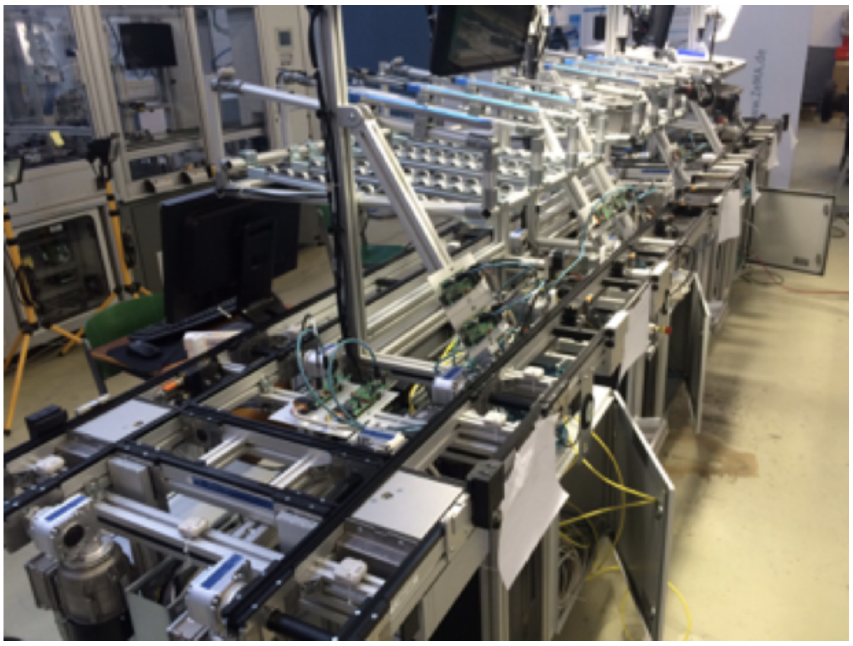


Figure 2.4.1: SmartF-IT Research Project Demo which is similar but a bit larger to the available one

in the past. Depending on support from Bosch-Rexroth, Stritzinger can evaluate the distributed caching system for the RFID tag content and other content associated with a tag (digital twin) on a prototype industrial transport system or a demonstrator.

We have secured access to a real-world demonstrator for a industrial transport system with RFID tag readers. The demonstrator allows multiple paths for workpiece carriers and has up to 15 networked embedded systems with RFID support attached to it. The demonstrator is located at Bosch-Rexroth in Stuttgart, Germany.

2.4.2 Evaluation Objectives

The aim of the experimental evaluation will be to validate the following top level requirements identfied in the SysML Requirements Analysis in deliverable D2.2:

- Allow concurrent writes with last writer wins semantics
- Communication with the manufacturing process
- Data of different RFID tags is independent
- Localize workpiece
- Mesh like network topology
- Persistence of information
- Runs on existing RFID reader hardware
- Store processing information on each workpiece

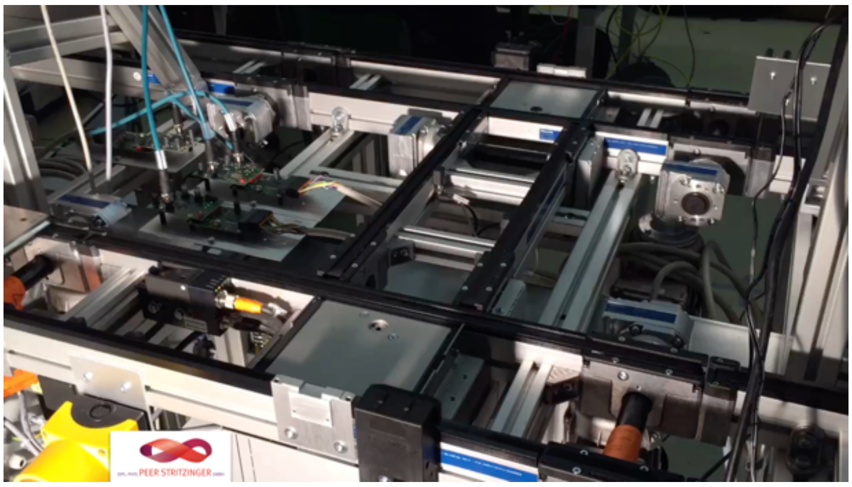


Figure 2.4.2: RFID Reader Prototype Boards controlling a converyor belt distributedly

- RFID tags shall stop as little as possible

If against all odds there will be no real-world demonstrator available, Stritzinger can evaluate the implementation also in a simulator running the Erlang based implementation on non-embedded hardware and simulating the movement of the tags. Simulating an application like this might also open up the possibility for collaboration with one of the academic partners.

2.4.3 Experiment Design

(a) Timeline

We plan to have a working prototype implementation by end of October 2019 and start testing on the Bosch-Rexroth hardware in November 2019

(b) Architecture

We have a maximum of 15 embedded nodes available with 200MHz PowerPC based CPUs and 64MiB of RAM. They have a Texas Instruments TRF7970 RFID transceiver and ISO15693 Antennas. The workpiece carriers are equipped with IS15693 tags with 2kiB FRAM which supports millions of writes.

(c) Experiment Description

The workpiece carriers will be moving along the conveyor belt passing by several RFID antennas where they can potentially stop. Since there is no real processing done on this demonstrator, we have to emulate processes writing to the tags with random data.

We will be able to verify in long running experiments that data integrity remains intact and that all invariants remain valid.

Furthermore, we will be able to demonstrate that stopping at the RFID antennas is substantially reduced.

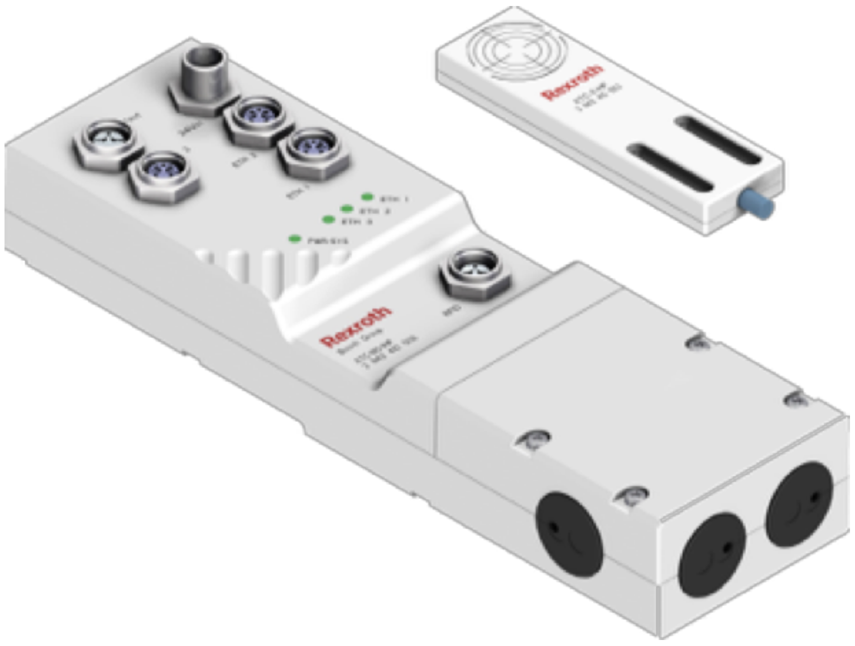


Figure 2.4.3: RFID Reader Prototype with Antenna

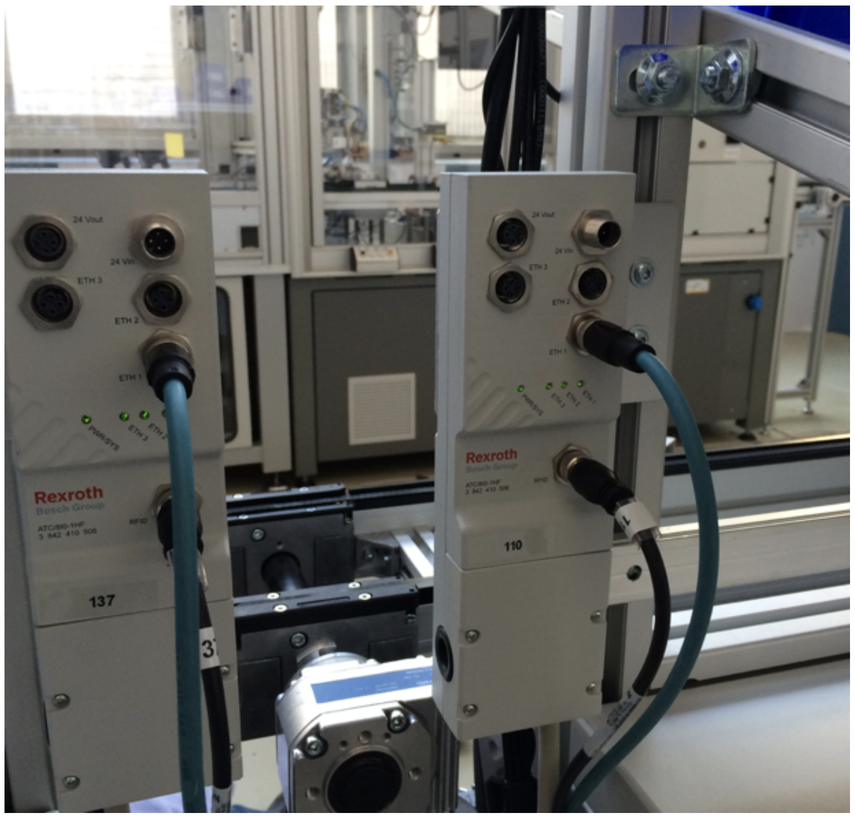


Figure 2.4.4: RFID Reader Prototype mounted at SmartF-IT Demonstrator

To show scalability of the application we will use a simulator that runs the full Erlang application in separate Erlang VMs and simulating the RFIDs moving between them.

(d) Further Contributions

Stritzinger has described several other use cases and is also contributing to the general technology base in WP3 and WP5.

(d).1 Smart Metering Use Case For the smart metering use case Stritzinger has been working on market research and a business case for a more concrete use case. After initial research it was decided to not pursue this business sector in the near future.

(d).2 GRiSP GRiSP is a software framework that allows running Erlang applications on small embedded systems which are common at the Edge of the network. This allows the same codebase to be deployed on these small systems as on either larger edge systems or in the cloud. We also developed a hardware platform to make it easier to build a prototype of such a system, reducing our time-to-market for Internet of Things and Cyber Physical Systems and other embedded systems. The hardware was already in development before the start of the LightKone project and the development costs are not financed by LightKone. However, the corresponding software platform was greatly improved and completed during the project and is used together with the hardware platform by research partners.

(d).3 LASP on GRiSP UCL has successfully ported their distributed systems platform LASP onto the GRiSP board in the last months and will continue the evaluation of wirelessly networked GRiSP hardware as sensor nodes together with LASP.

Initial evaluation to determine if GRiSP is a viable platform to run LASP and process sensor data, is complete at this point. Concrete plans for further evaluation are not known at this time since a new set of master students will start to continue the work in fall 2018.

(d).4 Antidote on GRiSP UPMC is starting to port the client-cache component, EdgeAnt, of Antidote on GRiSP and will start evaluating the possibilities of sensor nodes sharing their data through the Antidote database.

A first step of the evaluation will be the verification of the portability of the Erlang code to the GRiSP board. The evaluation of resource usage to examine the viability to run EdgeAnt on small Edge IoT devices will be the next step. Further plans are not known at this point, but will become known later in the project.

(d).5 GRiSP for research outside of LightKone Several institutions are using the GRiSP platform for IoT and other embedded systems research. The University of St. Andrews is using GRiSP for autonomous robotic vehicle research. The University of Bournemouth has a group that starts a project with GRiSP that researches small swarms of interacting robots which combines Edge networking with robotics. The University of Kent has plans to use GRiSP in research of which details are not yet known. GRiSP was also used for research at the Technical University Munich on hard real-time Erlang and

AGH University of Science and Technology Krakow for IEEE 802.15.4 Personal Area Networking.

(d).6 GRiSP for education To evaluate the reduced complexity of getting started with IoT and other embedded systems applications, we are successfully giving tutorials at conferences and at universities (Erlang User Conference Stockholm 2017, CodeBEAM Stockholm 2018, AGH University Krakow 2018) and help teaching initiatives like School of Erlang Krakow. Especially the students of School of Erlang, who have just started learning Erlang, reported ease of code deployment on the boards and sensor access.

(d).7 Improvements to Erlang transparent distribution protocol Stritzinger will continue to implement and evaluate its planned improvements to the Erlang distribution protocol. So far, they built a prototype to get around the head-of-line blocking problem in Erlang Distribution (a large message highly increases the latency of smaller messages). The next step will be working on the scalability issues of the protocol.

Stritzinger plans to evaluate the scalability improvements in larger mesh-networked cyber-physical nodes which run under the GRiSP platform if one of its customer projects proceeds at Bosch-Rexroth and will give them access to a network of industrial transport controllers. In absence of this, they will need to test either with cloud instances or GRiSP boards.

(d).8 IEEE 802.15.4 Protocol Stack for GRiSP As of recently Stritzinger received interest from potential customers or partner companies in mesh networking of GRiSP sensor nodes and gateways to other protocols. They plan to implement a protocol stack for IEEE 802.15.4 compatible hardware that can be connected to the GRiSP hardware boards (PMOD RF2 from Digilent). From there they want to explore if either one of the standardized protocols on top of 802.15.4 like Zigbee, 6LoWPAN or ISA100.11a are useful to implement or if they rather want to implement a proprietary protocol of their own design.

Chapter 3

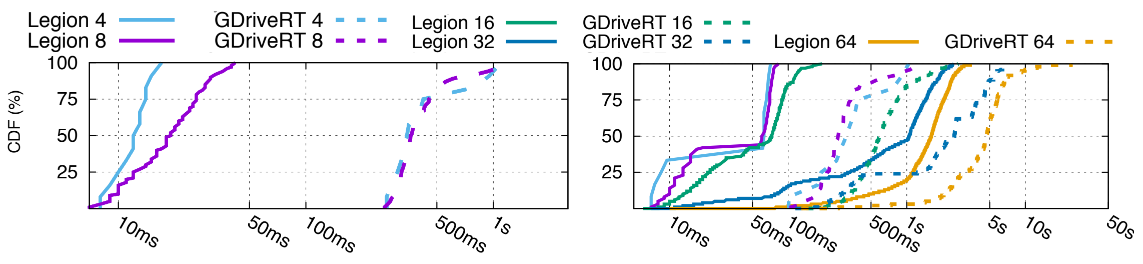
Evaluations in an Academic Context

In this section, we present preliminary evaluations of several software prototypes that have been developed in WP3, WP5 and WP6. These evaluations focus synthetic workloads; for Legion, we were additionally able to conduct a user study using a collaborative game.

3.1 Legion

Legion is a framework for developing web applications that leverage peer-to-peer communication, thus allowing to move web applications to the edge of the network. The web applications running on the client browsers can interact directly by using peer-to-peer WebRTC connection or indirectly through the server running in the cloud. The interaction with the server is mediated through a set of adapters that allow to connect to different cloud infrastructures. The development of Legion has started in the last period of the SyncFree project; a detailed description can be found in deliverable D5.1.

In this section we present the evaluation of Legion with an emphasis on the operation of Legion when using the adapters to inter-operate with the Google Drive Realtime (GDriveRT) infrastructure (unless stated otherwise in our experiments, we ran Legion with all GDriveRT adapters enabled and with support for legacy clients disabled). Our evaluation mainly focuses on two complementary aspects. We start with an analysis of our experience in adapting existing GDriveRT applications to leverage Legion. Then, we present an experimental evaluation of our prototype, comparing it with the centralized



(a) All clients within the same datacenter (b) Clients distributed over 2 datacenters

Figure 3.1.1: Latency for the propagation of updates.

infrastructure of GDriveRT regarding the following practical aspects: (i) What is the impact on update propagation latency? (ii) What is the impact on application performance? (iii) How does the system behave when the central server becomes (temporarily) unavailable? (iv) What is the impact of using Legion in terms of load imposed on the central component and on individual clients? (v) What is the overhead for supporting seamless integration with legacy clients?

(a) Designing Applications

We start by describing a set of web applications that we have ported to Legion using the GDriveRT adapters.

Google Drive Realtime Playground: The Google Drive Realtime Playground [1] is a web application showcasing all data types supported by GDriveRT. We ported this application to Legion by changing only 2 lines in the source code.

Multi-user Pacman: We adapted a JavaScript version of the popular arcade game Pacman 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 that maintain 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 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.

Along with extending and porting this application to use the GDriveRT API, we also implemented the same game (with all functionality) using Node.js as a centralized server for the game to which the clients connect using web sockets (this implementation does not leverage Legion). This enables us to investigate the effort in implementing such an interactive application using both alternatives. The Node.js implementation of the game is approximately 2.200 LOC for the client code, and 100 LOC for the server. In contrast, the implementation leveraging the GDriveRT API has approximately 1.620 LOC for the client code, and 40 lines of code for the server side (used to run multiple games in parallel). This shows that an API such as the one provided by GDriveRT and Legion simplifies the task of designing such interactive web applications.

Creating the Legion version (using the GDriveRT adapters) required to change only two lines of code to the GDriveRT version (as described before). From a user perspective, the Legion version runs much smoother, which is also shown by our evaluation presented further ahead.

Spreadsheet: We have also explored an additional application: a collaborative spreadsheet editor. Each spreadsheet represents a grid of uniquely identifiable rows and columns, whose intersection is represented by an editable cell. Each cell can hold numbers, text, or formulas that can be edited by different users.

A prototype of the spreadsheet web application was built using AngularJS and supporting online collaboration through GDriveRT. The spreadsheet cells were modeled using a GDriveRT map. Each cell was stored in the map using its unique identifier (row-column) as key. Porting this application to the Legion API only required the change of 2 lines of code (as discussed previously).

Discussion: Our experience with porting these applications to leverage Legion shows that doing so is simple, as the programmer can easily use our GDriveRT adapters. Furthermore, this shows that carefully designing our framework to expose (through adapters) APIs that are similar to existing Web infrastructures is paramount to promote easy adoption of our solutions.

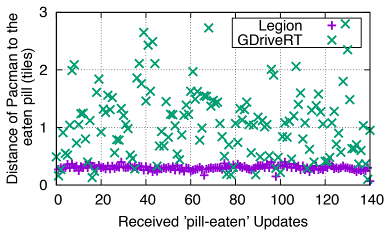
(b) Experimental evaluation

In our experimental evaluation, we compare Legion, with and without the use of adapters, against GDriveRT, as a representative system that uses a traditional centralized infrastructure.

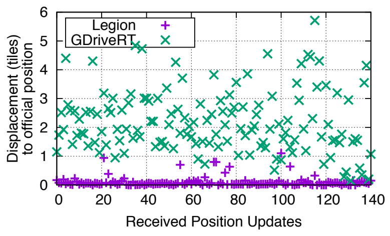
In our experiments, we have deployed clients in two Amazon EC2 data centers, located at North Virginia (us-east-1) and Oregon (us-west-2). In each DC, we run clients in 8 m3.xlarge virtual machines with 4 vCPUs of computational power and 15GB of RAM. Unless stated otherwise, clients are equally distributed over both DCs. The average round-trip time measured between two machines in the same DC is 0.3 ms and 83 ms across DCs.

Latency: To measure the latency experienced by clients for observing updates, we conduct the following experiment. Each client inserts in a shared map a key-value pair consisting of its identifier and a timestamp. When a client observes an update on this map, it adds to a second map, as a reply, another pair concatenating the originating identifier and the replier's identifier as the key, and as value an additional timestamp. When a client observes a reply to his message, it computes the round-trip time for that reply, with latency being estimated as half of that time. All clients start by writing to the first map at approximately the same time and reply to all identifiers added by other clients. Thus, this simulates a system where the load grows quadratically with the number of clients.

Figure 3.1.1 presents the latency observed by all clients for both Legion and GDriveRT. The results show that latency using Legion is much lower than using GDriveRT for any number of clients. The main reason for this is that the propagation of updates does not necessarily incur a round-trip to the central infrastructure in Legion. Furthermore, for 64



(a) Pacman displacement to eaten pills



(b) Pacman and Ghosts Displacement

Figure 3.1.2: Muti-User Pacman Performance assessment.

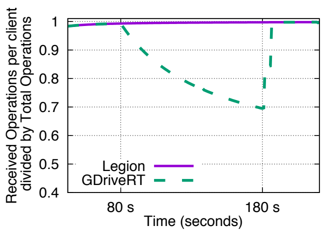
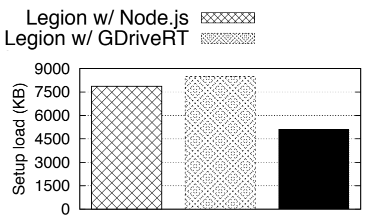
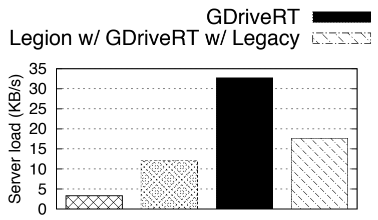


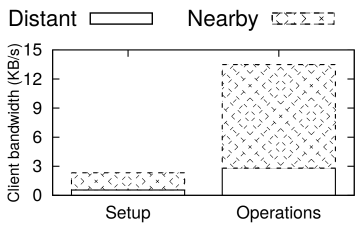
Figure 3.1.3: Effect of disconnection



(a) Server load during setup



(b) Server load during operations



(c) Client-to-client load

Figure 3.1.4: Network load.

clients, the 95th percentile for GDriveRT is almost an order of magnitude above Legion, suggesting that Legion's peer-to-peer architecture is better suited to handle higher loads than the centralized architecture of GDriveRT.

Multi-Player Pacman Performance: We now show the impact of Legion on the performance of applications in the context of our Multi-player Pacman game.

To that end, we conducted an experiment with volunteers, where we had five users playing Pacman (one player controlling Pacman, and four players for each of the ghosts). This experiment was conducted using five machines, in a local area network. Machines were running Ubuntu and clients executed in Firefox.

The goal of our experiment was to measure the displacement of entities in relation to their official position. As explained before, each client updates an object with the direction of its movement. The Pacman client computes and updates the official position of each entity periodically. Each client independently updates its interface based on the known direction of movement and the latest official positions. Displacement captures the difference between the position computed by a client and the received official position upon receiving an update. When displacement has large values, users see entities jumping on the game map. In particular, we measure: (i) the displacement of Pacman in relation to an eaten pill when an update reporting the pill being eaten is received by a client controlling a Ghost; and (ii) the displacement of Pacman and Ghosts when a client controlling a ghost receives an update for a position. Figure 3.1.2 reports the obtained results where the displacement is measured in tiles (the square unit that forms the interface). The board size of Pacman was 19×22 tiles featuring approximately, 59 turning points. Pacman and Ghosts move at approximately 3.33 tiles per second.

Figure 3.1.2a shows that when using Legion the interface is much more synchronized in relation to the real state of the system, showing that Pacman is visible by other players much closer to the eaten pill than when using the GDriveRT version of the game. Figure 3.1.2b reinforces these results showing that using Legion the displacement of entities in the game interface is significantly lower when compared with the game version that only uses GDriveRT, which is unable to send updates to all clients at an adequate rate.

Effect of disconnection: We study the effect of disconnection by measuring the fraction of updates received by a client. In the presented results, clients share a map object, and each client executes one update per second to the map (similar behavior was observed with other supported objects). We simulate a disconnection from the Google servers, by blocking all traffic to the Google domain using *iptables*, 80 seconds after the experiment starts. The disconnection lasts for 100 seconds, after which rules in iptables are removed so that connections can again be re-established.

Figure 3.1.3 shows, at each moment, the average fraction of updates observed by clients since the start of the experiment (computed by dividing the average number of updates received by the total number of updates executed). As expected, the results show that during the disconnection period, GDriveRT clients no longer receive new updates, as the fraction of updates received decreases over time. When connectivity is re-established, GDriveRT is able to recover. With Legion, as updates are propagated in a peer-to-peer fashion, the fraction of updates received is always close to 100%.

We note however that, while servers remain inaccessible, new clients cannot join the system. When leveraging Legion, clients that are active when the server becomes unavailable can continue operating regularly without noticing the server unavailability.

Network load: We now study the network load induced by our approach. To this end, we run experiments where 16 clients share a map object. Each client executes one update per second. The workload is as follows: 20% of updates insert a new key-value pair and 80% replace the value of an existing key selected randomly. The keys and values are strings of respectively 8 and 16 characters. We measure the network traffic by using *iptraf*, an IP network monitor.

In these experiments, we used the following configurations: *Legion w/ Node.js*: that uses our Legion server as backend. *Legion w/ GDriveRT*: that uses GDriveRT documents as backend. *GDriveRT*: that uses the original GDriveRT document as backend.

Figure 3.1.4a shows the total network load of the setup process, which entails making the necessary connections to the infrastructure and peer-to-peer connections. The incurred load using our own backend server is due to clients requiring to use this component to connect to each other initially (WebRTC signaling). Legion using GDriveRT as backend has a slightly higher cost due to the overhead of performing signaling through the infrastructure, which is less efficient. In both cases, only few clients obtain the initial object and propagate to other clients. Finally, in GDriveRT all clients download the shared data from the infrastructure.

Figure 3.1.4b shows the network load of the server without considering the initial setup load (computed by adding the traffic of all clients to and from the centralized infrastructure) for all competing alternatives. Results show that the load imposed over the centralized component is much lower when using Legion with GDriveRT as backend than when using only GDriveRT. This is expected, as only a few clients (active clients) interact with the GDriveRT infrastructure, being most interactions propagated directly between clients. Interestingly, the use of our server leads to an even lower load on the centralized component. This happens not only because the signaling mechanism used to establish new WebRTC connections among clients and the process for replica synchronization with the server is more efficient, but also because the data representation used by our backend is significantly more compressed. We run an additional configuration, (*Legion with GDriveRT w/ Legacy*) that uses GDriveRT documents as backend and synchronizes with the original document every 5 seconds. Supporting legacy clients (i.e., synchronizing with the original document) incurs a non-negligible overhead. This happens because the mechanism used requires a large number of accesses to the centralized infrastructure as to infer which operations should be carried from legacy clients to the Legion clients and vice versa. However, even with support for legacy clients enabled, Legion induces lower load on the centralized component when compared with GDriveRT.

Figure 3.1.4c reports the average peer-to-peer communication traffic for each client during the setup of WebRTC connections (*Setup*) and while clients issue and propagate operations (*Operations*). The results show that the traffic of each client is larger than the traffic of each client with the server in GDriveRT (which can be approximated by dividing the server load – in Figure 3.1.4b – by the number of clients). This happens because our dissemination strategy has inherent redundancy, whereas in GDriveRT there are no redundant transmissions between each client and the centralized infrastructure. However, an average under 14KBps does not represent a huge fraction of available bandwidth nowadays. Furthermore, the use of our location-aware overlay leads to a network-usage pattern where the amount of data sent to distant nodes is significantly lower than that sent to nearby nodes.

3.2 Antidote with Non-uniform Replication

Non-uniform replication is a mechanism to optimize the replication process in geo-replicated settings, by storing and propagating only a subset of the updates among the multiple replicas. Our integration of this mechanism in Antidote has been presented in deliverable D3.1, with its application in light-edge and heavy-edge solutions being discussed in D5.1 and D6.1. In this section, we present the evaluation of non-uniform replication, first by simulation and then with results from a real deployment in AntidoteDB.

3.2.1 Simulation

We start by comparing non-uniform replication CRDT designs against state-of-the-art CRDT alternatives: delta-based CRDTs [3] that maintain full-object replicas efficiently by propagating updates as deltas of the state; and computational CRDTs [9] that maintain non-uniform replicas using a state-based approach.

Our first evaluation is performed by simulation, using a discrete event simulator. To show the benefit in terms of bandwidth and storage, we measure the total size of messages sent between replicas for synchronization (total payload) and the average size of replicas.

To this end, we simulate a system with 5 replicas for each object. Both our designs and the computational CRDTs support up to 2 failing replica by propagating all operations to, at least, 2 other replicas other than the source replica. We note that this limits the improvement that our approach could achieve, as it is only possible to avoid sending an operation to two of the five replicas. By either increasing the number of replicas or reducing the fault-tolerance level, we expect that our approach would perform comparatively better than the delta-based CRDTs.

(a) Top-K with removals

The Top-K data type allows access to the top-K elements added to some sorted collection object and can be used, for example, for maintaining the leaderboard in online games.

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 add operations that have not been propagated to the source replica of the remove). This leads to an *add-wins* policy [11], 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.

We compare our Top-K design (*NuCRDT*) with a delta-based CRDT set [3] (*Delta CRDT*) and the top-K state-based computational CRDT design [9] (*CCRDT*).

The top-K was configured with K equal to 100. In each run, 500000 update operations were generated for 10000 Ids and with scores up to 250000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Given the expected usage of top-K for supporting a leaderboard, we expect the remove to be an infrequent operation (to be used only when a user is removed from the game). Figures 3.2.1 and 3.2.2 show the results for workloads with 5% and 0.05% of removes respectively (the other operations are adds).

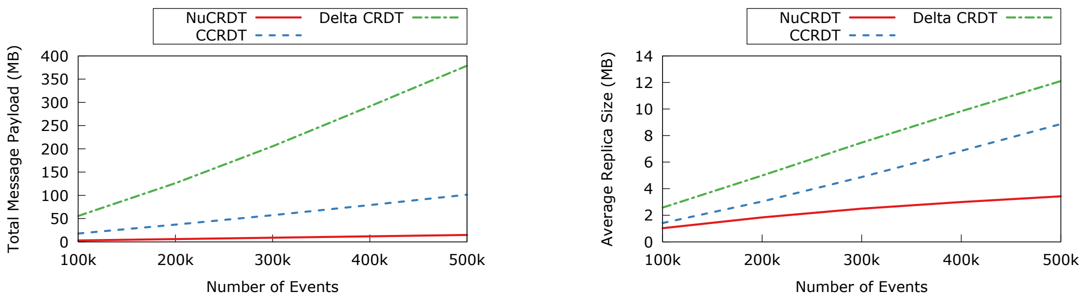


Figure 3.2.1: Top-K with removals: payload size and replica size, workload of 95/5

In both workloads, our design achieves a significantly lower bandwidth cost when compared to the alternatives. The reason for this is that our design only propagates operations that will be part of the top-K. In the delta-based CRDT, each replica propagates all new updates and not only those that are part of the top. In the computational CRDT design, every time the top is modified, the new top is propagated. Additionally, the proposed design of computational CRDTs always propagates removes.

The results for the replica size show that our design is also more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information from remote operations received for guaranteeing fault tolerance and those that have influenced the top-K at some

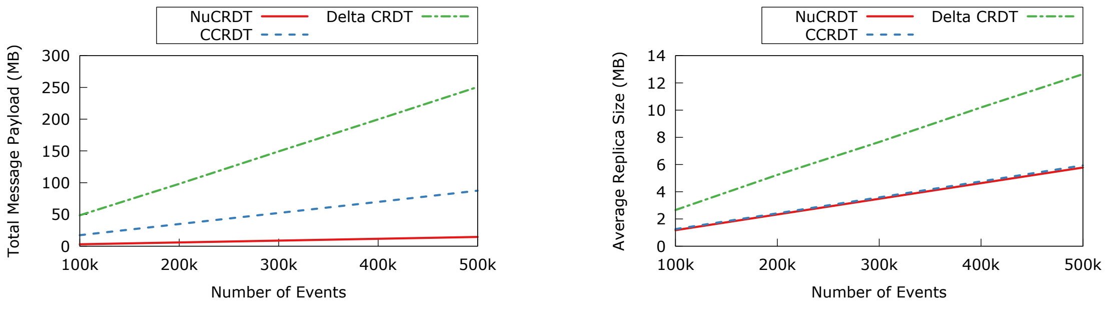


Figure 3.2.2: Top-K with removals: payload size and replica size, workload of 99.95/0.05

moment in the execution. The computational CRDT design additionally keeps information about all removes. The delta-based CRDT keeps information about all elements that have not been removed or overwritten by a larger value. We note that as the percentage of removes approaches zero, the replica sizes of our design and that of computational CRDT starts to converge to the same value. The reason for this is that the information maintained in both designs is similar and our more efficient handling of removes starts becoming irrelevant. The opposite is also true: as the number of removes increases, our design becomes even more space efficient when compared to the computational CRDT.

(b) Top Sum

To evaluate our Top Sum design (*NuCRDT*), we compare it against a delta-based CRDT map (*Delta CRDT*) and a state-based computational CRDT implementing the same semantics (*CCRDT*).

The top is configured to display a maximum of 100 entries. In each run, 500.000 update operations were generated for 10.000 Ids and with challenges awarding scores up to 1000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

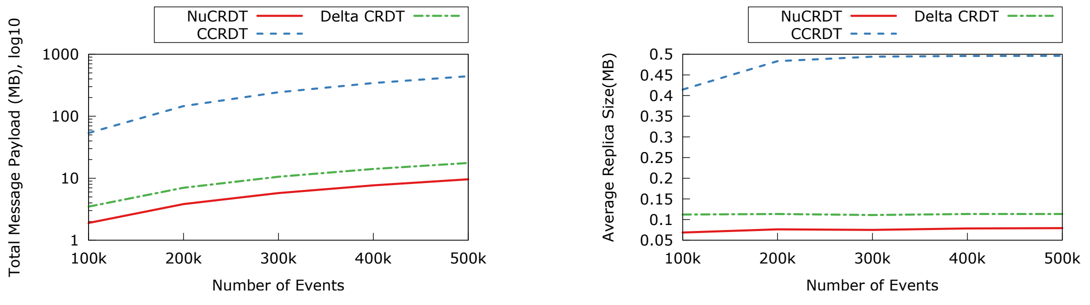


Figure 3.2.3: Top Sum: payload size and replica size

Figure 3.2.3 shows the results of our evaluation. Our design achieves a significantly lower bandwidth cost when compared with the computational CRDT, because in the computational CRDT design, every time the top is modified, the new top is propagated. When compared with the delta-based CRDTs, the bandwidth of NuCRDT is approximately 55% of the bandwidth used by delta-based CRDTs. As delta-based CRDTs also include a mechanism for compacting propagated updates, the improvement comes from the mechanisms for avoiding propagating operations that will not affect the top elements, resulting in less messages being sent.

The results for the replica size show that our design also manages to be more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information of remote operations received for guaranteeing fault-tolerance and those that have influenced the top elements at some moment in the execution.

3.2.2 AntidoteDB

In this section, we evaluate the performance of Non-uniform CRDTs in AntidoteDB. To this end, we compare the Top-K and Top-K with removals Non-uniform CRDTs with the current available solution that uses an add-wins set CRDT. The add-wins set is implemented by generating a new unique token for every insert operation. A remove operation will remove all tokens associated with an element known in the replica where the operation was executed.

The experiments we present in this chapter try to assess whether the introduction of non-uniform replication in a geo-replicated database system allows to: (i) reduce the size of database replicas; (ii) reduce the bandwidth used for synchronizing replicas. Furthermore, we study the scalability of a system that uses non-uniform replication in comparison with a system using full replication.

(a) Dissemination overhead and replica sizes

We started by measuring the size of the replicas and the bandwidth consumed for synchronizing replicas. To this end, we modified AntidoteDB to store in each data center the total size of messages transmitted for a given object. To measure the size of data type replicas we have introduced support for accessing the full object representation.

The experiment executes a sequence of randomly generated updates to different objects, where all different objects receive the same updates. The values used in each operation (regardless of the data type) were randomly selected using a uniform distribution. In the experiment we compare the Non-uniform CRDTs proposed in this work with the operation-based CRDTs currently available in AntidoteDB. Data points were recorded every 5,000 operations, by obtaining the total message size each data center had transmitted so far for each object and the size of the objects, which we later used to compute the mean size of each object. All results represent the mean result of three independent runs.

The experiments were ran on Amazon Web Services EC2, using *m3.xlarge* machine instances for both the AntidoteDB nodes and the node issuing the benchmark operations. Each of the machines were launched in the *eu-west-1c* region. A total of 5 AntidoteDB nodes were used, each one forming its own data center (containing only one node). Each AntidoteDB node was configured to buffer transactions for a period of 200 milliseconds.

(a).1 Top-K We first evaluated the performance of the Top-K design. In this case we compared our design against an add-wins set that models the same semantics on the client side, by explicitly removing elements from the set which become masked. This experiment used the following configuration: K was configured to 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000. The results are shown in Figure 3.2.4.

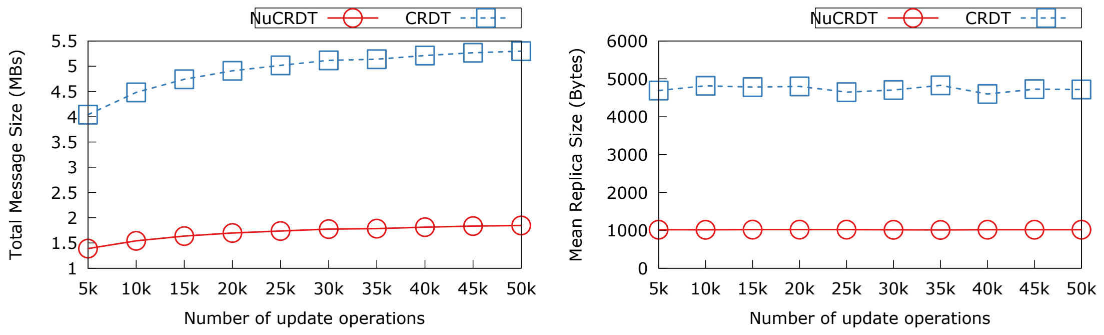


Figure 3.2.4: Top-K: total message size and mean replica size

For the total message size our data type achieved up to a 3 times lower dissemination cost. This is expected since to model the semantics of the top-K in the add-wins set, elements have to be explicitly removed once they are no longer part of the top.

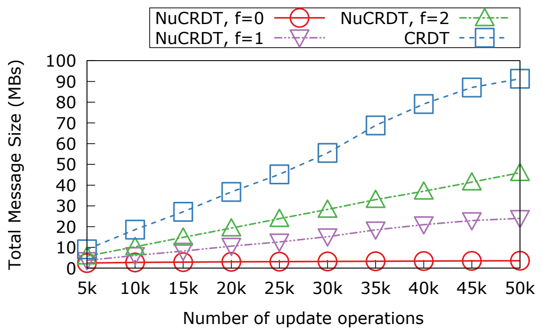
The results for the replica size show the efficiency of the state representation of our data type (up to a 4.8 times reduction). Even though both objects mostly have the same number of elements (roughly 100 for the add-wins set and always 100 for the Top-K) our design implementation manages to have a better state representation since it does not require unique tokens like the add-wins set.

(a).2 Top-K with removals We now compare the design of the Top-K with removals against an add-wins set which models the same semantics on the client side, by explicitly managing the removal of each affected element as would occur in the Non-uniform CRDT.

In this experiment, K was configured to be 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000. Furthermore, the system was configured to support from zero to two faults ($f = 0, f = 1, f = 2$) by propagating masked operations to f replicas.

As for the simulations in section 3.2.2, given the expected usage of a top-K for supporting a leaderboard, we expect the remove to be an infrequent operation (to be used only when a user is removed from the game). Thus, the workload was chosen with this in consideration. Figure 3.2.5 shows the results for a workload of 95% of adds and 5% of removes.

Our design achieved both a significant lower bandwidth cost (up to a 96% reduction for $f = 0$) and a lower replica size (up to a 67.7% reduction for $f = 0$) when compared to the add-wins set. This happens primarily because the add-wins set needs to propagate all elements to all replicas (even the ones that do not fit in the top) while our design only propagates the required elements to all replicas and the remaining elements are only propagated to a subset for durability.



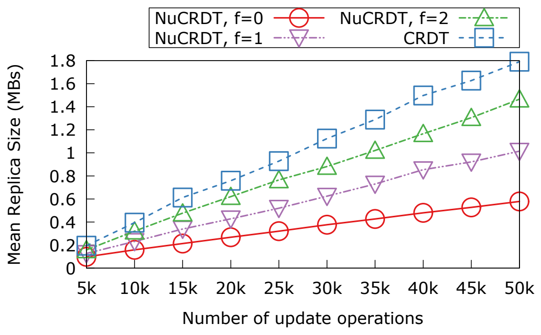


Figure 3.2.5: Top-K with removals: total message size and mean replica size with a workload of 95% adds and 5% removes

When specifically comparing between the various instances of the Top-K with removals with varying degrees of replication, the total message size and mean replica size increases as expected. We note that, when the data type tolerates up to 2 faults, its mean replica size reaches 78% of the mean size of the add-wins set, while each element that is not in the top is replicated only in 60% of the replicas. This happens due to a few reasons: 1) The Top-K with removals must explicitly maintain more information regarding each element including the replica id and the replica timestamp, and 2) the Top-K with removals must maintain an explicit registry of removals.

(b) Scalability

To evaluate the scalability of non-uniform replication, we have used the Basho Bench [4] benchmarking tool developed by Basho Technologies. To use Basho Bench, we developed a driver that specifies what operations can be executed for our use cases. For using AntidoteDB's original CRDT, the driver specification was extended to model the same semantics as NuCRDTs. To give an example, when modeling a top-K using an add-wins set, the elements which are no longer part of the top must be removed one by one – this is done explicitly by the client driver.

We now describe the benchmarking setup. These experiments were ran on Amazon Web Services EC2, using *m3.xlarge* machine instances for both the AntidoteDB nodes and the Basho Bench nodes. All benchmarks were run against 5 AntidoteDB nodes, each one forming its own data center (containing only one node), resulting in a total of 5 data centers. The benchmark runs using 5 Basho Bench instances, each one in its own machine. Each Basho Bench instance spawned a configurable number of clients and connected to the data center node running in the same EC2 region.

Machines that ran AntidoteDB nodes were launched on the following region/availability zones: *eu-west-1c*, *eu-central-1a*, *us-east-1d*, *us-west-1c*, and *ap-northeast-1c*. Machines that ran Basho Bench nodes were launched in the same region and availability zone as the AntidoteDB node they were connecting to. The mean round-trip time over 100 Ping requests between each machine is shown in table 3.2.1.

Each AntidoteDB node was configured to buffer transactions for a period of 200 milliseconds. All benchmarks ran for 3 minutes. Each data point for each experiment represents the mean result of three independent runs. Prior to each run, the AntidoteDB

	eu-west	eu-central	us-east	us-west	ap-northeast
eu-west	0.421	22.42	71.537	145.875	210.989
eu-central	22.42	0.417	89.241	156.304	254.216
us-east	71.537	89.241	0.459	61.699	143.058
us-west	145.875	156.304	61.699	0.453	117.695
ap-northeast	210.989	254.216	143.058	117.695	0.493

Table 3.2.1: Mean round-trip time between Amazon Web Services EC2 instances

nodes were shutdown and their data was deleted; the software was then recompiled, the nodes were relaunched, and the data center nodes were reconnected. This ensured a fair benchmarking environment.

(b).1 Top-K We now present the evaluation results for the top-K and the add-wins implementation of a top-K. This experiment used the following configuration: K was configured to 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000. The results are presented in Figure 3.2.6.

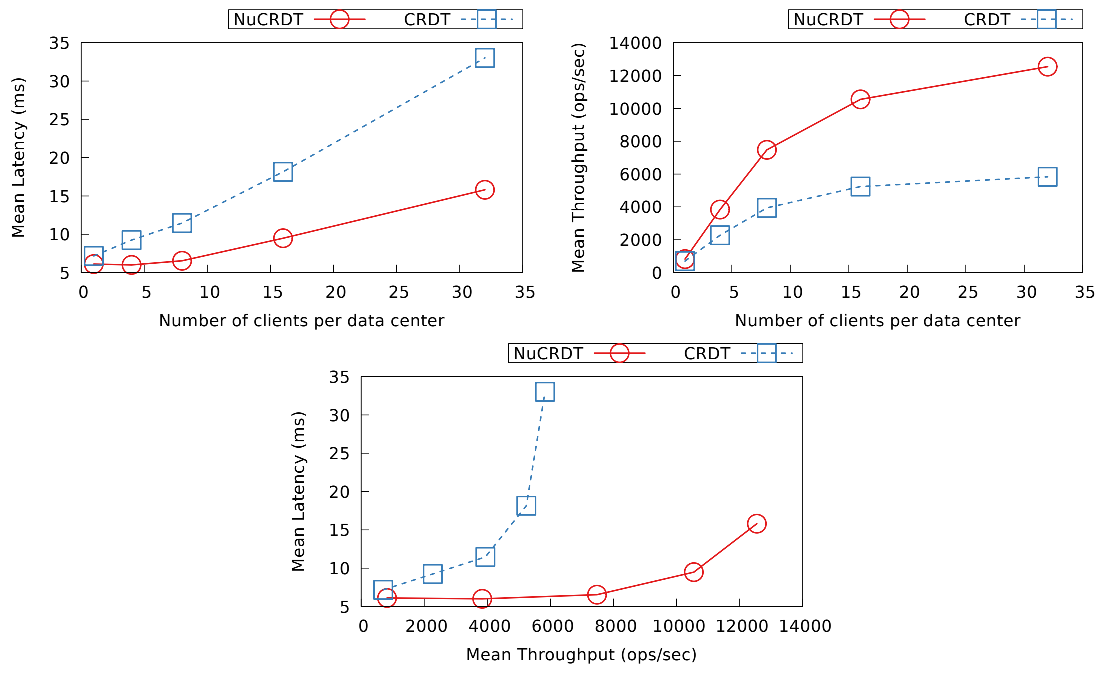


Figure 3.2.6: Top-K experiments

The results show that our non-uniform replication design scales much better than the add-wins set-based implementation of top-K. The reason for this is the fact that in the add-wins-based implementation, it is necessary to remove an element whenever a new element is added to the top, resulting in a larger number of operations being executed. Additionally, in our design, as the top is populated with elements with large scores, the number of operation that are not immediately masked tends to zero.

(b).2 Top-K with removals We now present the evaluation results for the top-K with removals. In this case, we need to maintain all inserted scores as a remove may delete only some of the scores. The configuration used in the experiments is the following: K was set to 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000. Furthermore, the system was configured to support from zero to two faults ($f = 0, f = 1, f = 2$) by propagating masked operations to f replicas. Similarly to the measurements of total message size and mean replica size, Figure 3.2.7 presents the results of a workload of 95% of adds and 5% of removes.

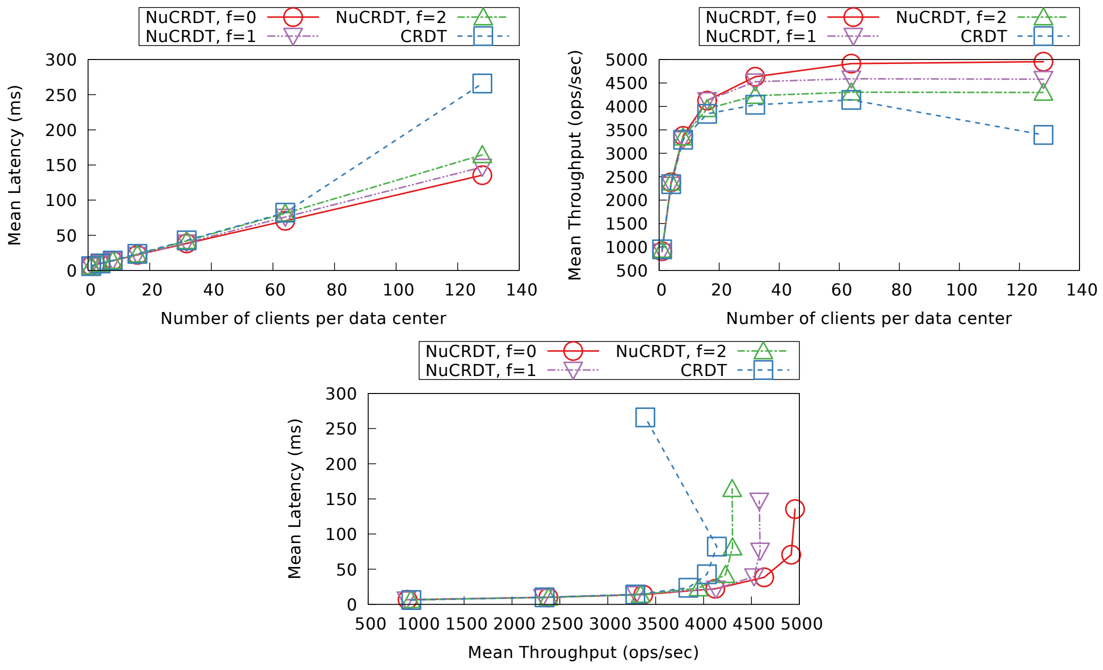


Figure 3.2.7: Top-K with removals experiments with a workload of 95% adds and 5% removes

The results show that both design behave similarly under low load (up to 16 clients). For a larger number of clients per data center, the mean latency of the add-wins set more than doubled while the mean latency of the NuCRDT design remained linear. For 128 clients per data center the add-wins set could not keep up with the increasing load and as result suffered a throughput drop. The NuCRDT design did not exhibit this behavior.

When comparing between the varying degrees of replication both latency and throughput were initially similar. However, after the number of clients per data center increased to more than 32, the latency also slightly increased for both $f = 1$ and $f = 2$. Correspondingly, the throughput also had a slower growth and could not reach the same value as $f = 0$.

We suspect that the marginal increase in scalability for this design is due to the top-K computation being executed inside the database, which could perhaps be optimized further.

3.3 Yggdrasil

Yggdrasil is a framework designed to aid in the development and execution of distributed protocols and applications for wireless ad-hoc networks in commodity devices, as the Raspberry Pi3 - model B. To this end, the Yggdrasil framework provides protocol and application developers with a set of abstractions that include: an event driven execution model; low level communication primitives; interaction mechanisms between protocols and applications; and mechanisms for multi-threaded execution that hide concurrency issues for the developers. Yggdrasil is reported in detail in D5.1, whose evolution is detailed in D5.2.

In this Section we report our preliminary experimental evaluation over Yggdrasil. Our experimental evaluation is divided in two parts. In the first part we discuss the implementation of a popular routing protocol for wireless ad-hoc networks and a moderately complex application using Yggdrasil. The goal of this first part is to answer the question: *How useful and easy is to leverage Yggdrasil to implement distributed protocols and applications?* The second part of our evaluation is focussed on answering the question: *What is the overhead generated by employing Yggdrasil?* To this end, we have conducted an experimental evaluation that measures the of the amount of time required to deliver a message to kernel to be sent to the network, with and without employing Yggdrasil. We also present the observed values of CPU and memory consumptions of this experiment.

3.3.1 Protocol and Application Implementation

To answer the first question, we have conducted two qualitative experiments where we measured the amount of effort in implementing a protocol and an application in the number of lines of code. In the following we present our methodology and results.

(a) Experimental Methodology

(a).1 Routing Protocol In the first experiment we have implemented a simplified version of a popular routing protocol for wireless ad-hoc networks: *Better Approach to Ad Hoc Networking*, or simply B.A.T.M.A.N. using Yggdrasil. The reason for this is two fold. First, Yggdrasil was missing a routing protocol. Second, we took this opportunity to compare our implementation of B.A.T.M.A.N. with an existing and well documented implementation of the same protocol that operates as a kernel module. This implementation is an adequate term of comparison since it is also written in the C language (we remind the reader that Yggdrasil is written in C and that currently protocols and applications also have to be written in C). The baseline implementation of B.A.T.M.A.N. can be found online at https://www.open-mesh.org/projects/open-mesh/wiki. We note that we did not study the baseline implementation of the protocol before implementing our own. Instead our implementation in Yggdrasil was conducted by following the specification of the protocol in the RFC [10].

(a).2 Test Application In the second experiment we developed a simple application that leverages our implementation of B.A.T.M.A.N. and other protocols already implemented in Yggdrasil. This application combines the functionalities of a discovery, broadcast, routing, and aggregation protocols, to periodically compute, in a particular node, the average number of messages sent by each device in a deployment.

The discovery protocol employed in the implementation of the test application is the exact same employed in the evaluation of MiRAge (described further ahead in this document), also enriched with fault detection. The broadcast protocol is a simple protocol based on flooding of the network, where every time a node receives a message for the first time, it immediately retransmits it. The routing protocol, as stated above, is our own implementation of the popular wireless ad-hoc routing protocol B.A.T.M.A.N.. Finally, the aggregation protocol that we employ to aggregate the average number of transmitted messages is the implementation of GAP used in MiRAge's evaluation.

In this deployment, each node periodically, and with a given probability, broadcasts (using the broadcast protocol) a message with random information. The number of messages broadcast by each node is aggregated in a single node (using the aggregation protocol), and routed to a backup node (using the routing protocol), as the application executes. This test application, although synthetic, presents a moderate level of complexity that other more realistic applications could also employ.

(b) Experimental Results

(b).1 Routing Protocol Our implementation of B.A.T.M.A.N., after conducting debug and testing, had less than 800 lines of C code. We note that our implementation has a number of simplifications when considering the original specification however, these have no significant impact on the execution or correctness of the algorithm. In particular, our implementation of the protocol's sliding windows is represented as an array of shorts (rather than a bit mask), and the messages exchanged between processes are slightly larger than the original specified messages (however, they are still smaller than the maximum size of a network frame, 1500 bytes).

When comparing the number of lines of code of our implementation with the existing baseline implementation, we discovered that, ignoring code specific to the interactions with the kernel and interfaces, the baseline implementation of the version 4 of the protocol has more than 2.000 lines of C code. We then inspected the code to understand the reason for this large number of lines. Some lines are indeed dedicated to optimizations that we did not implement in our implementation. However, most of these lines were dealing with low level aspects, such as message serialization, timer management, concurrency management, network interface management, among others. In Yggdrasil most of these aspects are handled by the framework through a simple API. This not only justifies the lower number of lines in our implementation, but also points towards the adequacy of the abstractions provided by Yggdrasil.

(b).2 Test Application Our implementation of this application, using all protocols as described above had, after debugging and initial testing, less than 180 lines of C code. Furthermore, the code was written and debugged in approximately two hours[1]. We believe that this is a positive indicator, since the envisioned test application already presents a moderate level of complexity, and it was possible to code with low effort.

[1]We do note that the person coding the application was one of the framework developers, and hence was highly familiar with the code base.

3.3.2 Yggdrasil Overhead

To measure the overhead generated by employing Yggdrasil, we have conducted a simple preliminary experiment where we measured the amount of time required for a message to be delivered to the kernel to be sent to the network. In the following we detail our experimental methodology and obtained results for this experiment.

(a) Experimental Methodology

In this experiment we have developed a simple application that sends a message to the network every second. This application is executed for more than $10,000$ seconds (slightly bellow 3 hours), sending a total of $10,000$ messages. This application is implemented and executed in four different settings:

A: The application is not implemented using Yggdrasil, as such it sends messages by directly interacting with the kernel.

B: The application is implemented using Yggdrasil. The application delegates the functionality of sending the message to the dispatcher protocol of Yggdrasil.

C: This setting is similar to the previous one, with the addition that the application executes concurrently with a protocol.

D: This setting is similar to the previous one, with the exception that before the message is delivered to the dispatcher protocol it passes through the protocol executing concurrently, we call this mechanism *intercepting events* (in this case a message).

The protocol that executes with the application in our experiments is named a *ghost protocol*. This is because the protocol does not perform any operation on its own. Hence, the protocol is configured to remain idle until the application terminates in setting **C**, to verify the effects of a protocol executing concurrently; and in setting **D**, the protocol is configured to intercept the messages created by the application, to verify the impact of an event passing through multiple protocols.

We executed these experiments using a Raspberry Pi3 - model B and a GRiSP board. In each setting we logged to a file a timestamp immediately before the creation of the message, and a timestamp immediately after the message was delivered to the kernel to be sent to the network. We correlated these data points for each message offline. Furthermore, during each experiment, the CPU and memory consumptions where gathered using standard profiling tools (e.g., perf, top, built-in tools in RTEMS). In the following we present the obtained results.

(b) Experimental Results

Table 3.3.1 reports the 95^{th} percentile delay of sending a message in milliseconds for each of our experiments detailed above. The results shows that employing Yggdrasil in a Raspberry Pi incurs in an increased delay of around 0.07 milliseconds, whereas in a GRiSP board, the increased delay is around 1.15 milliseconds. This is to be excepted as the GRiSP board has less resources than the Raspberry Pi nonetheless, both present an increase of 60% of delay. The overhead increase is to be expected, since more mechanisms are provided to the application. We believe that the 60% overhead could be decreased with simple optimizations to the implementation of the framework, such as reducing the amount of systems calls necessary to manage the events within the framework.

	A: W/out Yggdrasil	**B:** W/ Yggdrasil	**C:** W/ Ghost idle	**D:** W/ Ghost intercept
Raspberry Pi	0.110729	0.182161	0.186275	0.211146
GRiSP	1.983731	3.137988	3.155992	3.418891

Table 3.3.1: 95^{th} percentile delay of sending a message in milliseconds

Executing another protocol concurrently results in a relatively small overhead that is not significant in both platform. When the protocol intercepts the message it results in an overhead of 10%. This overhead is justifiable by the fact that the intercept behavior requires two additional memory copies for the message to be delivered to the kernel to be sent to the network.

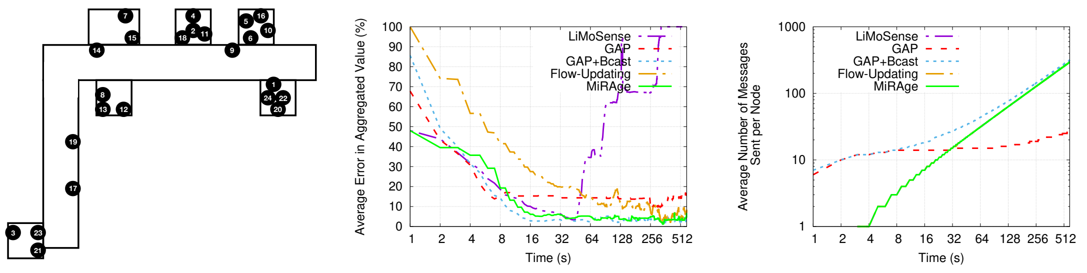
Regarding the CPU and memory consumptions we have observed few variations of values. The CPU consumption in the Raspberry Pi, which has a CPU with four cores with a clock rate of $1.2Ghz$, was observed to be 0.01%, while in the GRiSP board, which has a microchip CPU with a clock rate of $300Mhz$, it was observed to be approximately 1%. This low CPU usage is caused by the process being idle most of the time, having small spikes of usage. This however, is to be expected has most of the components in Yggdrasil are idly waiting for events.

Regarding memory consumption, we observed that the processes used approximately 700 KiB (1 KiB = 1024 bytes) of resident memory in both devices. This, in the GRiSP board, does not take into consideration the memory footprint of RTEMS. The memory footprint of Yggdrasil is relatively low (bellow 1 MB) nonetheless, we believe that we can improve this by applying simple optimizations to the event queues used by protocols (e.g., not initializing parts of the event queue that are not relevant for the protocol), as they are the data structures with the highest footprint of memory (higher than $15,000$ bytes).

Overall, our first assessment shows that Yggdrasil presents an acceptable overhead for the functionalities it provides to protocols and applications to lower the effort of implementation for developers, with a margin of opportunity to improve its performance.

3.4 Mirage

For supporting aggregation in light-edge nodes, we have design the $\underline{M}$ult$\underline{i}$ $\underline{R}$oot $\underline{Ag}$gr$\underline{e}$gation protocol, or simply MiRAge. Our protocol is inspired in the design of GAP [5], but generalizes its design to remove the dependence of a single root. To this end, our protocol leverages a self-healing spanning tree to *support* efficient continuous aggregation. In our protocol, all nodes compete to build a tree rooted on themselves. This competition is controlled via the identifier of each node (a large random bit string) and a monotonic sequence number (i.e., a timestamp) controlled by the corresponding root node. Additionally, our protocol was designed to ensure that all nodes in the system are able to



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

Figure 3.4.1: Disperse Deployment

continually compute and update the result of the aggregation function.

MiRAge is presented in deliverable D5.2. In this section, we present an extensive experimental evaluation of MiRAge comparing its performance against state-of-the art solutions for continuous aggregation.

We start by noting that, contrary to the large body of the literature regarding wireless ad-hoc protocols, our experimental evaluation does not resort to simulation. Instead, we have implemented prototypes of both MiRAge and the relevant competing baselines using the C language. All protocols rely on similar code bases, being implemented on the Yggdrasil framework reported in D5.1 and D5.2. All also use the same fault detector mechanism, and execute in similar conditions. We believe that this is an essential step to demonstrate the applicability of our solution.

In the following, we discuss in more detail our experimental methodology and present our experimental results composed of two different deployments that exercise these protocols in varied conditions. These include fault-free, input value changes, and multiple-node failures scenarios. All experiments reported here where nodes were scattered across a floor of a building were performed with the help of the current prototype of the Yggdrasil Control Process, introduced in D5.2.

3.4.1 Experimental Methodology

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

The baselines employed in our experimental work are **Flow-Updating** [7] which is a protocol that can compute the average function; a version of **LiMoSense** and [6] published by the authors, where counters maintained by nodes are never garbage collected. LiMoSense is a representative of the well known Push-Sum protocol [8] that can compute the sum, count, and average functions, that is enriched to ensure fault tolerance; **GAP** [5] which is the protocol that mostly resembles our own solution, being able to compute any aggregation function but unfortunately, unable to tolerate the failure of its static tree root. Since GAP does not enable every node in the network to obtain the aggregated value, we also developed a simple variant of GAP, that we named **GAP+Bcast** where the root of the tree also piggybacks its aggregated value in all messages disseminated by it. This value is then propagated in piggyback along the tree used by GAP, enabling all nodes to learn the result of the aggregation. All protocols were configured to perform their periodic communication step every two seconds. In both GAP and GAP+Bcast experiments the root is fixed in experiments with faults, and randomly assigned in experiments with no faults.

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

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

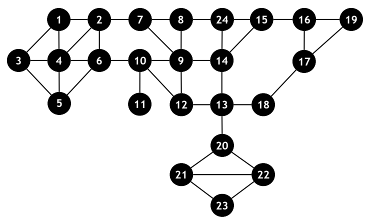
We have conducted experiments in two different settings:

Disperse Deployment: In this deployment, we have positioned the nodes across multiple rooms in two hallways in our department building. Figure 3.4.1a illustrates the distribution of nodes in the space. Each of the hallways has approximately $30m$. This is a particularly challenging environment, as there are multiple factors that affect transmissions among nodes in a very dynamic fashion. This deployment attempts to illustrate a scenario where the radio signal strength is highly variable among devices, which we have observed that could trigger our fault detector multiple times.

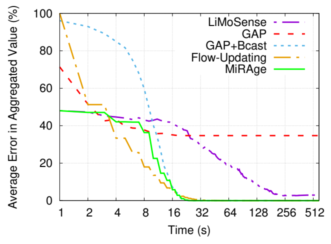
Dense Deployment with Overlay: In this deployment, we have positioned all nodes within a single room. However, we have added to all test prototypes a filter that restricts at each node the set of nodes from which it can receive messages. Effectively, this produces a logical network that defines (potential) neighboring relationships among nodes. This overlay is represented graphically in Figure 3.4.2a. We note that in this setting transmissions by any device can still produce collisions. This setting tries to illustrate a scenario where there are multiple sources of interference that can produce a somewhat higher number of collisions in the wireless medium.

3.4.2 Experimental Results

We now report our experimental results. In our experimental work we focus on the *Average Error in the Aggregated Value*, abbreviated *AvgErr*, that illustrates how far on average

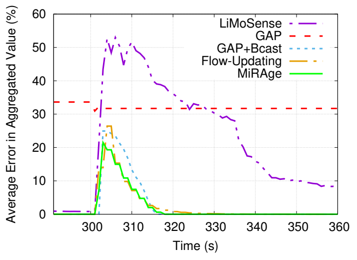


(a) Overlay Topology

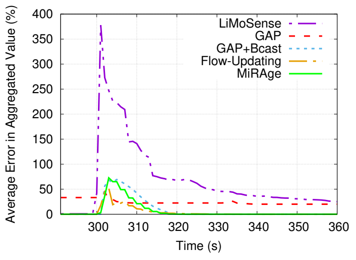


(b) Average Error in the Aggregated Value in Fault-free Scenario

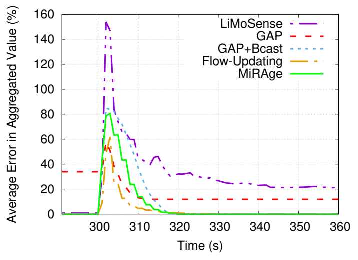
Figure 3.4.2: Deployment Configuration with an Overlay Topology



(a) 1 node



(b) 12 nodes



(c) 24 nodes

Figure 3.4.3: Average error in aggregated value with dynamic input values at different number of nodes

are all nodes from the correct average value, being defined as:

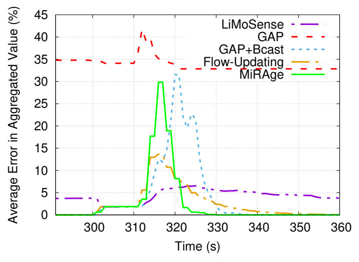
$$AvgErr = \frac{\sum_{i=1}^{n}(|Avg_{real} - Avg_i|)}{n \times Avg_{real}} \times 100$$

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

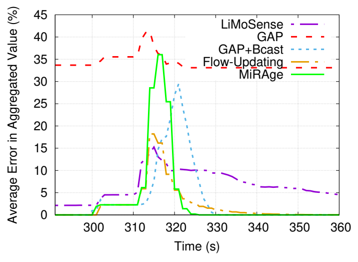
(a) Disperse Deployment

Figure 3.4.1 presents the schematics and results for our experiments in the disperse deployment. In this experiment we have deployed the nodes as represented in Figure 3.4.1a. Each experiment was conducted for a period of 10 minutes (600 seconds).

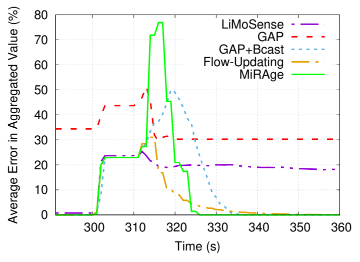
Figure 3.4.1b reports the measured $AvgErr$ across all nodes as the experience progresses. Note that the x-axis is in logarithmic scale, as the main point of this plot is to



(a) 1 node



(b) 6 nodes



(c) 12 nodes

Figure 3.4.4: Average error in aggregated value with variable number of node failures

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

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

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

Figure 3.4.1c shows the total number of messages sent over time for each protocol.

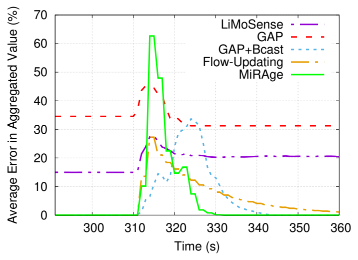


Figure 3.4.5: Average error in aggregated value with 12 link fault

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

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

(b) Dense Deployment with Overlay

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

(b).1 Fault-Free Scenario Figure 3.4.2a reports the overlay configuration employed in this scenario. Figure 3.4.2b presents the *AvgErr* in a fault free execution for all protocols.

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

(b).2 Dynamic Input Values In these experiments we have introduced variations on the input value of different amounts of nodes in the system. We have conducted experiments where we modify the input value of 1, 12, and 24 nodes concurrently. Results are summarized in Figure 3.4.3 and show consistent results for all experiments. LiMoSense is the protocol that is more susceptible to input value variations, whereas MiRAge, Flow-Updating, and GAP+Bcast all present somewhat similar results, being able to converge to the new aggregate result in less than 20 seconds. The reason why only these three protocols are able to cope in a timely fashion with the change of input values is nuanced. These protocols are the only whose computation of the aggregate result directly depends on the (original) input value. This implies that as soon as the input value changes, nodes start propagating aggregate information that reflects completely the change. Therefore, it suffices that messages propagate through the system to ensure that this effect reaches all nodes.

(b).3 Node and Link Failures In these experiments we introduce a variable number of node faults and measure the impact on the *AvgErr*. In the first experiment we introduced concurrently a number of node crashes that vary from 1, 6, and 12 nodes around 300 seconds in the experience. In these we have fixed the root of the trees used by GAP and GAP+Bcast to be node 1, and made sure that this node was not selected to become faulty (as these protocol would not tolerate that fault). Figure 3.4.4 depicts the results for these experiments. In the second experiment we introduce 12 concurrent link failures, where 12 pairs of nodes become permanently unable to communicate. This simulates the existence of obstacles or radio pollution in the environment. Figure 3.4.5 reports the obtained *AvgErr* in this scenario.

In both faulty scenarios (reported in Figures 3.4.4 and 3.4.5), all protocols suffer a significant increase in the *AvgErr* after the introduction of faults. With the exception of LiMoSense, all protocols can converge to the new aggregated value. LiMoSense takes significantly more time to converge because, following the Push-Sum strategy, at each communication step it only exchanges information with a single neighbor. While MiRAge is the protocol that consistently shows a higher *AvgErr* immediately after faults, it is also the protocol that converges faster. This happens due to our mechanism to manage and repair the support tree in the presence of faults, which reconfigured the tree in an expedite way, during this period however, computed results are affected by the (temporary) inconsistency of the tree topology.

Chapter 4

An Overview of Formal Evaluations

The purpose of this chapter is to discuss how formal method will be used in the evaluation phases of the Lightkone project. Formal methods are mathematical and logic models of the systems being designed and studied. The expectation is that these methods provide useful insight into the behavior of complex systems. The results can contribute to a higher level of confidence in the chosen approach or in defining the likely pitfalls. Ideally, formal methods, well applied, should results in more reliable and rapidly developed systems. One of the key areas of research in the Lightkone project is the implementation and application of formal methods for the design and evaluation of distributed systems and notably edge systems. The ongoing work is described in much greater detail in WP 2.2, but here their utility in the evaluation phases of the project will be briefly discussed.

4.1 Tools for Formal Models

A number of different tools for formal modeling have been developed for many years with the earliest temporal logic models dating from the 1950s. In spite of their long history and the existence of credible tools, the definition of models remains a something of a daunting task. The writing of appropriate specifications remains a fairly academic pursuit. One of the goals of Lightkone is to bring these methods closer to the selected real-world use cases.

4.1.1 TLA+ and TLC

The TLA+ formal specification language allows the key invariants of a particular use case to be expressed and then used with the TLC model checker which permits the state space of all possible outcomes to be studied in order to determine under what conditions the invariants are violated. This approach has been applied to the Guifi network where the intention is to greatly improve the reliability of the system monitoring. One key invariant is to assure that all components of the system are being monitored by at least one agent, even in the event of failures. Testing of all possible failure scenarios is impossible so this approach is expected to provide meaningful feedback to guide the physical evaluation phases discussed in this document. The initial TLA+ models are somewhat simplistic and as their work progresses, they are expected to progress to greater levels of detail.

4.1.2 Abstract Execution Formulations

As are the other approaches, the abstract execution approach is described in greater detail in WP 2.2. Generally speaking the approach consists of defining a logical chain of executions that make up transactions. The distinction is made between events that are visible to all and those that are not and must be arbitrated using more sophisticated consensus or resolution methods. This approach has been applied to the Scality use case and has already provided good insights into what will be possible and realistic and areas that need to be evaluated with greater care in the upcoming physical evaluation phase. The abstract execution model allow tests to be derived that could potentially cause consistency issues and assist in the verification of the correctness of the models during the evaluation. The results have already guided the selection of a replication model using Antidote across multiple data centers to provide relatively strong consistency models that can be tested during the evaluations.

4.1.3 Timed Automata Formalization

The timed automata approach is one of the most well known, based on a finite set of clocks and temporal logic. In some ways this approach appears straightforward as the before and after of timed events is part of daily life, but concurrent timing issues can be exceedingly complex. These methods seem well adapted to certain use cases and notably the Stritzinger use case where time optimization is a key priority. As discussed in WP 2.2, the UPPAAL tool has been applied to the problem. This modeling has already given results by clarifying the likely limits of gains that can be expected in the physical system. These results will prove useful in the practical phase of the evaluations in determining how close to optimum outcomes the real system is able to perform. For the Gluk use case, the need is defined by the requirement of having timely results, and specifically in determining important stage changes in a timely fashion. A somewhat different temporal method has been applied here in referred to as Discrete Event Simulation (DES). Early results of the use of this modeling approach have provided insights into the appropriate redundancies in the sensor and servo networks to guarantee robust outcomes. The results of this analysis will also be used in better defining the failure scenarios used in the evaluation phase of the project. The evaluations for this particular use case will be less based on performance and more based on robustness using low power and potentially unreliable components.

Chapter 5

Evaluations and Security

Digital security is under ever-increasing scrutiny and plays an essential role for the technology and systems developed in Lightkone. Even though the main focus of this evaluation work-package is the correctness, performance and efficiency of the developed solutions, we can consider performing a selection of suitable security tests or, alternatively, an in-depth review of potential security risks.

The security testing strategies could include:

- traditional penetration tests,
- code and security reviews, or
- use of industry standard verification tools such as wireshark, w3af, nmap.

Deliverable D3.2 contains a document with a case-by-case threat identification for each of the industrial use cases. This threat analysis can provide guidance for developing specific security evaluation tests. At the very least, during the evaluations of the physical pilot systems, a review of these threats will be conducted and reported. A complete review of system security in the event that one or several of these systems become commercially viable would be appropriate, but it is generally far more credible to have a security audit performed by an independent 3rd party. Such an effort was not budgeted for this project and likely would only be appropriately carried by the partner likely to benefit commercially from the developments.

Within the context of the project a review of the threat models described in D3.2 will be performed for each of the solutions providing recommendations in response to the most likely threat vectors.

In the case of the Gluk and Stritzinger use cases, security is a concern as in all environments, but in manufacturing and agricultural settings security has historically been handled via physical access restrictions rather than deep security of information systems. Securing ad-hoc and self organising networks is a useful topic, but in the scope of the evaluations performed here, no additional security evaluation is planned.

In the context of Scality's use case, where the AWS S3 protocol has been implemented, extensive security recommendations have been provided in documents such as this: AWS Security Whitepaper. Their recommendations point out a notable challenge via the very sophisticated rights manangement system referred to as IAM. This technology offers very complete and granular control of all resources, but in so doing creates a great deal of complexity and many opportunities for configuration errors. This complexity is often pointed out as a key risk vector in using their system as configuration errors and misunderstandings are common. In light of these considerations, a minimimal and simple configuration will be used during testing as they provide a baseline for typical user behaviours. No specific experiments are planned to further validate the security of this protocol.

In the case of the Guifi no specific security investigations are planned during the evaluation phase. While security is very important in this kind of environment, the scope of the investigations planned will consume all resources available within the context of the project.

Chapter 6

Summary

As has been discussed in this document, there are a number of evaluations that will be performed in the second half of the project. The other work streams are coming together to allow the deployment of the different pilot implementations, that will be evaluated. The majority of the work on these evaluations remains to be done, but this is as expected. A significant amount of progress has been made on evaluating the theoretical methods as was discussed in chapter 3. These evaluations do indicate that progress has been made both in the efficacy of the methods as well as understanding their limitations. These tools will be used extensively in the implementation of the evaluation platforms across the different use cases. As was described in this document, each of the four different use cases has a plan set forth to provide concrete results before the end of the project. With the official adoption of the GDPR across the EU, questions of security have grown in importance since the initiation of this project. An evaluation of the types of risks to be considered was included here, and ongoing efforts are being made to include more analysis of questions of security in this evaluation.

Bibliography

[1] Google Drive Realtime Playground. github.com/googledrive/realtime-playground.

[2] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkilä, X. Wang, K. Hamily, and S. Bugoloni. Affordable and energy-efficient cloud computing clusters: The bolzano raspberry pi cloud cluster experiment. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 170–175, Dec 2013.

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

[4] Basho Technologies, Inc. Basho Bench.

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

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

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

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

[9] 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.

[10] Axel Neumann, Corinna Aichele, Marek Lindner, and Simon Wunderlich. Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.). Internet-Draft draft-openmesh-b-a-t-m-a-n-00, Internet Engineering Task Force, March 2008. Work in Progress.

[11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proc. 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, 2011.

Appendix A

List of acronyms

AWS Amazon Web Services
CN Community Network
DIY do-it-yourself
GCP Google Cloud Platform
S3 Simple Storage Service
SBC Single-Board-Computer