Ref. Ares(2017)4776403 - 30/09/2017

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

D2.1: Informal Requirements

Deliverable no.: D2.1
Title: Informal Requirements
Due date of deliverable: September 30, 2017
Actual submission date: September 30, 2017

Lead contributor: Stritzinger
Revision: 1.0
Dissemination level: PU

Start date of project: January 1, 2017
Duration: 36 months

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

Revision Information:

Date	Ver	Change	Responsible
01/06/2017	0.1	1st draft with outline and ToC	Stritzinger
30/09/3017	1.0	Final version	Stritzinger

Contributors:

Contributor	Institution
Peter Van Roy	UCL
Igor Zavalyshyn	UCL
Bradley King	Scality
Dimitrios Vasilas	Scality
Giorgos Kostopoulos	GLUK
Kostis Kounadis	GLUK
Roger Pueyo	UPC, Fundació Guifi.net
Felix Freitag	UPC
João Neto	UPC
Leandro Navarro	UPC
Roc Messeguer	UPC
Peer Stritzinger	STRITZINGER
Adam Lindberg	STRITZINGER
Stefan Timm	STRITZINGER
Deepthi Akkoorath	KL
Annette Bieniusa	KL
Ali Shoker	INESC TEC
João Leitão	NOVA ID
Nuno Preguiça	NOVA ID
Bernardo Ferreira	NOVA ID

Contents

1 Executive Summary **1**

2 Introduction **3**

1 Motivation for the use cases 3
2 The importance of crosscutting topics 4
3 Requirements elicitation and the questionnaire 5

3 UPC **7**

1 Coordination between servers for the Guifi.net monitoring system 7
1.1 Common background for the Guifi.net use cases 7
(a) Guifi community network environment 7
(b) Edge computing in Guifi.net 7
(c) Current monitoring system for Guifi.net nodes 8
1.2 Overview of the use case 9
1.3 Current development 10
(a) Conflicting operations 10
(b) Invariants that exist in the application state 10
(c) Performance results/figures 11
(d) Persistence 11
(e) Security threats 11
(f) Current deployment details 11
1.4 Detailed description 11
(a) Architecture 11
(b) Edge computing requirement 13
1.5 Data model 13
1.6 Detailed description of the computations 14
1.7 Conflicting operations and invariants 15
1.8 Divergence and divergence control 15
1.9 Network partitions 15
1.10 Operational requirements 15
1.11 Security requirements 16
1.12 Data protection requirements 16
1.13 Implementation 16
2 Data storage service for the Guifi.net monitoring system 16
2.1 Overview of the use case 16
2.2 Current development 17
(a) Conflicting operations 17

(b) Invariants that exist in the application state 17
(c) Performance results/figures 17
(d) Persistence 17
(e) Security threats 17
(f) Current deployment details 17
2.3 Detailed description 17
(a) Architecture 18
(b) Edge computing requirement 18
2.4 Data model 19
2.5 Detailed description of the computations 19
2.6 Conflicting operations and invariants 19
2.7 Divergence and divergence control 19
2.8 Network partitions 19
2.9 Operational requirements 19
2.10 Security requirements 20
2.11 Data protection requirements 20
2.12 Implementation 20
3 Service provision support for the Cloudy platform 20
3.1 Overview of the use case 20
3.2 Current development 21
(a) Conflicting operations 21
(b) Invariants that exist in the application state 21
(c) Performance results/figures 22
(d) Persistence 22
(e) Security threats 22
(f) Current deployment details 22
3.3 Detailed description 22
(a) Architecture 23
(b) Edge computing requirement 23
3.4 Data model 24
3.5 Detailed description of the computations 24
3.6 Conflicting operations and invariants 24
3.7 Divergence and divergence control 24
3.8 Network partitions 25
3.9 Operational requirements 25
3.10 Security requirements 25
3.11 Data protection requirements 26
3.12 Implementation 26

4 Scality **27**
1 Pre-indexing at the edge 27
1.1 Overview of the use case 27
1.2 Current development 28
1.3 Detailed description 29
(a) Architecture 29
(b) Edge computing requirement 29
1.4 Data model 30

1.5 Detailed description of the computations . . . 30
1.6 Conflicting operations and invariants . . . 31
1.7 Divergence and divergence control . . . 31
1.8 Network partitions . . . 32
1.9 Operational requirements . . . 32
1.10 Security requirements . . . 32
1.11 Data protection requirements . . . 32
1.12 Implementation . . . 33
2 Lambda functions at the edge . . . 33
2.1 Overview of the use case . . . 33
2.2 Current development . . . 34
(a) Conflicting operations . . . 34
(b) Invariants that exist in the application state . . . 34
(c) Performance results/figures . . . 34
(d) Persistence . . . 35
(e) Security threats . . . 35
2.3 Detailed description . . . 35
(a) Architecture . . . 35
(b) Edge computing requirement . . . 36
2.4 Data model . . . 36
2.5 Detailed description of the computations . . . 37
2.6 Conflicting operations and invariants . . . 37
2.7 Divergence and divergence control . . . 37
2.8 Network partitions . . . 38
2.9 Operational requirements . . . 38
2.10 Security requirements . . . 38
2.11 Data Protection requirements . . . 38
2.12 Implementation . . . 38
3 S3 local cache of central data . . . 39
3.1 Overview of the use case . . . 39
3.2 Current development . . . 39
(a) Conflicting operations . . . 39
(b) Invariants that exist in the application state . . . 39
(c) Performance results/figures . . . 40
(d) Persistence . . . 40
(e) Security threats . . . 40
(f) Current deployment details . . . 40
3.3 Detailed description . . . 41
(a) Architecture . . . 41
(b) Edge computing requirement . . . 42
3.4 Data model . . . 42
3.5 Detailed description of the computations . . . 43
3.6 Conflicting operations and invariants . . . 43
3.7 Divergence and divergence control . . . 43
3.8 Network partitions . . . 44
3.9 Operational requirements . . . 44
3.10 Security requirements . . . 44

3.11 Data protection requirements 44
3.12 Implementation 45

5 Stritzinger 47
1 No-Stop RFID 47
1.1 Overview of the use case 47
(a) Conflicting operations 49
(b) Invariants that exist in the application state 49
(c) Performance results/figures 49
(d) Persistence 50
(e) Security threats 50
(f) Current deployment details 50
1.2 Detailed description 50
(a) Conflicting operations 51
(b) Invariants and other rules that govern the system 52
(c) Expected performance 54
(d) Persistence 54
(e) Architecture 54
(f) Edge computing requirement 55
1.3 Data model 56
1.4 Detailed description of the computations 57
1.5 Conflicting operations 58
1.6 Divergence and divergence control 59
1.7 Network partitions 60
1.8 Operational requirements 60
1.9 Data protection requirements 60
1.10 Implementation 61
2 Smart metering gateways 61
2.1 Overview of the use case 61
2.2 Current development 62
2.3 Detailed description 62
(a) Architecture 62
(b) Edge computing requirement 63
2.4 Data model 63
2.5 Detailed description of the computations 64
2.6 Conflicting operations and invariants 64
2.7 Divergence and divergence control 65
2.8 Network partitions 65
2.9 Operational requirements 65
2.10 Security requirements 66
(a) Meters to gateways 66
(b) Gateways to cloud 66
(c) Gateway to gateway 66
2.11 Data protection requirements 66
2.12 Implementation 67
2.13 Extension: Swarm of small satellites 67
(a) Current development 69

(b) Detailed description . . . 69
(c) Data model . . . 69
(d) Conflicting operations and invariants . . . 69
(e) Network partitions . . . 70
(f) Operational requirements . . . 70
(g) Security requirements . . . 71
(h) Data protection requirements . . . 71
(i) Implementation . . . 71

6 Gluk **73**
1 Agriculture sensing analytics . . . 73
1.1 Overview of the use case . . . 73
1.2 Current development . . . 74
(a) Conflicting operations . . . 75
(b) Invariants that exist in the application state . . . 75
(c) Performance results . . . 75
(d) Security threats . . . 75
(e) Current deployment details . . . 76
1.3 Detailed description . . . 76
(a) Architecture . . . 77
(b) Edge computing requirement . . . 78
1.4 Data model . . . 78
1.5 Detailed description of the computations . . . 79
1.6 Conflicting operations and invariants . . . 80
1.7 Divergence and divergence control . . . 80
1.8 Network partitions . . . 80
1.9 Operational requirements . . . 81
1.10 Security requirements . . . 82
1.11 Data protection requirements . . . 83
1.12 Implementation . . . 83

7 Data Protection **85**
1 Introduction . . . 85
2 EU legal framework for the right to data protection . . . 85
2.1 Directive 95/46/EC . . . 86
2.2 EU General Data Protection Regulation (GDPR) . . . 86
(a) Increased territorial scope (extra-territorial applicability) 86
(b) Penalties . . . 87
(c) Consent . . . 87
(d) Data subject rights . . . 87
(d).1 Breach notification . . . 87
(d).2 Right to access . . . 87
(d).3 Right to be forgotten . . . 87
(d).4 Data portability . . . 88
(e) Privacy by design . . . 88
(f) Data protection officers . . . 88
(g) Data protection per use case . . . 88

8 Security Analysis **93**
1 Introduction . . . 93
1.1 Security versus data protection . . . 93
1.2 Methodology . . . 93
2 Coordination between servers and data storage for the Guifi.net monitoring system . . . 94
3 Service provision support for the Cloudy platform . . . 95
4 Pre-indexing at the edge . . . 97
5 Lambda functions at the edge . . . 98
6 S3 local cache of central data . . . 99
7 No-Stop RFID . . . 100
8 Smart metering gateway . . . 102
9 Agriculture sensing analytics . . . 103

9 Deep Learning **107**
1 Introduction to deep learning . . . 109
1.1 Three-step design process . . . 109
1.2 Why is DL successful now and not before? . . . 109
1.3 Introduction to the design of a deep neural network . . . 110
1.4 AlexNet: a practical deep neural network . . . 112
1.5 Deep learning compared to other disciplines . . . 113
2 Relevance of deep learning to LightKone use cases . . . 114
2.1 Computation model requirements for deep learning . . . 115
2.2 Generalized convergence property . . . 116
2.3 Training on edge networks . . . 116
2.4 Data protection and anonymization . . . 116

Bibliography **119**

A List of Acronyms **123**

B Questionnaire **125**
1 Overview of the use case . . . 125
2 Current development . . . 125
2.1 Conflicting operations . . . 125
2.2 Invariants that exist in the application state . . . 125
2.3 Performance results/figures . . . 125
2.4 Persistence . . . 126
2.5 Security threats . . . 126
2.6 Current deployment details . . . 126
3 Detailed description . . . 126
3.1 Architecture . . . 126
3.2 Edge computing requirement . . . 126
4 Data model . . . 126
5 Detailed description of the computations . . . 127
6 Conflicting operations and invariants . . . 127
7 Divergence and divergence control . . . 127

8 Network partitions 127
9 Operational requirements 128
10 Security requirements 128
11 Data protection requirements 128
12 Implementation 128

Chapter 1

Executive Summary

This deliverable presents real-world use cases of the industrial partners that require significant edge computing and documents their informal requirements. We have identified a large number of use cases covering the entirety of the project's design space; in particular they cover both light edge and heavy edge. We summarize the use cases as follows, grouped per industrial partner:

- *Coordination between servers for the Guifi.net monitoring system (UPC)* (Chapter 3, Section 1). Guifi.net is a community-driven project with the objective of creating an open telecommunications network based on a commons model. In the Guifi model, users collaborate actively in the provision of services, and contribute to sustain edge microclouds. In this use case, we reimplement Guifi's current centralized monitoring system at the edge, to improve its resilience and reliability.

- *Data storage service for the Guifi.net monitoring system (UPC)* (Chapter 3, Section 2). This use case is complementary to the previous use case and focuses on the monitored data. It will reimplement Guifi's current centralized data storage system.

- *Service provision support for the Cloudy platform (UPC)* (Chapter 3, Section 3). The Cloudy platform allows Guifi users to provide services and applications themselves at the network edge. This use case will replace the current service provision platform by a new platform that leverages LightKone's edge computing support, to increase autonomy and reliability of the service provisioning.

- *Pre-indexing at the edge (Scality)* (Chapter 4, Section 1). Scality provides a peta-scale hybrid cloud storage across multiple clouds. This use case will add metadata search at the edge, to reduce latency and improve search quality.

- *Lambda functions at the edge (Scality)* (Chapter 4, Section 2). This use case will extend the storage system to allow applications to themselves define arbitrary functions (i.e., serverless applications, also called lambda functions) to be executed by the storage system when ingesting new data and retrieving queries. These functions will be performed at the edge, allowing the system to be more scalable and cloud agnostic, and provide enhanced data freshness.

- *S3 local cache of central data (Scality)* (Chapter 4, Section 3). This use case will extend Scality's storage system to enable local caching of data on client sites at the

edge. This will improve latency and availability, reduce the volume of data being transferred to the client site, and allow temporary offline operation.

- *No-stop RFID (Stritzinger)* (Chapter 5, Section 1). Industrial manufacturing uses transport systems to move workpieces (materials and partial products) between processing stations. Each workpiece is identified by an RFID tag. Whenever a workpiece arrives at a processing station, the RFID tag is read and written which takes significant time. This use case will extend the current transport system by implementing a distributed cache of RFID contents. This will allow the system to process RFID data without stopping, which will significantly increase performance of the manufacturing system.

- *Smart metering gateways (Stritzinger)* (Chapter 5, Section 2). Digitization of utility metering is a growing market that promises significant cost savings and improved resource usage, and is a stepping stone towards smart grids. This use case will add smart gateways between the digital meters and the cloud. These gateways will improve reliability and scalability, and enable local decision making for smart grid applications such as battery management and controlling electric car charging. Section 2.13 gives a related use case focusing on swarms of small satellites.

- *Agriculture sensing and analytics (Gluk)* (Chapter 6, Section 1). This use case targets precision agriculture, in which sensor and actuator networks are used to improve management of agriculture. The domain chosen for this use case is winery management, which refers both to the wine cellars and the vineyards where the grapes are cultivated. The use case will aid the farmer by moving the analytics on the edge, which will improve visualization, prediction, and decision making.

The main body of this document gives detailed requirements for each of the above use cases. To aid uniformity and completeness, the requirements were identified by presenting and discussing the use cases with all partners in project meetings and by presenting to all industrial partners a questionnaire with detailed questions. This approach took significant time, but it was necessary to create a common discussion language between industrial and academic partners. In addition to the above use cases, we have identified three areas that affect all use cases:

- *Data protection* (Chapter 7). Many use cases require personal data as part of their operation. The implementation must therefore respect the users' right to privacy. We summarize the users' rights and explain how this affects each use case.

- *Security analysis* (Chapter 8). While data protection safeguards individual users' data, information security protects the information system itself. Security is therefore a necessary prerequisite for data protection. We do a preliminary security analysis for each use case.

- *Deep learning* (Chapter 9). Deep learning is a branch of machine learning that attempts to model high-level abstractions in data that are close to human-level understanding. For reasons of scalability and personalization, deep learning is increasingly used directly at the edge. The LightKone edge computing framework must therefore take into account the needs of deep learning at the edge. We do a preliminary study of deep learning with respect to edge computing.

Chapter 2

Introduction

Since the beginning of LightKone in January 2017, edge computing is continuing its rapid growth. At the Net Futures conference in June 2017, Jorge Pereira made the prediction that in 2027 there will be more than 1000 edge devices for every human being on earth [35]. This document therefore does not only present the use cases we identified at the start of the project, but also does an effort to keep up with the continuing development of edge computing, and even to predict where we think edge computing is going. We have identified new use cases, such smart metering (and its connection to smart grids) and swarms of microsatellites, and we have identified one important new technique, namely deep learning, that is poised to significantly affect edge computing. It is clear that by the time our edge computing platform is mature, the societal use of edge computing will have changed greatly.

This document presents the detailed informal requirements for the use cases defined by the four industrial partners in LightKone, in collaboration with the academic partners. These use cases will drive the development of our edge computing platform and its programming model. The use cases cover the specific challenges of both light edge computing and heavy edge computing. We have chosen to present a large number of use cases, all compatible with the expertise of our industrial partners. This gives us a better overview of the industrial applications of edge computing. It also gives us the most flexibility to select appropriate use cases depending on the future evolution of the Internet of Things. In addition to the use cases themselves, we identify three areas that affect all use cases, namely data protection, security analysis, and deep learning. It is important to understand these three areas on their own, independent of the limitations of particular use cases. We have therefore devoted an individual chapter to each area.

1 Motivation for the use cases

The Executive Summary has already introduced the use cases; we will not repeat that text. Instead, we will focus on the motivations of the industrial partners and how these motivations inspire the use cases.

In Chapter 3, UPC presents three use cases in the context of the Guifi.net community network. The first and second use cases tackle two independent components of Guifi.net's network monitoring system, with the objective of achieving higher resilience and reliability. The first concerns the coordination of the monitoring servers themselves,

whereas the second concerns the distribution and replication of collected data to support local and global analytics. The third use case aims to enable smarter service provision in edge microclouds, which increases the scope of their application scenarios.

Chapter 4 presents Scality's use cases. Scality provides large-scale object storage platforms to a variety of companies. One of the promises of object storage is richer metadata than traditional filesystems can offer. However, to truly benefit from the richer metadata, the storage system needs to offer sophisticated search capabilities. Centrally indexing billions of objects in a timely fashion can become an intractable problem. That is why Scality's use cases focus on increasing the abilities of their storage system. By supporting the ability to perform partial indexing at the edge by the system's clients, up-to-date search becomes realistic. With the addition of remote partial replicas, and by leveraging new edge and distributed serverless functionalities, as well as machine learning based inferencing at the edge to augment the object metadata, Scality will provide new abilities that no other object storage platform can offer.

Chapter 5 focuses on industrial manufacturing. Stritzinger provides solutions for industrial transport in factories. The first use case shows how to overcome a limitation of these solutions, which is the transport time: no-stop RFID removes this key bottleneck. Stritzinger also envisages an extension to the no-stop RFID with distributed online planning; this document does not present this extension further because it may be out of the scope of LightKone. In addition to industrial manufacturing, Stritzinger is exploring how to use its embedded systems technology for other areas, namely smart metering and swarms of microsatellites.

Compared to the other three industrial partners, Gluk's application domains are in what is traditionally considered to be Internet of Things, namely sensor-based edge applications. In Chapter 6, Gluk presents its vision of a sensor-based platform with analytics on the edge. They plan to use this platform in their products for many areas of daily life, such as agriculture, health, smart homes, etc., depending on their business strategy. For LightKone, Gluk have selected a use case focused on precision agriculture and the challenges it offers for edge computing.

2 The importance of crosscutting topics

In addition to specific use cases, this document presents three crosscutting topics that can potentially affect all use cases:

- Data protection (Chapter 7). Edge computing, by its very nature, is often directly connected to individual users. Since LightKone is developing edge applications, it is essential that we take the necessary precautions so that the user data is protected. Each use case addresses its own data protection issues individually; this chapter complements these explanations by presenting the guidelines and legal framework that are common to all.

- Security analysis (Chapter 8). This chapter presents an initial security analysis of each use case, based on the informal requirements given in the other chapters. Security is a necessary prerequisite to data protection; whereas data protection safeguards the users, security safeguards the application itself.

- Deep learning (Chapter 9). This chapter presents a short introduction to deep learning and an initial analysis of how it can affect the use cases. Deep learning is a branch of machine learning that can model high-level features in data that are close to what human beings perceive. It is increasingly being used in edge applications and current trends imply that it will have a significant effect on edge computing. It is therefore necessary for LightKone to make sure that our edge computing platform can support deep learning computations.

Data protection and security will be further elaborated in the rest of the project, during the design and implementation of the edge computing platform, and its evaluation using selected use cases. Deep learning is more speculative: we do not really know to what degree it will affect the edge computing platform and the use cases we will implement. What is likely is that we will need to implement some existing deep learning algorithms, for example to do the computations needed for a deep neural network. This implies that we need to make sure that our edge computing platform can support these computations.

3 Requirements elicitation and the questionnaire

In order to ensure consistent and complete use case definitions, we prepared a common questionnaire that was presented to each industrial partner. Appendix B presents this questionnaire. The questionnaire was designed by partner Stritzinger in concertation with the other partners. Its questions can be divided into three groups:

- The first set of questions focuses on the goal of the use case and its general properties including the need for edge computing.
- The second set of questions focuses on the data model, the computations done, invariants maintained and possible conflicting operations.
- The third set of questions focuses on the environmental conditions, such as network partitions, data protection requirements, security requirements, and operational requirements including constraints on implementation by the industrial partners.

Based on this questionnaire, each industrial partner defined a first set of use cases. These use cases were presented and discussed during a project meeting. Subsequently, the most relevant use cases were selected and their requirements further elaborated in concertation with the academic partners. An academic partner was appointed to work together with each industrial partner for the elaboration of the use cases. We decided to give broad explanations of each use case, not only including information that is obviously important (such as the data model, consistency requirements, and nonfunctional requirements such as divergence), but other information as well, so the use case would be complete.

Not all use cases are equally well understood; some are still exploratory, whereas others are more solid. This is why the use case descriptions are not all equally detailed. When we considered a use case to be interesting, we included it in this report, even if its explanation was less detailed than some others. We do not want to preemptively narrow the scope of this document; we prefer to cast a large net and to decide later on in the project what use cases to elaborate further.

Chapter 3

UPC

1 Coordination between servers for the Guifi.net monitoring system

1.1 Common background for the Guifi.net use cases

(a) Guifi community network environment

Guifi.net is a bottom-up, citizenship-driven technological, social and economic project with the objective of creating a free, open and neutral telecommunications network based on a commons model. [1] The whole network infrastructure can be seen as a **crowd-sourced, multi-tenant collection of heterogeneous wired and wireless network devices with an Internet Protocol (IP) address**, interconnected between them and forming a [partially-]meshed network.

(b) Edge computing in Guifi.net

Edge computing builds upon the advantages of cloud computing, but extends the traditional cloud services with the capacities of local processing. Edge computing solutions already operational in many industrial and consumer-oriented scenarios, provided by major Internet service providers and customised by specialised enterprises.

In Guifi.net, a different edge computing model is considered in which, by contrast to the above approaches, the users of edge services are enabled to collaborate and actively participate in the service provision, and contribute to sustain *edge microclouds*. The aim is a cloud which is formed by user-provided computing and communication resources to allow providing services of local interest. Services include Internet access [14], but also applications deployed within the community network. This concept perfectly matches the geographical distribution and the multi-tenancy of the underlying Guifi.net infrastructure.

The cloud infrastructure for edge computing in Guifi.net is located at the network edge and most devices that contribute to the resource pool are at the premises of the users or in installations of municipalities. The software platform installed on these devices is the Cloudy platform, which is open and can be extended with additional services by the participants. The hardware used to form the infrastructure is heterogeneous, ranging from

[1] What is Guifi.net? - https://guifi.net/en/what_is_guifinet

Single-Board-Computer (SBC) to desktop PCs. While the user can continue to access through the edge device the traditional cloud services, the collaboration and contributions among user devices enable a microcloud of edge resources and services.

This approach for edge computing in Guifi.net started to be researched and developed in the last few years and nowadays counts with tenths of operational devices in the community network [37].

(c) Current monitoring system for Guifi.net nodes

The monitoring system nowadays in production in Guifi.net is built around a central DB (which is coupled with the Guifi.net website) that lists all the nodes in the network (i.e. network devices, such as routers) and assigns them to the servers spread all over the network, which are in charge of monitoring them. Each of these servers periodically fetches a list from the central database (DB) containing the information about which nodes it has to monitor to then check their status (uptime, ping RTT and traffic on their interfaces). The information collected stays local to the servers, not being automatically replicated or distributed anywhere else. The current system has a few shortcomings that make it not fully reliable: each router is monitored, at most, by a single server; when a monitoring server goes down, this is not automatically reported to the Guifi.net website or the central DB so network nodes are left unmonitored because they are not automatically reassigned to another server; data are stored at a single location only; etc.

The current Guifi.net monitoring tool, called *SNPServices*, was developed by the community around the central DB and Guifi.net website, where the assignation of network nodes to the different monitoring servers occurs. The Guifi.net website's DB contains a list of all the network devices with an IP address (i.e. wired and wireless routers, Customer Premises Equipments (CPEs), switches, servers, etc.) that must all be monitored. This list also includes the monitoring servers running the SNPServices tool themselves, since they are also part of the network infrastructure and have their own IP address.

Each SNPServices instance (i.e. each monitoring server) periodically fetches an updated list with the nodes it has been assigned. The server will only monitor the nodes in the list, ignoring the rest of nodes in the network, no matter how close of far they are, its current workload, etc. Additionally, each network node will only be monitored by a single SNPServices instance (because of the way the Guifi.net website is implemented, a network node can only be assigned to a single monitoring server).

Figure 1.1 depicts the current implementation of the monitoring system for the Guifi.net nodes, based on the SNPServices tool.

While, in general terms, the SNPServices monitoring system works and fulfils the basic monitoring needs, it has several limitations and shortcomings, as it does not leverage technologies for automation, distribution of the workload and decentralisation of coordination and decision-making. To name a few:

- Need for manual intervention: besides declaring the SNPServices server on the Guifi.net website, the network zones and nodes usually require to be manually assigned to a specific monitoring server.
- No automatic reaction to server failures: if one SNPServices instance crashes, is stopped or removed, there is no automatic way to detect it from within the Guifi.net

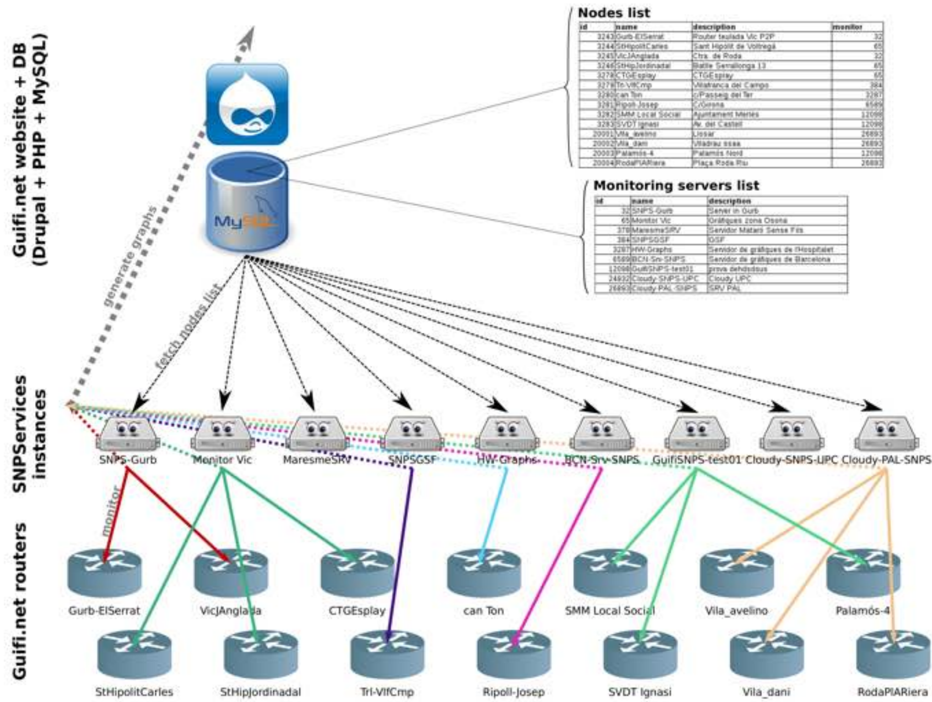

Figure 1.1: Architecture of the current Guifi.net network nodes monitoring system, based on the SNPServices tool.

central DB, in order to reassign nodes to another monitoring server.

- No redundancy: every node is monitored by (at most) a single SNPServices instance. Therefore, if this instance fails, the nodes assigned to it will stop being monitored, requiring manual intervention.

- No load balancing: a monitoring server might be overloaded or resources-constrained (in terms of computing power, storage or network bandwidth) while others might be idle, leading to an inefficient use of resources.

- No replication or distribution of collected data: each SNPServices instance collects and keeps monitoring data for a set of nodes, but no other instance has a copy of the data gathered.

As a result of this, some of the network nodes eventually end up not being properly monitored (if at all), single points of failure and risk of data loss appear, etc. Furthermore, the difficulty to access the collected data in a coherent and uniform way prevents taking advantage of automation and big data analysis techniques to reduce human intervention on the network, improve accounting and billing processes, etc.

1.2 Overview of the use case

In this section we describe the servers coordination use case for the Guifi monitoring system. Please refer to Subsection 1.1 for a general background about the Guifi.net environment, its relationship with edge computing and the description of the current monitoring system.

This use case will do the following: It will create and use distributed data structures to support the coordination of the server to router assignments in the monitoring configuration. The system will help to dynamically update and optimize these assignments, which can be done according to various criteria, such as server load, network status, resilience of the assignment with regards to the router position or accounting criteria.

This use case is part of the re-implementation of the current monitoring system, and aims at improving its resilience and reliability by means of automation, distribution and decentralisation, whose operation will be supported by Antidote DB. Four pieces of software are envisioned in the whole system: a first piece related to how the central DB (i.e. the Guifi.net website) feeds the monitoring system with the required information; a second piece to enable the monitoring servers to coordinate between them and distribute the workload on their own, according to different criteria; a third piece to allow the servers to store and share collected data in a redundant fashion; and a fourth piece for reporting information back to the central DB and the Guifi.net website. Of these four pieces of software, this use case covers the second one.

The monitoring servers will keep a distributed *monitoring servers* ⇔ *network devices* mapping which they will use to dynamically assign (and unassign) themselves which nodes to monitor in function of different criteria (network distance, workload, etc.). Such mapping will potentially be concurrently modified by any of the participating servers, at any moment. The main computations will consist in editing this mapping between the two sets of devices (network nodes and monitoring servers), while keeping it in a consistent status and ensuring that every node is being actively monitored by at least one server at any time.

In the context of a heterogeneous and geographically spread network such as Guifi.net, partitions of diverse duration might eventually appear (seconds, minutes or -in the worst case- hours). This could trigger isolated writings on the DB that would require reaching a consistent state later, when the network is restored.

Antidote DB's *Eventual Consistency* may allow to implement the mapping of monitoring servers to network devices in a distributed and decentralised way, without sacrificing the reliability of the system (e.g. avoiding all the monitoring servers suddenly withdrawing from monitoring a specific network node).

1.3 Current development

Please refer to Subsection 1.1-(c) for a description of the current development of the monitoring system for Guifi.net nodes. We now discuss specific issues that can occur in the current development.

(a) Conflicting operations

Currently there are no operations that manipulate the state of the application which need coordination. The current *monitoring servers* ⇔ *network devices* mapping is centrally generated, and there are no conflicting operations regarding which monitor watches each node.

(b) Invariants that exist in the application state

Every network node must be actively monitored by a functioning monitoring server.

(c) Performance results/figures

There are no known performance figures for the current monitoring system. The list of nodes contains $\sim 34,000$ active nodes and ~ 25 nodes are added daily. There are 285 monitoring servers, of which are 200 known - or considered - to be operative.

(d) Persistence

Both the *monitoring servers* $\Leftrightarrow$ *network devices* mapping and the collected data need to be persistent in the way they are implemented now. The mapping is currently stored in the central Guifi.net website and DB and the collected data is stored at the different monitoring servers.

(e) Security threats

A malicious user could set up a fake monitoring server that would not actually monitor network nodes, or that would do it inaccurately. This would be difficult to detect since, right now, each network node is only monitored by a single server and collected data is stored at a single location.

(f) Current deployment details

Currently there are around 200 monitoring servers known or considered to be active. These devices are low- to mid-end power devices, ranging from embedded ARM computers like the Raspberry Pi [2], low-power x86 Intel Atom-based devices like the Minix NEO Z64 [3], x86 virtual machines to refurbished bare-metal machines. The servers are spread all over the Guifi.net network in an organic way, usually without a carefully planned distribution. Monitoring servers are connected to the Guifi.net network through their local node; the physical connection is performed via cable (Ethernet). Guifi.net nodes are interconnected by a mix of mid- to long- distance wireless links (WiFi) and wired (fiber optics) links. Network latency between servers typically is in the $0.5\ to\ 10ms$ range, and throughput typically is between $1Mbps$ to $1Gbps$ (usually being at least in the $10s$ of $Mbps$).

1.4 Detailed description

The application detailed in this use case is part of the new monitoring system for the Guifi.net nodes aimed at automation, decentralisation, distribution, reliability and resilience. In particular, this use case covers the shared distributed storage that allows different monitoring servers to coordinate between them in order to spread the workload.

(a) Architecture

The general architecture of the proposed use case, showing the monitoring servers and their integration with the Guifi.net website and network nodes, is depicted in Figure 1.2. On the top of the picture appears the Guifi.net website and its central DB, which contains

[2]Raspberry Pi - Teach, Learn and Make with Raspberry Pi: https://www.raspberrypi.org
[3]Minix NEO Z64: http://minix.com.hk/en/products/neo-z64-windows

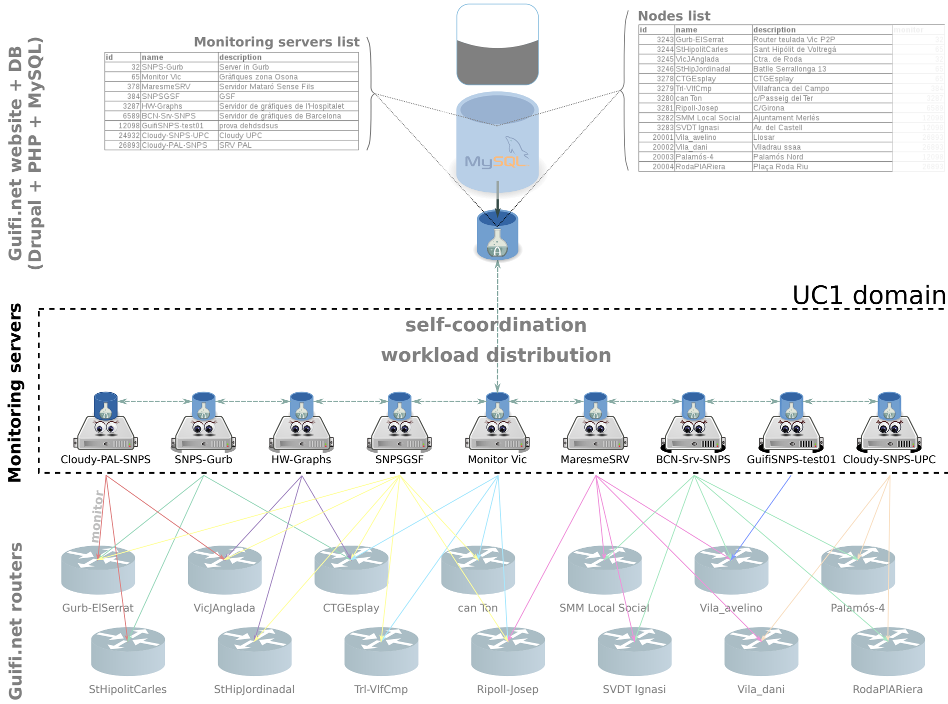

Figure 1.2: Architecture of the new monitoring system showing the different components in UC1

the lists of network nodes and monitoring servers. These two lists are always provided by the website, and can be considered to be correct and readily available at any time. Compared to the current implementation, the main difference lays on the proposed *monitoring servers ⇔ network devices* mapping, which is not generated centrally, but in a distributed and collaborative way, by the monitoring servers themselves. To achieve this, all the monitoring servers run a local instance of Antidote DB.

In this use case, the monitoring servers spread around the network share a distributed data space, implemented by means of Antidote DB. The servers use this data space to know which is the current *monitoring servers ⇔ network devices* mapping and update it according to different criteria, making sure that all nodes are being actively monitored and reacting to eventual failures (decommissioned servers, network partitions, etc.). Additionally, the main Guifi.net web and DB server also runs an Antidote instance, connected and synchronised with those running in the monitoring servers. This instance is used as the single authoritative entry point for updates on the nodes list. The list may also contain additional details or information about network nodes (i.e. their role in terms of network topology, performance or economic interest) in order to prioritise its monitoring or dedicate more resources to it.

The monitoring servers, once they have the list of nodes to watch, have to coordinate between them in order to perform the actual nodes monitoring. First, they have to assign every single node to - at least - one monitoring server. This task can be performed in many ways (for instance, each monitoring server could start picking, at random, nodes not yet assign and assign them to itself). Another method could be that monitoring servers probed their reachability and round-trip time (RTT) to every network node to then link each node to its closest monitoring server (or the closer ones).

In the context of a large Community Network (CN) like Guifi.net, where a combination of wireless and wired (fiber optics-based) links exist, it is possible to observe changes in the network topology as the dynamic routing protocols react to the status of the links between the nodes (link quality, RTT, throughput, usage, etc.). Despite unlikely, network partitions may also eventually appear, even if for short periods of time only.

(b) Edge computing requirement

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. In this context, local, autonomous edge applications that do not require or rely on the cloud better match the current operation of the infrastructure. Additionally, edge computing can provide more reliable monitoring data, especially in case of network partitions, low-throughput environments, etc. Nevertheless, backing the application with Cloud-based or datacenter-based computing might be considered for improved performance or reliability.

1.5 Data model

The data manipulated by the application consists of two sets of objects and a mapping between these objects. A simplified depiction of the data model and the application components involved in data manipulation is shown in Figure 1.3.

The first set contains a list with all the nodes in Guifi.net that have to be monitored. All the Guifi.net nodes are identified by a unique numeric ID (e.g. 58266), which remains immutable through all its lifespan. Additional information, such as an associated IPv4 address (e.g. 10.1.33.33) may be attached as a string-formatted JavaScript Object Notation (JSON) item. The current nodes list contains around 34,000 nodes, and grows at a rate of 25 nodes per day. Objects in this list are immutable (each network node is identified by its unique ID, which does not change through all its lifespan). The data in this first set is only modified by authoritative updates issued from the Guifi.net website; the monitoring servers only read it but do not modify it.

The second set contains a list with all the active monitoring servers. Servers are also identified by a unique numeric ID, being the servers list a subset of the nodes list (a monitoring server is indeed a device inside the network, with its own IP address, etc. that must be monitored too). The data in this second set is only modified by authoritative updates issued from the Guifi.net website; the monitoring servers only read it but do not modify it.

The mapping between the nodes list and the servers list can be seen as a collection of relations between objects, one in each set (one network node and one monitoring server). Any monitoring server may modify the mapping between nodes and servers (add, update or remove these relations at any time). According to different criteria -such as current workload, network status and other- each monitoring server will, for instance, assign itself a number of nodes and will update the *monitoring servers* ⇔ *network devices* mapping accordingly. This assignation may change over time, as new nodes are added to the list, the network conditions change, workload is redistributed, monitoring servers join or exit the pool, etc.

Given the nature of the application, and in order to successfully deal with concurrent updates of the mapping, eventual data consistency and integrity between the different DB instances are required. By leveraging these properties, it can be ensured that all network nodes end up being properly assigned to monitoring servers.

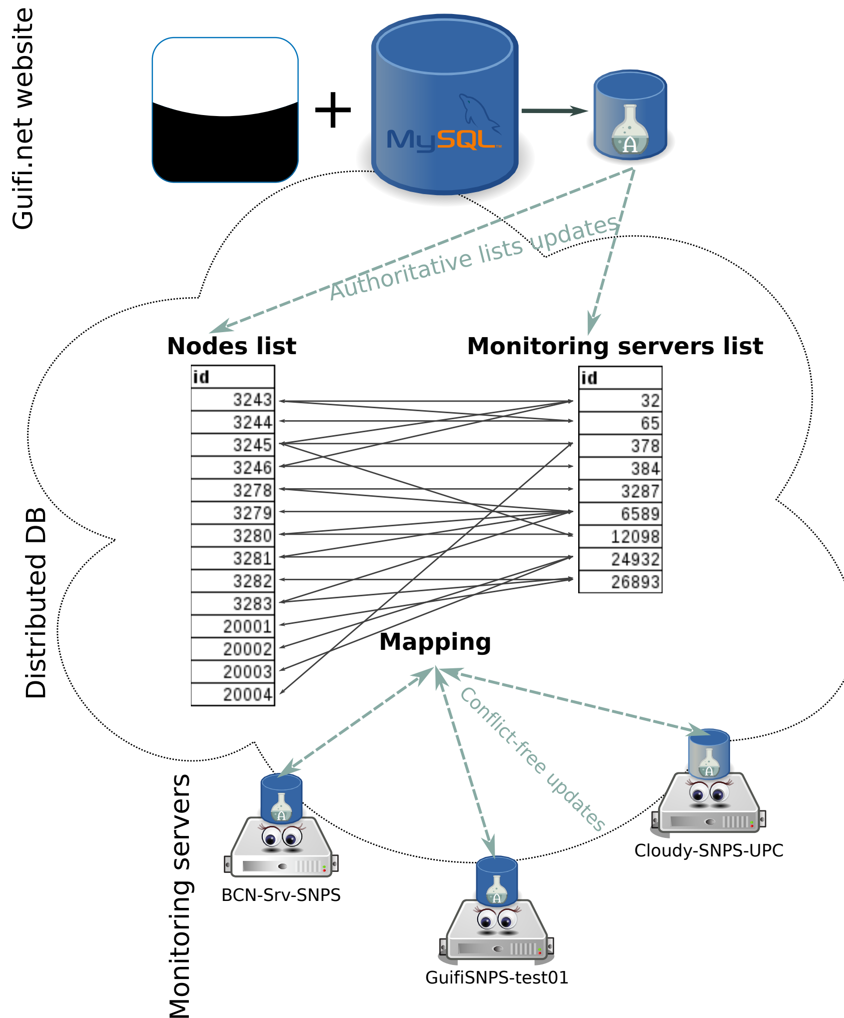

Figure 1.3: Representation of the data model and the different components involved in data manipulation

1.6 Detailed description of the computations

Given the network nodes and the monitoring servers sets (i.e. lists), and the mapping between the items in them, the monitoring servers will add, read, update or remove items to this mapping according to different criteria or triggering actions cited above (changes in the number of monitoring servers, addition of new network nodes, network partitions, etc.).

The depictions of the architecture and the data model in Figures 1.2 and 1.3 illustrate the data structures and roles. Due to network dynamics and server load variations, the mapping will need to be managed permanently by the monitoring servers. Therefore, each of the monitoring servers will require both lists and also their mapping.

1.7 Conflicting operations and invariants

The most important requirement for this application is that **every network node must be actively monitored by a functioning monitoring server** at any time.

Servers shall update the mapping between network nodes and monitors in the replicated DB without breaking the previous invariant. However, conflicting operations may appear when two servers try to update an item in the mapping list simultaneously. For instance, all the servers monitoring a given network node might report simultaneously they quit doing it. In case a [long enough] network partition occurs or if a monitoring server suddenly disappears from the pool, the remaining servers may proceed to delete the node assignations the failing server might had left.

Given the nature of the application, conflicting operations leading to temporarily inconsistent data would be acceptable, as long as the conflict is eventually resolved in a certain period of time, and every network node is monitored at any time.

1.8 Divergence and divergence control

The application is expected to work appropriately with a propagation latency in the order of a few seconds. An ”offline” mode or, similarly, a ”network partition mode” may exist for a while (minutes, hours...) but might not make sense after a period of time if, for instance, the nodes list becomes very outdated.

An indication on the divergence of data is not a must, but it might be interesting to have information about the degree or severity of the divergence (e.g. as a metric for knowing how much a monitoring service is ”integrated” in the whole system or out of it).

Towards this end, knowing that the data is not stale by more than some amount of time would be interesting.

1.9 Network partitions

Natural partitions exist in the system, as the monitoring servers are spread around a large network with different performance and throughput at every location. Some of the servers may be placed close to or by the core network routers while others may be placed further to the edge of the network.

1.10 Operational requirements

The application is running under Eventual Consistency (EC) for availability and resilience. It runs on low- to mid-end fixed infrastructure, in a few dozens of locations which may have datacenter (DC) conditions or not. For details on the infrastructure please refer to Section 1.3-(f).

In order to improve tolerance to network partition and churn, tens of full replicas of the mapping data could be desirable.

The operational application may have tens of thousands of objects, each object being a unique numeric ID, probably needing around 100 bytes per object. The total amount of data should be around a few megabytes, growing a few kilobytes per day. The objects universe is not partitioned.

1.11 Security requirements

Security is not a main concern right now, but features like access control on write operations permissions might be needed in the future. Auditing and rolling back offending updates or overwriting by authoritative entities would be positive.

1.12 Data protection requirements

The application does not monitor personal data, but the mapping of devices connected to a network. Therefore, there are no data protection requirements.

1.13 Implementation

Almost all the software developed for / in the context of Guifi.net is open source, therefore so should be this application.

2 Data storage service for the Guifi.net monitoring system

In this section, we describe the storage service UC for data collected by the monitoring system from the Guifi.net network nodes. Please refer to Subsection 1.1 for a general background about the Guifi.net environment, its relationship with edge computing and the description of the current monitoring system.

2.1 Overview of the use case

This use case covers the third piece of software of the monitoring system, which was presented in Section 1.2.

The use case will do the following: It will use a distributed data storage service structures to collect the data about the routers obtained from the monitoring servers. For monitoring servers, the system will keep the collected data in a replicated and distributed storage for analysis. Each server has gathered data locally, and will leverage the shared storage to distribute it and replicate, while also helping to store data generated by other servers.

Since more than one server may be monitoring a given node simultaneously, data concurrently generated will need to be compared, agreed and coherently merged. The main computations will be related to keeping a persistent distributed shared storage between the different monitoring servers, where the data concurrently collected will be stored for further analysis (performed locally or remotely).

The servers will, in general, store only a part of the whole data storage, but will be backed by more powerful *cloud-like* servers with increased storage able to store the data being collected by all the servers.

In the context of a heterogeneous and geographically spread network such as Guifi.net, partitions of diverse duration might eventually appear (seconds, minutes or - in the worst case - hours). This could trigger isolated writings on the DB that would require reaching

a consistent status later, when the network is restored. Additionally, this could lead to divergence in the data reported by different monitoring servers for the same network node; conflict resolution will be needed to deal with such situations.

Leveraging Antidote DB's features will allow to automate the replication and distributed across the different servers in a decentralised way, while keeping the data persistent and reliably stored.

2.2 Current development

Please refer to Subsection 1.1-(c) for the description of the current development of the monitoring system for Guifi.net nodes.

(a) Conflicting operations

The current monitoring system is not affected by conflicting operations since each monitoring server operates on its own, isolated from the other servers.

(b) Invariants that exist in the application state

The data collected by a monitoring server about a node must not be deleted until the node is removed from the Guifi.net central DB.

(c) Performance results/figures

There are no known performance figures for the monitoring application. There are approx. $\sim 34,000$ nodes that must be monitored, and around ~ 25 nodes are added daily. The current monitoring system uses $\sim 1MB$ per monitored device.

(d) Persistence

The collected monitoring data must persist server reboots, service software updates, etc. and not be lost when a monitoring server fails or is decommissioned.

(e) Security threats

A malicious user could set up a monitoring server that reports fake data for some or all the nodes. This could lead to an unreliable monitoring service or unreliable data on which to elaborate stats, graphs, etc.

(f) Current deployment details

Please refer to Section 1.3-(f) for details on the infrastructure.

2.3 Detailed description

The application detailed in this UC is a new monitoring system for the Guifi.net nodes aimed at decentralisation, distribution, reliability, resilience and automation. In particular, this UC covers the persistent storage for servers to share and store collected data in a redundant and distributed way.

(a) Architecture

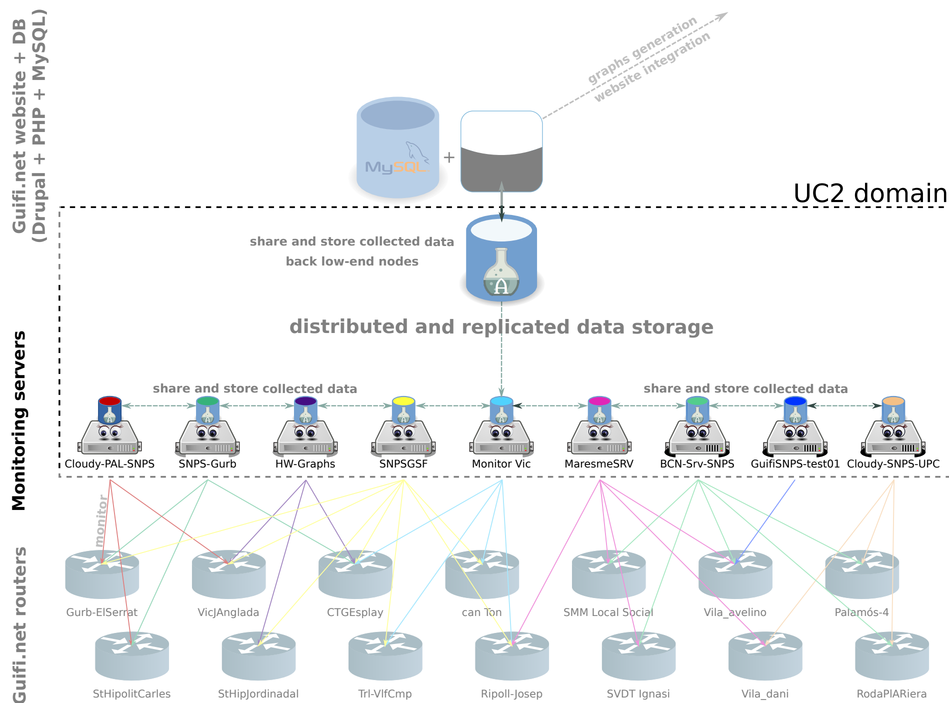

Figure 2.1: Architecture of the new monitoring system showing the different components of UC2

The overall architecture of this second UC is depicted in Figure 2.1, which shows the monitoring servers distributed among the Guifi.net network and the *cloud-based* storage backing them. The monitoring servers shown in the middle of the picture are in charge of checking the different network nodes below, and store the gathered data in a replicated and distributed way by leveraging Antidote DB. On top of it lies a *cloud-based* DB instance, with a greater storage capacity, that is able to store all the data being gathered and generated by the monitoring servers themselves. This DB also provides the data needed to elaborate stats and graphs to be shown via the Guifi.net website, depicted on top.

The different DB instances are interconnected and synchronised, and are able to share different data sets, depending on their hardware characteristics and available resources. For instance, one low-end monitoring server might only store the data collected by itself, while more powerful devices can store theirs and other servers' data.

(b) Edge computing requirement

The edge computing requirement described in the first use case in section 1.4-(b) apply also to this second use case. In addition, in this particular case, the edge devices are to be based by *cloud-like* or *datacenter-grade* infrastructe, with enough computing and storage resources to manage all the data being generated by the application, for increased performance, reliability and ease of access to the data.

2.4 Data model

The data managed by the application consists of tens of thousands of time-series data points, corresponding to the different aspects of network nodes being monitored (RTT, interfaces traffic, etc.). These data are stored in Round-Robin Database (RRD) format, with a fixed maximum size per attribute of 1 MB. This means that, as data are continuously collected, the required storage is kept constant. To achieve this, older values are averaged over time, resulting in less fine-grained numbers.

2.5 Detailed description of the computations

Given the network nodes monitored by the servers (i.e. lists), these servers will add the collected monitoring data to stored objects. On this data, aggregations will need to be performed. Additional operations may become of interest for the detection of events or anomalies.

2.6 Conflicting operations and invariants

For higher resilience, each router should be monitored by more than one servers. Each measurement of the different servers is obtained at a specific moment identified by a timestamp. This measurement data should be merged and correctly ordered taking into account the clock of each monitoring server.

2.7 Divergence and divergence control

As in the previous use case, the application can probably work appropriately with a propagation latency in the order of a few seconds. An ”offline” mode or, similarly, a ”network partition mode” may exist for a while (minutes, hours...) but might not make sense after a period of time if, for instance, the nodes list becomes very outdated.

Having information about divergence is not a must, but might be interesting to have information about the degree or severity of the divergence (e.g. as a metric for knowing how much a monitoring service is ”integrated” in the whole system or out of it).

Regarding the quantification of divergence, knowing that the data is not divergent by more than some amount would be interesting.

2.8 Network partitions

As in the previous use case, natural partitions exist in the system and may also temporarily happen, since the monitoring servers are spread around a large network with different performance and throughput at every location. Some of the servers may be placed close to or by the core network routers while others may be placed further to the edge of the network.

2.9 Operational requirements

The application runs under EC for availability and resilience. It runs on low- to mid-end fixed computing infrastructure, in a few dozens of locations which may have DC conditions or not. For details on the infrastructure please refer to section 1.3-(f).

The storage system should have tenths of full replicas of recent data in order to allow accounting and billing operations to be conducted.

Tens of thousands of objects, each object being a unique numeric ID, probably needing 100 bytes per object. The total amount of data should be around a few megabytes, growing a few kilobytes per day. The objects universe is not partitioned but it could be interesting to have it partitioned.

2.10 Security requirements

Security is not a main concern right now, but features like access control on write operations might be needed in the future, as well as digital signatures for authentication. Auditing and rolling back offending updates or overwriting by authoritative entities would be positive.

2.11 Data protection requirements

The applications do not use personal data, but stores data about monitored networking equipment. Therefore, there are no data protection requirements at this point in time.

2.12 Implementation

Almost all the software developed for / in the context of Guifi.net is open source, as should be this application.

3 Service provision support for the Cloudy platform

3.1 Overview of the use case

This use case aims to improve the Cloudy service provision platform used in Guifi.net in the provision of services and applications through microclouds at the network edge.

The use case will do the following: It will create and use distributed data structures to support service publication and service discovery in the microclouds formed by the Cloudy platform[4]. As a result, the services offered in microclouds are instantly found by other Cloudy nodes. Data analysis may be performed on the persistent historical service data to support the development of new features of the Cloudy platform such as service placement and predictions.

On top of the Guifi.net network infrastructure, participants (e.g. individuals, Small and Medium Enterprises (SMEs), organisations) run their own public and private network services (backups, Virtual Private Networks (VPNs), file sharing, Internet access, etc.). Some of these are offered for free, usually in a best-effort manner (e.g. a user offering bandwidth-limited proxy-based Internet access), while others are offered for a monthly fee (and including some guarantees and adequate Service Level Agreement (SLA)). This use case focuses on the framework for the provision of these services beyond the network operation itself (e.g. monitoring). These services complement the Internet connectivity, and should add value to the network.

[4]Cloudy - A community networking cloud in a box: https://cloudy.community/

The services publicly and openly offered on top of Guifi.net (i.e. by the community, for the community) need to be publicly announced through the Guifi.net website. This process, however, is performed mostly manually, which leads to many services not being published, updated or unpublished when they are shut down.

Historically, community services in Guifi.net have been hosted in heterogeneous hardware, such as low-end x86 computers, refurbished desktop Personal Comupters (PCs) and servers. More recently users in Guifi.net have started using Virtual Machines (VMs) and lately Docker [5] thanks to the Cloudy platform.

The servers providing services to the community are deployed in an organic fashion all around the network, without a careful distribution plan, as communities or individuals have installed them to fulfil their needs. They are found, therefore, both at the network edge (at the users' premises) and at more central nodes with better network connectivity. On the one hand, this heterogeneity in terms of hardware and software might add an extra degree of complexity but, on the other hand, almost all of the services run in Linux-based platforms and are based on open sourced code.

3.2 Current development

A limited version of the use case proposed here is already materialized and is currently in production [6]. Nowadays, the Cloudy platform leverages Serf as the backend for the servers to publish their running services and discover the others all over the network. Serf is a decentralised, fault-tolerant, lightweight and highly-available tool for cluster membership, failure detection and orchestration based on a gossip protocol.

While Cloudy has relied on Serf for the few past years and has proven very convenient in the few past years, having a multi-tenant persistent-storage weakly-consistent backend (similar to Amazon DynamoDB, powered by Antidote DB) would be one of the steps to investigate for simplifying such deployment of applications, running alongside a container/Virtual Private Server (VPS)/server provisioning service.

This storage backend could improve on some of the limitations we face with the current Serf-based service publication and discovery (current limitations: limited message size in Serf, no historic data saved, no analytics on service data to trigger smart ”actions”).

(a) Conflicting operations

The number of services per service provider can be updated concurrently by the different service providers. Currently, however, the data objects for this information are not shared for writes. The resulting list of services per service providers is obtained from the above data object.

(b) Invariants that exist in the application state

In the current implementation, the list of services and service providers is always available from the search operation.

[5] Docker - Build, Ship, and Run Any App, Anywhere: https://www.docker.com/

(c) Performance results/figures

In the current use case development, there are no qualitative performance metrics. The usage is for the information of end users and using a best effort model. This approach excludes other scenarios which have already been envisioned [7], such as commercial services, which may need performance metrics to operate with SLAs.

(d) Persistence

The information on services, which is currently available in a Cloudy microcloud, refers to *instantaneous values* of the services which is queried to the messaging service (implemented by Serf). This data includes the availability and service characteristics. This data is not persistent. It is not stored at the Cloudy nodes nor at the Serf daemon, but disappears if the remote service providing Cloudy node disappears or is not reachable. Regarding the instantaneous values, it could be interesting to dispose of a copy persistent within a limited time in order to compute indices from the service availability, such as contributions which could be used by a billing and accounting system.

Historical data on services is currently not stored nor available, though it is desirable to have it, which would allow to conduct analysis and predictions to improve the service performance. What the historical data concerns, the usage could include 1) the mentioned billing and accounting, for which several months of data should be stored, and 2) data analytics for prediction, for which persistence for a data size suitable for the *training* of machine learning tools would be required.

(e) Security threats

Currently there are no specific measures in place to address security threats. As a consequence, there is the possibility that users can provide incorrect or manipulated data. It is not an issue at this point of time, but may become an issue for the envisioned advanced role in the future of such microclouds at the network edge.

(f) Current deployment details

The Cloudy nodes are deployed in the Guifi community network. Tenth of nodes use the Cloudy platform. The devices on which this software runs are heterogeneous. There is no specification for the hardware of a Cloudy node. Nevertheless, the scenario suggests as scope of hardware mini-PCs, which the users can run in a 24/7 mode with low energy consumption. The Cloudy devices communicate with each other over IP.

3.3 Detailed description

A distributed framework for the deployment of services, where the different server machines or instances are interconnected and share a common space, would allow the deployment of automatically orchestrated services, especially those publicly available to any user, but would also allow for commercial services to appear and be offered and requested on demand. Having a multi-tenant persistent-storage weakly-consistent backend, e.g. powered by Antidote DB, would be one of the obvious steps in simplifying

such deployment of applications, running alongside a container/VPS/server provisioning service.

The envisioned needs for this use case are components that enhance the service provision function of the Cloudy platform with improved scalability, reliability and availability, described in the following subsections.

(a) Architecture

The common software platform in Guifi.net to enable a microcloud at the network edge is Cloudy[6]. Cloudy runs on diverse hardware in Guifi.net as illustrated in Figure 3.1. Cloudy nodes within a microcloud have different administrative domains, where each node often belongs to a different community network member. There is also a multi-tenant usage of the cloud node's resources, since the resources are shared with the community, therefore used by the community and the node owner.

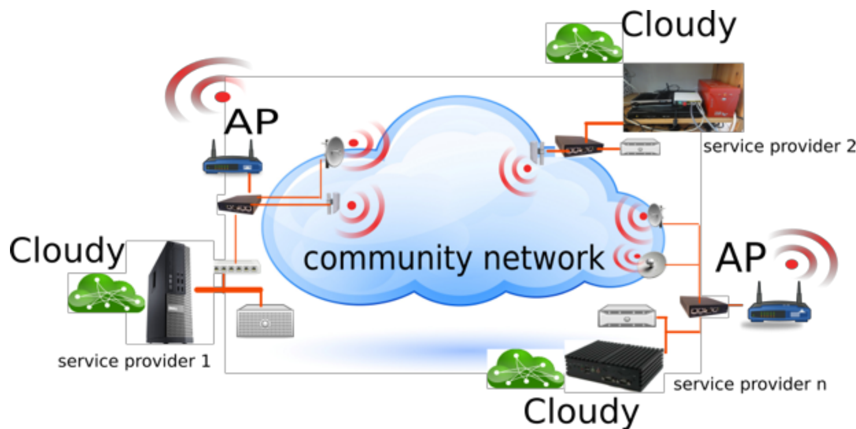

Figure 3.1: Heterogeneous Cloudy devices in Guifi.net

(b) Edge computing requirement

The infrastructure used to form the microcloud environment is composed by the Cloudy devices located at the network edge. The edge computing scenario is therefore the resulting model from Guifi.net's service provision approach.

Data center cloud computing would also be compatible with the Guifi.net approach in the sense that servers could be achieved by a crowd-funding approach [8] and being available as a common good. From an organizational point of view, however, such crowd-sourced servers seem to be more difficult to achieve at this point of time. First, users will need to be satisfied and convinced of the value of microcloud-based services. After this has happened, the interest and usefulness perception of for instance backup services would increase the willingness to conduct crowd-funded hardware purchase.

From a technical point of view, data center cloud infrastructure within Guifi.net could provide a valuable support to an edge microcloud. Many services could be offered with improved guarantees and functionalities if a stable data center infrastructure can be leveraged by the edge-based services.

[6] http://cloudy.community/

From a conceptual point of view, edge computing with microclouds can also be seen as a consequent application of Guifi's network provision model, which provides the communication network as a common good by the contribution of many single networking devices, to service provision.

3.4 Data model

The main data objects are first the service availability object referring to the services and the list of providers (Cloudy nodes) where they are offered, and secondly, the service provider object, referring to the service providers and the list of services each of them offers. From operations using these basic data objects, additional information could be extracted and stored in order to support specific functions which this use case could enable.

3.5 Detailed description of the computations

The main computations for this use case will be related with keeping a *distributed and persistent storage layer* to improve the communication and coordination between the different crowd-sourced servers and applications. In addition, functions like orchestration, smart service placement or workload distribution, which are not present as of today, could be fostered by operations on the data objects. While the main operations that need to be supported include write, read, update and delete of data, additional operations, which would implement specific steps of a recognition mechanism, could target to extract information from the data.

3.6 Conflicting operations and invariants

The service availability object as well as the service provider object should contain consistent data to reflect the instantaneous service availability. Using inconsistent data by users would lead to reduced Quality of Experience (QoE). In M2M interactions, inconsistent data could lead to incorrect results and decisions.

Concurrent writes on the data objects could for instance produce incorrect service availability information. A service may appear to be available to other nodes in a moment when the service providing nodes is not reachable.

3.7 Divergence and divergence control

In the edge environment found in Guifi.net there is latency in the network layer, which is caused by the network characteristics. Divergence within near-real seems acceptable when the range of the quantities, on which this divergence happens, are not critical for the service provision. For instance, if for a determined service a large amount of service providers are available, a temporal divergence of a observed value from the real value will not be critical. Differently, if the quantities are very small, with divergence a service availability may be suggest while it is not the case, which would produce incorrect operations and should be avoided.

If there is divergence, then information or qualification of the information with regards to divergence may be interesting, but it is not a must.

A probabilistic quantification for the divergence could be a nice to have feature, but is not essential while the services are provided as best effort and without SLAs. However, for future scenarios which could involve commercial operations, such probabilistic quantification could be important for cloud service providers to decide on measures for SLA compliance. Specific requirements on the metrics, however, would possibly be service provider dependent.

3.8 Network partitions

In the community network the servers are spread around a large network with different performance and throughput at every location. Network partition happens from time to time such that some servers are unreachable.

Even if there is no partition, latencies may vary during the network operation. Very different latencies between the service providing nodes can happen during the day between two determined nodes and in general, between service provisioning nodes, there will be different latencies within the community network scenario.

3.9 Operational requirements

The application is running under EC because of the characteristics of our scenario, where most servers are crowd-sourced and deployed at the edges of the network. The application currently does not involve data centers, but may do so in the future to add additional storage capabilities.

The devices on which the service provision framework is deployed are mainly low- to mid-end devices (embedded ARM computers like the Raspberry Pi [7], low-power x86 Intel Atom-based devices like the Minix NEO Z64 [8], x86 virtual machines and refurbished bare-metal machines).

Service data objects should be replicated in the future to address churn, network partition and performance.

The number of service data objects can initially be estimated as being in the range of hundreds of objects. However, if the current trend towards microservice provision gains a stronger momentum in the future, we can expect that the number of data objects will greatly increase.

3.10 Security requirements

At the current point of the use case, security issues are not a main concern now, but required for future extensions. For instance, access control on write operations might be needed in the future, as well as authentication. The Guifi.net Lightweight Directory Access Protocol (LDAP) server could be used for authentication and approval or denial of operations. Message integrity could also be a desirable feature. Due to the transparency concept used in Guifi.net, confidentiality may only be needed for specific cases.

[7] Raspberry Pi - Teach, Learn and Make with Raspberry Pi: https://www.raspberrypi.org
[8] Minix NEO Z64: http://minix.com.hk/en/products/neo-z64-windows

3.11 Data protection requirements

The purpose of the service is to make service announcements to the public (other Cloudy users). Data protection requirements for the use case operation are not identified at this point of time. While the services/servers or applications may handle user-sensitive data, the use case itself does not manipulate user data.

3.12 Implementation

The Cloudy platform leverages PHP and Bash Shell scripting, and interacts with other components and applications through a Representational state transfer (REST) Application Programming Interface (API). As most of the software developed for / in the context of Guifi.net is open source, released under the GNU General Public License (GNU GPL); further contributions to the platform shall follow this approach.

Chapter 4

Scality

1 Pre-indexing at the edge

1.1 Overview of the use case

While most enterprise data today originate from and is stored in on-premises storage solutions, use cases for hybrid cloud storage are emerging in many industries. For example, in media, the creation of content in on-premises private clouds leveraging object storage has become prevalent, but the use of public cloud services for content distribution (CDN) or compute-bursting for transcoding is growing. There exists thus a need for a centralized abstraction of multiple storage services, either on-premise or 3rd party object storage infrastructure platforms.

An important capability of a hybrid cloud storage solution is to enable metadata and semi-structured data search, across the federation of all underlying clouds. The ability to perform searches may be a preferred way for applications to retrieve objects, and would be a natural retrieval mechanism that can complement the usual by-key semantics of object storage systems.

Scality has introduced the Zenko Multi-Cloud Controller (Fig.1.1), an open-source project that provides a unifying storage interface (an Amazon S3 compatible API) while supporting multi-cloud backend data storage systems. Backend storage systems include both on-premise and as well as other cloud services, including: Amazon S3, Microsoft Azure and Google Cloud Platform. Zenko provides an engine that federates metadata to enable policy-based data management. It enables data replication across clouds, data migration services and will be extended to allow cloud workflow services, including cloud analytics and content distribution. Currently, work is ongoing on Zenko to provide an Apache Spark-based metadata search tool on application-defined S3 metadata. The objective of that development is to unify metadata search across the namespaces/clouds managed by the system.

There are a number of challenges associated with providing metadata search in a geo-distributed multi-cloud storage system. Approaches like the one currently being developed, that do not involve indexing, require significant resources at query time and can suffer from excessive delays. Indexing while data is being written increases both the latency of write operations as well as the IO requirements on the backend systems. If batch or background indexing is performed, the staleness of the indexes increase, thus reducing the usefulness of the indexing.

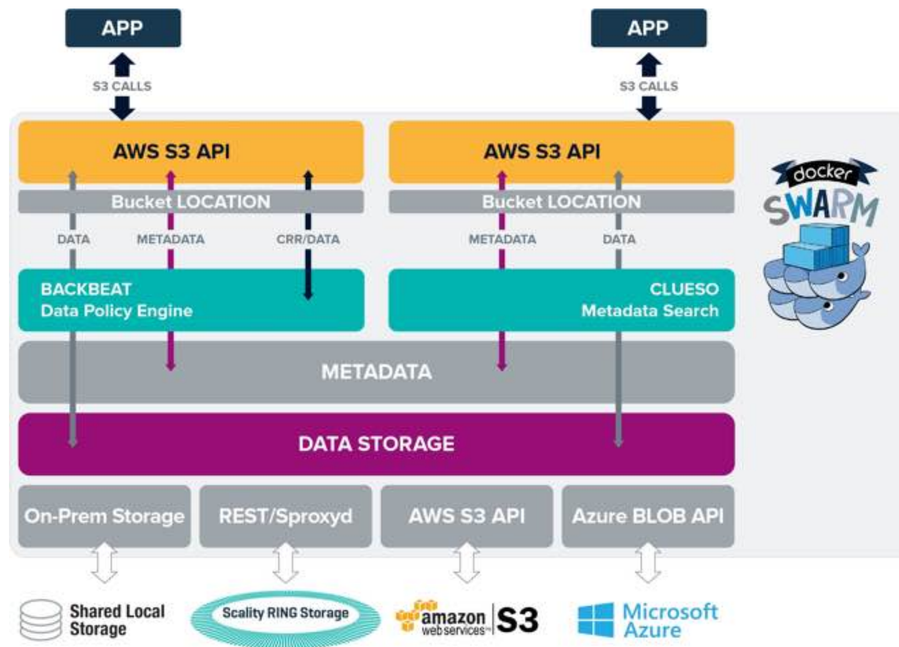

Figure 1.1: Zenko Multi-Cloud Controller Architecture.

In this use case partial indexing will be performed at the edge using the clients that are storing the data or possibly using other edge resources. These indexes will be created in real time or as nearly so as possible. The partial indexes can either be combined into a global index in the background or combined at query time to provide complete results. Ideally, a measure of the incompleteness of the indexes will be provided together with query results. Results of such queries can be used to produce views of the data, for lifecycle management, and for improving compliance by identifying sensitive data from metadata tags. The current desired scope is to provide indexing on all object metadata, but full text indexing of stored objects might become realistic if sufficient edge resources can be made available.

By performing partial indexing at the edge, staleness and write latencies can be significantly reduced, and the sophistication of the indexing can be increased. It becomes possible to implement advanced techniques such as machine-learning inferencing-based indexing on different data types. Several possibilities for more sophisticated indexing options will be discussed further in later sections of this document.

1.2 Current development

As of today, Zenko includes an Apache Spark-based metadata search component. While the basic metadata search feature exists, there are a number of challenges associated with the model. The current implementation does not index data as it is being stored and does not distribute indexing work across the system. One key challenge is to generate partial indexes in such a way that global search results can be obtained in a timely fashion, even in the event of a network partition.

This use case aims at addressing the limitations of the current implementation of metadata search in Zenko, as well as exploring further enhancements to the existing sys-

tem which could be made possible by performing computations at the edge. Potential benefits that edge computing can bring to the system are further discussed in 1.3(b).

1.3 Detailed description

We are envisioning a new system that will work along with Zenko, and improve the system's search functionalities. The system should perform pre-computations on data at the client side, before storing them to the backend storage systems. Pre-computations may include maintaining indexes, in order to enable more efficient search on S3 metadata and semi-structured data, generating object hashes, and encrypting data. These computations could be performed on a per-client operation basis. The computed sub-indexes and hashes would be transferred to the storage backends along with the encrypted data, reducing central site computation loads, in the case of on-premise storage systems that support this functionality. For other clouds that do not enable indexing, the system could perform merging of the per-operation sub-indexes, maintain and store the resulting index, thus enabling metadata search across all clouds.

(a) Architecture

Figure 1.2 depicts a sample architecture for the proposed system. A layer of pre-computation nodes operates as an intermediate layer between client applications and the backend storage systems. These nodes may act as index caches, storing partial indexes generated from recent client writes. When a client performs a write, the index entries generated from the write will be stored on a pre-computation node. Pre-computation nodes may then communicate with each other in order to aggregate partial indexes. Eventually, indexes should be flushed from the cache and merged to the global index, which will be stored at the backend systems.

The clients accessing the system are likely the most numerous component and are consequentially the most interesting for use in edge computation. The backend servers are typically deployed in groups of five, so that they number less than 20 servers, even in a very large deployment. It remains to be determined exactly how and where the pre-computation nodes should be deployed.

(b) Edge computing requirement

The proposed system can benefit from the use of edge computing by:

- Reducing computation load at the core of the system and allowing real-time updates of indexes;
- Improving responsiveness in case of partitioning;
- Accommodating different cloud systems, providing different functionalities and data models;
- Enabling more sophisticated indexing techniques. Deep learning inferencing could be used to introduce more sophisticated index elements in complex data such as images, videos or text documents;

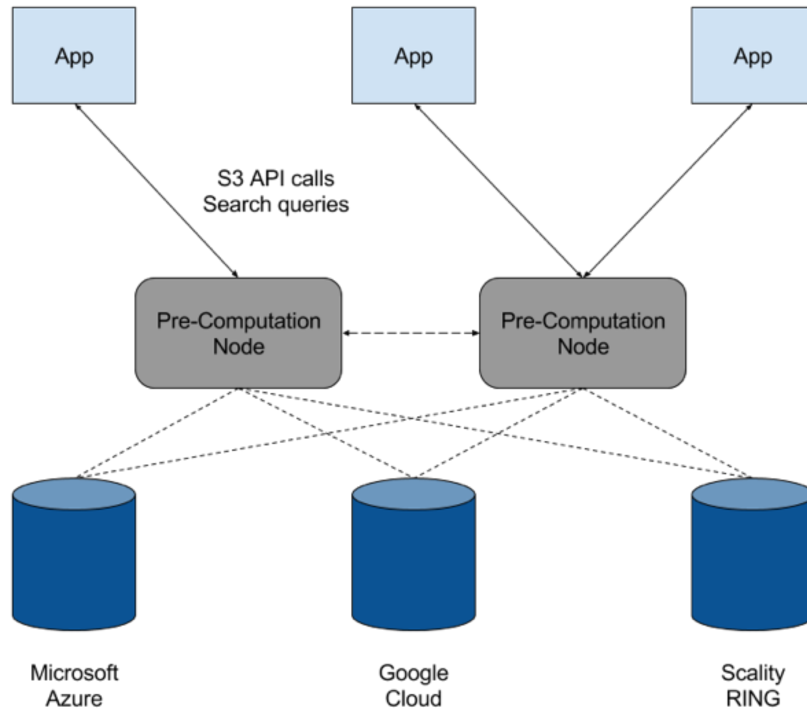

Figure 1.2: Proposed system architecture.

- Pre-indexing before encryption allowing search on encrypted data, which can only be done at the edge before data is encrypted.

1.4 Data model

The system's data model is that of an object storage. The system stores objects, composed of a unique identifier (key), an uninterpreted blob of data (content), and a set of metadata attributes. Metadata consist of both attributes generated by the storage system (content size, time of last modification, author, access control lists), and custom, user-defined attributes, represented as arbitrary key-value pairs (tags). Objects are organised in collections called buckets, which form a flat namespace.

Object data is immutable, while metadata is mutable. Writing to an already existing key creates a new version of the object. However, object's metadata can be modified (adding and removing tags, modifying ACLs) without creating a new version.

The indexes proposed here allow searches potentially based on all the metadata available. The exact form that these indexes should take, remains a somewhat open topic. Precise schemas allow fast searches of specific attributes, but limit the flexibility of of the searches that can be performed. As much as possible an indexing model that supports mapping of index data onto different schemas should be preferred.

1.5 Detailed description of the computations

Computations to be executed at the edge include:

- Hash signature generation: Hash message authentication is used for authenticating

S3 uploads;

- Encryption of the data;
- Index generation: Generating inverted indexes on metadata attributes or semi-structured content, possibly involving bitmap, or compressed bitmap computations. Potentially index generation on more complex data such as images or text;
- Index lookups in response to search queries.

Additional computations may involve creating query execution plans including sub-queries to multiple indexing nodes and storage backends, and joining the retrieved results.

Typical data flow patterns include applications writing data to the storage backends and retrieving search results from the system. Data are transferred from client applications to an intermediate layer, where the above computations are performed and then to the backend storage systems. In the case of search queries, a list of object keys (and not the actual data) is returned from the intermediate layer to the client application.

1.6 Conflicting operations and invariants

Invariants involved in the operation of the indexing system are:

1. If a condition holds for an object, then an index lookup for this condition should contain the object as part of the results.
2. If an object is contained in the results of an index lookup for a given condition, then the condition holds for this object.
3. *As an option*, if access control does not allow objects to be viewed, they are filtered from results. This requires causal consistency for ACL updates (see Sec. 1.10).

Violations of (2) (false-positives) should be allowed as they can be corrected by checking the source data. Violations of (1) (false-negatives) may occur temporarily, but the invariant should eventually be true, once the index is up-to-date with the source data.

Concurrent updates may result in conflicting application states. As an example, two applications may concurrently update an object's metadata attribute with different values. The pre-indexing system will separately generate index entries for both tag values. If the storage system then resolves the conflict by selecting one version of the object's metadata, this will result in a situation where both version exist in the index, and only one of them is true.

1.7 Divergence and divergence control

Indexes may be out-of-date, but can be allowed to gradually converge. Applications will benefit from knowing how divergent search results are, and possibly will be able to specify a maximum amount of divergence allowed, or generate system alarms or warnings when limits are exceeded. Useful data on divergence includes:

- The number of objects stored but not yet indexed;
- The percentage of the total data not yet indexed or updated;

- Time since most recent data sync;
- Estimation of time to full convergence.

Additionally, client applications will benefit from being able to specify a maximum amount of divergence that can be allowed.

Probabilistically Bounded Staleness [9] presents a model for providing expected bounds on data staleness. This work can be studied as a starting point for quantifying the divergence between index and source data.

1.8 Network partitions

The system should be able to tolerate partitions between pre-computation nodes. Disconnected partitions could maintain sub-indexes independently, and merge them once the connection between them is re-established. The ability to continue writing data in the event of a partition is a highly desirable characteristic, if the indexes can converge after the partition is resolved.

However, partitions between the backend storage and the rest of the system may not be tolerated, as the system would neither be able to persist the source data nor the indexes.

1.9 Operational requirements

The system should:

- Be able to scale to billions of objects, and petabytes of data;
- Be geo-distributed across different geographic locations;
- Remain available even in the presence of network/server/data centre failures.

1.10 Security requirements

Updates to access control objects are handled on other parts of the system. However, access control should be applied to query results to prevent data leakage via indexes. It can be managed either on object or bucket granularity, with bucket granularity being the minimum requirement. Returning search results for inaccessible data should be configurable, but if activated, search results must be filtered to hide private data. A corresponding ACL invariant is mentioned in Section 1.6 of this chapter.

1.11 Data protection requirements

The sensitivity of the stored data varies greatly across the product's user base. All data is expected to be treated with care, but highly-sensitive users are expected to encrypt data at the source. At rest, encryption also remains a possibility and should be considered as an edge function. Notably, as mentioned, index creation before encryption is an attractive functionality.

1.12 Implementation

Long-term plans include integration of the indexing system into the core Scality technologies for storage at scale of immutable data and object metadata. A fully distributed LevelDB implementation with RAFT-based distributed session management is used in this model. An open source version with single LevelDB instances will be used for much of this work. The code is primarily Javascript and Typescript based Node.JS code. The sources are available on Github as well as Docker images on Docker Hub. More details are available on the the Zenko.io website. The existing development only provides reliable storage of the metadata associated with the objects and provides no indexing capabilities. An implementation of indexing that does not increase latency or reduce availability is highly desirable. For this, leveraging Antidote and SyncFree with CRDT based semantics currently appears to be the most promising approach.

2 Lambda functions at the edge

2.1 Overview of the use case

Serverless compute services enable cloud-based applications to run application code on cloud infrastructure without the need for infrastructure management. They employ an event-driven model, enabling users to define functions to be executed in response to events. These services are suitable for a number of different use cases, including data analytics, log filtering, and data transformations.

An important extension to a hybrid cloud storage system would be to enable applications to define lambda functions that will be executed in the path of data ingestion in response to specified events, and produce results that can be then retrieved by queries. Performing these operations at the edge, and then aggregating the results to produce system-wide results can enable the system to be more scalable, cloud agnostic and enhance data freshness. There are similarities with a map-reduce approach, but the concept is more a data-flow model, where all data is analyzed or transformed as the data is being persisted. The outcome may or may not involve a reduce operation that generates summary information for all data.

An example application is the generation of execution logs from multiple sources in the cloud. In order to generate analytics, the application needs to pull the logs from the cloud on a regular basis and process them. Using the proposed service, a function will be triggered when a new log file is uploaded. It will run at the edge and calculate partial analytics over the uploaded data. Partial analytics from multiple sources may then be merged to produce system-wide results.

Examples, where no summary results are generated could involve distributed processing performing computations on individual pieces of data, such as:

- Stream-data processing for validation, encryption, addition of digital rights controls.
- Data transformations including thumbnail images, adding tags to media files, separating audio and video tracks etc.
- Reformatting diverse log data into a consistent form.

Other potential use cases include both cases where clients perform local computations and then aggregate the computed results to produce global results, such as:

- Log filtering (count event occurrences)
- Statistical analysis of metadata attributes (calculate summary statistics such as sum/max/min/average of numerical attribute values)

2.2 Current development

Public cloud infrastructures today include serverless or Lambda function options, and could be used to provide part of this functionality [30]. Serverless functions are becoming a popular notion, but the appropriate place to execute them remains somewhat uncertain. However, the long-term objective is to make the answer to that question immaterial. For this use case, the thin or thick edge could potentially be exploited for these types of operations. The key characteristic is the ability to simply request or mandate operations on the data path either inbound, outbound, or time based. The workflow engine currently available in Zenko could be adapted to trigger and manage serverless operations.

(a) Conflicting operations

In the current development any such operations, if performed at all, are performed centrally using batch methods, which limits conflicts, but also restricts the timeliness of the results. As the results of this work push the computations further towards the edge in a more distributed fashion risks of conflicts are bound to arise; this is discussed more fully in section 2.6.

(b) Invariants that exist in the application state

The creation of derivative objects whose existence is linked to the original object mandates that derivatives must be deleted together with the original object. A mechanism to establish and respect this dependent status should be implemented. Maintaining the dependency in object metadata appears to be the most reasonable approach to do this.

In the case that a derivate object or value is retained and the original object is destroyed, an alternative invariant is a causal relationship requiring the creation of all derivative objects before the deletion of the original object.

(c) Performance results/figures

Typical platform sizes are in the range of 500 terabytes to several petabytes of data, with update rates in the range of 1,000 to 20,000 updates per second. The availability of sufficient resources for real time updates is the expectation, but as discussed in the divergence section 2.7, the ability to maintain availability in the event of network partitions or resource constraints is highly desirable. Certain serverless functions may be trivially simple, while others, such as transcoding or compression, may be very computationally intensive.

(d) Persistence

Generally speaking, the initial data will be preserved and derivative data will also need to be persisted. It is assumed that this data will be persisted within the same storage model as the initial data. In its simplest form, derivative data will be persisted as additional metadata tags associated with the initial data. More sophisticated forms could include multiple additional copies of data stored in the same or parallel containers on the system.

(e) Security threats

A number of security issues may arise:

- If the serverless function is not local to the platform storing the data, the data must be protected in transit to and from the function's execution point.
- Derivative works should, by definition, be protected with the same rights as the original data, unless specifically changed. As an example of this, thumbnail previews of protected works might be made publicly readable to allow anonymous browsing for the purpose of discovery.
- In certain instances, the derivates or results of such functions might involve persisting or publishing the results to entirely different systems which may or may not have similar levels of security or data protection.

2.3 Detailed description

The application described in this use case is a new service, extending the Zenko Multi-Cloud Controller, which will enable cloud-based applications to define lambda functions, in order perform custom data transformations or data analysis at the path of ingestion.

The system will allow applications to publish custom lambda functions. A client application will be able to publish a function by uploading the function's code and specifying which event will trigger it. When an operation triggers a function, it will be executed at the edge, on the data involved with that operation. In the cases where the function's output is a value or data structure, local results will be aggregated to produce system-wide results.

(a) Architecture

Figure 2.1 depicts a potential architecture design for the proposed system, containing the different parts of the architecture and the data flow for a simple lambda function calculation. An architecture component is placed as an intermediate layer between the application and the storage system, receiving client writes and executing lambda functions when triggered. A second architecture component is used as an additional layer where local function results are aggregated. Depending on the client's computing capabilities, these components can be situated either in the client or in points of presence between the clients and the data center.

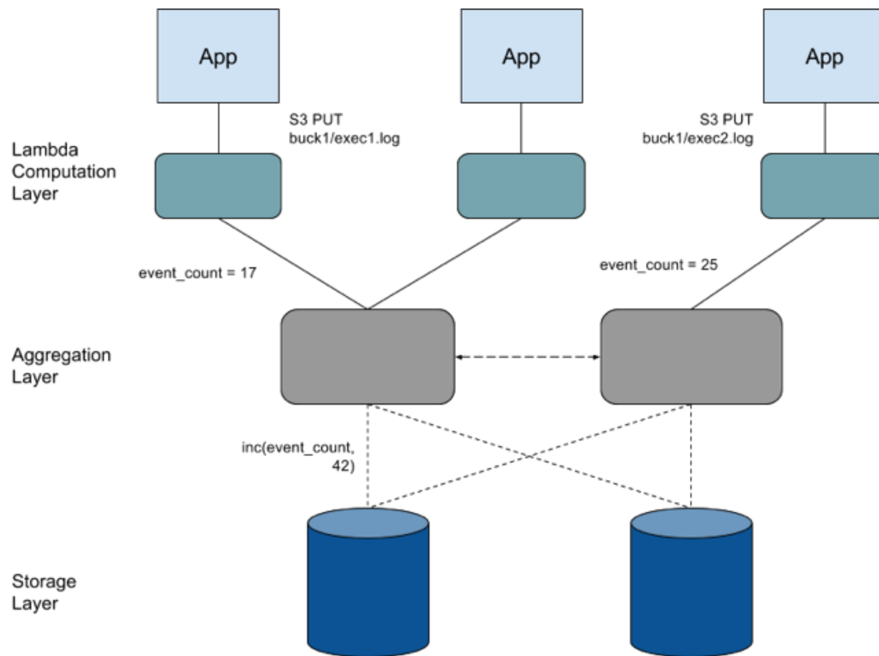

Figure 2.1: Basic Lambda Functionality.

(b) Edge computing requirement

The proposed system can benefit from the use of edge computing by:

- Distribution of compute load across many servers or "clouds";
- Lambda functions are conceptually designed to scale with compute requirements;
- Accommodating different systems, providing different types of functionalities, and allowing optimization of choices;
- Providing timely updates of data rather than more traditional batch-based methods.

Cloud environments, and especially those providing serverless function capabilities, might well be considered for this usage, and might constitute a component in certain situations. In many cases, unless the data is destined for a cloud environment, sending the data to a remote node may not be practically feasible, either for reasons of cost and bandwidth or data security. In certain cases, it would be pertinent to push the functionality all the way to the client at the edge.

2.4 Data model

The data model is the one described in 4.1. Data are immutable while metadata are mutable. Updating an existing object creates a new version.

There is no apparent need for updating more than one object atomically, as lambda calculations are triggered after writes to objects are executed.

2.5 Detailed description of the computations

The computations in this use case are given by the specific lambda functions defined by the system's clients. A common requirement for various functions may be the aggregation of local results.

However, there will be resource limits bounding each local invocation. These limits may include:

- Memory allocation range;
- Disk capacity usage; and
- Execution time.

Data-flow patterns in applications include writing data to the storage system. During an update, when a function is triggered, data is transferred to the intermediate layers, where the computation occurs. Finally, results for lambda function are stored in the storage system, and can later be retrieved by client applications.

2.6 Conflicting operations and invariants

An invariant of the system is that lambda functions should be invoked when the specified triggering conditions are true. Since these conditions are evaluated based on each update locally, concurrent updates will generally not violate this invariant. These operations would generally be idempotent in nature.

However, concurrent invocations of a lambda function may result to incorrect state at the function's results. An example would be performing cumulative statistical analysis over a collection of data where the same piece of data is written by different sources. Some conflicts could be avoided by the function's definition or in some cases, the results should mimic the approach applied to the initial data such as last-writer-wins or the use of versioning to avoid loss of information.

2.7 Divergence and divergence control

Ideally, lambda functions will be triggered by writes or reads of data and will happen in or near real time with no practical divergence. Most operations will not permit divergence that is not controlled; this is especially true of mathematical analysis of inbound data. Certainly, it is possible to allow the results of such calculations to diverge, but their validity is no longer assured. In this case, the importance of the information must be measured against the availability requirements of the system.

In the event of unavailability of lambda function resources, either due to resource constraints, or due to network partitions, the most desirable behaviour would be to remain available for writing data, if and only if it remains possible to queue and later execute lambda functions. At any point in time that data will be stored without guarantees that functional execution will occur, the system should be deemed unavailable for updates, or generate alarms if the system is configured to accept the loss of information as preferable to loss of availability.

2.8 Network partitions

Partitions can occur in the system, as lambda function calculations are performed in a distributed way and global results are calculated by aggregating local partial results. In case of a partition, the respectively partitioned nodes should be able to maintain a higher degree of consistency, aggregating local results. Upon reconnection, partitions should eventually propagate their partial results and converge.

2.9 Operational requirements

If public cloud serverless infrastructures are used, the compute capacity of those resources is assumed practically unbounded. The budget allocated to consume those resources may not be infinite, though, and as is the case for private infrastructure, the ability to reduce or stop the actions of the lambda functions to retain availability for reading or writing primary data is a highly desirable feature. The ability to control the point of execution of serverless functions is operationally interesting, allowing load to be distributed intelligently based on resource availability or cost.

2.10 Security requirements

Access control to data is managed by access control lists. The access control mechanism should be extended to include execution of lambda functions, since results can reveal information about the input data. Therefore, only data objects which the author of the lambda function has access to, should be included in the computations.

2.11 Data Protection requirements

Data protection generally remains in the realm of the end users who store data, implement the specific lambda functions and define access rights. Ownership of derivate data should respect rules associated with the original data so that lifecycle policies that might be applied to expunge original data also act on derivative data. The most complex elements might result from non-anonymized data stored within a particular set of assets that need deletion. This could include image data, personal data, or logs of transactions, such as web logs.

2.12 Implementation

Currently, Lambda functions are available on AWS, Google and Azure platforms. The management of simultaneous access and federation of multiple environments is currently not managed. The addition of the triggering of Lambda Functions at the edge would be a great benefit for the solution. The possibility of performing certain functions on the edge by clients should be considered. Containerized architectures that push these serverless functions to the edge client devices or to network components with compute resources are interesting options.

3 S3 local cache of central data

3.1 Overview of the use case

Scality's storage system can be deployed across multiple, geographically distributed data centers (DCs) in order to maintain data availability in case of DC or network failure. Moreover, geo-replicating data in several data centers across the world improves latency as client applications can connect to the DC closest to them.

Nevertheless, contacting the data center for each client operation incurs significant latency and requires potentially costly bandwidth to synchronize data that may never be used. Furthermore, cloud applications may become disconnected from any data center, due to network partitions, and still need to make progress.

An extension to Scality's storage system can be to enable local caching of data on local sites. Caching data locally can improve latency and availability, and significantly reduce the volume of data being transferred to the local site. Additionally, a very useful option is to allow temporary-offline operation.

This scenario presents a number of challenges:

- Providing consistency guarantees for client applications operating on locally cached data at a reasonable cost and at scale.
- Maintaining these guarantees when client-DC connections are interrupted, and possibly re-established with a different DC.
- Detecting cases where applications require access to large amounts of data, that do not fit in the cache, and thus local caching is not sufficient. In these cases, bypassing the cache and accessing the data center directly may be more efficient.

3.2 Current development

Scality currently has an open-source S3 server interface that functions locally and provides local indexing and local storage. This development would be a realistic starting point for the edge cache components. Protocol and data management functionalities are present, but no partial replication of data or indexes is currently supported in the available solution.

(a) Conflicting operations

The reading of existing data creates no conflicts, but any modification or creation of new data potentially creates conflicts both with the core platform and for other cache instances. If there are no partitions, the system can make progress, but with potentially unacceptable latency, especially in the event that multiple edge-cache instances are acting on the same data set (i.e. bucket in S3 parlance). In order to provide a correct outcome, eventually-consistent behaviour is expected to be required.

(b) Invariants that exist in the application state

The following invariants need to be maintained:

- Data indexes subscribed to locally must be consistent with central indexes.
- If the consistency invariant cannot be respected, it must be signalled. If liveliness is preferred to consistency operations can then be allowed to continue.
- Data that is created or modified in more than one local cache must not destroy data created elsewhere. A renaming or versioning policy should be preferred over a last-writer-wins policy.

(c) Performance results/figures

The typical central platform will contain something in the range of 500 terabytes to several petabytes of data, with billions to hundreds of billions of objects. A local platform would realistically be limited to less than 10 terabyte of data, with network bandwidth of no more than 1 gigabyte and latencies in the range of 10 to 200 milliseconds. The subscription to the full central dataset is probably generally unrealistic and a subscription of something in the range of 10 to 30% of the full data set is realistic in the remote office/branch office situation. In a small office usage the subscription would likely be limited to a single S3 bucket with millions of objects at most.

(d) Persistence

All data created locally must be persisted to the core platform in an eventually consistent fashion. The local store is expected to provide some level of data protection such as Raid or replication schemes. If the system operates in a partitioned mode where updates cannot be propagated transactionally, the possibility exists that data could be lost without in an untraceable fashion should the local site fail completely. Clearly establishing the guarantees of persistence in different degraded scenarios will need to be clearly documented and communicated.

(e) Security threats

Security policies that are applicable centrally for users should be synchronized to the remote (local) sites as well. Additionally, the creation of privileged users must only be possible using the centrally located system's rights management tools, to prevent rogue privileged users from accessing, deleting or modifying data locally that would in the end be propagated centrally. A useful feature would be a local infraction detection mechanism that would prevent central synchronization until the system can be secured. This would prevent a local breech or attack from being propagated globally.

(f) Current deployment details

The current technology is deployed in three different situations:

- The S3 server interface is deployed centrally on one site with a highly scalable backend storage platform. It is accessible locally and provides required functionalities centrally. It may be accessed from remote sites over WAN links using standard clients. No data is persisted locally in this situation.

- The S3 server interface is deployed in a distributed fashion across a limited number of sites. Generally, it will be deployed across two or three sites. In this case, there are two different synchronization options:

  1. An active-active synchronous replication between sites providing a fully consistent view across sites. For the case of interest here, this multi-site system can be considered as a single central instance. If the central system becomes partitioned, it follows a well-defined set of rules to avoid divergence.
  2. An active-passive asynchronous replication between sites providing an eventually-consistent view on the remote site after the synchronization has finished, but the replication can be of a subset of data. This model can include subsets of the data that synchronize in opposite directions, but for any given subset the replication is uni-directional.

- The open source S3 server component Zenko.io is deployed locally on a site, but today all indexes and data are stored locally. The Zenko framework allows for data to be stored on remote public object storage sites, but no metadata synchronization is performed and remote data is not cached locally today. Mechanisms are available in this model to asynchronously replicate data to a remote site, but as in the case above, the replication is strictly uni-directional.

3.3 Detailed description

The application described in this use case is an extension to Scality's storage system, implementing data caching at client machines. Client applications will read and write data objects and their metadata, using the AWS S3 API.

Read operations performed by an application should fetch data objects to the cache, so that subsequent operations can be executed there. Write operations should be executed in the cache and propagated to the data center asynchronously in the background. Every data object should eventually be stored and replicated at the data centers.

The system should also provide support for strongly consistent operations, enabling clients to enforce consistency with the data center when necessary.

The system may include a number of mechanisms for making the cache more efficient:

- Prefetching data objects to the client cache intelligently based on executed operations;

- Pushing data to the client cache based on search queries.

SwiftCloud [40], a cloud storage system that proposes an approach to access data replicas at client machines and cloud servers, can provide a starting point for the system described in this use case.

(a) Architecture

Figure 3.1 describes the architecture of this use case. The data storage system spans both client nodes and data center storage servers. The core of the system consists of a set of data storage servers that form data centers (DCs), and store replicas of the system's

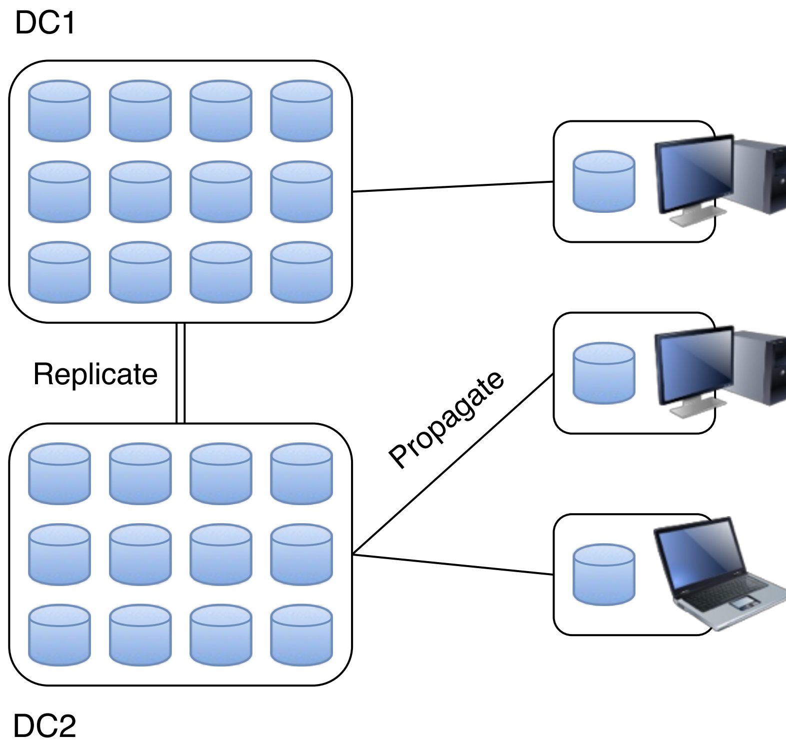

Figure 3.1: Basic System Architecture.

objects. At the edge, client nodes cache a subset of the objects, in order to locally execute reads and writes.

(b) Edge computing requirement

This implementation is a direct implementation of a heavy-edge design where the local servers (remote to the central platform) perform as much of the work as possible. This includes index lookups and maintenance as well as serving data that is stored locally. In addition, the other use cases described previously can complement this usage, with partial indexes being computed locally, and potentially providing local lambda function calculations if sufficient infrastructure is available locally. It is expected that tens or hundreds of such instances could be connected to the system. This scenario is especially pertinent for branch office solution where the central data store has much more data than can be stored locally. This is distinct from a small office, home office (SOHO) gateway solution that replicates all its data to a central platform for the purpose of securing the data in the event of a failure. The current Zenko open-source solution responds, at least functionally, to those requirements.

3.4 Data model

The data model is the one described in UC1. Users read and update objects and their metadata through S3 PUT/GET operations. Data are immutable, and updating an object create a new version, while metadata such as user-defined tags or ACLs are mutable.

3.5 Detailed description of the computations

This use case concentrates on distribution and storage of data locally at the edge, rather than computations at the edge. At the same time, computations performed locally at the clients will be required to manage object metadata and maintain listings of objects in each bucket. The generation of partial indexes at the edge should be both realistically attainable and desired. More sophisticated lambda functionality at the edge or deep-learning inference-based operations, can certainly be considered based on the computational resources available at the edge.

3.6 Conflicting operations and invariants

A number of conflicts are possible for such a system, for instance:

- Writers on distant sites can create the same object simultaneously;
- A writer creates an object locally while disconnected; however, the object was created remotely during the partition;
- A reader attempts to read an object in the local index that has already been deleted centrally;
- A reader retrieves a version that is different from the one advertised.

The extent to which these conflicts can occur will depend on the extent to which full consistency is respected, at expense of affecting availability. Ideally, the consistency should be tuneable, but with the invariant that no new data creation destroys existing data. This can be assured by allowing versioning and renaming of data.

3.7 Divergence and divergence control

The data stored in the local cache will naturally lack data from the central system, since the local cache cannot support all of the central storage. The indexes providing lists and versions of all existing data are expected to be consistent with the central system, but must support some level of divergence in the event of a network partition, if liveliness is desired. The indexes would ideally remain synchronized with a time lag that is not more than one order of magnitude more than the network's round-trip latency. In the event of a partition or other event that potentially causes divergence, information providing a clear indication of the level of inconsistency would be very important. Pertinent metrics could include:

- Fraction of the data in % that is not synchronized;
- The time since the last synchronization;
- The number of objects not yet included in the indexes.

When partitions are resolved the return to a consistent state would include bi-directional updates and resolution of any conflicts using the established rules such as renaming, creation of versions or last writer wins strategies.

3.8 Network partitions

Generally speaking, the most likely partitions is expected to occur between the central site or sites and the more numerous remote sites. The central sites frequently have multiple redundant links connecting them. Since the central site is considered to be authoritative, a fully consistent view is only expected to be available when there is no partition with the central site. In the event of a partition with the central site, it is important that local sites can continue to make progress writing data and reading any data they have locally. A potentially interesting extension would be the ability to interact between multiple local sites in order to obtain data that they have cached, or to repopulate an out-of-date index in the event of a partition.

3.9 Operational requirements

The centrally located components are deployed as described, either in a consistent fashion or using active-passive replication; eventually-consistent synchronization is not currently supported on the central system, but it remains an interesting alternative to the methods permitted today. The remote clients can potentially be little more than a Docker container instance with a limited amount of storage, as little as a few tens or hundreds of gigabytes. They can be numerous, as in the case of branch office installations, where there might be up to thousands of instances. The volumes of data stored centrally can reach billions or hundreds of billions of objects on multi-petabyte platforms. In the case of a very large multi-billion object platform, the ability to support indexes of partial subsets of the full dataset, a single bucket or container or collection of them would be the most reasonable subset to support, so that remote instances need not carry indexes of multiple billions of objects. Even a single bucket can contain around a billion objects, but more common numbers for buckets are millions of objects.

3.10 Security requirements

The access control mechanism should be extended for data residing in local caches. ACL propagation may be an issue. Clients should be able to use strongly consistent operations in order to enforce changes to the ACLs to invalidate ACLs in other clients caches. Alternative more weakly consistent methods for ACL propagation are under study and could prove useful in this context.

3.11 Data protection requirements

Because the system is generally operated in the context of a public or private cloud-storage-as-a-service offering, the management of individual data protection primarily falls on the users themselves. The end users of the system have the ability to change access controls for the data in such a way that it can be private or entirely public, the system is accountable for protecting the data as configured by the users. It is however important to be able to distinguish ownership for data in such a way that the data can be explicitly deleted or secured if required. Tagging of data by its origin country or other pertinent tags having a bearing on data protection should be possible. In the specific case of partial replicas on remote systems, it could become possible to require partial indexing

of data based on geographical criteria, should constraints of data sovereignty be required. Requests for such controls have been enumerated by customers and the functionality would permit a definitive response should this be required on a platform.

3.12 Implementation

The most realistic option for the implementation of the protocol component of this functionality is to build on the open source components provided by Scality's Zenko platform. The server is available in the form of Docker containers or in source code form on github S3 server. Interesting development work already underway in the SyncFree SwiftCloud project, which provides many of the required eventually consistent synchronization principles mentioned here.

Chapter 5

Stritzinger

1 No-Stop RFID

1.1 Overview of the use case

A transport system in industrial manufacturing is any means of handling transport of materials and partial products between processing stations. These consist of all kinds of conveyor belts, turn tables, self-driving carts on rails as well as free-driving, and human pushed carts. Common among them is some means of more or less flexible movement of a workpiece carrier and some way to identify the workpiece and keep information about necessary processing steps, progress and measurement data, together with other logistics data. Mostly, a form of read-only or writable RFID tag is used for this. The RFID tag can be mounted on the workpiece carrier or on the product itself [22].

While the product is traveling along the transport system from processing station 1.1 to processing station, data associated with the workpiece carrier is used to decide what processing step needs to be taken next and to route the product towards a station that can do that step. When processing is done, this data is updated accordingly, e.g. steps done, quality test result and calibration information 1.1.

RFID tags are used to at least associate the data with a product in the making and often also to store the information itself. In the case where this data is stored on the RFID tag itself, the time needed to read and write these RFID tags over and over again is a substantial overhead in manufacturing that adds to the cost.

The No-Stop RFID system we are planning to build will implement a distributed cache of the RFID contents. By doing this we can avoid this time loss. Ideally, we can achieve all RFID communication with the tag in drive by mode without stopping.

The processing stations, transport system, and decision making systems will communicate only with this distributed cache instead of directly with the RFID tag, and can thus observe much reduced latency. Consequently, the RFID tag itself is only used as token to detect physical location of the workpiece carrier and its non-volatile storage acts as backup when in sync with the cache.

The distributed cache will be integrated in the embedded systems that are mounted along the transport system and contain the RFID antenna needed to communicate with the RFID tags over short distances (for HF tags 5-30mm). These embedded systems are networked with Ethernet connections along the transport system forming a mesh network 1.3.

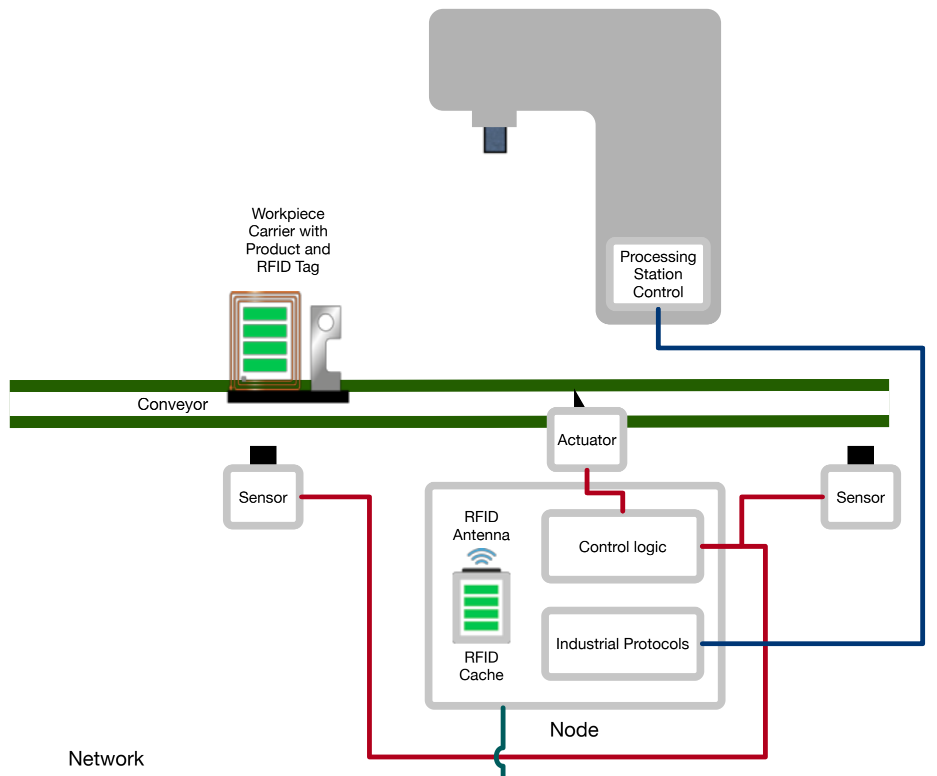

Figure 1.1: Node at Processing Station

Routing decision making will be done on these embedded systems directly. Processing stations are connected to a local embedded node, and communicate only with the local cache. Updates to the cache, either from reading from the tag or from writes from processing stations, are to be distributed with the necessary metadata over a gossip protocol (communicating only with neighbours) to all nodes in the network. Whenever a RFID tag is in communication distance with an antenna, data in the local cache is synced in both directions with data on the tag.

Since we have a large network of embedded devices and no server or cloud components at all this is a light edge use case.

The challenge is to keep eventual consistency between all the caches and the RFID tag contents. There are strict requirements to always detect when a cache is not yet consistent when a RFID tag is located at an antenna. Causal ordering needs to be provided in case of conflicting updates, and failure to do so needs to be always detected and flagged as an error. While the network is mainly reliable because it is mostly wired, we need to deal with hardware failures nonetheless, that can cause network partitions. Assembly lines are often maintained and changed while parts of them need to keep working. This results in partial network outages and change of topology, which needs to be dealt with.

The current RFID systems work as isolated systems. There is a short-term non-distributed cache to make data accessible during glitches in radio accessibility. It also speeds up re-reads of the same data, and supports a pre-fetch operation that fills the cache, when no other operations are done. A configurable short while after the RFID tag

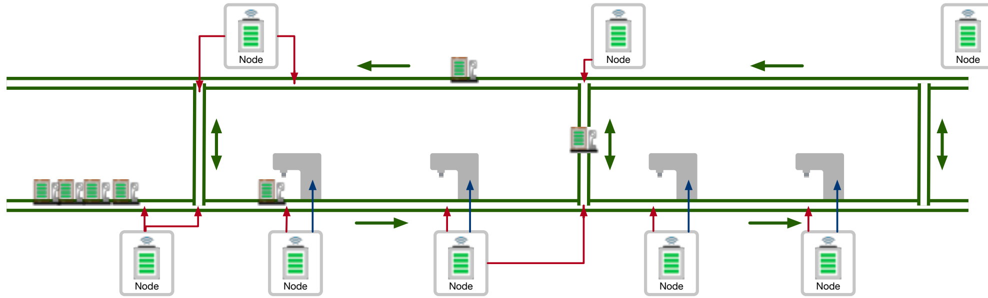

Figure 1.2: Part of a assembly line

is not accessible, the cache gets deleted, since the tag could have been at another reader in the meantime.

Other application aspects are distributed over several nodes (e.g. IEC61499 distributed PLC subsystem for user programmability). For the other distributed aspects, we use Erlang's transparent distribution protocol. There are two test installations on small industrial transport systems (small meaning dimensions of $< 10m$) with about 15 nodes each.

We have a current hardware prototype RFID system that runs Erlang directly on a small embedded device, which is being tested in an industrial environment at Boschrexroth. These embedded devices use the software stack of GRiSP (www.grisp.org), running Erlang very close to the hardware on a hard real-time unikernel system.

Development of the new system has not begun yet, since Stritzinger has only been a partner in LightKone for 2 weeks.

(a) Conflicting operations

In the non-distributed system usually no conflicts occur, since physical presence of a RFID is required at one of the antennas (associated with a processing station) during a complete transaction.

(b) Invariants that exist in the application state

The non-distributed cache we now have is only valid, while the RFID tag is present in front of the antenna. It only exists to speed up duplicate reads or read/modify/write operations of blocks of data.

Valid data in the cache needs to be identical with the data on the RFID tag while it is present at the antenna. Writes are immediately validated in sync with the operation. Reads of data not present in the cache result in immediate read from the RFID tag.

(c) Performance results/figures

The current, unoptimized system can transfer about 200Byte/s to and from the RFID tag. Currently we are working on improving this number to make the proposed distributed system work better, and we plan to reach data transmission rates in the order of 1KiB/s. When data is in the cache already, latency towards the application is so low it does not matter anymore.

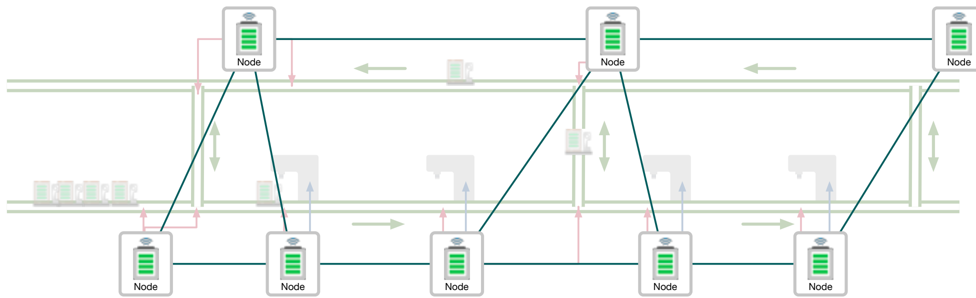

Figure 1.3: Network along a assembly line

The usefulness of a non-distributed cache is very limited and was also meant as a first stepping stone towards a distributed cache.

(d) Persistence

In the current system data in the cache is only kept for a maximum of a few seconds after a tag physically leaves the antenna to avoid the possibility that in the meantime a tag has been written at another antenna. Since the caches do not communicate with each other in the non-distributed case, it would result in incorrect data in the cache.

The only long-term persistence is on the RFID tag.

(e) Security threats

The environment is currently considered to be shielded from threats. However, in practice this is no longer true, but the general level of security awareness in manufacturing has been low in the past. Eventually moving towards an encrypted and authenticated communication between nodes will improve shielding of all application areas.

(f) Current deployment details

The RFID readers are small embedded systems, mounted alongside the industrial transport system. The physical environment can be harsh industrial conditions. These embedded systems are the nodes in our network.

Currently, we have access to two demo transport systems with 15 nodes each at Boschrexroth and Zema) developed in the SmartF-IT research project.[13] These are considered small demo systems. However, the capacity for very large factories are up to 1,000 nodes. Each node is capable of about 200 MIPS and has 64 MB RAM. Communication between machines is currently a mesh network of 100 Mbit Ethernet. Every node has three network connectors, linking it to neighbouring nodes. The network graph has a large diameter requiring many hops. In many cases, network connections go in parallel with physical transport system (e.g. the conveyor belt).

1.2 Detailed description

From our experience we know that there are two main schools regarding the use of RFID tags in this setting: one only uses a unique ID on the tag and stores everything in a

database. The other keeps all relevant data on the tag itself, and updates it as it goes through the processing steps. Both approaches have advantages and disadvantages.

The decentral approach, that keeps all data on the RFID tags, can work without a network. It is the older approach, since until a decade ago there was no or very limited networking of processing station along a conveyor. Since all data transfer is done locally, it is very robust against outages. However, read/write speed of RFID systems are very slow, resulting in the need to stop in front of the antennas and wait until the data transfer is done.

The centralised approach, that keeps all data in a central database, can speed up read/write operations given sufficient network and database performance. The price for this is a single points of failure (the database server) for costly outages of the whole factory.

We seek a middle ground between these two approaches by developing a distributed cache of the contents of all RFIDs deployed on an assembly line. This avoids costly waiting time of workpiece carriers in front of antennas along the transport system.

By distributing the cache, we can read and write data for a RFID tag whenever it drives by or is stopped at any antenna in the system. Writing can be done to the cache also when a tag is no longer at a local antenna, e.g. when it's leaving or traveling between processing stations. Before a write can be done, the writer needs to obtain a handle which needs to stay active until all the writes are done, and afterwards needs to be closed. This will be presented to the users like a file operation with open/read/write/close semantic which can also be seen as a transaction. When the tag is present at another station where the written data is needed, the distributed cache needs to ensure that the data in the cache is consistent with the last closed transaction on the tag. To achieve this, a transaction id needs to be reliably written to the tag during or after each transaction.

When a local cache stays behind the RFID tag transaction id for a longer time, an error condition is signalled to the operators (e.g. red or yellow lamp) and manual intervention is required. Since normally the data along the network should travel much faster than the physical workpiece carriers, this should only happen when a carrier is transported across a network partition border or during system failures.

Every time a RFID tag passes a reader, as much data as possible is written or read without stopping to closer match the RFID tag and the cache. Some mechanism needs to be developed to quickly determine what transactions were successfully finished on the tag, as it can leave the radio field with partial writes of an unknown status.

Smaller sized database nodes can be optionally added in a distributed way, to give increased persistence of data which is not on the RFID tag in case of power outages.

(a) Conflicting operations

Reads from the cache need to make sure to always return the most recent writes. Transactions group related reads and writes to be executed atomically. Processing stations access the copies of the cache close to the nodes they are connected to. Transactions need to be applied in causal ordering and cannot be rolled back.

In the non-distributed system, conflicts are avoided since physical presence is required at one of the antennas associated with a processing station during a whole transaction.

This model of operation needs to be emulated by the distributed cache. The limitation

is loosened in the way that the RFID tag needs only be shortly present at an antenna associated with a processing station to identify it and possibly write a transaction id if it is to be written.

In normal operation, a causal ordering between transactions at different stations is ensured by physical presence at the station. Based on an error in the processing station applications, there could be conflicting writes with no detectable causality. It is sufficient to flag these errors when detected. If possible, the conflicting writes should be prevented right away, signalling an error to at least one conflicting writer. If a conflict can only be detected after the fact, the workpiece carrier should flag an error, and be stopped as soon as arriving at a station where it is possible.

It is very important that these inconsistencies are detected in all cases, since progressing in the face of inconsistent data can result in broken products or even in physical damage to the machinery!

Another conflict might arise between RFID tag transaction id and cache transaction id when at an antenna. When the cache is ahead in transactions compared to the RFID tag, the new transaction id needs to be reliably written to the tag prior to any other operation. When the cache is behind in transactions compared to the RFID tag, any operation needs to be delayed until the cache has caught up. After a timeout, an error needs to be indicated at the station, since this conflict cannot be resolved until either the cause for the cache being behind is fixed (e.g. network partition) or the problematic palette needs to be taken away and manually inspected like in the case of production failures detected. In almost all factories there are repair stations for this. Instead of manually taking away the workpiece carrier, it can also be transported towards a repair station, while this inconsistency exists.

Automatic error flagging, which results in skipping all processing stations and driving towards a repair station, could also result in the tag being consistent again when at another antenna. Then the problem with the tag can be considered fixed and the error condition cleared. This is very robust to partial network failures, since observed behaviours would be that workpiece carriers just drive through a station skipping it when it is disconnected from the network. It is even possible, that the RFID tag is written fully at another station, and being fully in sync with the cache, which should be detectable by having a special flag in the tag memory, which is set every time the tag is fully synced. Then a partitioned station could read the whole tag memory and still operate consistently, only with degraded performance. Through this mechanism, cache information can travel from one network partition to another.

When the system is aware of the physical transport topology, this can be improved by stopping and writing tags before they leave network partitions, avoiding this kind of error condition altogether. Knowledge of transport topology needs to be optional though, since it cannot always be provided. Even when topology is known, workers can move palettes from one network partition to another.

(b) Invariants and other rules that govern the system

- RFIDs have an id that is globally unique, which is identical to the id for its distributed cache.
- Caches for different RFID tags (different Id) do not share any data with each other.
- Transaction ids for a single RFID are linearly ordered.

- Transactions associated with an id are either open or closed.
- Open transactions are bound to one cache instance and cannot be migrated to work on their caches (all transaction operations open, read, write and close work only against a single cache instance)
- For a RFID tag that is physically present at a reader, when the transaction id on the tag is the same as the one in the local cache, the following holds:
  - any cache byte that is marked as ”valid” must have the identical values on cache and RFID tag
  - any cache byte that is not marked ”invalid” (i.e. ”valid”, ”dirty” or ”flushed”) must have the value of the last write in the transaction id.
- A new transaction id can only be obtained, when the old transaction is closed and this is known to the local cache.
- A new open transaction id can only be obtained when the RFID tag is physically present and the cache transaction id is either the same or higher (by the ordering relation) than the one on the RFID tag. The cache can be ahead of the tag, but not the other way around.
- The new transaction id is higher by the ordering relation than the old transaction id in the cache.
- Writing requires to hold an open transaction id bound to the local cache.
- Reading can be done anytime, but the values might not reflect the last write. To ensure reads to read the last write, an open transaction id needs to be obtained also. If a transaction id is opened only for reading, it can stay the same value and need not be a higher one.
- To avoid undetectable data loss, new transaction ids need to be written safely to the RFID tag. This makes it necessary that a new transaction id can only be obtained while the tag is in communication distance of a local antenna (”present” state).
- So far, no known pre-existing cache coherence protocol was selected. Due to the special nature of a cache with moving ”memory” (or a database with moving ”disk”) a new protocol might be needed. The cache data including all metadata needs to be distributed to all other nodes, since it cannot always be predicted where the RFID tag will show up next.
- If a transaction cannot be obtained for any of the reasons above, it will wait with an optional timeout, until all necessary conditions are true.
- There is also an operation to open a read or read/write transaction for the next RFID tag that is detected at the antenna. Such an operation would return the RFID id and an open transaction id. This allows transaction start on drive by.

(c) Expected performance

The only performance that matters here is factory throughput.

Data in all caches should be able to catch up with the physical tag location almost always. If not, the tag has to wait before the next processing station until the data caught up with it. This reduces factory throughput and therefore waiting needs to be minimized, best eliminated.

Minimum time between processing stations depends on transport speed and distance between stations and is around 1 to 10 seconds. Conveyor speed for conventional systems gets up to 20m/min, standard is 12m/min.

The system needs to scale to a size of up to 1,000 nodes without degrading performance.

(d) Persistence

All cache and metadata needs to stay on all nodes indefinitely.

If a workpiece carrier is removed permanently from the factory, it should be booked out of the system. Otherwise caches that contain only bytes that are either invalid or valid could be removed after a while without data loss.

Caches which contain dirty and flushed data need to be held on to for a long time, since it is not uncommon that workpiece carriers are taken off the system in the case of a throughput problem.

Recommended practice will be to use special exit readers, which make sure all data is flushed and validated on the RFID tag before removing them. The same goes, if palettes can regularly pass out of the cached system into a part of the assembly lines which do not have caches.

Because normally caches would be flushed and validated when workpiece carriers are removed, the systems need to hold on to any caches that are dirty or only flushed, since it needs to make sure no data is lost.

There will also be a procedure to manually remove RFID tags from the caches in case a RFID chip breaks. Knowing the time how long the RFID tag has not been seen by any antenna is an important information to find a tag to remove, so it should be kept if possible.

For additional protection against power outages, nonvolatile storage can be added to the system (e.g. a small server running specialized software or simply a NAS connected to one of the nodes). This can be just log storage without quick access, since it will be used mainly for backup in case of whole factory power outages.

To shut down an assembly line in a controlled way, the caches should have a special shutdown mode which flushes all caches whenever possible (stopping workpiece carriers where it can). It needs to give a signal whenever all data is persisted nonvolatile on RFID tags.

(e) Architecture

The RFID reader nodes that contain the caches are networked in mesh topology. Network diameter is large and the topology partly mirrors the physical transport topology. The reader nodes are small embedded systems all alike. Since assembly lines can get quite long, this can result in a high diameter network. Each node can be connected to up

to three neighbouring nodes. We will implement a link state routing protocol allowing shortest path routing, but want to limit the cache synchronisation to nearest neighbour communication.

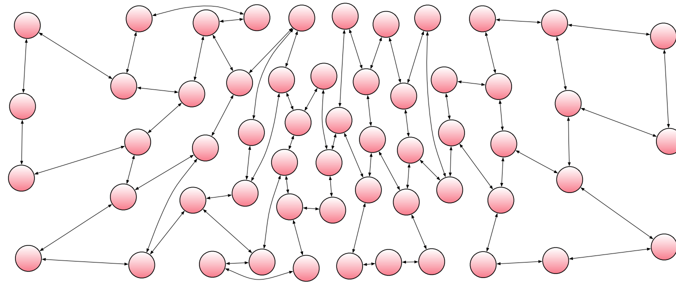

Figure 1.4: Network topology example

For other applications we plan to have shortest path routing between the nodes using IS-IS [27] link state routing protocol, which has some internal similarity with the gossip protocols discussed in LightKone.

For the RFID caches however, we want to limit ourselves to nearest neighbour transmission of the cache operation, to keep network traffic and routing effort down. We do not expect many random outages (this would impair other system functionality greatly and therefore always needs to be avoided). What can happen is the occasional failure of the node hardware, which will be replaced soon.

What will happen regularly is that different sized subsets of the nodes are taken down for maintenance or factory rebuilding. It is important that the rest of the assembly line keeps working as expected, while parts of it are changed or being taken down.

(f) Edge computing requirement

Our edge nodes make local decisions about routing the workpiece carriers to one of the next processing steps. These decisions are based on the data associated with the RFID tag, sensor input and communication with a processing station. This decentralised decision making has been the traditional way factories were and are still built. The decisions need to be made in real time with hard real-time constraints, and therefore need to be made locally. It also reduces wiring cost, when sensors, actuators, processing stations, and RFID antennas are only wired to a local node. In addition, there are no single points of failure for the whole assembly line this way. Therefore, we need to have the computation at the edge.

In manufacturing RFID systems, there are two major approaches: keeping data centralized on a server using only the unique id of the tags to get data; or always keep all necessary data on the RFID tag and be completely independent of the server.

The centralized approach has the disadvantage of a single point of failure for the whole factory for the server and the network infrastructure between the assembly line and the server. Also, latency occurs when getting the data from the server to the processing station and back. Reliable writes need to be done synchronously on the server, which increases latency. Also, the server is an added cost factor which can be noticeable when a reliably database server cluster needs to be purchased and maintained.

The traditional decentralized way keeps all data on the RFID tags, making a server for this data unnecessary. Since every workpiece carrier has the essential manufacturing information on its attached RFID tag, operation is independent of any central infrastructure. Unfortunately, latency does also occur due to the slow read/write operations to the RFID which need to be done in sync. It can be more cost effective than the centralized since it does not need all the server infrastructure and is quite resilient to outages.

By adding a distributed cache, we can reduce the latency greatly compared to both the centralized solution and the traditional decentralized solution. Every node has in the ideal case all the data already available once it has ever been read or written from the RFID tag at another station. At the same time, we plan to remain as robust as the traditional decentralized way, and have no single point of failure.

1.3 Data model

The data model, that forms the basis of all functionality, is just arbitrary mutable addressed bytes on the RFID tag. Addresses are an integer offset starting at 0 and going up to the size of the RFID tag minus 1. For each byte, some state metadata is kept for helping the cache loading and flushing 1.6.

It is essential, that the distributed cache works reliably with the basic data model of bytes, since some users will want to implement their own data structures on top if it.

The data is generated and updated by processing stations which are communicating with their local edge node. It is fully replicated together with the metadata. The updates to the data need to preserve causal ordering. We can work with eventual consistency, but we need to always detect, if a local cache is not yet consistent when a RFID is present at a antenna. If it's not consistent yet, the workpiece carrier needs to wait at the antenna or the processing station until it is.

We will need to provide transactions that provide atomicity, consistency (eventual) and isolation. Whenever a RFID tag is in reach of a antenna and if the local cache has caught up with all updates, we try to update the data on the RFID. For all RFID tags that are fully in sync with their distributed cache we get durability also.

Cache data is flooded to all participating nodes. Since the RFID tags are on the palettes which get reused, expiry needs to be very slow (workpiece carriers removed permanently from the system without telling the system).

In addition to the "just bytes" data model, we can provide higher level structures as a convenience library, which is useful mainly if we can provide better properties for these higher level data structures.

Possibly useful data structures on the RFID:

Data stored on RFID

- Key/Value mapping with the keys are fixed for a whole production run (after which the RFID tag is erased to be reused) being distributable as meta-data to the caches and only the values are stored at offsets and sizes and types that can be part of the meta-data. The other data structures in this list could be structured types stored at these keys.

- Push-only stack of fixed maximum size. Read operations should be provided at offsets relative to the top of the stack.

  - Generalization: also allow mutating writes relative to the top of the stack.
  - Generalization: also allow popping elements off the stack again.

- Processing steps necessary for a product with completion information 1.5

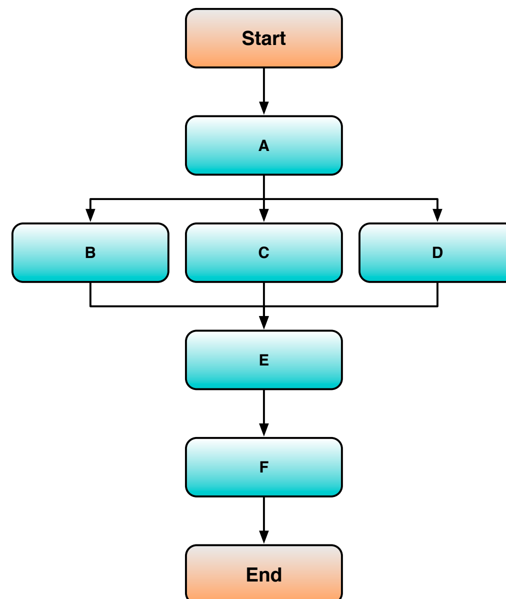

Figure 1.5: Example partial ordering of processing steps

Processing steps could be seen as a partial ordering of available steps out of a set of available steps for a factory. Not all steps need to be taken in order in production giving some flexibility in machine utilization.

Which steps are completed also needs to be stored on the tags can be seen as a portion of the partial order which is limited to the ordering of the processing steps allowed.

1.4 Detailed description of the computations

Computation with the data associated with a RFID tag is programmable by the users. These computations take the data in the cache, sensors in the physical area as inputs, communicate with local processing stations if there (input and output) and make decisions resulting in outputs to actuators that influence the transport system. These computations have hard or soft real-time requirements.

Besides these user defined arbitrary computation, there is the state model of the caches as shown in 1.6:

- invalid: only bytes that have never read or written for a certain cache are invalid.

- valid: only bytes in the cache that are guaranteed to be equal to their RFID counterpart are valid. They get only valid by reading their value from the RFID tag either a first time read or a validation read after a write.

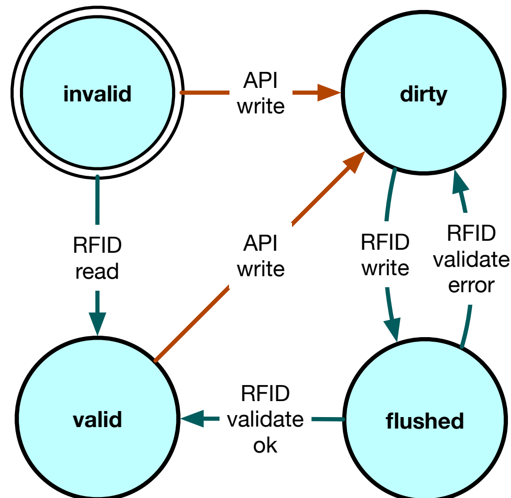

Figure 1.6: Cache state transitions

- dirty: bytes written through the API by some computation are marked as dirty. Valid bytes that are unchanged can remain valid. Bytes that are dirty need to get flushed to the RFID tag as soon as possible. However, most RFID tags can only be read and written in blocks so we might need to read a block to make sure none of the bytes are invalid before writing them back to the RFID tag.

- flushed: Bytes that we have tried to write to the RFID tag. We cannot be sure that blocks written to the RFID tag made it into non-volatile memory there (it could have lost power after receiving a block but before saving it by leaving the antenna field which powers it over the air). Therefore, we read bytes that are marked flushed back from the RFID tag and validate if they are identical with the cache contents. If they are identical they are marked valid if not they go back to dirty to trigger another write attempt later.

For the partially ordered processing steps we need to provide the following functionality:

- Encode the partial order in a compact way and write it to the cache. This is usually done at a starting point in a assembly line when available empty workpiece carriers get associated with a new product to build. All processing steps are marked as not done yet.

- Get the set of possible next processing steps taking in regard the set of processing steps ”done” and the partial order. There is always at least one step available for an active product in making (after the last step the workpiece carrier is empty again and can be reused).

- Mark a processing step as “done”.

1.5 Conflicting operations

In normal operation, conflicting operations can be avoided by causal ordering of transactions. This causal ordering can be ensured by only starting a transaction when the RFID

is in range of the local antenna. When the transactions are all finished (closed) in timely order, and the cache updates propagate fast enough, conflicting operations on the data in the cache can be avoided. However, too long running (or hung) transactions as well as network partitions can result in conflicting operations, that can not be causally ordered anymore.

The cases where there are confliction operations that lead to a incorrect state must be very rare since they need manual resolution by a human, if they occur. There are several mechanisms planned to prevent such incorrect state, therefore multiple problems need to occur at the same time to result in unresolvable incorrect states.

Most important is that all conflicting operations are detected and flagged as error.

1.6 Divergence and divergence control

There are two kinds of divergence possibility in the system:

- Local cache transaction id is behind the transaction id on the RFID tag

  The workpiece carrier needs to wait until the local cache caught up. If the wait time is too long (configurable) there are several possibilities to resolve the problem

  - Error signals are triggered locally, calling a human to attention to remove the blocking workpice carrier and resolve the problem at a repair station by inspection of the partial product.
  - An error is flagged on the RFID tag causing it to travel to the next repair station by itself.
  - The workpiece carrier is sent on its way without processing, and travels on the assembly line until the problem fixes itself by reaching a cache that is up to date, e.g. in another network partion. Care needs to be taken that workpice carriers are not traveling forever without resolving the problem by e.g. a error counter on the RFID tag with a threshold that gets reset every time when some progress is made.

- Data content on the RFID tag is not in sync with the cache when crossing system or network partition borders.

  An assembly line can be partially equiped with No-Stop RFID system and partially have traditional RFID systems (e.g. during a upgrade of the factory or it's only worthwhile to upgrade part of the factory). In this case the antennas that are at places where tags can leave the distributed cache areas need to be configured specialy and be able to stop the workpiece carrier, and make sure all the data is flushed and validated to the RFID tag.

  The same can be done in the case of network partitions, but only if the physical and network topology and ther relationship of the two are known. Smart transport systems, which are automatically routing workpiece carriers, have this knowledge. When we are able to write all cache contents to the RFID when leaving a network partition, we can effectively bridge the air gap between the partitioned systems.

When the distributed cache is offline, all the data always need to be written to the RFID and the system degrades to the old non distributed system but is still able to make progress producing products.

1.7 Network partitions

When there are network partitions, it is a requirement that all cached data is written back to the RFID when the tag leaves the physical area of the partition.

If this cannot be achieved, divergence of cache content needs to always be detected at the RFID location. After an application specific timeout waiting for convergence, this can be flagged as an error, which needs human intervention.

People working on the transport system might not be aware of a network partition and pick up palettes from one partition and insert them in another partition, when their content is not yet consistent with the cache. Normally, there are stations where palettes can be removed safely (writing all data to the RFID tag before). But especially when the system is not in optimal operating state and palettes back up, human error while trying to fix the problem can lead to such problems.

1.8 Operational requirements

The node characteristics can be seen in table 1.1. We need to be able to store all RFID caches on each node. With up to 8kiB RFID memory and up to 2,500 RFID tags per assembly line active, we have a 20 MB raw storage need. It would be good if the whole cache would fit in 50MB of RAM including metadata and distribution housekeeping.

Data transmission to/from RFID tags will be in the order of 1KiB/s (current system is much slower with about 200Byte/s but this is unoptimized).

Table 1.1: Node Characteristics

Scale	Large Assembly line up to 1,000 nodes
CPU	200 MHz PowerPC
Firmware	running GRiSP (Erlang on RTEMS)
RAM Size	256MiB
”Disk”	NAND Flash on PCB with filesystem (can’t write often)
Network	100 Mbit Ethernet, 3 links

The tags used by the customer are conforming to [25], [23] and [24] and have FRAM memory (with 10^{12} times read/write endurance) to allow the high number of write cycles required by the application domain [20]. Example of a RFID tag used in the system is MB89R112 [19]. Characteristics can be found in 1.2.

Table 1.2: RFID Properties

Storage	2..8 KiB per Tag
Block Size	16 or 32 Bytes
I/O Speed	drive by read+write+verify 25..150 Bytes
Write guarantee	Only when read back
Active tag count	500 .. 2,500 per assembly line

1.9 Data protection requirements

None of the data stored in the system are related to persons. Therefore, we expect no data privacy issues.

1.10 Implementation

We plan to implement the cache in the programming language Erlang and build on top of the GRiSP software system (running Erlang close to the hardware using RTEMS embedded operating system) which we already used for the predecessor system.

For other functionality, we use the transparent Erlang distribution protocol and plan to use it for the communication needed for the distributed cache too. As soon as the extensions we are working on in WP3 are available for the distribution protocol, we want to build on top of these extensions.

Possible usage of Lasp or Antidote (or what will evolve from it in the project) will be researched. We will build on top of any useful Erlang libraries or applications that are developed in LightKone.

2 Smart metering gateways

2.1 Overview of the use case

Digitization of utility metering is a growing market that promises to bring cost savings and provides important data for efficient resource usage, and are an important stepping stone towards smart grids [12]. Metering has always been an important component of electrical grids, historically used for billing purposes, and it's increasingly being adopted for water grids, hot water grids, gas grids, etc. Therefore, smart grids, digitally enabled, self- balancing electrical grids, are becoming an important building block for efficient renewable energy usage.

Smart sensors with a fine granularity is an important data source within smart utility metering. The higher quality and finer granularity, the better decisions can be made which impact the bottom line, cost effectiveness and environmental impact. The easiest way to reach the metering devices is sub gigahertz radio protocols, because of their range and them being license free frequencies[33]. In Europe, Meter-Bus (M-Bus)[1] is the standardized protocol that is supported by many metering device manufacturers, such as Siemens and Deutsche Telekom[1]. The protocol was initially designed for drive-by or walk-by solutions to read meters in a house from the street. Due to its limitations and costs, a wireless extension, Wireless M-Bus (WM-Bus) [2], has been standardized to support smarter ways for metering data collection through wireless gateways. Our customers are increasingly demanding gateways which are connected to mobile and wired Internet. This allows a much finer grained continuous data collection needed for the new big data applications.

The workflow in this use case is simple (see Figure 2.1): digital meters are placed throughout properties, measuring physical processes such as electricity or water consumption. The meters then transmit data by wireless to gateways which collect and aggregate the data to be transmitted into the cloud (for further processing and presentation).

The gateways are small embedded devices that are distributed evenly over residential and industrial areas and are connected via mobile or wired Internet. They can also communicate with each other via WM-Bus or other radio protocols. Gateways are in charge

[1]The OMS Group lists several other prominent companies: http://oms-group.org/en/about-oms/members/

of the main computations and operations on data, such as data thinning by aggregation to save bandwidth, and redundant data transmission routing (between the gateways and towards the server) through mobile and wired Internet connections for reliability. Gateways can also take part (along with the cloud) in decision making for smart grid applications, e.g. switching on batteries, controlling electric car charging and other flexible energy sinks. Consequently, this use case can span the whole spectrum from light edge to heavy edge, with more emphasis on light edge, since future applications can involve smart grid decision making right at the edge without server involvement.

Data loss and duplication are two main challenges in the use case. While it is sufficient that data eventually arrives at the server, data loss must be avoided to ensure correct data analysis and avoid under- or over-billing which is critical in this business. Given that, enough gateways should be installed to cover all the meters in the region by a required redundancy factor. This likely causes multiple gateways overlaps with meters which can lead to duplication of collected data.

2.2 Current development

Although this project is still in its very early stages at Stritzinger — having only recently entered the smart metering gateway market — the project has been given high priority because of being in contact with a strategic European customer. The customer is operating the metering infrastructure and is in need of a communication network interfacing with the actual meters. Stritzinger is also planning to use the technology to build smart grid applications with decision making at the edge. To this end, we are also in communication with another potential customer who is in the smart grid and energy trading business.

In-house, we have been developing our own embedded systems and smart Internet of Things (IoT) devices, and have extensive experience with networking and cloud services. The smart metering use case is a natural extension of those strengths, but raises the bar when it comes to handling of data and computations at the edge.

2.3 Detailed description

(a) Architecture

The architecture of a smart metering network can be divided into three layers; the sensor layer, the gateway layer and the Internet layer. Sensors are positioned at the edge and are collecting data, such as electricity or gas usage. They report metrics to gateways that are close to the sensors (10-100 meters range). Gateways perform operations on the data, communicate with each other, and to the Internet. They report data to servers in the cloud, such as auxiliary cloud servers that perform additional analytics or directly to the customers data centers. For this use case, Stritzinger is involved in building the gateways and possibly additional cloud server analytics.

The WM-Bus gateways we are planning to build are small, cost sensitive devices with form factor constraints (DIN rail, 11 cm max width). The gateway hardware and software is designed and implement by us (see Section 2.12). This gateway shall be able to communicate with up to 1,000 metering sensors via WM-Bus using sub gigahertz radio and will have at least 100 Mbit Ethernet connectivity and either 3G or LTE mobile connection capability. It should be able to communicate with arbitrary servers via Transport

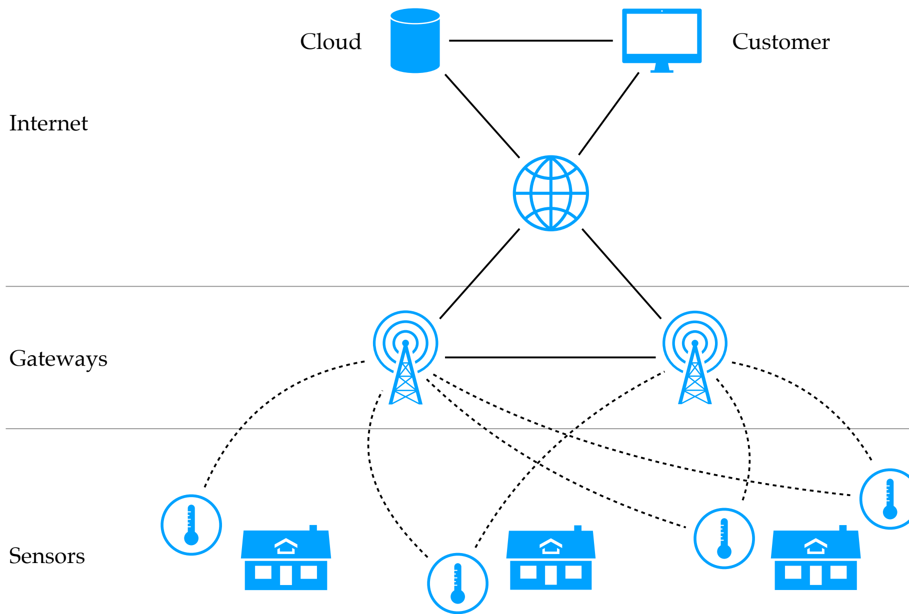

Figure 2.1: High level overview of smart metering infrastructure.

Layer Security (TLS) since we do not control the server infrastructure in all applications.

It is expected that the wireless ranges of neighbouring gateways overlap. If this overlap is high enough, we can provide tolerance against gateway failure. In addition, having WM-Bus radio messaging between the gateways themselves facilitates use of gateways as a repeater to collect data from more distant sensors and provides failure tolerance against mobile or wired up-link outages. It also allows us to attempt to reduce mobile bandwidth by moving data between neighbouring gateways to reach a wired gateway with much larger and cheaper bandwidth.

(b) Edge computing requirement

Since the gateways are, by physical necessity, at the edge in radio reach (up to 100 meters), we need to have edge computation together with a cloud solution to collect all the data, e.g., before delivering it to the billing software. Because of the potentially low bandwidth availability and cost savings opportunities, edge computing has a big advantage over sending raw data to the cloud.

2.4 Data model

Two distinct types of data flows through the system. The first is read-only metering data collected by meters, which is then sent via gateways to the cloud. The second is control data originating in the cloud, sent to the edge to affect actuators. Data is transmitted via WM-Bus. An example of metering data is electrical power or gas usage, and an example of actuator data is a control signal to turn electricity on or off. Both types of data flow can go between multiple gateways, in order to achieve failure tolerance and mobile data

savings.

Metering data is expected to be tuples of a sensor ID and a monotonically increasing counter, which might occasionally be reset on a battery change in some of the cases, and in other cases the whole device is replaced and basically the old ID disappears and a new one appears. IDs of metering sensors are immutable, metering and sensor values are associated with these IDs. Normally the mapping between IDs and sensor locations is done by the customer in the cloud, but for decision making at the edge this mapping needs to be known in the edge nodes as well.

In a networked gateway scenario where several gateways co-operate to provide redundancy and bandwidth, consistency becomes important. Aggregation of data must be performed between the gateways without data loss and duplication. Control decisions affecting a single local actuator must be atomic between gateways, either by connecting the actuator to a single gateway or by some kind of atomic update between gateways.

2.5 Detailed description of the computations

There are three typical types of computations in this use cases: aggregation, data thinning, and analytics. The former two are basically needed to reduce the transmission overhead of the sensed data towards the cloud; whereas analytics is needed for autonomous applications where actuators can be fed with control data based on decisions originating from the cloud.

Aggregation means collating data from several sensors, as a form of lossless data reduction (e.g. summing up the heating measurements of all room sensors in a flat). In this case, loss of some sensor data can result in mis-billing and care must be taken to ensure aggregation is only done when the correct set of data is available.

Data thinning is a lossy form of data reduction, where data points in time series are removed to save bandwidth. Since meter sensors are usually monotonically increasing counters this cannot result in mis-billing (however, data might shift between billing periods based on latency). Depending on the data format, data reduction might be achieved using delta encoding where occasional absolute metering values are interspersed with data points only containing the difference in values.

Computations for decision making through analytics are likely to be originating at the cloud level using more powerful data analytics tools. However, based on high level decisions in the cloud, local feedback loops might be given autonomy over decisions based on local real-time readings of data. For simple applications, this will be a one-way dataflow, that will instruct a gateway to affect an actuator based on the readings. For more advanced use cases, there might be the need for more advanced mechanisms, such as state machines. Both these types of actuator manipulations needs be synchronized reliably between multiple gateways if redundancy is to be achieved.

2.6 Conflicting operations and invariants

In the case of monotonically increasing counters or gauges, the data must arrive in a timely fashion. Delays of more than an hour or a day are not tolerated, depending on customer requirements. Missing data can be acceptable if later new data arrives to correct the current measurement within a certain time frame defined by the customers need. Depending on the customers data processing methods it might be acceptable to receive

duplicate data points, but it should be avoided. More complex sensors or advanced time series meters (where the change in value is as important as the value itself) generally cannot have missing data. Here data is also not allowed to be duplicated since it would change the meaning. Time series must also have a minimum guaranteed resolution for them to be useful.

In addition, aggregation must not change data precision. Some sub applications might be able to live with probabilistic values if the distribution is known but most metering use cases require exact data.

Invariants also exist when it comes to actions and decision making, since the objects they operate on exist in the physical world. For example, to avoid breaking batteries it is important that charging decisions are taken synchronously between involved nodes, so that one node does not undo an action taken by another node immediately after. Some actions might be possible to correct in feedback loops, but others might need harder guarantees.

2.7 Divergence and divergence control

Divergence is mostly accepted for collecting data, and latency for pure metering application is uncritical and can be minutes to hours. If infrastructure control decisions are made, required latency can be as low as seconds no matter where the decision is made. Important is that decisions are not based on outdated information.

As control and aggregation moves towards the edge, software update and configuration becomes a critical path when it comes to synchronization and version divergence. Steps must be taken so that even in case of diverging software version or software configuration versions, the system cannot enter a corrupted or incompatible state.

2.8 Network partitions

Partitions are to be expected at all points in the architecture. Connectivity to cloud servers can come and go based on wired or mobile Internet connectivity issues. Connections between gateways and meters is affected by quality and interference of radio signals. Tiered hand-off is done upwards in the case of collecting data, and downwards in the case of control decisions (see Figure 2.1).

2.9 Operational requirements

The meters would number in the millions and generate about 10-100 readings per day, at sizes of around 10 bytes per reading. There should be an overlap of getaway coverage of meters, providing a fixed partitioning with some fuzziness due to varying radio range. The gateways consist mainly of small, immobile, embedded systems. They are either driven on battery power using long-living batteries or connected to mains electricity. Performance and scalability on the edge can be controlled manually when deploying more gateways. These devices would number in the thousands.

A cloud layer for this use case is optional, and would reside in a smaller number of data centers with off-the-shelf hardware. They will be exposed to the Internet for communication with the gateways. Only a small number of cloud servers would be needed,

such as a cluster of ten machines. Any eventual consistency requirements between these machines is out of scope for this use case.

Customer cloud servers and databases are also out of scope and can be seen from the outside as a single big database, while it might internally not be.

2.10 Security requirements

There are three layers of different security requirements described as follows.

(a) Meters to gateways

Meters are supposed to be tamper-proof and have some built in encryption and identifications to ensure data integrity and privacy—which we will implement but is outside of the scope of our work. Limitations are set by the WM-Bus standard and what metering sensor suppliers provide.

(b) Gateways to cloud

Typical security measures as those used between clouds and applications are needed. The most likely option is to use a two-way securely authenticated and encrypted channel, e.g. with standard Hypertext Transfer Protocol (HTTP) over TLS (HTTPS) connections, ensuring stateless transfers and providing decent security primitives. An alternative would be TLS encrypted Transmission Control Protocol (TCP) over Internet Protocol (IP) (TCP/IP) connections.

(c) Gateway to gateway

Security support here is probably limited to what is available in the WM-Bus standard, thus additional security measures will be needed. For instance, readings—if not aggregated but potentially thinned—can retain the authentication and encryption of the metering sensors. Anything else originally processed by the gateways needs to be encrypted, authenticated and possible signed by the gateway.

In addition, software and configuration updates are done over the air and require highest security standards such that only signed code is booted once signature checks succeed. Update needs to be authenticated after the update is written as well. Denial of service attacks are a possibility. Flooding a gateway with unauthenticated metering requests or replaying of data are two likely methods. Malicious attacks are relevant in the context of command and control, where a gateway's normal remote-control mechanisms might be subverted.

Finally, physical access to gateways is an issue here, which needs to be dealt with by the customer—who installs the gateways. Furthermore, tamper proofing might be built-in in the gateways, but it is usually hard to get right; whereas secure storage of certificates and other keys is included in the hardware.

2.11 Data protection requirements

Metering data values are highly privacy sensitive. Ideally, they are anonymized at the metering source in a secure way—it depends on the WM-Bus standard and what is im-

plemented by the vendors. We assume sensors used by the customer have sufficient security measures, and securing them is out of scope for our work.

Anonymized aggregated data can potentially be disclosed if there are no ways to tie it to individual users, however data between gateways and the cloud should be encrypted by default. We also make the assumption that the gateways and the optional cloud service provided by us to the customer does not have to deal with individual customer IDs and thus can work purely on the anonymized data. Control decision data have identical requirements.

2.12 Implementation

We have selected the Erlang[2] programming language as the basis of our implementation, both for our gateways and for any additional cloud services. This is very helpful since most software in LightKone will also be implemented in Erlang. Erlang is a language developed at Ericsson[3] for use in telecommunications switches and is well suited to networked applications with fault tolerance requirements[4], and has proven to be much more productive compared to more traditional systems languages[39].

The challenge using Erlang for this use case is that Erlang has traditionally been seen as a high level language less suited to close to the hardware development[5]. We have extensive experience using Erlang in this context, and have developed the GRiSP[4] software and hardware stack to meet those demands. GRiSP runs Erlang close to the hardware without a traditional Operating System (OS). Instead it utilizes Real-Time Executive for Multiprocessor Systems (RTEMS)[5] to provide low-level OS functionality. Parts from other operating systems can be included such as network functionality and drivers, but implementing these in pure Erlang is usually beneficiary if it can be achieved without too much effort.

Erlang is also suited for cloud applications, therefore we will also use Erlang on the optional cloud aggregation layer, running on the FreeBSD[6] operating system.

We are in contact with Ericsson to enhance the intra-communication capabilities of Erlang, which can allow us to more easily achieve native Erlang communication between gateways over WM-Bus. This way, we do not have to implement a custom communication protocol between gateways.

2.13 Extension: Swarm of small satellites

Swarms of small satellites, also known as microsatellites, is another business area that Stritzinger has plans to enter, and is proposing a corresponding use case to study within LightKone. The use case is indeed similar to smart metering in many fundamental and practical aspects; and it is very likely to use the same prospective software base of the

[2]Erlang is a programming language developed by Ericsson, used to build massively scalable soft real-time systems with requirements on high availability: http://www.erlang.org

[3]Ericsson is a multinational networking and telecommunications equipment and services company headquartered in Stockholm, Sweden: https://www.ericsson.com

[4]GRiSP: Dive into a new experience of building wireless embedded systems: https://www.grisp.org

[5]RTEMS is an open source Real Time Operating System (RTOS): https://www.rtems.org

[6]FreeBSD is an advanced computer operating system used to power modern servers, desktops, and embedded platforms: https://www.freebsd.org

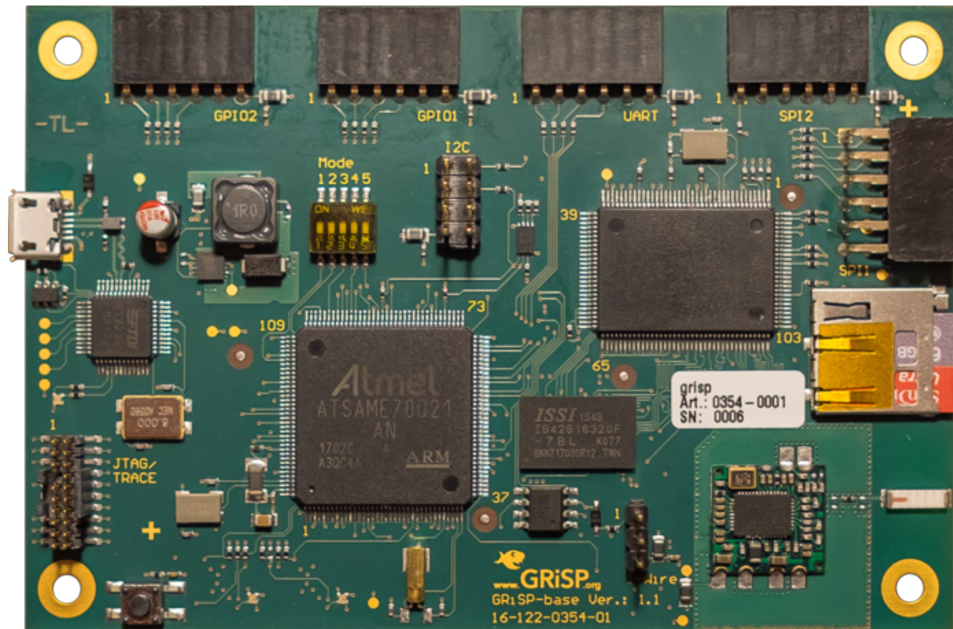

Figure 2.2: The GRiSP base hardware board.

metering use case to address potential new challenges in the satellites use case. To avoid redundancy, we opt to include the satellites use case as an extension section. In the following, we first overview the use case showing the conceptual and implementation similarities with the metering use case; and then we highlight the major points where special tackling may be required due to possible differences in functional and nonfunctional properties.

Traditional satellite development, launching heavy and well-tested satellites, is very expensive. As the need for more devices in space increases, ways to develop cheaper satellites and applications for space involve making smaller and cheaper devices with more off-the-shelf hardware and software. This market is expected to get very large in the coming years[10]. In this terminology microsatellites and nanosatellites refer to satellites in the size range 10-100 Kg and 1-10 Kg respectively[28]. Specialized versions include CubeSats which are devices that measure 10x10x10 cm. Such devices when acting as satellite swarms (or "fractionated spacecrafts") share many similarities to the smart metering scenario:

- They usually have sensors for ingesting data, such measurement devices or cameras. Actuators are possible, but rare.
- Computations in space are power constrained.
- Satellites move around, only being in range sporadically. This is similar to connection loss or the drive-by scenario for smart metering.
- Bandwidth to ground—downlink—is very limited (Kbit/s) compared to bandwidth between satellites (Mbit/s).
- Satellites are positioned at the edge, but are sometimes paired with larger, more capable satellites acting as downlink gateways.

- The goal is to collect and aggregate data to ground based servers (which have no constraints in regards to power or capability).

Next, we address the potential differences with the smart metering use case which will be tackled within Lightkone. A single exception that will not be addressed in the project is the following. In addition to satellites communication with each other and acting as devices in an edge network, they are often constructed with several processors for redundancy. This poses interesting sub-problems in regards to consistency, but in these cases the design usually opts for full consistency to mitigate potential hardware problems that are more common in harsh environments such as space (bitflips, faulty hardware, temperature extremes). One common solution is to have several copies of the system running, voting on decisions based on calculation results similar to much software in airplanes and other critical use cases. Since this is a different paradigm, and local to the individual devices, it can be ignored for our work.

(a) Current development

Stritzinger does not currently have any active projects related to nanosatellites but have realized the potential of the market and the big overlap with the current work with smart metering networks. Our developments so far in the smart metering use case and other projects have partly been done with the future nanosatellite use case in mind.

(b) Detailed description

The tiers in the smart metering use case apply here as well, where the gateway and sensor tiers are fulfilled by satellite objects in orbit. Bigger satellites or satellites with special equipment —acting as gateways— can be responsible for transmitting information to ground, whereas smaller or simpler satellites can be responsible for collecting data. One extra pseudo-tier exists in the additional equipment needed to receive and send radio signals on the ground. This is part of the transportation layer and can be seen as wireless link directly to servers on the ground for the purposes of this use case.

In addition to the smart metering use case, control operations become more common and more critical in space. These include, but are not limited to, changing of orientation (direction, orbital position etc.), configuration of instruments, integrity checks and power state changes. The same semantics does still apply and we expect any research results in regards to the smart metering use case will be well applicable here as well.

(c) Data model

The data model is equivalent to that of smart metering, where data should converge eventually, since strict consistency is not critical. In addition, although the available memory and storage may be more or less different to those of smart metering, they still share the same constraints.

(d) Conflicting operations and invariants

The semantics are identical to the smart metering use case with the addition of control operations being more common. Physical invariants exist, similar to those in the parent

use case, such as changing a power state must not be done in parallel by several neighbouring nodes. In this case the individual satellites would perhaps have more autonomy over their local control state, requiring only consensus internally among the hardware components in a single satellite.

(e) Network partitions

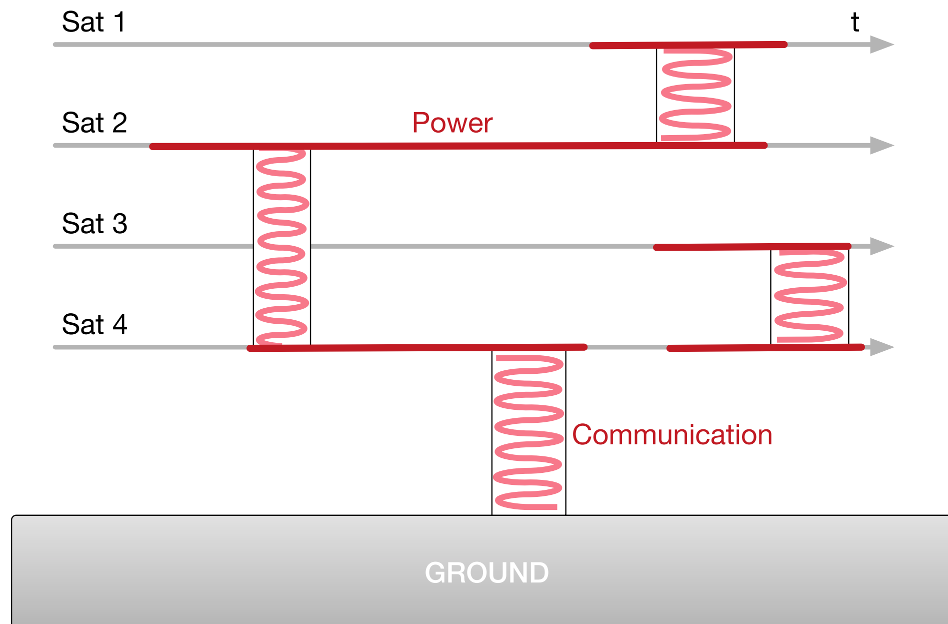

Figure 2.3: Communication between satellites and ground.

Network partitions would be more frequent, since there is more possible interference both between ground and the satellites and among the satellites themselves (see Figure 2.3). Satellites might also frequently go in and out of low power states to preserver battery, which will limit or disable any communications possibilities.

(f) Operational requirements

Providing power is a major difference to the smart metering use case. Battery power is the only way to power devices in space, and they have to be recharged via solar power which might not always be available. On the other hand, communication in space, though different, is operationally similar to non-reliable mobile Internet connection. In particular, Radio communication will have to be adopted for space, selecting appropriate hardware and frequencies. Other options includes using traditional frequencies, such as Wi-Fi where experiments have been done between spacecraft[36], or optical frequencies such as laser[11]. Uplink and downlink communication is subject to the atmospheric radio window and international and local regulations.

As mentioned before, another main difference to the requirements described in Chapter 2.9 is the possible addition of intra-device consistency and fault-tolerance by using consensus voting between redundant copies of hardware. This is a common way of handling faults in flight hardware and software. However, this should be treated as a hardware internal issue and is thus not part of the scope of this use case.

In all other aspects, storage and other hardware capacity will be identical to the smart metering use case as similar classes of hardware will be used.

(g) Security requirements

For uplink connections —data sent from ground to the satellites— command authentication needs to be performed to prevent unauthorized access to the hardware. Downlink connections —data sent from satellites to ground— normally does not need any advanced protection for most civil applications. Encryption will be used for valuable commercial data to protect it from competitors. For satellites, the physical attack scenario can be ignored since it in most cases will not be feasible to perform and thus not commercially interesting to protect from.

(h) Data protection requirements

Data sent is not tied to a user and does not need any privacy protection.

(i) Implementation

In initial prototyping stages, we plan to use the same technology stack as with the smart metering use case.

Chapter 6

Gluk

1 Agriculture sensing analytics

1.1 Overview of the use case

In this specific use case we present GLUK's IoT, sensor based analytics platform. This platform is planned to be used in many areas of our daily life, such as agriculture, health, smart homes, etc, depending on the business strategy of the Company. To the current deliverable, and based on the LightKone proposal, we will present our use case designed for precision agriculture. This information and technology based agriculture management system aims at the application of technologies and principles to identify, analyze and manage spatial and temporal variability associated with all aspects of agricultural production within fields. The domain we have chosen and have made our current installation is the winery industry. The installation refers both to the wine cellars where the wine is being made, but also to the vineyards where the grapes are being cultivated.

In order to achieve the above, our platform consists of the following main components:

1. **Wireless sensor nodes (WSN):** integrated sensors for data acquisition enabled with wireless interfaces for sending the data to a field basestation/gateway,

2. **Basestation/gateway** enabled with radio interfaces for communication with WSN and cellular radio interface for connectivity with GSM and/or GPRS/LTE for internet connectivity,

3. **Open APIs for data handling**, storage capabilities on the network keeping an archive of the measurements, open APIs for developers to develop new applications per farming case,

4. **Dashboard/GUI** for personal computers, tablets and smartphones which presents the history of collected data and business analytics for decision making.

In figure 1.1 the top level architecture is depicted, that explains how the aforementioned components are interacting per installation. In the current and every future installation the number of nodes could differ, and this has to do with every client's requirements. For this and for visualization reasons in figure 1.1 we have used the specific number of

nodes as an example.

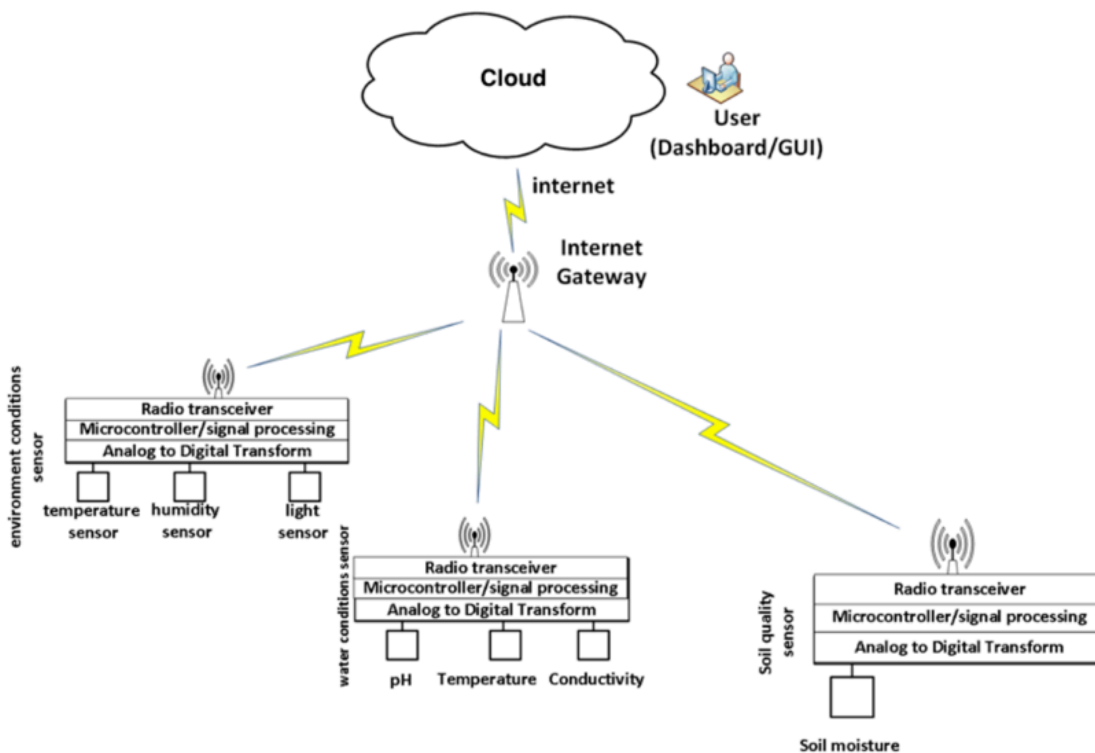

Figure 1.1: Current Setup - Architecture

1.2 Current development

In the current development we are using sensors that are collecting only immutable data. These data are coming from the vineyard and the cellar, and they have to do with parameters such as temperature, soil moisture, pH, conductivity, humidity, and light. All these data are transmitted through the gateways that are installed (in the vineyard and the cellar) and sent to the cloud. There we gather the date and forward them to the end user, in case they need static data, but we also apply business analytics so as to be used in future case. In the cloud we apply business analytics using machine learning algorithms to generate statistical results for future use.

After the deployment, we are always receiving new requirements from the end users and respectively we have to upgrade our system by installing new sensors and bringing up new services.

The current flow is depicted in figure 1.2.

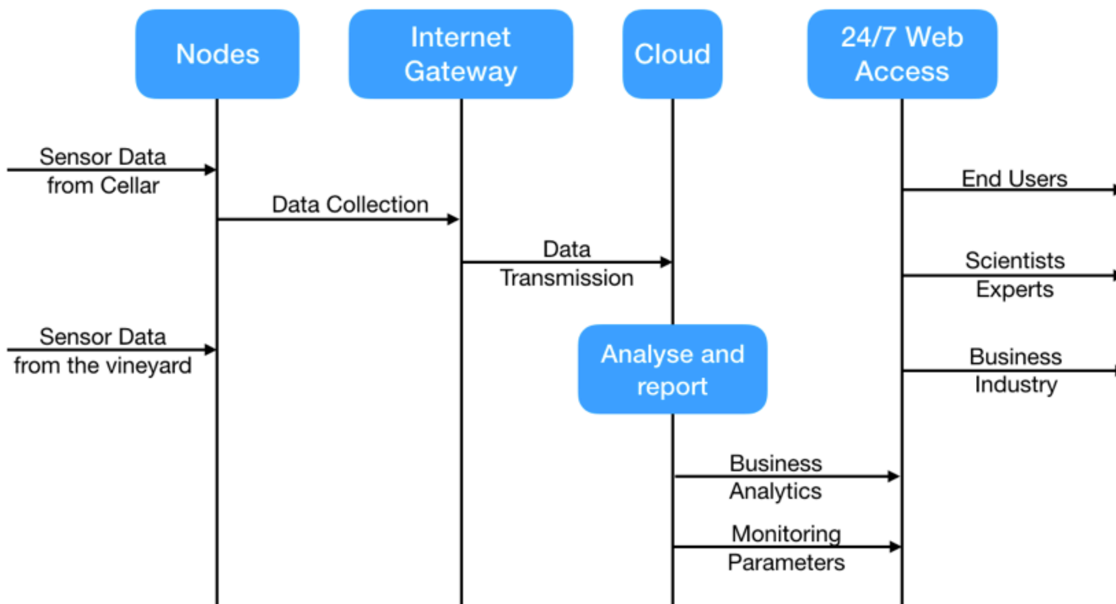

Figure 1.2: Flow in the current installation

(a) Conflicting operations

In the monitoring process of the inputs from the sensors, we do not meet any conflicting operations.

(b) Invariants that exist in the application state

At the current installation there are no invariants that could affect the system operation.

(c) Performance results

In the current setup we perform a data sampling rate of up to three samples per 10 minutes. The typical medium distances are 70 to 300 meters (inside the cellar smaller than outside) and the data transition rates from the nodes to the gateway are from 250 to 5,470 bps. This varies and has to do with many parameters, one of the most important being the environmental (weather) conditions.

(d) Security threats

In the current setup we use all the security features provided by the hardware in use. In the agriculture domain and in the current installation there are no major security concerns.

(e) Current deployment details

In the current development of the wireless sensor network system we are using the open-source hardware platforms Arduino [3] and Raspberry Pi [18]. The system is low-cost and highly scalable both in terms of the type of sensors and the number of sensor nodes, which makes it well suited (effective and cost-efficient) for our customers. We are using Microsoft Azure [32] cloud for hosting and running our scripts. In the Azure cloud we are using Microsoft SQL Server 2012 and the Entity Framework (version 6.1.3) for the management of the database layer.

1.3 Detailed description

In our development we want to replace cloud by performing the computation in the edge. At this moment, all the critical data is transmited to the cloud as input to the machine learning algorithms, stored, processed, etc., a fact that makes the whole procedure sluggish. We would like to use the stored data in the cloud in order to create the patterns (e.g. models for optimization of the grape cultivation, models for optimization of the wine maturing, etc.), but also to use them statistically, in order to create business analytics rules to be used by other industries e.g. insurance industry. Furthermore, we want to upload the rules to the edge (e.g. rules for triggering an actuator) and the patterns. That approach could be more efficient in a remote location-solution, such as in the precision agriculture field, where many constrains exist in the communication part.

This use case, **in the future deployment** must fulfill the following challenges and functional requirements:

- Easy to be installed and used (farmers are people with low technological skills).
- Scalable solution. Covers from a small family cellar to large vineyards (multiple buildings, etc).
- Plug n’ Play (turn on the power, set the sensors and get values).
- Low maintenance cost.
- Low power solution. Due to the fact that the sensors and nodes will be distributed in not easily accessible areas, and fixed power lines will not easy accessible, batteries are going to be used and therefore we need a minimal power consumption.
- Extendable. The user can add extra sensors any time he wants, without having any technical knowledge.
- Data Visualization. For example, the wine maker should be able to get a projected heat map of the cellar on his tablet.
- Weatherproof. The solution must be able to withstand exposure to weather without damage or loss of function.
- Power and communication backup plan.
- It must be able to process static measurements from sensors (e.g. temperature, humidity, etc) but also it must be able to control devices (e.g. actuators, surveillance cameras, etc).

- Near real-time data for decision making.
- Remote programming.
- Automatic different types of network handoff.
- Immediately to detect and heal a potential failure (reset, replace, or recalibrate the sensor, etc.).
- Software easy to be used for people from age range 20 to 65 (studies show that elderly people are in favor of using tablets rather than other devices – it is also easy for a farmer to carry a table around and read on it, even for people with presbyopia).
- Bidirectional control: For example, the user will have the option available to choose the period for acquiring new measures.
- Engage scientists and update rules and patterns. Through a web interface the scientists/experts must be able to upload the new rules or patterns at the edge. E.g. temperature level in the cellar before hit an alert, quality parameters for the wine inside the tanks, etc).
- Store the data for business analytics.
- Reduce cloud traffic and hosting costs.
- Insect detection and sound classification acoustic sensors.
- It should be able to incorporate data that make the decision more accurate and effective from external sources, e.g. meteorological data - predictions, imagery from drones, etc).

(a) Architecture

Figure 1.3 shows the use case architecture we have envisaged. Depending on the installation, there could exist more than one gateway and the number of nodes could be up to a hundred.

What we want to realize in the system is the following (best case scenario):

Edge nodes:

- Receive feeds from IoT devices using any protocol, in real time
- Run IoT-enabled applications for real-time control and analytics, with millisecond response time
- Provide transient storage, often one to two hours
- Send periodic data summaries to the cloud

Cloud:

- Receives and aggregates data summaries from many edge nodes

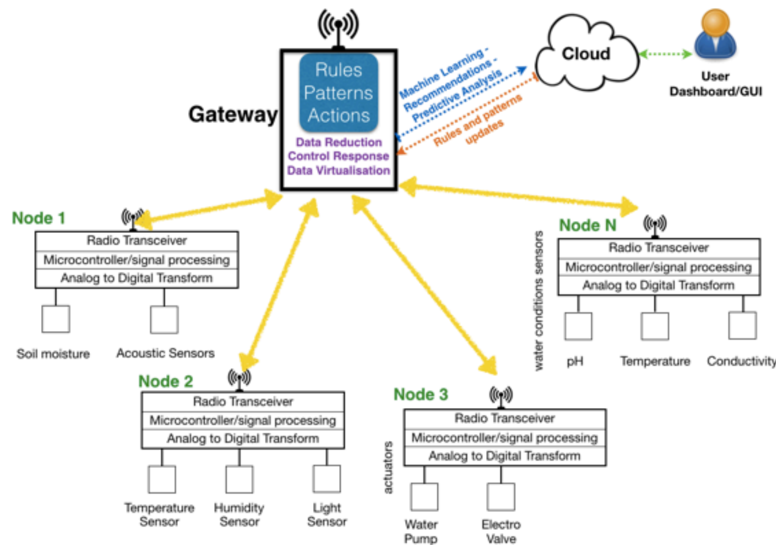

Figure 1.3: Flow in the edge

- Performs analysis on the IoT data and data from other sources to gain business insight
- Sends new application rules to the edge nodes based on these insights

(b) Edge computing requirement

By 2020, 43 percent of all IoT generated data will be processed at the edge. Our use case is going to use "edge intelligence" to make real-time analytics-based decisions. E.g. an insurance company (which is a target customer for Gluk) calculates a potential risk in the insured product (e.g. vineyard) and also receives the cultivation data to cross check any frauds. Another example has to do with the online and remote update of the rules and the patterns from scientists or experts for an optimized cultivation. Machine learning routines will be applied and used at the edge. In order to realize the above suggestive scenarios, we need edge coomputing in our platfrom in order to enable faster and local decision making but also realtime constraints in future installations. The business target for the company is to offer a sophisticated services portfolio, not "yet another hardware solution", as defined and presented in the functional requirements in a previous section. The company strategically plans to use the aforementioned technology and equipment also in more demanding and competitive applications compared to the agriculture case.

1.4 Data model

The system will mainly manipulate data received by sensors and actuators.

Sensors may be physically hardwired or communicate via a short-haul communication protocol like Bluetooth Low Energy (LE) or ZigBee.

Actuators will affect the electromechanical or logical state of the product and the environment. They are going to be the system's hands and feet. System commands sent to embedded applications—such as remote reboot, configuration updates, and firmware distribution—should also be considered actuation because, by changing its software, the system is in fact changing the physical reality of the product.

In the current implementation the wireless sensor network was treated as a relational database. In our developments each sensor produces multiple tuples. The node that generates the tuple is termed the source. For example, the temperature sensor produces a tuple of the form <nodeLocation, timestamp, temperature>. All the data are stored in Microsoft SQL Server 2012 Database.

Dependencies

Dependencies between sensors and actuators are going to exist, since the actuators status could be change if specific parameters are being changed (e.g. the soil moister sensor can affect the water pumps for irrigation purposes).

1.5 Detailed description of the computations

Following, an example is provided:

The winemaker must receive feedback from the scientists about the quality of the grapes inside the tanks. Until now the winemaker has to receive a sample of the mixture and deliver it to chemists in order to analyze it (through spectroscopy) and take the results. This procedure that takes long, cost time, money and increases the risk of spoiling the grapes in the tank. Using the edge solution the spectroscopy could be installed in the tank immediately, receive the values and make the computation at the edge by providing the result immediately to the winemaker. The rules and patterns in order to interprete the results would have been uploaded to the edge. So, the solution could deliver results even though in an offline mode. Later on the data could be transmitted to the cloud in order to be used for statistical purposes. Furthermore, in that way, we want to avoid any privacy implications that could arise by transmitting data in the cloud (in another scenario of this topology).

The hierarchy of the computations we want to achieve is the following:

- The most time-sensitive data is analyzed on the edge node closest to the things generating the data. For example, one of the most time-sensitive requirement is to verify that protection and control loops are operating properly. Therefore, the edge nodes closest to the sensors can look for signs of problems and then prevent them by sending control commands to actuators.

- Data that can wait seconds or minutes for action is passed along to an aggregation node for analysis and action. This type of data could be for example the values from the sensors that do not affect the actuators or any other immediate decision making.(e.g. last hour temperature values in order to be used in the heat map projection).

- Data that is less time sensitive is sent to the cloud for historical analysis, big data analytics, business analytics and long-term storage. For example, each of thousands or hundreds of thousands of edge nodes might send periodic summaries of weather data to the cloud for historical analysis and storage.

1.6 Conflicting operations and invariants

In the case that there is a blocked actuator and we don't know it on time, it could be huge problem for the production. Or in the case that an actuator is being or not triggered because of a false or non-measurement it could lead to major problems. E.g. The air-climate control in the cellar, in a very warm environment, is not triggered because the temperature measurements have failed. The system as from the moment that is going to use actuators and rules will be applied it is going to involve invariants. These invariants should not be violated and must be true all the time. Otherwise a critical failure in the monitoring and alerting will happen, fact that will create damages in the client.

1.7 Divergence and divergence control

It is important for the deployment to know any time the real status of the network. In case of the vineyard or the precision agriculture the deployment could reach a large geographical areas and the nodes could be difficult to be monitored. It is quite important to know the measurements per area, since this is a key element for the quality of the crops. E.g. it is quite important to know the temperature in the cellar from multiple sampling points, so in case those halves of the sensors are off, it is a problem. Furthermore, we must know in which geographical position the problem exists so as to easily find it and repair it.

In the case of the measurements form the sensors (e.g. temperature, humidity, moisture, etc.) a small delay does not make significant problem in the system. We could live with a 5 minute divergence in this case but in the case of the actuators it could be a huge problem, e.g. in case of a blocked water pump a water overflow could happen, and we cannot live with that. In case of actuators only a divergence of 1-2 seconds could be affordable.

The data that are going to be sent to the cloud for further processing and statistical analysis could be transmitted with some delay. One hour it would be also acceptable. However, we want to avoid such long delays.

An offline mode is acceptable if the computations take place on the edge, fact that could happen in real life (e.g. loss of network connectivity due to heavy weather conditions).

1.8 Network partitions

It could be possible to have network partitions. The system could be receiving data, which are stored locally in the network device and after a time period, transmitted to the main database. When the partition goes away it is acceptable to merge data. This procedure should taking place seamlessly. Also, it could exist a node hierarchy in the installation in case of multiple nodes. Each node (hardware feature), could be used in

a different role in the deployment (e.g. end device, router, coordinator, etc) and can be deployed either in a **tree** or **mesh** deployment.

1.9 Operational requirements

The application currently runs in the infrastructure, and this is how planned to do. Currently in a small number of data centers in the cloud. Edge computing will manage to increase the performance in more competitive scenarios (more nodes and respectively sensors and data, especially in the automated decision making process).

A typical example of the hardware that we use in experiments (more expensive solution comparing to installations in clients) is presented below:

Gateways
Processor: 1 GHz Quad Core (x86) RAM memory: 2 GB (DDR3) Disk memory: 16 GB Power: 6 to 12 W (12 V) Power source: PoE (Power Over Ethernet) Security: Authentication WEP, WPA, WPA2, HTTPS

Nodes
Microcontroller: ATmega1281 Frequency: 14.7456 MHz SRAM: 8 kB EEPROM: 4 kB FLASH: 128 kB SD card: 2 GB Weight: 20 g Dimensions: 73.5 x 51 x 13 mm Temperature range: [-10 ^{o}C, +65 ^{o}C] Clock: RTC (32 kHz)

Nodes Consumption
On: 17 mA Sleep: 30 microA Deep Sleep: 33 μA Hibernate: 7 μA Operation without recharging: 1 year (Time obtained using the Hibernate mode as energy saving mode)

In the application of the solution in client's side, always we keep in mind the hardware cost parameter. The client must pay for the services and not an extremely high initial purchase cost. Therefore we are looking for cost-effective hardware solutions.

Currently we are in investigations of printing out our own board.

Furthermore, the following parameters must be considered:

- The network could be Ethernet, wifi or GSM/3G/4G. Also, we have to consider the fact that the system will be deployed in a **wide geographical area with variant environmental conditions** and a **standard minimum bandwidth must be preserved**.
- Each installation it should be capable to support up to 100 nodes per 1 gateway. Each node could act as end device, router, coordinator, etc. Each node is considered to support minimum 10 sensors or actuators.
- The size of each object is considered to be mostly **bytes**.For example in the current installation each frame is 20bytes. In case we integrate cameras in the installation the size could be increased but this is not a priority. The rate of data growth follows a linear curve.
- The objects could be composed of independent databases but also in one big database. This has to do also with the hosting/cost parameter.

1.10 Security requirements

High security is not a must in the specific Use Case and computation power could be saved in favor of other computation. However, security issues must be considered in terms of:

- Integrity. Corrupted or manipulated data will affect the provided services.
- Authentication. We want to avoid injecting additional packets, and nodes accepting false administrative tasks (e.g. network reprogramming).
- Availability. It is essential that the users of the sensor network must be capable of accessing its services whenever they need them.
- Freshness. The data produced by the sensor network must be recent. Consequently, the messages of the network should aim to reduce the network delay to the smallest possible value, even in unfavourable situations where the network is under attack or other emergency. Furthermore, if an adversary success on replaying an old message inside the network, the data not only will be useless, but also harmful (e.g. it may inform of a non-existent alarm).
- Auditing. In order to adjust their behaviour, sensor nodes must be able to know the state of their surroundings.
- Privacy and Anonymity. These security properties are very important only for the scenarios where the location and identities of the base station and the nodes that generated information should be hidden or protected.This item is related to section 1.11.

1.11 Data protection requirements

In the current development and use case presentation there is no need to consider data protection issues. However, in future installation using the same topology, but different requirements, they could arise data protection issues. For example, if the infrastructure is being used in a physical security scenario, in vehicle plates recognition, face recognition if cameras are being used, in smart homes-health environment, when the user will have to transmit personal data or for physical security purposes cameras are being used.

1.12 Implementation

We implement our scripts in C++ and Java (programming part in sensor network). For the publication to the outside world we use Microsoft Azure Cloud and for the deployment the IIS version 8.5. All the business analytics have been generated using the ASP.NET MVC, version 5.2.3. We are open to use the technologies that have been developed in Syncfree project (e.g. Antidote db).

Furthermore, we are always taking account the following objectives during our implementations:

Technical Objectives	
Objective	Details
Selection of commercial off the shelf sensors and actuators for the selected agriculture applications	Selection criteria will include: Depending on the applications sensor types such as soil moisture, temperature, humidity, light, pH, conductivity, hive weight, soil temperature, trunk diameter, stem diameter, fruit diameter, weather station etc. and actuator types such as water pump, electro valve, oxygen pump, growing Light, etc. Power consumption evaluation of sensors and actuators for autonomy evaluation. Sampling frequency of sensors, accuracy and sample size. Resistance on harsh environment (temperature, humidity, rain)
Selection of wireless technology for connectivity of the remote devices	Selection criteria will include: Low-power connectivity solutions. Depending on the use case corresponding design parameters such as radio transceivers density per square meter, and wireless link distance. Low packet error rates, and wireless capacity to fulfil sensor sampling/data rates. Mesh and star topologies with the corresponding routing protocols.
System components (boards) development for the agriculture fields	Depending of use case we will propose and introduce the corresponding system components (boards). Main characteristics of the system components will include: Power management, supporting sleep modes for low power consumption, energy autonomy or prolonged lifetime if is battery-based, introducing solar energy harvesting if is needed Low power microprocessor(s) for data processing, storage and receiving transmitting data. Low latency interfaces with wireless transceivers, sensors and actuators.

Chapter 7

Data Protection

1 Introduction

LightKone use cases rely on modern information and communications technology, which in some cases could raise a lot of questions regarding legal, ethical and even social implications. Social exclusion, trespassing of individuality and privacy, malicious use of sensitive data and decisions made towards their management are only some general indicators of potential misguidance when applying new technological innovations. Additionally, the use of personal data such as acquisition of images and location information, entails the risk and responsibility of being aware of a citizen's profile. Even so, how can it be assured that the technology applied to improve business activities does not ultimately result in harming of the other citizens' well-being?

In this section we will make an introduction to the current data protection framework, and further considerations per each LightKone's use case with reference to existing legislation will be defined.

2 EU legal framework for the right to data protection

The protection of personal data is one of the most important issues in the European Union regulatory framework, and extensive work has been performed in order to obtain a comprehensive European Directive that regulates the treatment of personal data in the European Union. In fact, personal data is collected and used in many aspects of everyday life and can be collected directly from the individual or from existing databases; moreover, personal data can be used for purposes different from the original ones. Additionally, personal data may be available in places different from the original location, so that data related to the citizens of one member state can be used in other member states of the EU or even in every country of the world. This raises the necessity of a European regulation framework to handle and protect this data by overcoming potential discrepancies among national laws. Furthermore, some member states did not have laws on data protection. For these reasons, there was a need for action at the European level, and this took the form of EC Directives.

2.1 Directive 95/46/EC

The Protection of Private Data Directive 95/46/EC[16] is a directive adopted by the European Union designed to protect the privacy and protection of all personal data collected for or about citizens of the EU, especially as it relates to processing, using, or exchanging such data. It includes all key elements from article 8 of the European Convention on Human Rights, which states its intention to respect the rights of privacy in personal and family life, as well as in the home and in personal correspondence. The Directive 95/46/EC was developed to harmonize national laws for personal data protection and movement of data, based on the existing national legislation of the EU Member States. The Directive 95/46/EC is the reference text, at European level, on the protection of personal data. It sets up a regulatory framework which seeks to strike a balance between a high level of protection for the privacy of individuals and the free movement of personal data within the European Union (EU). To do so, the Directive sets strict limits on the collection and use of personal data and demands, that each Member State sets up an independent national body responsible for the protection of these data. The Directive 95/46 EC includes the main legislative principles applicable to all processing of personal data and all the use cases and scenarios of the LightKone project are subject to this directive, and after 25th of May 2018 they will be subject to the new EU General Data Protection Regulation (GDPR). In the context of the Directive, personal data is defined as "*any information relating to an identified or identifiable natural person. An identifiable person is one who can be identified, directly or indirectly, in particular by reference to an identification number or to one or more factors specific to his physical, physiological, mental, economic, cultural or social identity*" (Article 2a).

2.2 EU General Data Protection Regulation (GDPR)

The EU General Data Protection Regulation (GDPR) [17] replaces the Data Protection Directive 95/46/EC on the 25th of May 2018 and was designed to harmonize data privacy laws across Europe. It protects and empowers all EU citizens' data privacy and reshapes the way organizations across the region approach data privacy. The aim of the GDPR is to protect all EU citizens from privacy and data breaches in an increasingly data-driven world, that is vastly different from the time in which the 1995 directive was established. Although the key principles of data privacy still hold true to the previous directive, many changes have been proposed to the regulatory policies; the key points of the GDPR as well as information on the impacts it will have on business can be found below.

(a) Increased territorial scope (extra-territorial applicability)

Arguably the biggest change to the regulatory landscape of data privacy comes with the extended jurisdiction of the GDPR, as it applies to all companies processing the personal data of data subjects residing in the Union, regardless of the company's location. Previously, territorial applicability of the directive was ambiguous and referred to data process 'in context of an establishment'. This topic has arisen in a number of high profile court cases. GPDR makes its applicability very clear - it will apply to the processing of personal data by controllers and processors in the EU, regardless of whether the processing takes place in the EU or not. The GDPR will also apply to the processing of personal data of data subjects in the EU by a controller or processor not established in the EU,

where the activities relate to: offering goods or services to EU citizens (irrespective of whether payment is required) and the monitoring of behaviour that takes place within the EU. Non-EU businesses processing the data of EU citizens will also have to appoint a representative in the EU.

(b) Penalties

Under GDPR, organizations in breach of GDPR can be fined up to 4% of annual global turnover or €20 million (whichever is greater). This is the maximum fine that can be imposed for the most serious infringements e.g.not having sufficient customer consent to process data, or violating the core of Privacy by Design concepts. There is a tiered approach to fines e.g. a company can be fined 2% for not having their records in order (article 28), not notifying the supervising authority and data subject about a breach or not conducting impact assessment. It is important to note that these rules apply to both controllers and processors – meaning 'clouds' will not be exempt from GDPR enforcement.

(c) Consent

The conditions for consent have been strengthened, and companies will no longer be able to use long illegible terms and conditions full of legalese, as the request for consent must be given in an intelligible and easily accessible form, with the purpose for data processing attached to that consent. Consent must be clear and distinguishable from other matters and provided in an intelligible and easily accessible form, using clear and plain language. It must be as easy to withdraw consent as it is to give it.

(d) Data subject rights

(d).1 Breach notification Under the GDPR, breach notification will become mandatory in all member states where a data breach is likely to "result in a risk for the rights and freedoms of individuals". This must be done within 72 hours of first having become aware of the breach. Data processors will also be required to notify their customers, the controllers, "without undue delay" after first becoming aware of a data breach.

(d).2 Right to access Part of the expanded rights of data subjects outlined by the GDPR is the right for data subjects to obtain confirmation from the data controller as to whether or not personal data concerning them is being processed, where and for what purpose. Furthermore, the controller shall provide a copy of the personal data, free of charge, in an electronic format. This change is a dramatic shift to data transparency and empowerment of data subjects.

(d).3 Right to be forgotten Also known as Data Erasure, the right to be forgotten entitles the data subject to have the data controller erase his/her personal data, cease further dissemination of the data, and potentially have third parties halt processing of the data. The conditions for erasure, as outlined in article 17, include the data no longer being relevant to original purposes for processing, or a data subjects withdrawing consent. It should also be noted that this right requires controllers to compare the subjects' rights to "the public interest in the availability of the data" when considering such requests.

(d).4 Data portability GDPR introduces data portability - the right for a data subject to receive the personal data concerning them, which they have previously provided in a 'commonly use and machine readable format' and have the right to transmit that data to another controller.

(e) Privacy by design

Privacy by design as a concept has existed for years now, but with the GDPR it is now becoming part of a legal requirement. At it's core, privacy by design calls for the inclusion of data protection from the onset of the designing of systems, rather than an addition. More specifically - 'The controller shall .. implement appropriate technical and organizational measures .. in an effective way .. in order to meet the requirements of this Regulation and protect the rights of data subjects'. Article 23 calls for controllers to hold and process only the data absolutely necessary for the completion of its duties (data minimisation), as well as limiting the access to personal data to those needing to act out the processing.

(f) Data protection officers

Currently, controllers are required to notify their data processing activities with local DPAs, which, for multinationals, can be a bureaucratic nightmare with most member states having different notification requirements. Under GDPR it will not be necessary to submit notifications / registrations to each local data protection authority (DPA) of data processing activities, nor will it be a requirement to notify / obtain approval for transfers based on the Model Contract Clauses (MCCs). Instead, there will be internal record keeping requirements, as further explained below, and DPO appointment will be mandatory only for those controllers and processors, whose core activities consist of processing operations which require regular and systematic monitoring of data subjects on a large scale or of special categories of data or data relating to criminal convictions and offences. Importantly, the DPO:

- Must be appointed on the basis of professional qualities and, in particular, expert knowledge on data protection law and practices.
- May be a staff member or an external service provider.
- Contact details must be provided to the relevant DPA.
- Must be provided with appropriate resources to carry out their tasks and maintain their expert knowledge.
- Must report directly to the highest level of management.
- Must not carry out any other tasks that could results in a conflict of interest.

(g) Data protection per use case

In the following table are presented in a total all the data protection considerations per use case (identified by their chapter and section number). Guifi does not implicate any privacy issues in their use cases, therefore their use cases references are not mentioned in the table.

Traceability matrix of data privacy implications		
Use case	**Implication**	**Compliance to legal framework**
4.1 4.2 4.3	Fairly and lawfully processed personal data	Article 6 of the Directive 95/46/EC, Article 5 of the Regulation EC COM (2012), GDRP
4.1 4.2 4.3 5.2	Processed, for limited purposes, personal data	Article 7, 8 of the Directive 95/46/EC, GDRP
4.1 4.2 4.3 5.2	Adequate, relevant and not excessive personal data	Article 6 of the Directive 95/46/EC, GDRP
4.1 4.2 4.3 5.2	Accurate personal data	Article 6 of the Directive 95/46/EC, GDRP
4.1 4.2 4.3 5.2	Data should not be kept longer than necessary	Article 6 of the Directive 95/46/EC, GDRP
6.1 4.1 4.2 4.3	Data have to be processed in accordance	Article 7 of the Directive 95/46/EC, GDRP
6.1 4.1 4.2 4.3 5.2	Data must be securely exchanged via encryption mechanisms	Article 16, 17 of the Directive 95/46/EC, GDRP

Traceability matrix of data privacy implications (cont.)		
Use Case	**Implication**	**Compliance to legal framework**
4.1 4.2 4.3	Data must not be transferred to countries without adequate data protection	Article 25 of the Directive 95/46/EC, GDRP
4.1 4.2 4.3 5.2	Data must be securely destroyed after their usage with absolutely no chance of retrieval	Article 12 of the Directive 95/46/EC, GDRP
4.1 4.2 4.3	Citizens must be aware of being under surveillance and have access to their personal data	Articles 6, 12 of the Directive 95/46/EC, GDRP
4.1 4.2 4.3 5.2	Secure storage and management of the personal data	Article 16, 17 of the Directive 95/46/EC, GDRP
4.1 4.2 4.3	Respect the citizen's privacy and individuality – try to keep the anonymity	Article 6 of the Directive 95/46/EC, Article 5 of the Regulation EC COM (2012), GDRP
4.1 4.2 4.3	The citizens must be aware of the fact that they are monitored , of their legal rights and of the impact on their lives	Article 6 of the Directive 95/46/EC Article 5 of the Regulation EC COM (2012), GDRP

Traceability matrix of data privacy implications (cont.)		
Use Case	**Implication**	**Compliance to legal framework**
4.1 4.2 4.3	Use of personal data according to human rights and democratic practice	Article 6 of the Directive 95/46/EC Article 5 of the Regulation EC COM (2012), GDRP
4.1 4.2 4.3	Ensure the end-users that their personal data will not being used against them (Confidentiality)	Article 16 of the Directive 95/46/EC, GDRP
4.1 4.2 4.3	Data must be securely exchanged via encryption mechanisms	Article 16, 17 of the Directive 95/46/EC, GDRP

Chapter 8

Security Analysis

1 Introduction

1.1 Security versus data protection

In the previous section we discussed the legal aspects related to data protection. However, in addition to these aspects, information security must also be considered from a technological viewpoint. Both aspects are closely related areas that complement each other but belong to different domains. While *data protection* refers to the set of actions used to safeguard the identity of system users, requiring their explicit consent for processing, storage and transmission of sensitive data, the topic of *information security* is clearly distinguished since it refers to the protection of assets that are essential for the functionality and purpose of the information system itself. In particular, information security ensures the confidentiality, availability and integrity of information assets, while protecting them from a wide range of threats [26]. It aims to guarantee system functionality and minimize the risks and possible consequences associated with loss or leakage of these assets.

1.2 Methodology

For LightKone, information security is an important design and implementation aspect, standing as a relevant concern that is transversal to all of the project's use cases. The purpose of this section is therefore to perform a security analysis that will allow us to understand the security threats and requirements for the project and for each of its particular use cases.

In terms of methodology, we will conduct the security analysis by following a set of steps. We will start by providing, for each use case, a high level system model overview in which we will identify its main components, key functions and actors. Then, leveraging these overviews, we will identify the assets to be protected and define each use cases' security goals. Finally, we outline the security assumptions made, define our trust model, and list potential threats and the adversary model considered.

2 Coordination between servers and data storage for the Guifi.net monitoring system

System model

The system described in Sections 1 and 2 of Chapter 3 is composed of three main components. First, there are thousands of nodes that collaboratively provide or consume various network services (e.g. Internet connectivity). Their state and available resources need to be monitored for the purpose of billing, capacity planning and service provision. This is done by the second component of the system - the monitoring servers. They coordinate with each other in order to distribute monitoring tasks and maintain a shared database containing the results of monitoring. Finally, the main Guifi.net web and database server aggregates and visualizes the data provided by the monitoring servers. It also maintains the list of nodes to monitor which it shares with the monitoring servers.

There are several actors involved in the system. These are the system administrators that control the central web and database server. They have full access to the nodes' statistics and the list of nodes to monitor. Then, there are administrators of the monitoring servers that have access to the shared database of monitoring results as well as to a pool of monitoring tasks. Similarly, the owners and/or administrators of the network nodes control available resources and the performance of the provided services.

Sensitive system assets and security goals

The main asset we are willing to protect is a shared database of the nodes' status and performance maintained by the monitoring servers and replicated at the central web server. While the data in this database may be regenerated if lost, this could have negative effect on the services that depend on its availability (e.g. network provisioning). The mapping between the monitoring tasks and the monitoring servers responsible for those is also an important asset. Every node in the network must be monitored by at least one monitoring server and failure to do so might lead to late incident response and loss of potential profit. The nodes' status information as reported by network nodes is another essential asset of the system. Our goal is to ensure the correctness and authenticity of this information and its secure transmission and storage at the database. We also aim to protect the database itself as well as the mapping between the monitoring tasks and the monitoring servers responsible for those.

Assumptions and trust model

We assume the central web and database server administrators to be trusted and not maliciously collude with the administrators of monitoring servers. The list of nodes to be monitored that is provided by central server is assumed to be initially complete and containing accurate nodes' description.

Threat and adversary model

Within the described system we can identify several potential attackers. On the one hand, the administrators of the nodes might tamper with the nodes' settings and report

fabricated information. By doing so, they might appear providing a better service and thus attract more user traffic. Bogus data can also affect the network provisioning and maintenance decisions. On the other hand, the monitoring servers' administrators might as well modify the monitoring data especially for the nodes they control. This could be done for the purpose of improving their nodes' statistics and/or lowering the statistics of other nodes in the network. Additionally, they could abuse the monitoring tasks pool by assigning themselves to the new tasks as they appear and immediately reporting fake data. Considering that normally checking a node's status takes some time, such technique may potentially leave benign monitoring servers out of work. Consequently, this could lead to data conflicts and ultimately to an unreliable monitoring service.

Security requirements

We outline the following security requirements for the described system:

- The system must ensure that each network node is monitored by at least two independent monitoring servers for cross-checking. If any persistent divergence in the reported data is detected the corresponding server must be flagged and reported.
- The system must protect the integrity of the monitoring database. All operations must be logged and authenticated. The monitoring servers can only write or modify the database entries for the nodes they were assigned to. Any modification of monitoring data for other nodes must be forbidden.
- The status data reported by the network node must be checked for correctness, integrity and authenticity before being added to the database. The monitoring software running on the node must be protected and verified for each report.
- The monitoring tasks distribution must not only be fair and efficient but also abuse-resistant. No monitoring server must be responsible for the majority of nodes.
- The system must be resistant to network partitions and database corruption. The collected data should be persistent and remain available when any of the monitoring servers fails or disconnects.

3 Service provision support for the Cloudy platform

System model

The system described in Section 3 of Chapter 3 relies on a network of nodes geographically distributed across the Catalonia region. Each of these nodes provides a certain service to the rest of the network (e.g. Internet connectivity), and needs a way to advertise it. At the same time the users of the network need a way to discover such services. The designed system, therefore, aims to satisfy both of these needs. Through the use of a distributed shared database each user and service provider can discover or advertise a certain service. The database contains the up-to-date information on the offered services, their providers, and additional information needed for accounting and billing.

Sensitive system assets and security goals

The key asset to be protected is a service data which is stored in the distributed database. The service data objects may be written and read by both users and service providers (currently there is no limited access). We, therefore, aim to protect the integrity of the database and ensure its availability.

Assumptions and trust model

We assume the services provided by the designed system and the data they operate with to be out of this analysis' scope. While some of the services might operate with sensitive user data, we leave it for the service provider to address the potential security and privacy issues.

Threat and adversary model

Within the threat model we can identify three potential attackers. On the one hand, there are service providers who can deliberately modify the entries in the database in a way so that users looking for a competitors' services would be pointed to their ones. Such an attack can lead to the profit loss of the respectful providers. Additionally, malicious providers could delete the entries pointing the users to the competitors' services, effectively creating a monopoly in the network for certain services. Finally, the billing information can be secretly manipulated in order to charge the users additionally after they have already subscribed to the service (e.g. the service price, the amount of resources used, etc.). On the other hand, the users might as well act maliciously and launch similar attacks. The billing information might as well be changed to lower or avoid the service bills. This way the users might be using the services without being charged for those. Lastly, there are independent malicious entities that are willing to advertise and benefit from the services that are not approved or supported by the network. This services might be offered by unregistered users or providers and can be potentially insecure.

Security requirements

We define the following requirements for the designed system:

- The database must be protected from unauthorized access and manipulation. The service providers must be able to change only the information about the services they provide. The users must be able to view the service description but not modify it. Any modification of the entries in the database must be logged and visible to everyone. The users must be notified of any changes in the billing cycle or prices of the services they consume.
- The integrity of the database must be guaranteed in order to avoid manipulation of the service data in the process of writing to the remote database, as well as for reading the data from the database.
- The system must be able to authenticate both service providers and service consumers to avoid unapproved or unsupported service deployment and ensure accurate billing and accounting.

4 Pre-indexing at the edge

System model

The system described in Section 1 of Chapter 4 comprises several main components, namely, client applications that write data to the storage system and retrieve the search results, precomputing nodes at the edge that aggregate client requests and forward data to higher levels of the system, and backend storage systems where the client data are stored. The precomputing nodes also perform various computations on client data including encryption, hash signature generation, indexing and index lookups.

Within the described system we can identify several main actors. First, there are users of the system who upload their data and run queries over it. Second, the applications and their corresponding developers that allow for efficient data management at the client side. Third, the provider of the precomputing edge service and the service itself. Finally, the cloud storage provider and maintainer and the service it provides.

Sensitive system assets and security goals

Having outlined the model of the system we can now identify its critical and most sensitive assets. There are four main system assets we are willing to secure. The client data and metadata may contain sensitive user information and must be treated with care at all the stages of the operational cycle. At the same time, an index information and the results of the client search queries, while not being sensitive per se, might provide insights useful to the attacker. Finally, the authentication tokens for accessing cloud storage backends must be stored securely as they might be used for unauthorised access to the application data without the user consent. Considering all the described sensitive assets the design of the system must protect the integrity and confidentiality of not only user data and metadata, but also search queries results and access tockens for cloud storage services.

Assumptions and trust model

We assume the software and hardware platform where the client applications are executed to be secure. We do not address attacks on user data performed by any malicious applications running on the same platform that are not involved in the described use case scenario. We also assume that the edge precomputing service provider does not maliciously collude with the application developer or cloud storage provider.

Threat and adversary model

In this specific scenario, our threat model consists of malicious application developers who try to exfiltrate or temper with sensitive user data. Having direct access to the unencrypted data, such an adversary might leak it to unauthorized parties or modify it according to his needs (e.g. avoid encryption process). At the same time, in case of pre-encryption indexing, the edge node maintainer and service provider can perform the same attacks. Edge node maintainer can also abuse its cloud storage access rights and manipulate the data without the user consent. Cloud storage provider on the other hand, can have unlimited and untracable access to the user data stored at his premises. In both

cases the user data can be analyzed for user profiling, shared with third parties, or even deleted. Finally, an external attacker can intercept and eavesdrop on the communication between the client applications and the edge computing nodes, or between the edge nodes and the cloud storage systems.

Security requirements

Considering the above threat model the security requirements for the described system are as following:

- The system must provide defense mechanisms that prevent access to unencrypted dataflows.
- The system must ensure availability of user data in case of network partitions and partial data deletion.
- All the parties involved in the system must be authenticated and authorized by the user
- The system must provide the ways for the user to verify the integrity of data at any stage of processing.

5 Lambda functions at the edge

System model

The use-case scenario described in Section 2 of Chapter 4 shares a lot of similarities with the previously described use case. The model of the system and its components are mainly the same with the exception of additional component that is responsible for executing various Lambda functions in response to certain events (e.g. new log file is uploaded). It might be deployed along the edge precomputing node or run at the client side.

This new component introduces additional member to the list of actors interacting with the system. In this case it is an entity which writes the code of Lambda functions and offers it to the system clients. The application developer or an edge computing service provider both can provide such services. The user himself might be acting as this entity if he or she has appropriate expertise in that.

Sensitive system assets and security goals

While the critical system assets defined in the previous section are still valid in this scenario, the integrity of Lambda function code and its output results are additional assets the system must protect. We also aim to ensure the confidentiality of the data generated, consumed and transmitted within the system.

Assumptions and trust model

Similarly, our assumptions and trust model from the previous scenario remained unchanged. However, we add additional assumption that Lambda functions are running in isolated sandboxed environments and cannot interact with each other. This is will ensure that malicious version of the functions running side by side could not collude.

Threat and adversary model

In this scenario, our previously described threat model is complemented with an additional adversary. The malicious application developer or an independent entity publishing custom Lambda functions can pose a threat to the system security. They can abuse access to sensitive application data and trigger actions that might cause harm or financial losses. They can perform denial-of-service (DoS) attack preventing the application and potentially other service providers to operate as intended.

Security requirements

We define the security requirements for such use case scenario as following:

- Lambda functions' actions must be properly logged and constantly verified for compliance with the stated functionality.
- The access control system must ensure that the Lambda function has access only to the resources needed for computation.
- The user data the function processes and acts upon must be properly protected from tempering and eavesdropping.

6 S3 local cache of central data

System model

The system reviewed in Section 3 of Chapter 4 is similar to the one we have described previously with a difference in the functionality of the edge node. In contrast to previous scenario, the edge node performs local caching of read data objects and while being closer to the client applications allows for faster writes operations. A client machine with sufficient computing resources might perform these operations.

The list of actors in this specific use case scenario depends on the amount of entities that have access to the edge node. Application developers and their corresponding applications, as well as system users themselves must be considered. In the situation when caching takes place at a the edge node provided by the independent service provider (e.g. cloudlets) this provider and its maintenance personnel are considered as additional actors.

Sensitive system assets and security goals

The personal user data and metadata cached at the edge is the main asset we wish to protect. We aim to ensure the confidentiality and integrity of these data. We also aim to secure the data in transit between the central data storage and a client cache.

Assumptions and trust model

We assume that the client machines are protected from physical access by unauthorized parties that could otherwise gain full control over the cached data. In cases when the cache data is stored at the premises of a third party storage provider, we assume that the application developers do not maliciously collude with this provider.

Threat and adversary model

Several malicious adversaries must be considered in the security analysis of this system. The applications developers might configure their applications in way so that the cached user data is leaked to a third party. The same developers might try to access the cached data of other applications and either eavesdrop on it or manipulate with it according to their interests and needs. On the other hand, the cached data stored outside of user's premises at a local third party storage, might be tampered or leaked to third parties.

Security requirements

Considering the described adversarial actions we define the security requirements for the designed system:

- The system must ensure the integrity of the cached data at all stages of the application's life cycle.
- It should also prevent any unauthorized entity from eavesdropping on or tempering with it. This involves protecting the data in transit and at a third party storage.
- The system must provide the ways for the user to verify the integrity of cached data at any stage of processing.

7 No-Stop RFID

System model

Within this use case scenario a system described in Section 1 of Chapter 5 is composed of the following components: the RFID tags that are moving on the conveyor belt, the RFID readers that can read and write the data to and from the RFID tags, and a distributed cache of RFID content featuring completed and missing steps. The readers communicate with each other through Ethernet network by flooding all the latest updates to keep the cache data consistent. Unfortunately, due to resource constraints of the readers this communication is unencrypted and solely relies on efficiency of firewall rules at the higher level of the network and physical access control at the factory.

Given a restricted environment of the factory premises the only actors constantly interacting with the system are the factory workers. The RFID readers manufacturers and firmware developers have limited access to the system during its deployment, testing or upgrading.

Sensitive system assets and security goals

The main and most important system assets we wish to protect is the cache data that must not only be consistent from reader to reader, but must also be accurate. Any inconsistency between the readers cache data might slow down the manufacturing process or even stop it completely. Wrong or corrupted data in the cache may lead to broken products or even physical damage to the machinery or personnel. We therefore seek to protect the integrity of the cache data and ensure communication security.

Assumptions and trust model

We assume that the factory premises are protected from physical access by unauthorized parties that could otherwise gain full control over the cached data.

Threat and adversary model

We consider several malicious adversaries in this use case scenario. First, the competitors of the factory might be willing to intervene in the manufacturing process to intentionally corrupt the products and/or gain the insights that can be of a commercial value. The factory workers might be intentionally or unintentionally tampering with the cache data and RFID readings. RFID readers manufacturers and their respective software developers might also carry a stealthy attack. There are several attack vectors by means of which such attackers can exploit the vulnerabilities of the system. First, an attacker can compromise the firewall and gain unrestricted access to the readers' network. By having access to this network an attacker can effectively launch DoS attack or manipulate the data transmitted in clear text. While the DoS attack can slow down or stop the manufacturing process, the tampered cache data can cause damage. The malicious software running on the RFID reader can be configured in a way so that it periodically skips one or more production steps. Such an attack will be hard to detect, since it will not show up during short-time debugging, and will eventually affect the quality of manufactured products and company's market value.

Security requirements

Considering the above attack vectors we define the following security requirements for the system:

- The system must be able to protect the cache data from eavesdropping or manipulating by unauthorized parties.
- The cache data must be consistent between the readers and delivered in a timely manner.
- The communication between the readers must be secured and resistant to DoS attacks.
- The system must provide a way for the factory owners to verify the authenticity of RFID reader software and hardware.

- Any modifications to the cache data must be detected and flagged without affecting the manufacturing process.

8 Smart metering gateway

System model

The smart grid system described in Section 2 of Chapter 5 comprises the following components: the smart meters that monitor resource consumption; the gateways that aggregate the sensor readings reported by smart meters in their vicinity; and the cloud server of service provider that orchestrates the gateways and meters. The gateways also communicate with each other and share collected meter readings for data aggregation and thinning. Such design allows to achieve fault-tolerance and data savings. At the same time, the gateway nodes may also act as actuators reacting to the changes in meter data.

We identify two actors involved in the system. These are the home owners whose activity is monitored by the smart meters (e.g. electricity or water consumption), and the service provider that installs gateways and meters, provides the desired service, and charges users according to the resources they consume.

Sensitive system assets and security goals

For the described use case, the main system asset are the meter readings. They must be accurate and represent the latest values measured. These readings aggregated at the gateway nodes and a cloud backend are essential for the provided service. Timing is also important, and while temporal delays in meters readings' delivery might not affect the billing cycle, they might be critical in cases when gateways act as actuators and respond to changes in resources consumption. Additionally, meter readings represent a highly sensitive information that might affect the privacy of their users when exposed to unauthorized parties. The mapping between the meter readings, meters IDs and their corresponding locations shared between the gateways and the cloud must be protected. Overall, our goal is to ensure the integrity, availability and confidentiality of the metering data.

Assumptions and trust model

We assume that gateway nodes' manufacturers and firmware developers do not maliciously collude with the service provider. We also assume that only service provider's employees have physical access to the gateway nodes.

Threat and adversary model

There are several potential attackers we can identify within this use case scenario. First, the home owners who have physical access to smart meters might tamper with the meters' hardware and software in order to lower utility bills. Alternatively, they can intercept the meter to gateway communication and modify the reported values. Secondly, the service provider might be leaking privacy sensitive information of its customers to the third parties for targeted advertisement or new offers. The service provider can also

configure smart meters to report higher consumption values to increase profits. Finally, external attackers might intercept communication between the meters, gateways and the cloud backend in order to manipulate the meter data in transit. Besides leaking these data to third parties, the criminals might use it to predict when the customers are not at home and plan their illegal activities accordingly. The attackers can also carry out a DoS attack to jam communication between the gateways and the cloud backend or to exhaust the gateway resources (e.g. wireless bandwidth). The software and configuration updates might be another attractive target for the attacker. With the modified firmware update an attacker can potentially gain full control over smart meters and/or gateways.

Security requirements

We therefore outline the following security requirements for the designed system:

- The system must provide the ways for service provider and a home owner to verify the authenticity of the smart meters' and gateways' software.
- The authenticity of software updates for meters and gateways has to be verified at install time.
- The smart meters must be able to verify the identities of the gateways they communicate with and visa versa. The same applies for the gateway to gateway and gateway to cloud server communication.
- The communication between all the components of the system must be protected from eavesdropping or manipulation by unauthorized parties.
- The system must ensure the authenticity, correctness and integrity of the reported meter readings at all stages of processing.
- The meter readings must be properly anonymized to prevent utilities and third parties from linking the collected data to the identities and locations of the customers that generated them.
- The system must be able to withstand network partitions and provide fault-tolerance against gateway failures.
- It must support high volumes of data from potentially thousands of meters and ensure reliable data delivery to the service provider.

9 Agriculture sensing analytics

System model

The use case scenario described in Chapter 6 presents a sensor-based platform for precision agriculture which allows for collecting, analyzing and reacting to field sensor data in near real time. It collects the readings from thousands of field sensor nodes and actuators, aggregates these values at the edge gateway and finally sends them to the cloud service for further processing. The cloud backend provides an interface for statistical analysis and

data visualization, and creates rules and patterns for actuators using machine learning. These rules are then uploaded back to the edge gateway which allows for local real-time decision making and action without the cloud support.

The proposed system is planned to be used by several parties. Farmers rely on the system to facilitate and automate the process of farming, generate agriculture analytics and provide useful insights on their activity. Insurance companies review statistical data in order to calculate a potential risk for an insured product (e.g. crops). Scientists and researchers share trigger rules and patterns that allow for an optimized cultivation. Software developers create applications for edge nodes and actuators to react to changes in the environment according to the configured rules and patterns.

Sensitive system assets and security goals

There are several sensitive system assets that are essential for a functionality of the system. The most important assets are raw and aggregated sensor readings. They are collected and aggregated by the edge nodes, with backups stored in the cloud backend. The whole system depends on availability, correctness and accuracy of these readings. Failure to access and compute on readings within a certain time frame might lead to crop losses or damage. Inaccurate sensor readings will affect data analytics' utility and might trigger erroneous actuator actions. Finally, inconsistent or a lack of sensor readings might affect the decision of the insurance company. We therefore seek to protect the integrity of sensor readings during their generation, transmission, storage and processing.

Additionally, the system depends on the correctness and integrity of rules and patterns used by actuators to react to environmental changes. Poorly written or intentionally manipulated rules might cause crop damage or even loss. The outdated or inaccurate patterns might lead to wrong decisions made by farmers and insurers. The rules and patterns must be protected from tampering in the same way the sensor readings are protected.

The connectivity between sensor nodes and gateways is essential for coordination and computation on shared sensor data. Insecure or poor connectivity, or a lack of it, might lead to inconsistent computation results, inaccurate statistics and erroneous actuators actions. We seek to protect the data in transmission and provide the means of authenticating the communicating nodes.

Finally, the battery and bandwidth resources of the sensor and gateway nodes are limited and must be used with care. Sensor nodes in the field are usually battery powered. The power supply is thus limited and requires precise radio communications planning. Constant transmissions will eventually deplete the battery resources effectively making the sensor node useless. In case of GSM/3G/4G connectivity, wireless carrier might charge the farmer according to the bandwidth consumed, so efficient bandwidth usage is needed. Poorly written or malicious version of application running on the gateway node might exhaust the bandwidth resources with constant transmissions. Our goal is to protect the nodes' resources from abusive actions and verify the authenticity of applications using them.

Assumptions and trust model

We assume the hardware of sensor and gateway nodes to be trusted and operating as stated in the specification sheets. At the same time, the developed system might be able

to detect by means of fingerprinting if the original hardware has been modified without the user consent.

Threat and adversary model

Potential attackers might be competitors of the system owner, application developers, rules and patterns creators, cloud backend service and wireless connectivity providers. The system owner himself might appear as an attacker. For instance, by manipulating the statistical data before releasing it to the insurance company. Considering all the potential attackers, we therefore define four possible attack surfaces: (1) sensor node software, (2) gateway node software (applications, rules and patterns), (3) cloud backend, and (4) external resources (e.g. wireless medium).

For a sensor node attack surface there could be several potential attacks. First, an attacker might send crafted sensor reading to the gateway node to trigger certain automations that might have devastating effects on the crops. For instance, false high moisture level sensor readings will cause the water pump to shut down and potentially cause severe damage to crops. This attack might be possible if the software of the sensor node is compromised by the attacker or node's legitimate owner. At the same time, erroneous sensor data might originate from uncalibrated or broken sensor.

Secondly, an attacker might manipulate the sensor node software so that it constantly transmits sensor readings or any arbitrary data to the gateway node or other sensor nodes. Such an attack will not only exhaust the scarce battery resources but will also effectively overload the wireless medium with redundant messages preventing benign nodes from communicating.

At the gateway node level, an attacker might configure a malicious version of the gateway application to leak the sensor data to the competitors and other third parties. Moreover, an attacker could reprogram the gateway to accept the commands from these unauthorized entities. By modifying the rules and patterns it is also possible to manipulate the actuators state and affect the utility of the provided services. Finally, bandwidth resources can be abused by the malicious application running on the gateway by generating significant volumes of traffic and causing higher bills.

Cloud backend can be vulnerable to several attack vectors. The database with all the sensor readings collected during the whole period of system operation can be an attractive target. By leaking the data from this database the competitors could get insight information on the agriculture methods and techniques. Such information might be of a commercial secret, thus leaking it might cause the company to loose its competitive advantages on the market. An attacker could also control the state of actuators from the cloud backend and send false administrative commands. Similarly to gateway and sensor nodes attacks this can lead to severe damage or loss of the crops.

In all the attack surfaces mentioned above, we assumed the attacker to have access to at least one of the system components. For some type of attacks this is however not necessary. An attacker with sufficient resources and competence could carry a Denial-of-Service (DoS) attack on wireless medium. By simply broadcasting radio signals on the same frequency the nodes use for communication and overpowering the original signal, an attacker can efficiently jam the network. This kind of attack will prevent the nodes from communicating with each other and the gateway, leading to missing sensor readings and, as a consequence, invalid actuators state. At the same time, an attacker could also

intercept and replay legit actuator commands triggering certain actions. Finally, Internet service provider that provides connectivity for a gateway node could perform a man-in-the-middle attack and intercept the traffic to the cloud backend.

Security requirements

Considering all the described attacks, we define the requirements for the system to comply with in order to prevent or minimize the risks from any of these attacks:

- Every node in the network must be identified and checked for any software or hardware modifications before being added to the network. All nodes failing this check must remain isolated from the rest of the network.
- The gateways must be able to identify the sensor nodes abusing the wireless and battery resources and have ways to isolate those from the rest of the network.
- The system must prevent any changes to the sensor readings and the actuators commands at the time they are generated, transmitted and processed. Any manipulation with the data along the way must be detected and reported.
- The data generated by the system must be only accessible to the authorized entities (e.g. network nodes, applications, system administrators). The system must ensure secure communication channels and storage.
- The system must be robust and fault tolerant. It must adapt to changes in the wireless environment and withstand the DoS attacks. Each communication channel must be replicated to ensure data availability and timely delivery.
- The sensor data must be verified for correctness before the action is taken upon it. The system must detect the deviations in the sensor reading reported by the sensors that are close to each other, cross-check the values and discard suspicious readings. Any repeated, conflicting or inconsistent actuators commands must be detected and discarded as well.

Chapter 9

Deep Learning

Deep learning is a branch of machine learning that attempts to model high-level abstractions in data by using deep neural networks (DNNs), which are layered architectures based on multiple nonlinear transformations. In the last few years, deep learning has achieved remarkable theoretical and practical success. It is able to achieve goals that were long considered to be impossible. It is increasingly being applied in commercial computing systems. Its use in data centers is well-known and ubiquitous, for example in targeting advertisements, predicting user behavior, or providing an interactive voice interface (e.g., Apple's Siri). At the current time, barring a few exceptions, it is almost always implemented in data centers because of its high computational and data needs. The exceptions are recent and motivate this section. A prominent exception is the iPhone X released in September 2017 which does on-phone deep learning computations to do real-time three-dimensional face recognition. It is able to do these computations because its custom A11 processor has targeted support for them.

In this section we give an overview of deep learning for nonexpert users, i.e., for those who wish to use deep learning and are not themselves deep learning researchers, and we explain how deep learning is becoming increasingly relevant for edge computation. Several LightKone use cases have learning components, and it is likely that these learning components will become increasingly important in the next few years. For brevity, we assume basic knowledge of standard machine learning (classification and regression) as can be obtained from many introductory articles (a particularly good introduction is given in [15]). For an in-depth discussion of the topics of this chapter, we recommend the book *Deep Learning*, published in 2016, which is the most comprehensive recent presentation available at the time of submission of this deliverable [21].

Health monitoring: a killer app for deep learning on the edge

Personal health monitoring is an application of deep learning on the edge that is quickly becoming a killer app. The goal of personal health monitoring is to continuously monitor each human being for early signs of health problems, such as cancer, heart problems, and degenerative diseases such as Alzheimer's and Parkinson's. The human is set up with a personal sensor array that continuously monitors vital signs and sends data streams to a set of personal computation nodes (a "personal data center"). These nodes analyze the data streams and notify the human if any health problem appears. This analysis is sophisticated; state-of-the-art systems that do such analysis run on data centers and use

deep learning algorithms. For example, clinical studies are currently (in 2017) being done for detecting signs of Parkinson's disease using the accelerometer data from smartphones and analyzing them in a data center [38]. Another recent example is the Apple Watch, for which new software released in September 2017 is able to diagnose certain kinds of heart ailments through data collected by the watch sensors. With the release of its HealthKit, Apple is clearly betting on health monitoring and fitness as a major application area for the Apple Watch.

The advantages of such an early monitoring system are numerous, both for improving users' health and reducing healthcare costs (early treatment is almost always much cheaper than late treatment). To make a reliable diagnosis, the analysis must continuously do large computations on large data streams. It can be shown that the aggregate computation and storage power of data centers are insufficient to do the analysis for all human beings on earth. On the other hand, the number of edge devices and their computation and storage abilities are growing exponentially with time. J. Pereira predicts that in 2027 there will be more than 1000 Internet of Things devices for each human being on earth [35]. It follows that a natural place to do the computations for health monitoring is directly on the edge. An increasing number of major companies have reached the same conclusion, for example, Huawei and its Kirin 970 chipset (released in 2017) supports deep learning computations on mobile phones, as well as Nvidia's recently released Jetson TX2 deep learning processor for embedded systems, and Apple's new A11 processor in the recently released iPhone X. Nvidia used to be focused on building graphics boards, but it has completely transformed itself into an AI company focused on building high-performance processors for deep learning algorithms.

Acknowledgements and caveats

This section was written by Peter Van Roy, using material from the DeepLearn 2017 Summer School complemented by other sources and by contributions from the other LightKone partners. While all the factual assertions in this section have been taken from the work of experts and are assumed to be correct, the extrapolations on the future relevance of deep learning to edge computing are specific to the LightKone project. Given that LightKone is still at an early stage, we consider that thinking about the future is highly important: it is essential for a project at the state-of-the-art such as LightKone to make motivated extrapolations as to what will exist beyond the state-of-the-art, even if we do not yet know how some of these extrapolations will be implemented.

Most of the material in this section (with a few exceptions) is summarized from the lectures given at the DeepLearn 2017 Summer School, held in Bilbao, Spain, from July 17-21, 2017. This summer school was outstanding, both in the quality of the lectures and in the number of attendees. There were more than 1300 attendees from all over Europe. There were 28 lecturers, all of whom are either top researchers in deep learning, or top researchers in an area that uses deep learning. These lecturers gave 86 lectures organized in 3 or 4 parallel sessions.

1 Introduction to deep learning

Deep learning is a branch of machine learning that uses deep neural networks (DNNs) to model high-level abstractions in user data. Subsection 1.1 first gives a high-level overview from the viewpoint of a developer. Subsection 1.2 then explains what is making deep learning successful now, when work has been going on in this area for several decades. Subsection 1.3 elaborates on how to design a deep neural network. Subsection 1.4 gives an example of a practical DNN, namely AlexNet. Finally, subsection 1.5 concludes by comparing deep learning to other disciplines.

1.1 Three-step design process

A DNN consists of a set of building blocks combined into a sequence of layers. Building a practical DNN consists of three steps:

1. *Design*. The first step is to determine which building blocks to combine and in which order to combine them.

2. *Training*. The second step is to determine the parameters in the building blocks. The layers in between the input and output layer are commonly called *hidden layers*. The training step therefore determines the values of the parameters in the hidden layers. This is done by running a computation-intensive training algorithm with a large set of training data for which the output is known. This is called *supervised learning* and is currently the most common training approach used.

3. *Inference*. The third step is to use the DNN in an application. This also requires significant computation, since the layered system can contain a large number of elements and many nonlinear functions connecting them. But it is much less computation-intensive than training because it is straight line code (no iteration over many examples, like in training).

1.2 Why is DL successful now and not before?

It is legitimate to ask why deep learning has become successful now and not earlier. What has changed now with respect to the past? To explain this, let us first give a brief history of neural network research. There are three main periods. The first period ended around 1969, when Minsky and Papert analyzed an early form of neural network called a *perceptron*. They showed that a single-layer perceptron could not do many important tasks, such as the XOR function or determining whether a curve in two-dimensional space is connected or not. Because of this, research in neural networks was essentially abandoned for a decade. The second period started in the 1980s. There was a resurgent interest in neural networks because of the popularization of the backpropagation algorithm for training hidden layers. However, the vanishing gradient problem, which shows that training hidden layers becomes more and more difficult as the number of layers increases, again caused neural network research to languish. Finally, we come to the third period of popularity, which is considered to have started around 2012 and continues to the present day. The current period is characterized by three properties:

1. *Training needs massive computing power*. The massive parallel computation power needed by current training algorithms is now generally available through GPUs. GPUs were originally designed for high-performance graphics ("Graphics Processing Unit"), but they are now used very successfully for training DNNs. Still, even with fast GPUs, training often takes from hours to days.

2. *Backpropagation can train the hidden layers of a DNN*. The backpropagation algorithm can theoretically train the hidden layers, but in order to make this practical three modifications are currently used:

   - All the nonlinear functions in the DNN are differentiable. Typically, it is important to model discontinuous (step function) or piecewise continuous functions (rectifiers), so smooth versions of these functions are used.
   - The backpropagation algorithm uses the chain rule for differentiation together with gradient search to determine the hidden parameters. This works because a DNN is mathematically just a composition of nonlinear functions.
   - The architecture of the layers is designed to avoid the vanishing gradient problem. For example, the LSTM (Long Short-Term Memory) is a *trainable* memory cell. The discovery of the LSTM in the 1990s was a major step toward overcoming the vanishing gradient problem.

3. *An increasing number of highly visible successes*. We give three examples of successes that were highly mediatized:

   - The IBM Watson natural-language question answering system competed on the *Jeopardy!* TV game in 2011 and won $1 million against former human champions.
   - Real-time voice translation between English and Chinese was demonstrated by Rick Rashid, then head of Microsoft Research, in 2012 with an error rate of only 7%, far less than previous systems.
   - AlphaGo, combining deep learning and Monte Carlo tree search, won a three-game Go match in 2017 against the world's number one human player, Ke Jie.

1.3 Introduction to the design of a deep neural network

This section gives a technical summary of how to design a deep neural network. It is intended as a starting point for a LightKone developer who wishes to investigate the use of DNNs in his or her edge application. A deep neural network consists of a series of layers. Each layer has a matrix (or vector) of variables (which are real numbers). Layers are pairwise connected through transformation functions. If there are n layers in all, the first layer is called input layer, the final layer is called output layer, and the $n-2$ other layers are called "latent" or "hidden" layers. The transformation functions can either be linear or nonlinear. Each transformation function is parameterized by a weight matrix (or vector) that needs to be determined in the training phase for successful operation of the DNN. Designing a successful DNN has two main parts:

- *Determining the meaning of each layer and the transformation functions connecting layers*. This is a difficult design problem, comparable to algorithm design.

New and successful designs are still considered worthy of conference publication. There is a lot of folklore regarding design principles for this part, but no mature methodology yet. There is no substitute for experience.

- *Training each transformation function*. Training determines values for the coefficients of the weight matrices. This is commonly done by means of a training set, i.e., a large set of inputs for which correct outputs are known. Training is done by the backpropagation algorithm, which is a form of stochastic gradient descent that works as follows. A training item is fed to the DNN and the output is computed from the input to the output layer. At the output layer, the output is compared to the correct output. The differences between the computed and correct output are propagated backwards from the output layer towards the input layer. During this propagation, the algorithm modifies each coefficient in the weight matricies to make the computed output become closer to the correct output. The modifications are computed using the chain rule for differentiation. It is observed empirically that learning typically proceeds in two stages: first, a short "fitting" stage, where the network learns to correctly identify the input items, and second, a much longer "compression" stage, during which the network becomes good at generalization ("diffusing" its correct identifications through the space of possible inputs).

The most important transformation functions are:

- *Convolution*. Input and output layers are matrices, and each output value is a linear function of neighboring input values. Convolution is inspired by the visual cortex of the brain. It does feature extraction, i.e., it can transform an image containing only pixels into an image with edge, shape, or complex object detection. Using several successive convolutional layers gives more and more sophisticated features, up to whole objects. Many practical DNNs have several convolutional layers in the early stages, in order to raise the abstraction level of the input before other transformations are done.

- *Full connection*. Each output vaue is a linear function of all input values. This is not scalable in general; it is typically used only near the output of a DNN, where the size of the layers has been reduced.

- *ReLu (Rectified Linear Unit)*. This is a differentiable version of a rectifier, where a rectifier is defined so that each output value is 0 is the corresponding input is negative, otherwise it is equal to the input.

- *LSTM (Long Short-Term Memory)*. Each output value is a smooth nonlinear function of the previous output value and the corresponding input value. This provides a memory cell. To control the cell, the input, memory, and output are all gated using smooth functions. An LSTM can be seen as a memory cell that is designed in such a way that it is trainable using backpropagation.

- *Maxpooling, averagepooling, and subsampling*. When the input is a two-dimensional matrix, the output matrix is smaller in both dimensions than the input by factors $k \times l$. With maxpooling each output value is the maximum of the corresponding $k \times l$ input submatrix. It is called averagepooling if the output is the average of the

inputs, and subsampling if the input weights are learned. This is a way to reduce the data size without reducing output quality (if done right).

- *Softmax*. The softmax function is most often used in the last stage. It does an elementwise normalization: it returns a vector of values that are normalized so all add up to 1. It first does an exponential of input values before normalization. The exponential is important because it will separate otherwise close coefficients. Semantically, it makes the output be probabilities, if the inputs are energy values in a statistical model.

Several easily available software packages exist for designing and training deep neural networks. We mention Caffe, which is simple and fast but limited to its built-in functionality. PyTorch and TensorFlow are flexible toolboxes that support simple development and complex neural network topologies. Theano is a library optimized for high performance. Several libraries, including PyTorch and TensorFlow, provide a convenient syntax using pipe notation to define multilayer DNNs. This is highly expressive and can express most of the historically important DNNs in just one or two lines of code. To gain understanding in the design of DNNs, it is highly recommended to download one of these packages and experiment with it.

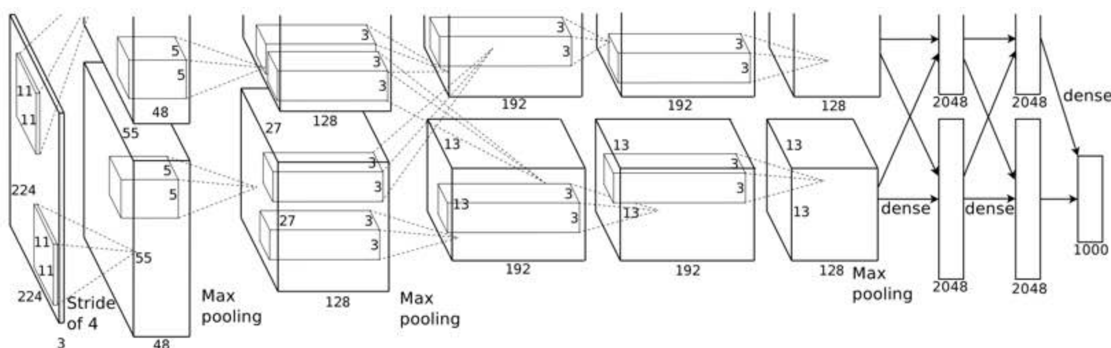

Figure 1.1: Architecture of the AlexNet deep neural network

1.4 AlexNet: a practical deep neural network

AlexNet is a deep convolutional neural network that classifies high-resolution images. The information about AlexNet in this section is summarized from [29]. This network was entered in the 2012 ILSVRC, an annual competition of image classification with 1.2 million images. It achieved the best score with an error rate of 15.3%, compared to 26.2% for the second-best entry. Since then, the techniques used in AlexNet have been improved, resulting in even lower error rates for more recent DNNs.

Figure 1.1 shows the architecture of AlexNet; this figure explicitly shows how the computations are partitioned between the two GPUs used for training. The GPUs communicate only at certain layers. Two GPUs were necessary because the network was too big to fit on a single GPU. The DNN contains eight layers with weights; the first five are convolutional and the remaining three are fully-connected. All layers are necessary; removing even one layer results in reduced performance. The last fully-connected layer

is fed to a 1000-way softmax, which produces a distribution over the 1000 class labels. The full DNN has 60 million parameters that need to be learned; several innovative techniques were used to reduce so many parameters without overfitting. AlexNet shows that a large, deep convolutional network is capable of achieving record-breaking results using purely supervised learning.

1.5 Deep learning compared to other disciplines

One of the surprising properties of deep learning as an engineering discipline is that it subsumes good solutions in many traditional areas. If the underlying layers are well-chosen, then the training process will automatically pick a reasonable solution from a lower level. Consider the following traditional areas:

- *Traditional computer vision view.* All the major results in computer vision can be reformulated as simple DL pipelines that can be executed in popular DL software frameworks such as PyTorch or TensorFlow. For example, John Canny's influential work on noisy step edge detection can be reformulated as the simple pipeline `CI(1,33) | Sigmoid()`, namely a convolution followed by a sigmoidal function. Perhaps surprisingly, training automatically "discovers" the clever tricks used by Canny in the hidden layers.

- *Traditional computer algorithm view.* Neural networks can be viewed as Boolean circuits. The revival of neural networks in the 1980s was made possible by the replacement of hard nonlinearities (piecewise nondifferentiable) by soft nonlinearities (smooth functions such as sigmoid), which are trainable and can converge back to hard ones during training. This means that many traditional computer science algorithnms (e.g., shortest path in a graph) will "emerge" automatically from the DNN after it is trained.

- *Pattern recognition and traditional ML view.* Traditional machine learning such as classification and regression is subsumed by deep learning. Traditional transforms, such as PCA (Principle Component Analysis), will appear automatically in a properly designed DNN pipeline during training.

- *Signal processing view.* Traditional properties for signal processing, such as translation invariance, are obtained by training a properly designed DL pipeline. Both linear and nonlinear filters are obtained simply by properly structuring the DNN layers. For example, a convolutional layer is a linear filter and maxpooling is a nonlinear filter.

- *Decision theory view.* Consider a DL system as a black box that outputs probabilities. After training, it can provide a solution that is optimal according to Bayesian analysis.

The main problems to solve when designing a DNN are (1) the structure and features of the pipeline (the DNN layers), and (2) the training data and training approach. Given that these two are properly chosen, then training will come up with reasonable solutions that are comparable in performance to best-of-breed algorithms in each traditional view.

Furthermore, it will come up with good solutions that combine ideas from different traditional views. Is deep learning then all you need? According to Nvidia deep learning researcher Thomas Breuel, the answer is surprisingly close to a *yes*.

Deep learning as part of software engineering

Given the observations of the previous section, it is clear that the design of a deep learning system can be seen as a form of software engineering. In traditional software development, the design of data structures, algorithms, and architecture (e.g., object-oriented design) is very important. Given the increasing success and applicability of deep learning, it is clear that many applications will have learning components inside. These learning components need only a training phase to be successful, and there is now enough computing power generally available to make this possible. In effect, traditional software design is replaced by deep learning design, and the training phase is a form of *compilation*, analogous to the compilation of an application's source code.

The conclusion is that the design of deep learning components will eventually become a general subfield of software engineering. The design of a learning component requires determining the features (abstractions) to be learned, and the architecture (pipeline) of the layers of a DNN. There does not yet exist a mature methodology for this kind of design (only bits and pieces exist), but we can expect such a methodology to emerge in the near future. Ideally, it should then become part of the software engineering curriculum and be taught to all computer science students. In the context of LightKone, this points to an ever increasing use of deep learning computations, which means that we should keep alert to the addition of these computations to our edge computing model.

2 Relevance of deep learning to LightKone use cases

The LightKone use cases are presented in detail in this deliverable. Several of these use cases as presented in the deliverable have a learning component, as briefly summarized below:

- Agriculture sensing analytics (such as winery management) (Gluk): this is based on sensor data streams, which are used for predictions, recommendations, and decision making.
- Community network monitoring (Guifi): the monitoring is used for management (including decision making) and analytics.
- Petascale database with intelligent queries (Scality): this database requires pre-indexing at the edge and lambda functions at the edge, which are important in the context of content-based search, image analysis.
- Distributed online planning for automated manufacturing (Stritzinger): the planning algorithm must monitor the environment in the factory and adapt the plan accordingly, in real time.

All these examples do significant computation with a learning component. The questions to be answered in LightKone are, how should these computations be spread between an

edge network and a data center, and how can the edge computations be done efficiently? We expect that as the project advances, these use cases will evolve to use more learning, since learning technology is being successfully applied more and more in edge computing. It is clear that the use cases in this document are just a starting point, and that we will follow their development as our technology progresses.

In the following subsections, we focus on several aspects of deep learning and edge computing that cut across the use cases.

2.1 Computation model requirements for deep learning

The first requirement on the edge computing model is that it must support the inference phase of deep learning. Recently introduced edge computing devices (see the health monitoring example discussed earlier) have hardware support for these computations. The vast majority of computations done in the inference phase are of two kinds: matrix operations on numeric data vectors, and elementwise smooth nonlinear functions on data vectors. Typical nonlinear operations are softmax, tanh, and sigmoid: these are smooth (sometimes called "soft") approximations to discontinuous functions and combinations of exponentials, parametrized so that the approximation is more or less precise.

Inferencing does not need loops; it is straight line code without conditionals or loops (except for LSTMs, which contain small loops inside one layer that implement a memory). Training on the other hand has a convergence criterium and uses many optimizations to reduce computation time (implying large numbers of conditionals and loops). While the inner loop of training is straight line code like inferencing, controlling it requires a full set of operations. Training is usually embarrassingly parallel, except for RNNs (Recursive Neural Networks) and LSTMs that are iterative and therefore less parallelizable than CNNs (Convolutional Neural Networks).

We now give a formal characterization of the computation done in the inference phase. Given a DNN with n layers $\overline{a}_1$ to $\overline{a}_n$, where each $\overline{a}_i$ is a vector or matrix of real numbers. The input layer is $\overline{a}_1$, the output layer is $\overline{a}_n$, and the $n-2$ remaining layers are the hidden layers. Then the DNN is characterized by $n-1$ transformation functions f_1 to f_{n-1}, where the function f_i computes $\overline{a}_{i+1}$ given $\overline{a}_i$ and a parameter $\overline{p}_i$. The parameter is a vector or matrix that is computed during the training phase. In brief, the inference phase computes $\overline{a}_n$ given $\overline{a}_1$ according to the following rule:

$$\overline{a}_{i+1} \quad = f_i(\overline{a}_i, \overline{p}_i) \quad 1 \leq i < n$$

If all parameters $\overline{p}_i$ are fixed, then this is a pure function from $\overline{a}_1$ to $\overline{a}_n$. Since it has no internal state, computing this function does not need to use convergent computation (such as Lasp) to achieve high availability. It can be computed with traditional non-convergent dataflow techniques, for example by Naiad [34]. Because the $\overline{a}_i$ almost always have large numbers of components, this is an embarrassingly parallel computation. Any failures or stragglers can be handled by simply repeating the computation.

In practical applications (such as Apple's three-dimensional face recognition in the iPhone X), the DNN adapts to slowly changing inputs, which means that the parameters are slowly changing over time and that occasional bursts of training are needed to update them. In this case, convergent techniques may be used to keep track of parameter changes. Common choices for the functions f_i are given in Section 1.3. Typically, for early layers (low values of i) this is a (linear) convolution or maxpool, for intermediate

layers this is a nonlinear function, and for the final layer a softmax can be used if the output is a vector of probabilities.

2.2 Generalized convergence property

There is a convergence property in our edge computation model and in deep learning. We currently have a prototype edge computation model called Lasp [31]. Because of its theoretical design, Lasp naturally does convergent computing: despite intermittent failures in the underlying edge network and its nodes, the computation is guaranteed to converge to a correct result if the network satisfies a connectivity property that is obtained by the implementation's hybrid gossip layer. Failures of part of the system have no effect on correctness; their only effect is to slow down convergence. In deep learning, there is also a convergence concept. When training is done correctly, the DNN improves with time and asymptotically converges to the best DNN possible given the training data and DNN structures. Note that randomness is part of the training data. This seems to imply that training could be done on unreliable networks and still converge. One of the evolutions of Lasp that we are discussing in LightKone is the addition of probabilistic operations, which would make the correctness probabilistic but will still converge. Training a DNN seems to be an interesting example case for this extension.

2.3 Training on edge networks

As we mentioned before, in 2027 there are predicted to be more than 1000 devices per person in the generalized IoT (as estimated by Jorge Pereira at NetFutures 2017) [35]. Many of these devices will be low power. These devices and their networks will be frequently offline or off, and only active when necessary. Some of these devices could be nodes with high computational power that can be used to do occasional intensive computation on an edge network. For example, Nvidia has recently announced the Jetson TX1 and TX2 modules, which are high-performance GPUs designed for embedded systems. This means that we would not be limited to doing training in a data center and inferencing on the edge, but that we could train directly on the edge. This will become increasingly important in future systems, where *training and inferencing will be interleaved* and run on the same system, in order to continuously adapt the system to its environment. This is exactly how the human brain works, and brain research continues to inspire DL research. The current situation, in which training is a large block of computation that is done infrequently and separately from inferencing, is temporary.

2.4 Data protection and anonymization

Deep learning is being used successfully for data anonymization. The basic idea is to train the DNN using real data, and then to use the trained system to *generate* fake data with the same abstract properties as the real data. (It is a general property of DNNs that they can be used both for identification and generation of input data.) This technique has been used for generating fake patient records in healthcare applications. The fake records are indistinguishable by doctors from real records. This works because the deep learning system able to represent the high-level features of the data, and not just the superficial statistical properties. This technique could be useful for the anonymization necessary for

evaluating the LightKone use cases with realistic user data (see the discussion of data protection in this deliverable).

Bibliography

[1] CEN/TC 294. Communication systems for meters and remote reading of meters - part 3: Dedicated application layer. EN 13757-3:2013, European Committee for Standardization, 2013.

[2] CEN/TC 294. Communication systems for meters and remote reading of meters - part 4: Wireless meter readout (radio meter reading for operation in SRD bands). EN 13757-4:2013, European Committee for Standardization, 2013.

[3] Arduino. Open-source electronics platform . https://www.arduino.cc, 2017.

[4] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, December 2003.

[5] Tomas Aronsson, Johan Grafström, and Ericsson Telecom AB. A comparison between Erlang and C++ for implementation of telecom applications. *LiTH/IDA*, 05 2000.

[6] R. Baig, J. Dowling, P. Escrich, F. Freitag, R. Meseguer, A. Moll, L. Navarro, E. Pietrosemoli, R. Pueyo, V. Vlassov, and M. Zennaro. Deploying clouds in the Guifi community network. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 1020–1025, May 2015.

[7] Roger Baig, Lluís Dalmau, Ramon Roca, Leandro Navarro, Felix Freitag, and Arjuna Sathiaseelan. Making community networks economically sustainable, the Guifi.Net experience. In *Proceedings of the 2016 Workshop on Global Access to the Internet for All*, GAIA '16, pages 31–36, New York, NY, USA, 2016. ACM.

[8] Roger Baig, Ramon Roca, Felix Freitag, and Leandro Navarro. Guifi.Net, a crowdsourced network infrastructure held in common. *Comput. Netw.*, 90(C):150–165, October 2015.

[9] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8):776–787, 2012.

[10] Bhavya Banga. Global nanosatellite market anticipated to reach $6.35 billion by 2021, reports BIS research. http://www.prnewswire.com/news-releases/global-nanosatellite-market-anticipated-to-reach-635-billion-by-2021-reports-bis-research-62989 html, Jun 2017.

[11] Joshua Buck. CubeSat to demonstrate miniature laser communications in orbit. https://www.nasa.gov/press-release/cubesat-to-demonstrate-miniature-laser-communications-in-orbit, October 2015.

[12] Sarah Darby. Smart metering: what potential for householder engagement? *Building Research & Information*, 38(5):442–457, 2010.

[13] DFKI. SmartF-IT project. http://www.smartf-it-projekt.de/?lang=en, Sep 2017.

[14] E. Dimogerontakis, J. Neto, R. Meseguer, L. Navarro, and L. Veiga. Client-side routing-agnostic gateway selection for heterogeneous wireless mesh networks. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 377–385, May 2017.

[15] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–86, October 2012.

[16] EU. The Protection of Private Data Directive 95/46/EC. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:31995L0046, 1995.

[17] EU. The EU General Data Protection Regulation (GDPR). http://www.eugdpr.org, 2017.

[18] Raspberry Pi Foundation. Open-source electronics platform . https://www.raspberrypi.org, 2017.

[19] Fujitsu. MBR89R112A, ISO/IEC 15693 compliant FRAM embedded high-speed RFID. http://www.fujitsu.com/uk/Images/MB89R112A-B.pdf, Aug 2016.

[20] Fujitsu. FRAM radio frequency identity chip (RFID). http://www.fujitsu.com/uk/products/devices/semiconductor/memory/fram/rfid/, Sep 2017.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, November 2016.

[22] Rexroth Bosch Group. Transfer systems from Rexroth. https://www.boschrexroth.com/en/gb/products/product-groups/assembly-technology/material-and-information-flow-technology/material-and-information-flow-technology, Sep 2017.

[23] ISO. ISO/IEC 15693-2:2006 Identification cards – Contactless integrated circuit cards – Vicinity cards – Part 2: Air interface and initialization. https://www.iso.org/standard/43467.html, 2006.

[24] ISO. ISO/IEC 15693-3:2009 Identification cards – Contactless integrated circuit cards – Vicinity cards – Part 3: Anticollision and transmission protocol. https://www.iso.org/standard/43467.html, 2009.

[25] ISO. ISO/IEC 15693-1:2010 Identification cards – Contactless integrated circuit cards – Vicinity cards – Part 1: Physical characteristics. https://www.iso.org/standard/39694.html, 2010.

[26] ISO. ISO/IEC 27000:2016 Information technology – Security techniques – Information security management systems – Overview and vocabulary. https://www.iso.org/standard/66435.html, 2016.

[27] ISO/IEC. Intermediate system to intermediate system intra-domain routeing information exchange protoco. https://www.iso.org/standard/30932.html, 2002.

[28] Gottfried Konecny. Small satellites—tool for Earth observation? In *XXth ISPRS Congress*, pages 580–582, January 2004.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, pages 1097–1105, 2012.

[30] AWS Lambda. What is AWS Lambda? http://docs.aws.amazon.com/lambda/latest/dg/welcome.html. Accessed: 2017-09-28.

[31] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP 2015)*, pages 184–195. ACM, 2015.

[32] Microsoft. Microsoft Azure Cloud Computing Platform & Services . https://azure.microsoft.com/en-us/, 2017.

[33] Vivek Mohan. An introduction to wireless M-Bus. http://pages.silabs.com/an-introduction-to-wireless-mbus.html, June 2015. Last accessed 2017-09-28.

[34] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '2013)*, pages 439–455, November 2013.

[35] Jorge Pereira, Frank Fitzek, Ingrid Moerman, Luiz daSilva, Antonio Fuganti, and Linda Doyle. Mobile connectivity: How will the 'edge' look like in 2027? (panel discussion). In *Net Futures 2017: Conference on Internet, the economy, and society in 2027*, Brussels, Belgium, June 2017.

[36] Fang-Ching Ren, F.-S Zhang, Jian Hui Bao, Bo Chen, and Y.-C Jiao. Compact triple-frequency slot antenna for wlan/wimax operations. *Progress In Electromagnetics Research Letters*, 26, 01 2011.

[37] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag, and L. Veiga. Practical service placement approach for microservices architecture. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 401–410, May 2017.

[38] C. Stamate, G. D. Magoulas, S. Kueppers, E. Nomikou, I. Daskapoulos, M. U. Luchini, T. Moussouri, and G. Roussos. Deep learning Parkinson's from smartphone data. In *Proceedings of the 2017 IEEE International Conference on Pervasive Computing and Communications*, Kona, HI, March 2017.

[39] Phil Trinder. Comparing C++ and ERLANG for Motorola telecoms software. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 51–51, New York, NY, USA, 2006. ACM.

[40] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno M. Preguiça. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *CoRR*, abs/1310.3107, 2013.

Appendix A

List of Acronyms

API Application Programming Interface
ARM Advanced RISC Machines
CN Community Network
CPE Customer Premises Equipment
DB database
DC datacenter
EC Eventual Consistency
GNU GPL GNU General Public License
HTTP Hypertext Transfer Protocol
HTTPS HTTP over TLS
IoT Internet of Things
IP Internet Protocol
JSON JavaScript Object Notation
LDAP Lightweight Directory Access Protocol
M-Bus Meter-Bus
OS Operating System
PC Personal Comupter
REST Representational state transfer
RRD Round-Robin Database
RTEMS Real-Time Executive for Multiprocessor Systems
RTOS Real Time Operating System
RTT round-trip time
SBC Single-Board-Computer
SLA Service Level Agreement
SME Small and Medium Enterprise
TCP/IP Transmission Control Protocol (TCP) over Internet Protocol (IP)
TLS Transport Layer Security
VM Virtual Machine
VPN Virtual Private Network
VPS Virtual Private Server
WM-Bus Wireless M-Bus

Appendix B

Questionnaire

1 Overview of the use case

Provide here a short and general description of the use case including:

- The context.
- The environment description where the use case is to be deployed.
- A justification of its relationship with edge computing (*i.e,* what are the benefits that are expected by addressing this within the Lightkone project..
- What are the most common types of computations are you expecting to have in this use case.
- What are the key challenges.

2 Current development

Explain if the use case being proposed here is already materialized in an early version. If so, explain what is its current state, what are the issues that one aims at addressing. Furthermore, if the use case is already implemented fill in the following subsubsections (other wise remove them).

2.1 Conflicting operations

Are there operations that manipulate the state of your application that cannot be executed without some form of coordination?

2.2 Invariants that exist in the application state

Does the state of your application contains invariants that cannot be violated?

2.3 Performance results/figures

Are there performance metrics known (latency, throughput, ...)?

2.4 Persistence

What data needs to be persisted?

2.5 Security threats

Are there known security concerns that have to be addressed?

2.6 Current deployment details

In which environment is your application deployed? How many machines? What are there capacities/specifications? What about communication between these machines?

3 Detailed description

Describe your application in detail (the future plans).

3.1 Architecture

Network topology/architecture and complete specifications of the devices over which these are planned to execute in the future. Identify concrete technologies that are relevant to achieve a solution (either from the Lightkone consortium or external). Please add a principal network diagram to give a basic understanding of the actors, system components such as mobile devices or database servers and the communication patterns between them. I.e. synchronous calls, messaging, 1 or 2–way state synchronization.

3.2 Edge computing requirement

Why is Edge Computing required? Why would a Cloud solution not be sufficient?

4 Data model

Please provide a complete description of the data being manipulated by the application. What data objects are imutable (i.e, their value will never change since their creation) and what data objects are mutable? Are there dependencies between immutable data and mutable data? If so, identify these dependencies.

If your case study includes mutable data, please explain if transactions are required to manipulate this state (transactions in the sense of ACID properties – Atomicity, Consistency, Integrity, and Durability).

Is there any need for updating more than one object atomically in your application? Please describe the example and explain why is it important for updates to be applied atomically.

5 Detailed description of the computations

Provide a complete description of the computations that are required in the context of the application, with emphasis on computations to be executed in the edge.

Additionally, provide information about the data flow patterns of all applications, and also which parts of the system need which data (if possible)

6 Conflicting operations and invariants

Can you identify conflicting operations that can lead your application to evolve into an incorrect state? For instance, are there concurrent updates that without synchronization might break invariants?

If so, what are these invariants in the application state? For each invariant identified, should conflict or invariant violations be prohibited, or is it OK to correct them after the fact? Do invariants need to be true at all times or only eventually? If invariant violations are to be forbidden (instead of repaired), how important is for all replicas to be able to execute operation that might lead to invariant violation?

7 Divergence and divergence control

What is the expected latency? Is there an “offline” mode? Are there limits to the degree of temporary divergence allowed?

Would it be interesting to know how divergent is the data you are reading (from the data in other replicas/data centers)? Please give examples. If yes:

- What type of information would be useful - number of operations not observed, difference in the value observed, how long your data is old, some other?
- Could you live with a probabilistic value (e.g. there is 90% chance that you are not missing more than 3 operations)?

Would it be interesting to include a mechanism that guarantees that data is not divergent by more than some amount? If yes:

- What type of information would be useful - number of operations not observed, difference in the value observed, how long your data is old, some other?
- What price would you be ready to pay for it - increased latency on reads, increased latency on writes?
- Could you live with a probabilistic value?

8 Network partitions

Does natural partitions exist in the system, such that global consistency can be lowered while upholding a higher degree of consistency the partitions? For instance, if all data in

a partition is handled by the same data center, then it can benefit from reduced latency between members of the same partition, and thus increased consistency.

Another example: Tiered data–handoff, where a hierarchy is made of nodes on different levels.

9 Operational requirements

Is your application running under EC? For availability? Performance? Scalability? Where is the application running: on client-controlled mobile machines or in the infrastructure? If the latter, in a small number of data centers, or in large numbers of DCs?

What types of machines (capacity, power, ...) and through which network do they communicate. Please provide as concrete specifications for hardware as possible.

How many replicas: tens? thousands? millions? Full replication or partial replication?

What is the expected number of objects: thousands? millions? billions? Size of each object? size of the data: bytes? megabytes? gigabytes? Rate of growth? Is the object universe partitioned (i.e, composed of discrete, independent databases that can be managed/replicated independently) or (one single big database)?

10 Security requirements

Access control? at what granularity (object, operations, operations arguments, users)? Information flow control? at what granularity? Auditing and rolling back offending updates a posteriori?

11 Data protection requirements

Is there sensitive data being manipulated by your use case? Can data identify users? Reveal private information about users? Which portion of the data model fits in these cases?

12 Implementation

What programming stack are you using/planning to use? E.g. Programming language, backends, communication libraries, serialization formats. Is it open source, available for Lightkone partners, closed source?