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

D2.2: Formal Requirements

Deliverable no.:	D2.2
Title:	Formal Requirements
Due date of deliverable:	January 15, 2019
Actual submission date:	January 15, 2019
Lead contributor:	Stritzinger
Revision:	2.0
Dissemination level:	PU
Start date of project:	January 1, 2017
Duration:	36 months

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

Revision Information:

Date	Ver	Change	Responsible
06/07/2018	1.0	Submitted version	Stritzinger
15/01/2019	2.0	Submitted revised version	Stritzinger

The complete `git`-based changes tracking details for this document are available at the https://github.com/LightKone/WP2 private repository.

Contributors:

Contributor	Institution
Peter Van Roy	UCL
Igor Zavalyshyn	UCL
Bradley King	Scality
Dimitrios Vasilas	Scality
Giorgos Kostopoulos	GLUK
Kostis Kounadis	GLUK
Roger Pueyo Centelles	UPC
Felix Freitag	UPC
João Neto	UPC
Leandro Navarro	UPC
Roc Messeguer	UPC
Peer Stritzinger	STRITZINGER
Adam Lindberg	STRITZINGER
Stefan Timm	STRITZINGER
Kilian Holzinger	STRITZINGER
Peter Zeller	KL
Annette Bieniusa	KL
Ali Shoker	INESC TEC
José Proença	INESC TEC
Carla Ferreira	NOVA ID
João Leitão	NOVA ID
Nuno Preguiça	NOVA ID
Bernardo Ferreira	NOVA ID

Contents

1 Executive Summary **1**
1 Summary of Deliverable Revision 1

2 Common Edge Computing Formalization **3**
1 Introduction 3
2 Generic architecture 4
3 TLA+ 5
4 Abstract execution formalization 6
5 Timed Automata formalization 7
5.1 Specifying Timed Automata 7
5.2 Verifying Timed Automata with UPPAAL 9

3 UPC **11**
1 Introduction: Network monitoring use case 11
2 Network monitoring use case description 12
3 First TLA+ Specification: Centralised Database 14
4 Second TLA+ Specification: Distributed Database 17
5 Verification 20

4 Scality **21**
1 Use case: Multi-cloud metadata search 21
1.1 Architecture 22
1.2 API 23
1.3 Specification 23
(a) Consistency levels 25
(b) Invariants 26
1.4 Discussion 26

5 Stritzinger **27**
1 The RFID-powered conveyor – overview 27
2 Distributed state over tags and nodes 28
2.1 The distributed challenges 28
2.2 The tag structure 29
2.3 Replicas of the tag structure at the nodes 29
2.4 Optimisations 30
3 Formalisation using Timed Automata in UPPAAL 31

3.1 General configurations . . . 31
3.2 Tag Automata . . . 32
3.3 Node Automata . . . 33
3.4 Cache Automata . . . 34
3.5 Verification . . . 34

6 Gluk **37**
1 Use Case: Self-sufficient precision agriculture management for Irrigation 37
1.1 The story . . . 37
1.2 Scientific Context of the Use Case . . . 37
(a) LightKone Innovation . . . 38
1.3 Description . . . 38
(a) Actors . . . 40
(b) Requirements . . . 41
1.4 Return of the Investment . . . 41
1.5 Agriculture Market Potential . . . 43
1.6 State of the art . . . 44
2 Abstract Modeling . . . 46
3 A Use-Case Example: Irrigation Control System . . . 46
3.1 Problem Description . . . 46
3.2 DES Model of the Irrigation System . . . 47
3.3 Controlled State Trajectory . . . 49
3.4 Deriving a Discrete Event Controller . . . 50
4 Conclusion . . . 52

7 SysML Requirement Diagrams **53**
1 Introduction . . . 53
2 Requirements Analyis of Use-Cases . . . 54
2.1 Requirements Analysis: UPC . . . 54
2.2 Requirements Analysis: Scality . . . 55
2.3 Requirements Analysis: Stritzinger . . . 55
2.4 Requirements Analysis: Gluk . . . 55

Bibliography **61**

A Source code **63**
1 Guifi.net TLA+ Specification . . . 63

Chapter 1

Executive Summary

This document presents the formal requirements of the use cases selected from the large number initially identified.

- Chapter 2 describes the common generic architecture and explains how the four use cases selected from Deliverable 2.1 refine it. It also introduces TLA+ and Timed Automata, the formalisms used in the following chapters.
- Chapter 3 describes the TLA+ specification of the Guifi.net network monitoring use case. We first specify the operations for the case of the centralized database. Secondly, we present the overall structure of the TLA+ specification and the operations revised for the case of the distributed database.
- In Chapter 4 we specify the Scality use case of pre-indexing at the edge. We use a specification technique based on abstract executions here. This allows us to separate the concerns of the general application behaviour and the different consistency levels the system must support to be applicable to different deployment scenarios.
- Chapter 5 proposes an approach to make the nodes from Stritzinger's RFID-based use case communicate with each other to avoid downtimes interacting with RFID tags. The contribution is two-fold. On one hand it examines and identifies key situations where the conveyor may have to stop, and how to avoid them. On the other hand, it provides a formalisation using Timed Automata of a concrete scenario, verifying several useful properties regarding the time each operation (and the full process) can take.
- In Chapter 6 we provide a state trajectory analysis for Gluk's irrigation system use case. We review the concepts of abstract modelling and a DES modelling approach. We then explore the system states and their transition trajectory. Finally, we formulate the constraints for the system edge controller that must account for high sensor failure rate, and reason about the controller design and functionality.

1 Summary of Deliverable Revision

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

made to the deliverable are as follows:

- Use cases from UPC Chapter 3 and Scality Chapter 4 have been updated to be independent of the LightKone technologies that will be used in the development step.
- A completely new use-case has been added from Gluk.
- Requirements analysis has been added for all use cases using SysML/UML in chapter 7, the requirement analysis contains the target requirements for the evaluation phase.

Chapter 2

Common Edge Computing Formalization

1 Introduction

For the formalization, each industrial partner chose one preferred use case among the use cases in Deliverable D2.1, giving four use cases in all. When the work of analysing these use cases started, we discovered that all four share a common generic architecture, despite being introduced completely independently. In this chapter we present this generic architecture and we explain how the four use cases refine it. We also introduce the formalisms we will be using in the remainder of this report by showing how they can be used for the generic architecture.

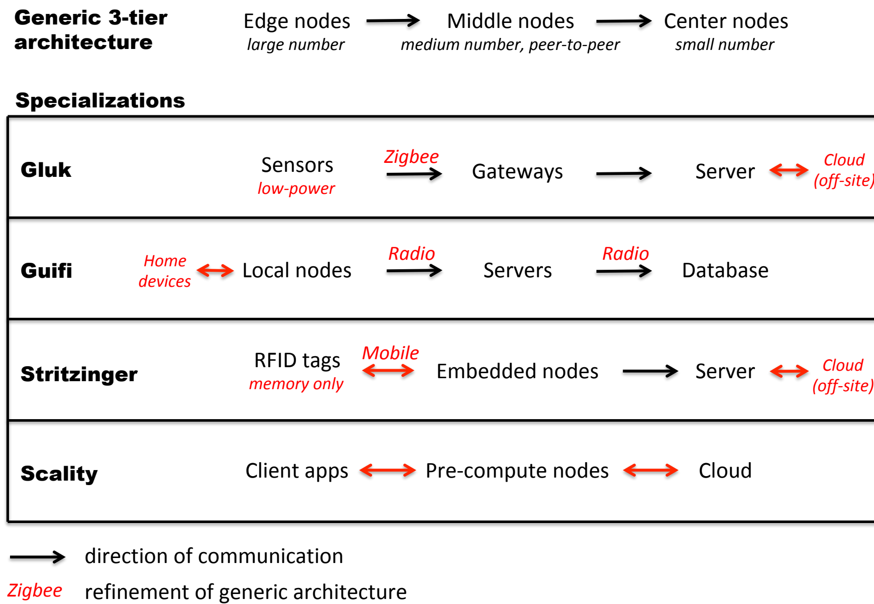

Figure 1.1: The generic edge architecture and its specializations by the four use cases.

2 Generic architecture

The generic architecture for edge computing has three tiers, namely edge nodes, middle nodes, and center nodes. Edge nodes are most numerous, with fewer middle nodes, and just a small number of center nodes. The main direction of communication is from edge to middle, and from middle to center. Edge nodes may or may not communicate with each other, while middle nodes always do. Edge nodes may or may not have computing power, while the middle nodes always do. An important task, realized most strongly in the Gluk and Guifi use cases, is data aggregation and monitoring, which is done with mostly one-way communication.

Each of the four use cases specializes the generic architecture in a different way. Figure 1.1 gives a diagram that shows the generic architecture and the four use cases. We explain individually each of the use case's specializations.

Gluk use case In the Gluk use case, sensors are low-power and mostly off; all they do is send data periodically to the middle nodes through a low-power protocol such as Zigbee. The Gluk use case is the only one of the four that monitors real-world properties (such as temperature, humidity, and CO_2 concentration). The current application is monitoring using an Azure service for analytics and display, however future extensions for doing management are planned. Management requires communication from the center toward the edge.

UPC use case In the UPC use case, based on the Guifi.net network, local nodes are specialized hardware boxes installed at each home, with a radio or fiber link to other local nodes in the vicinity and IP routing to the whole Guifi.net network nodes. All home

devices connect to this network, and to the Internet, through their local node. The first application will be monitoring certain aspects of the different local nodes (tens of thousands of them), such as availability and uptime, network traffic, etc., which will be the basis for further developments out of the scope of LightKone such as automated billing processes, smart network management, etc. To achieve this, a number of monitoring servers are scattered through all the network and concurrently access a distributed database.

Stritzinger use case In the Stritzinger use case, RFID tags have only memory and are mobile. When an RFID tag is physically close to an embedded node, there is a communication link and the RFID tag's memory can be read and written. The RFID tag is attached to a product being manufactured. It contains information on the product and its status in the manufacturing process. Information in all RFID tags must be consistent with each other and with the product. The first application is maintaining consistency in the RFID tags, even if they do not stop when close to an embedded node ("No-Stop RFID"): this means that data can be only partially read and written by the embedded node. A later application is distributed planning, which plans the tasks in the factory and is done in distributed fashion by the embedded nodes and the server. An important use for distributed planning is handling contingencies, such as processing stations being off-line.

Scality use case In the Scality use case, client apps are typically personal computers or smartphones and have significant compute power. They cache data and indexing information from the cloud database. The pre-compute nodes continuously maintain the data and index close to the client, so that the client perceives increased performance. Consistency between the pre-compute nodes and the cloud database is an important requirement.

3 TLA+

TLA+ [16] is a formal specification language based on basic set theory and predicate logic, especially well suited for writing high-level specifications of concurrent and distributed systems [15]. A TLA+ specification represents a system as a state machine, where the state of the system is an assignment of values to variables. Typically, a TLA+ specification is defined as

$$Spec \triangleq Init \land \Box[Next]_{vars} \land Liveness$$

Predicate *Init* is a formula that defines what are the acceptable initial states. From the initial state, every system transition, either leaves the state unchanged or performs a state transition as defined by the formula *Next*, which changes variables *vars* (a tuple of all state variables). The values assigned to variables on each state come either from inputs (constants), or libraries like *Naturals* or *Integers*. Note that *Spec* can generate multiple execution traces due to the number of possible initial states satisfied by *Init*, and all combinations of the possible system actions, defined in the *Next* with multiple disjunction clauses. *Liveness* is a temporal logic formula expressing the liveness properties of the system, defined as the conjunction of fairness conditions on actions.

TLC is a model checker for specifications written in TLA+. It finds all possible system behaviours, i.e, exhaustively checks all possible execution traces, and verifies if any of them violates the invariant properties such as safety and liveness. Safety properties can be described as what the system is allowed to do, while liveness properties can be described as what the system must eventually do [19]. Therefore, the TLC model checker provides a verification of the system specification and its properties [22]. The procedure used by the model checker to compute all possible behaviours uses a directed graph, whose nodes are states, and has 4 main steps:

1. Computation of all initial states, by computing all possible assignments of values to variables that satisfy *Init*;

2. For every state found in step 1, compute all possible next states by substituting the values assigned to variables, with the actions defined in the *Next* formula;

3. For every state found in step 2, if it is not already in the graph, it is added, and an edge is drawn from the state that generated it, to it.

4. Steps 2 and 3 are repeated until no new states or edges can be added to the graph.

When this process terminates, the nodes of the graph correspond to all the reachable states of the specification. The process either ends with a state that is supposed to happen and marks the end of the execution, or with a deadlock, a state from which there is no next state satisfying the *Next* formula, but corresponds to a situation not supposed to happen.

TLA+ has been increasingly use by projects in the industry for the ability to explore unexpected combinations of events, verify complex designs, make innovative performance optimisations, but mainly because it allows developers to precisely check the safety of a system, and therefore avoid serious bugs from reaching production.

4 Abstract execution formalization

Clients typically cannot observe the internal ordering of events and messages in a system. They can only observe when their operations are started and when and what they return. The history of operation invocations (including times and results) is called a concrete execution. From a client perspective, there can in general be multiple orderings of all system events, that would explain a given concrete execution. The explanation with the ordering of all events is called an abstract execution.

To specify a system using abstract executions, one has to specify conditions for the result values of an abstract execution and what constraints exist on the ordering of events in the system. Both can be specified using standard mathematical and logical notations (we mainly use first order logic in this document). The latter is usually called the consistency model. This technique of specification is described in detail in [9]. We only give a very brief overview here. Formally, an abstract execution consists of:

- A set E of events.

- A labeling *op*, which defines the operation (including arguments) for each event in E.
- A labeling *rval*, which defines the return value for every event in E.
- A partial order *rb* (returns before) on E. We write $e_1 \prec_{rb} e_2$ iff event e_1 returned a value before e_2 was started.
- An equivalence relation *ss* (same session) over E. This relation depends on how clients interact with the system. Typically, two events are in the same session, if they came from the same client using the same connection. Based on *ss*, we can also define a session order $so \stackrel{\text{def}}{=} rb \cap ss$.
- A relation *vis* (visibility) on E. We write $e_1 \prec_{vis} e_2$ if the effect of event e_1 is visible when execution event e_2.
- A total order *ar* (arbitration) on E, which can be used to order concurrent events to get a deterministic specification.

Using these elements, the system behaviour is specified. Everything that is nondeterministic from a client's perspective is captured in the relations *vis* and *ar*. A consistency model defines restrictions on *vis* and *ar*, which have to be respected by the system. This means, an implementation is correct, if for each concrete execution one can find relations *vis* and *ar* to get an abstract execution that explains the results of the concrete one.

5 Timed Automata formalization

Timed automata [4, 6, 8] is one of the most widely used formal models to specify and verify real-time systems. Using timed automata, one can model and verify a system whose behaviour is influenced by the passage of time. Formally, a timed automaton is a finite automaton extended with a set of dedicated real-valued variables called *clocks*. Transitions are labelled by clock-constraints, i.e., constraints defined on clock variables, restricting the behaviour of an automaton, and states are labelled with other clock-constraints specifying invariants. All clocks of an automaton are initially set to zero and increase synchronously whenever time evolves. Transitions may additionally reset selected clocks when being taken.

5.1 Specifying Timed Automata

Timed automata are labelled transition systems enriched with constraints over so-called *clocks*. A clock is a special variable capturing the time passed since it was last reset.

Definition 1 (Clock Constraint)**.** *A clock constraint g over a set of clocks C, written $g \in CC(C)$, is given by the grammar below, where $n \in \mathbb{N}$, $x,y \in C$ and $\odot \in \{>,\geqslant,=,<,\leqslant\}$.*

$$g ::= true \mid x \odot n \mid x - y \odot n \mid g \wedge g,$$

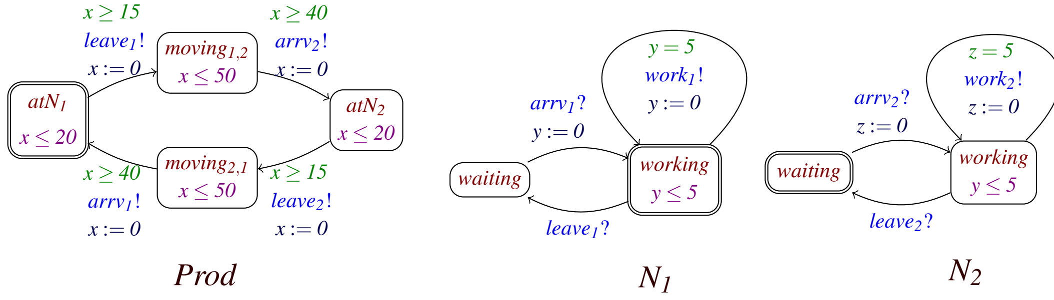

Figure 5.1: Network of three timed automata, modelling a product (left) and two nodes (right), communicating via the actions $leave_i$ and $arrv_i$.

Definition 2 (Timed Automata)**.** *A timed automaton is a tuple $\langle L, \ell_0, \Sigma, C, T, Inv\rangle$ where L is a finite set of locations, $\ell_0 \in L$ is the initial location, Σ is a finite alphabet of actions, C is a set of clocks, $T \subseteq L \times CC(C) \times \Sigma \times 2^C \times L$ is the set of transitions, $CC(C)$ denotes the set of all clock constraints over C, and $Inv : L \to CC(C)$ assigns invariants to locations.*

Informally, a connector in a location ℓ can evolve either by (1) letting time pass without leaving its location, by incrementing all its clocks without breaking the invariant $Inv(\ell)$, or by (2) taking a transition $\ell \xrightarrow{g,a,C} \ell'$ if the conditions g and $Inv(\ell')$ hold, going to the location ℓ', setting the clocks in C to zero, and leaving the remaining clocks unchanged (i.e., time does not pass when performing actions).

The actions in the alphabet Σ are used to synchronise with other automata. More precisely, two automata with a shared action $a \in \Sigma$ are only allowed to take a transition labelled with a when the other automata can also take a transition with a. A set of automata running in parallel, synchronising actions and evolving their clocks simultaneously, is called a *network of timed automata*. We follow the convention that action synchronisation can only occur in pairs, and we use their notation $a!$ and $a?$ to mean that performing $a!$ triggers $a?$ to be performed [7]. Furthermore, we omit clock constraints, actions, and reset sets from the labels whenever they are *true*, irrelevant, and $\emptyset$, respectively. This is illustrated in the example below with a network of two timed automata.

Example 1. *We depict in Figure 5.1 a network of two simple timed automata, where angle brackets $\langle\cdot\rangle$ denote guards on edges, and initial states have double lines.*

Initially all the first and last automata are in their left locations and the middle one is in its right location, and their clocks x, y, z are set to 0. The leftmost automaton cannot be longer than 20 time units in its initial location, and it can only perform a $leave_1$ action after at least 15 time units. Once it performs this action, it will move to the $moving_{1,2}$ location, where it will stay between 40 and 50 time units. The middle automaton will synchronise with the $leave_1$ action and will move into a waiting location, where it will stay until edge $arrv_1$ synchronise with the leftmost automaton. More generally, the first automaton describes the time to stay at each node and the time to move between nodes, and the other two describe the time needed to perform working actions, e.g., read or write blocks of data to the tag in the node.

We use the UPPAAL model checker to specify and verify timed automata [6, 8]. UPPAAL supports extra extensions that ease the formalisation, including notions such as

committed states, *internal variables*, and *parametric actions*, not described here.

5.2 Verifying Timed Automata with UPPAAL

This section presents a temporal logic—*Timed Computation Tree Logic* (TCTL) [4, 7]—, used by UPPAAL to describe desired properties of (networks of) timed automata. This logic consists of path formulas ϕ, which in turn use more dedicated state formulas ψ. State formulas are defined over automata locations and clocks.

Definition 3 (TCTL formulas). *A TCTL formula ϕ is given by the grammar below.*

$$\phi \ ::= \ \exists\Diamond\,\psi \mid \forall\Diamond\,\psi \mid \exists\Box\,\psi \mid \forall\Box\,\psi \mid \psi_1 \to \psi_2$$
$$\psi \ ::= \ A.\ell \mid g \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \psi_1 \Rightarrow \psi_2$$

$A.\ell$ represents the location ℓ in the automaton A, g is a clock constraint, and $\neg$, $\vee$, $\wedge$ and $\Rightarrow$ represent the usual logical negation, disjunction, conjunction, and implication. The temporal operators $\exists$, $\forall$, $\Diamond$, and $\Box$ describe the range of states for which the state formulas ψ must hold, and $\psi_1 \to \psi_2$ is a shorthand for $\forall\Box\,(\psi_1 \Rightarrow \forall\Diamond\,\psi_2)$ (which cannot be written in our syntax), read ψ_1 leads to *ψ_2.*

We explain some of these temporal operators using the timed automata in Figure 5.1.

- $\exists\Diamond\,\psi$ means that there must *exist* a sequence of transitions where ψ holds at *some point*.
Example: $\exists\Diamond\,N_1.working \Rightarrow (x > 17)$ – the clock x can become higher than 17 while in location *working* in N_1.

- $\forall\Diamond\,\psi$ means that *for every* sequences of transitions, ψ *can hold* at some point.
Example: $\forall\Diamond x > 17$ – at any given point of the execution, one can find a future state where x is higher than 17.

- $\exists\Box\,\psi$ means that there must *exist* a sequence of transitions such that ψ *always holds*.
Example: $\exists\Box\,(x = 11) \Rightarrow N_1.waiting$ – it is possible that the clock x reaches the value 11 only when N_1 is in its *waiting* location, or that x never reaches 11.

- $\forall\Box\,\psi$ means that *for every* sequence of transitions, ψ *must always hold*.
Example: $\forall\Box\,(x \geq 0 \wedge x \leq 40)$ – the clock x will always be within 0 and 40.

- $\psi_1 \to \psi_2$ means that whenever ψ_1 holds then ψ_2 *must hold* at some point.
Example: $N_1.working \to Prod.moving_{1,2}$ – every time $N_1.working_1$ is reached, it will eventually reach $Prod.moving_{1,2}$ at some point in time.

UPPAAL notation. In UPPAAL expressions and properties are written in plain ASCII. Our representation follows closely UPPAAL's one, writing: '⇒' instead of '`imply`', '→' instead of '`-->`', '∧' instead of '`and`', '∨' instead of '`or`', '∀' instead of '`A`', '∃' instead of '`E`', '□' instead of '`[]`', and '◊' instead of '`<>`'.

Chapter 3

UPC

1 Introduction: Network monitoring use case

The current monitoring system for Guifi.net's network devices is built around a centralised database that lists all the devices in the network and assigns them among a number of servers, which are geographically spread all over the network and take care of monitoring the infrastructure. Each monitoring server periodically fetches its assigned devices list from the central database; then it checks for their status, gathers information about their network interfaces, etc. The assignation of a network device to its monitoring server is mostly static and rarely changes over time. All the information collected stays local to the different monitoring servers, not being automatically replicated or distributed between them or elsewhere. This current system fulfils most of the basic monitoring needs but 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:

- No redundancy: every device is monitored by a single server
- No load balancing between monitoring servers
- No automatic detection of server failures and reassignment of devices to another monitoring server
- No replication or distribution of collected data
- Need for manual intervention: devices usually require to be manually assigned to a specific monitoring server

UPC's use case reimplements the current monitoring system to improve its resilience and reliability by means of automation, distribution and decentralisation. It will leverage distributed data structures to support the distributed coordination of monitoring servers in performing the assignation of network devices between them in a decentralized fashion. The monitoring servers keep a distributed *monitoring servers* $\Leftrightarrow$ *network devices* mapping which they use to dynamically assign (and unassign) themselves which devices to monitor in function of different criteria. Such dynamic mapping is to be concurrently modified by any of the participating servers, incurring in operations potentially leading

to inconsistent data. To overcome this, bringing the CRDTs technology into the application would allow delegating data synchronisation and consistency to the underlying data storage level, ensure certain properties (namely, consistent data replication) to the upper layers.

The main computations of the application consist in editing mapping between the two sets, network devices and monitoring servers, while keeping it in a consistent status and ensuring that every device is being actively monitored by a predefined minimum number of servers at any time.

2 Network monitoring use case description

Figure 2.1 provides a high-level overview of the monitoring system. A number (n) of *monitoring servers*, drawn on the top left corner, are the primary actors that perform actions within the system. These servers are composed by two items, the *assignment* and the *monitoring* components. The monitoring service has functional requirements regarding what a monitoring system should provide, which consist in a set of read and write operations to be carried out by the different primary and secondary actors. The required services operate on top of a distributed storage service.

The *assignment* component within each monitoring server carries out the mapping of the networking devices assigned to itself. For this, the assignment component shall read the current data of the assignments of all monitoring server from a distributed storage service in order to obtain the latest assignments. In order to take precise decisions, the assignment component shall obtain consistent data from the distributed storage service. The assignment component shall write the new assignment data of its own assignment, decided by itself, to the distributed storage service in order to communicate to the other monitoring server its latest assignment decisions taken. The decision that an assignment component takes on its updated assignment can be extended by taking into account contextual information and additional decision support functions.

The *monitoring* component reads the monitoring data from the networking devices (e.g., routers) and stores them in the distributed storage service. For this, the monitoring component shall read the current monitoring data from the network device in order to be able to store this data in the distributed storage service. Then, the monitoring component shall write the monitoring data to the distributed storage service. If several data sets are taken for the same networking device, the data to be stored shall be aggregated or merged.

Other actors are the Guifi.net database (Guifi DB), which supports the initialization of the assignment. For this the Guifi DB shall provide the information about the available monitoring servers and network devices to be written to the distributed storage service. Information about the network state and about network devices can feed contextual information. A distributed database shall provide the storage service to support the different read and write actions.

With regards to the features enumerated in the previous section on the existing monitoring system, its limitations are clear and the improvements of the new implementation target to overcome them, leading to achieve the following non-functional requirements:

- Automated assignment: no need for manual intervention to assign monitoring servers to network devices. The system shall do the assignment automatically. In a perma-

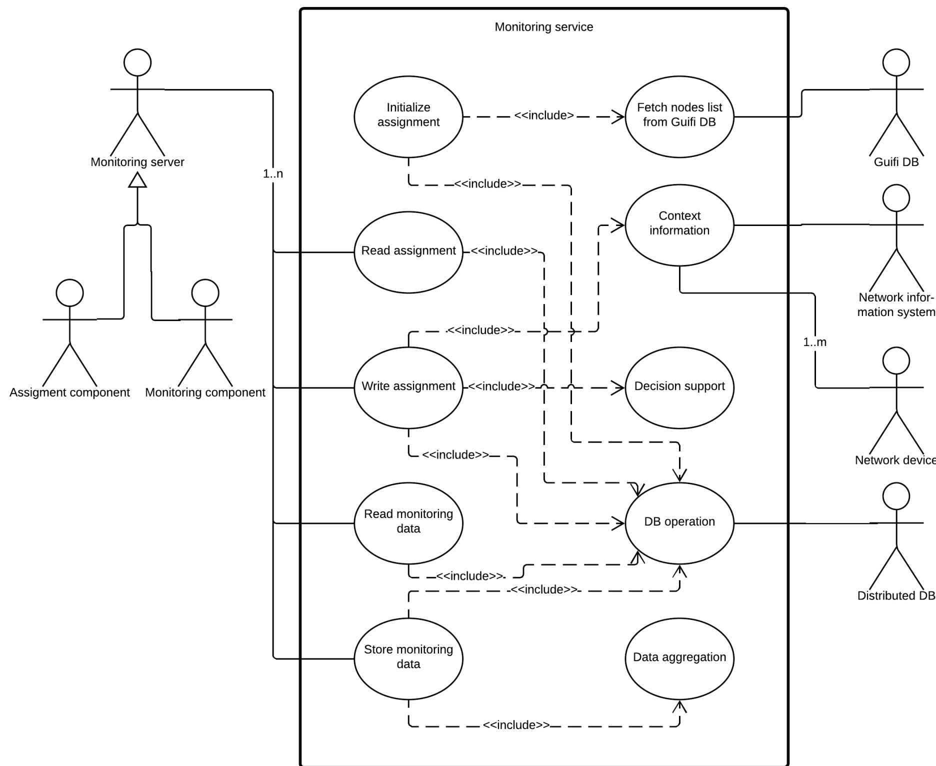

Figure 2.1: Monitoring system from the perspective of a UML use case diagram.

nent operation these services should run autonomously without manual intervention.

- Automated reconfiguration: automatic detection of server failures and reassignment of network devices to another monitoring server. The system shall carry out the reconfiguration automatically. Each monitoring server responds to the situation of the assignment of monitoring servers to network devices and takes a decision to compensate failures of servers or network devices.

- Redundancy: every networking device is monitored by a several servers (i.e., the monitoring servers check which network devices are less monitored and decide autonomously if they become a monitor for this device).

- Load balancing between servers: assignment decisions take into account server load. The load of the monitoring service on a monitoring server should not significantly affect the user experience if the device is used for multiple services.

- Data replications: the collected data are replicated or distributed. In the case of network partition or churn of some storage nodes the data should still be available for being retrieved by the monitoring service.

The fulfillment of the requirements by the new monitoring system implementation will be evaluated by performance metrics, the values of which will be positioned with

respect to evaluation targets or evaluation criteria to assess its success. The evaluation targets/criteria for the success of the new implementation will be determined by high-level aspects, and not by individual low-level technical performance metrics (see the corresponding chapter to UPC's use case in Guifi.net in the deliverable D7.1). The reason is that what the Guifi community network from the new monitoring service in practice needs is the reliable operation with improved usability, while tiny performance variations will typically not be the critical path for the success in the practical usage of the new monitoring system implementation.

We expect, however, that by the determined performance metrics and the measurements to be done in the experimental deployments through the tasks of WP7, also low-level performance characteristics of the network monitoring service will be obtained. The results can bring valuable feedback to the research on the distributed storage service design. The specific performance metrics and evaluation targets/criteria we expect to apply in the experimental evaluation of the network monitoring use case implementation are further explained in deliverable D7.1.

3 First TLA+ Specification: Centralised Database

The specification presented in this section is an abstraction of the case study implementation. In particular, it abstracts the main cause of complexity of the monitoring system, the fact that the system uses a geo-replicated distributed database. Instead, we are going initially to assume a shared centralised database. This simplified setting abstracts away all the complexities raised by distribution and is the ideal setting for defining the main functionalities and the global invariants of the monitoring system.

The specification includes the following constants: (*i*) SERVERS, the set of servers; (*ii*) DEVICES, the set of devices; (*iii*) MINRATE, the minimum monitoring device rate; (*iv*) MAXCAPACITY, that maximum monitoring capacity for each server; (*v*) MAXFAILURES, the maximum number of crash failures the system is able to withstand. To ensure that each device is monitored by at least MINRATE monitoring servers, a restriction is needed to limit the number of simultaneous crash failures that can occur (# is the cardinality operator):

$$(\#\text{SERVERS} - \text{MAXFAILURES}) * \text{MAXCAPACITY} \geq \#\text{DEVICES} * \text{MINRATE}$$

Beyond this limit the system cannot ensure the monitoring rate for all devices. Two simplifications are assumed: a server's maximum monitoring capacity and a device's minimum monitoring rate are the same for all servers and devices, respectively. These simplifications could easily be avoided, but they would make the specification unnecessarily complex while not bringing any relevant contribution to the verification of the system properties. Load balancing between servers is ensured by the constant MAXCAPACITY that limits the number of devices monitored by each server. More sophisticated load strategies could be specified, but this is the approach followed in the current implementation.

Figure 3.1 gives an overview of the Guifi.net TLA+ specification. The state represents a centralised database and is modelled by two variables, the *monitorisation* relation and the set of *crashed* servers. The *monitorisation* relation is a set of monitoring pairs of

$$
\begin{array}{lll}
vars & \triangleq & \langle monitorisation, crashed \rangle \\
Monitor & \triangleq & [server : \text{SERVERS}, device : \text{DEVICES}] \\
TypeInv & \triangleq & \land\ monitorisation \subseteq Monitor \\
 & & \land\ crashed \subseteq \text{SERVERS} \\
Init & \triangleq & \land\ monitorisation = \exists\, pairs \subseteq Monitor\ : \\
 & & \quad \land\ \forall\, d \in \text{DEVICES} : \#servers(d, pairs)\ \geq\ \text{MINRATE} \\
 & & \quad \land\ \forall\, s \in \text{SERVERS} : \#devices(s, pairs)\ \leq\ \text{MAXCAPACITY} \\
 & & \land\ crashed = \{\} \\
Next & \triangleq & \exists\, s \in \text{SERVERS}, d \in \text{DEVICES} : revoke(s,d) \lor assign(s,d) \lor crash(s) \\
Resilience & \triangleq & \forall\, d \in \text{DEVICES} : \#servers(d, monitorisation) \geq \text{MINRATE} \\
LoadBalacing & \triangleq & \forall\, s \in \text{SERVERS} : \#devices(s, monitorisation)\ \leq\ \text{MAXCAPACITY} \\
Inv & \triangleq & Resilience \land LoadBalacing \\
Spec & \triangleq & Init\ \land\ \Box[Next]_{vars}\ \land\ \Box Inv
\end{array}
$$

Figure 3.1: Overall structure of TLA+ specification of Guifi.net monitoring system (global database).

type *Monitor*, where the first element is a *server* and the second element is a *device*. The predicate *Init* initialises all variables. In particular the *monitorisation* relation is assigned a nondeterministically chosen set of monitoring pairs that satisfy the minimum device rate and maximum server capacity. The initialisation uses functions *devices* and *servers* not presented here. Function *devices* determines the set of devices monitored by a server. The dual function *servers* determines the set of servers that are monitoring a device. The *Next* action describes all possible system actions. Namely, *revoke* (resp. *assign*) updates the monitorisation relation by removing (resp. adding) a device from a server monitorisation.

The invariant *Resilience* expresses that all devices are monitored by at least MINRATE servers, while invariant *LoadBalacing* expresses that the number of devices monitored by any server does not exceed the maximum server capacity defined as MAXCAPACITY.

The formula *Spec* describes the overall behaviour of the system: it starts in one of the acceptable initial states defined by *Init*, and every state transition either leaves the state unchanged or performs one of the actions defined in *Next*. Moreover, every system state satisfies the safety invariants *Resilience* and *LoadBalacing*.

Next, we describe the system actions. Action *assign* in Figure 3.2 represents the assignment of a device to a monitoring server. The action is enabled, or executable, if the three following conditions hold: the server *srv* is not crashed, device *dvc* is not already monitored by server *srv*, and server *srv* has capacity to monitor one more device. The execution of the action will update the *monitorisation* relation by adding this new monitoring pair, while the *crashed* set stays unchanged.

Action *revoke* in Figure 3.3 unassigns device *dvc* to monitoring server *srv*. The action is enabled if device *dvc* is actually monitored by server *srv* and the device is currently monitored by at least MINRATE + 1 servers. This last condition ensures that after the unassignment the device remains monitored by sufficient servers to satisfy the *Resilience* invariant. The execution of *revoke* will update the *monitorisation* relation by removing

$$
\begin{aligned}
\mathit{assign}(\mathit{srv}, \mathit{dvc}) \triangleq\ & \texttt{(* preconditions *)} \\
& \land\ \mathit{srv} \notin \mathit{crashed} \\
& \land\ [\mathit{server} \mapsto \mathit{srv}, \mathit{device} \mapsto \mathit{dvc}] \notin \mathit{monitorisation} \\
& \land\ \#\mathit{devices}(\mathit{srv}, \mathit{monitorisation}) < \textsc{MaxCapacity} \\
& \texttt{(* update system state *)} \\
& \land\ \mathit{monitorisation}' = \mathit{monitorisation} \cup \{[\mathit{server} \mapsto \mathit{srv}, \mathit{device} \mapsto \mathit{dvc}]\} \\
& \land\ \textsc{Unchanged}\langle \mathit{crashed} \rangle
\end{aligned}
$$

Figure 3.2: TLA+ specification of action *assign*.

this monitoring pair, but does not modified the *crashed* set.

$$
\begin{aligned}
\mathit{revoke}(\mathit{srv}, \mathit{dvc}) \triangleq\ & \texttt{(* preconditions *)} \\
& \land\ [\mathit{server} \mapsto \mathit{srv}, \mathit{device} \mapsto \mathit{dvc}] \in \mathit{monitorisation} \\
& \land\ \#\mathit{servers}(\mathit{dvc}, \mathit{monitorisation}) \geq \textsc{MinRate} + 1 \\
& \texttt{(* update system state *)} \\
& \land\ \mathit{monitorisation}' = \mathit{monitorisation} \setminus \{[\mathit{server} \mapsto \mathit{srv}, \mathit{device} \mapsto \mathit{dvc}]\} \\
& \land\ \mathit{crashed}' = \mathit{crashed}
\end{aligned}
$$

Figure 3.3: TLA+ specification of action *revoke*.

Action *crash* represents a server crash (see Figure 3.4). In this first specification version, devices being monitored by the server that crashed are immediately reassigned to other remaining active servers. The action is enable if the server is not already crashed and the maximum number of allowed crashes has not been reached.

The reassignment is described in the *let* expression of Figure 3.4. It selects non-deterministically an assignment of the devices monitored by the crashed server to the remaining "live" servers. This non-deterministic reassignment guarantees that the monitoring rate of the devices is kept above the defined threshold, and that the capacity of "live" servers is not exceeded.

$$
\begin{array}{lll}
crash(srv) & \triangleq & \texttt{(* preconditions *)} \\
& & \land\ srv \notin crashed \\
& & \land\ \#crashed < \textsc{MaxFailures} \\
& & \texttt{(* update system state *)} \\
& & \land\ crashed' = crashed \cup \{srv\} \\
& & \land\ \textsc{let}\quad devs \triangleq devices(srv, monitorisation) \\
& & \qquad assignment \triangleq \{p \in monitorisation : p.server = srv\} \\
& & \qquad live \triangleq \textsc{Servers} \setminus crashed' \\
& & \qquad reassignment \triangleq \\
& & \qquad\quad \exists\, pairs \subseteq [server \in live, device \in devs] : \\
& & \qquad\qquad \land\ \forall\, d \in devs : \#servers(d, pairs) \geq \textsc{MinRate} \\
& & \qquad\qquad \land\ \forall\, s \in live : \#devices(s, monitorisation) \leq \textsc{MaxCapacity} \\
& & \quad \textsc{let}\quad monitorisation' = (monitorisation \setminus assignment) \cup reassignment
\end{array}
$$

Figure 3.4: TLA+ specification of action *crash*.

4 Second TLA+ Specification: Distributed Database

This second specification considers that each server has its own replicated database. In this setting, local operations are applied immediately to the server local database and propagated in the background to other servers. Moreover, it is assumed that no messages are lost and that remote operations are executed following a causal order.

Because there is no centralised global database, when executing an action within a server the action preconditions are checked using local information. A consequence is that several servers may decide concurrently to revoke the same device, because locally each action preconditions holds, thus breaking the global *Resilience* invariant. To maintain the invariant at all times it would be sufficient to use a *bounded counter* [5], a replicated abstract data type that enforces numeric invariants while avoiding most coordination. The key idea is to distribute *rights* to revoke a device between all servers. If a server has the *right* needed to execute a revoke action, the action can safely execute locally, knowing that the global invariant will not be broken. However, the special complexity of using bounded counters in the case study implementation is prohibitive, since each counter has a quadratic complexity on the number of servers and its necessary to have a counter per device. An alternative is to relax the *Resilience* property and allow this invariant to be temporarily broken. Assuming that the propagation time of remote operations is low, detection and recovery of the invariant violation should be fast. The TLA+ specification presented in this section checks the conditions under which the invariant is ensured to be recovered.

Figure 4.1 gives an overview of the revised Guifi.net TLA+ specification. The system specification has three variables, the configuration, the propagated messages, and the set of crashed servers. The configuration keeps the local state of each server, which is a record where the first element is the monitorisation relation and the second element is the server vector clock. The variable *msgs* is a function that keeps for each server the set of remote operations (messages) still to be executed. Messages propagated include all

necessary information to execute a remote operation: the local server, the device, and the type of action. The message vector clock is necessary to ensure that remote operations are causally executed. The predicate Init initialises all variables. The configuration is initialised such that all servers have the same monitorisation record, and the the vector clocks start at zero. Also, initially there are no messages and the set of crashed servers is empty.

Recovery is a temporal property that expresses that if the *Resilience* is broken in the current state, then eventually *Resilience* will hold again in the future. When checking temporal formulas it is necessary to include assumptions about the system's environment.

$$
\begin{array}{lll}
vars & \triangleq & \langle configuration, msgs, crashed \rangle \\
VectorClock & \triangleq & [\text{SERVERS} \rightarrow Nat] \\
State & \triangleq & [monitorisation : \mathbb{P}(\text{SERVERS}), vclock : VectorClock] \\
RemoteOp & \triangleq & [server : \text{SERVERS}, device : \text{DEVICES},\ op : \{\texttt{revoke}, \texttt{assign}\}, \\
 & & \ vclock : VectorClock] \\
TypeInv & \triangleq & \land\ configuration \in [\text{SERVERS} \rightarrow State]\ \land\ msgs \in [\text{SERVERS} \rightarrow \mathbb{P}(RemoteOp)] \\
 & & \land\ crashed \subseteq \text{SERVERS} \\
Init & \triangleq & \land\ configuration = [srv \in \text{SERVERS} \mapsto \\
 & & \qquad\qquad [monitorisation \mapsto initial, vclock \mapsto [s \in \text{SERVERS} \mapsto 0]]] \\
 & & \land\ msgs = [s \in \text{SERVERS} \mapsto \{\}] \\
 & & \land\ crashed = \{\} \\
Resilience & \triangleq & \forall\, d \in \text{DEVICES} : \#servers(d, monitorisation) \geq \text{MINRATE} \\
Recovery & \triangleq & \neg Resilience \Rightarrow \Diamond Resilience \\
Fairness & \triangleq & \forall\, s \in \text{SERVERS}, d \in \text{DEVICES} : \text{WF}_vars(assign(s,d)) \land \text{WF}_vars(remote(s,d)) \\
Spec & \triangleq & Init\ \land\ \Box[Next]_{vars}\ \land\ Fairness\ \land\ \Box Recovery
\end{array}
$$

Figure 4.1: Overall structure of TLA+ specification of Guifi.net monitoring system (distributed database).

Figure 4.1 gives an overview of the revised Guifi.net TLA+ specification. The system specification has three variables, the configuration, the propagated messages, and the set of crashed servers. The configuration keeps the local state of each server, which is a record where the first element is the monitorisation relation and the second element is the server vector clock. The variable *msgs* is a function that keeps for each server the set of remote operations (messages) still to be executed. Messages propagated include all necessary information to execute a remote operation: the local server, the device, and the type of action. The message vector clock is necessary to ensure that remote operations are causally executed. The predicate Init initialises all variables. The configuration variable is initialised such that all servers have the same monitorisation record (set *initial* not shown here), and the vector clocks start at zero. Also, initially there are no messages and the set of crashed servers is empty. As said at the beginning of this section, the *Resilience* invariant is going to be relaxed so it can be temporarily broken. This is expressed by the recovery *Recovery* property, which is a temporal formula stating that if the *Resilience* is broken in the current state, then eventually *Resilience* will hold again in the future. When

checking temporal formulas it is necessary to include assumptions about the system's environment. In the specification of Figure 4.1 it is assumed weak fairness, that says that when an action is continuously enabled, the action will be eventually executed. In TLA+ weak fairness means that the system is permitted only a finite number of stuttering steps (a step that leaves all variables unchanged) before executing an action. In our specification the system has to choose to execute actions *remote* and *assign* action regularly.

Next we present the *assign* action (Figure 4.2), while dual action *revoke* is presented in Appendix 1 where the complete TLA+ specification is listed. The *let* expression starts by accessing the local state of server *srv* and follows by defining the updated vector clock. The new vector clock only increments the server logical vector by one, while all other servers logical clocks remain unchanged. The *monitorisation* relation is updated by adding a new monitoring pair. Then the new local state is defined as a record with the updated monitorisation relation (*new_mt*) and vector clock (*new_vc*). Also, a message describing the action being executed is prepared. If the action preconditions hold, the local state of the server evolves and the action just executed is propagated to other servers. The message propagation is represented by adding message *op* in the message set of other servers. The set of crashed server remains unchanged.

$$
\begin{array}{l}
assign(srv, dvc) \triangleq \\
\quad \text{LET} \quad \begin{array}{rcl}
state & \triangleq & configuration[srv] \\
new_vc & \triangleq & [state.vclock \text{ EXCEPT } ![srv] = @ + 1] \\
new_mt & \triangleq & state.monitorisation \cup \{[server \mapsto srv, device \mapsto dvc]\} \\
new_state & \triangleq & [monitorisation \mapsto new_mt, vclock \mapsto new_vc] \\
op & \triangleq & [server \mapsto srv, device \mapsto dvc, op \mapsto \texttt{assign}, vclock \mapsto new_vc]
\end{array} \\
\quad \text{IN} \quad \texttt{(* preconditions *)} \\
\qquad \land [server \mapsto srv, device \mapsto dvc] \notin state.monitorisation \\
\qquad \land \#devices(srv, state.monitorisation)) < \text{MAXCAPACITY} \\
\qquad \texttt{(* update server local state *)} \\
\qquad \land configuration' = [configuration \text{ EXCEPT } ![srv] = new_state] \\
\qquad \land msgs' = [s \in \text{SERVERS} \mapsto \text{IF } s = srv \text{ THEN } msgs[s] \text{ ELSE } msgs[s] \cup op] \\
\qquad \land crashed' = crashed
\end{array}
$$

Figure 4.2: TLA+ specification of action *assign*.

Action *remote*, presented in Figure 4.3, describes the execution of a remote operation. The remote action to be executed is non-deterministically selected from the set of pending remote action (messages) that satisfy the causality relation. So, there is no pending remote action that happened before the action just picked. As in the previous action, the *let* expression starts by accessing the local state of server *srv* and updating the vector clock. The *monitorisation* relation is updated according the type of remote action being executed locally. If the remote action was an *assign*, then a new monitoring pair is added to the local *monitorisation* relation. Otherwise, it removes a monitoring pair from the local *monitorisation* relation. Finally, the local state of the server evolves, and the remote action just executed its removed from its pending remote actions. The set of crashed server remains unchanged.

$$
\begin{array}{l}
remote(srv) \triangleq \\
\quad \exists\, op \in msgs[srv] : \\
\quad \text{LET } state \triangleq configuration[srv] \\
\qquad new_vc \triangleq [s \in \text{SERVERS} \mapsto \text{IF } s = srv \text{ THEN } state.vclock[s] + 1 \\
\qquad\qquad\qquad \text{ELSE } max(state.vclock[s], op.vclock[op.server])] \\
\qquad new_mt \triangleq \text{IF } op.operation = \texttt{assign} \\
\qquad\qquad \text{THEN } state.monitorisation \cup \{[server \mapsto srv, device \mapsto dvc]\} \\
\qquad\qquad \text{ELSE } state.monitorisation \setminus \{[server \mapsto srv, device \mapsto dvc]\} \\
\qquad new_state \triangleq [monitorisation \mapsto new_mt, vclock \mapsto new_vc] \\
\quad \text{IN } \texttt{(* preconditions *)} \\
\qquad \land\ \forall\, op1 \in msgs[srv] : \neg happenbefore(op1.vclock, op.vclock) \\
\qquad \texttt{(* update server local state *)} \\
\qquad \land\ configuration' = [configuration \text{ EXCEPT } ![srv] = new_state] \\
\qquad \land\ msgs' = [msgs \text{ EXCEPT } ![srv] = msgs[srv] \setminus \{op\}] \\
\qquad \land\ crashed' = crashed
\end{array}
$$

Figure 4.3: TLA+ specification of action *remote*.

5 Verification

Both specifications where verified with model checker TLC. For the first specification a small model with three servers and five devices was defined and relevant invariants where checked in a few milliseconds. For the second specification, modelling a distributed database, it was necessary to impose restrictions on the number of actions executed to be able to check the *Recovery* invariant. The question is that messages (remote operations) can be read in many different orders, which leads to a state explosion. By restricting the number of operations to a small number, TLC was able to check the *Recovery* invariant in three minutes.

Chapter 4

Scality

1 Use case: Multi-cloud metadata search

Scality's open source multi-cloud framework, Zenko [1], enables applications to transparently store and access data on multiple public and private cloud storage systems using a single storage interface. Applications can use Zenko to access multiple cloud storage systems, including cloud services such as Microsoft Azure Blob Storage, Amazon S3 and Google Cloud, as well as private on-premise storage systems, using an Amazon S3 compatible API.

The focus of this use case is Zenko's capability to support federated metadata search across multiple cloud namespaces. This enables applications to retrieve data by performing queries on metadata attributes, independent of the data location.

We consider a system model composed of a small set of geo-distributed data centers, and a large set of client devices. Some of the data centers fully or partially replicate data, representing geo-replicated cloud storage systems, while others store disjoint datasets, representing different clouds storage systems (ex. DC1 & DC2 = AWS S3, DC3 & DC4 = Microsoft Azure). An instance of Zenko is deployed on one of those data centers.

Clients perform reads and writes using the S3 API either through Zenko, which then forwards operations to the appropriate clouds (in-band operations), or by communicating directly with a backend cloud storage service (out-of-band operations). Clients can perform metadata search queries through Zenko using an SQL-like interface.

In order to provide metadata search, Zenko captures and stores object metadata attributes in a database. This database is replicated within a data center for fault-tolerance. The current implementation of Zenko uses MongoDB as this metadata store. Metadata attributes are stored in MongoDB as JSON objects and Zenko takes advantage of MongoDB's indexing and search capabilities to support metadata search. For in-band write operations, metadata attributes are captured and stored in the metadata store. For out-of-band write operations, metadata attributes are eventually propagated to Zenko's metadata database using event notification mechanisms provided by the cloud services.

The current implementation provides the desired functionality, but has the limitation of assuming that most user operations are performed through Zenko, and are thus local to the data center where Zenko is deployed. However, for out-of-band write operations, the

[1] https://www.zenko.io/

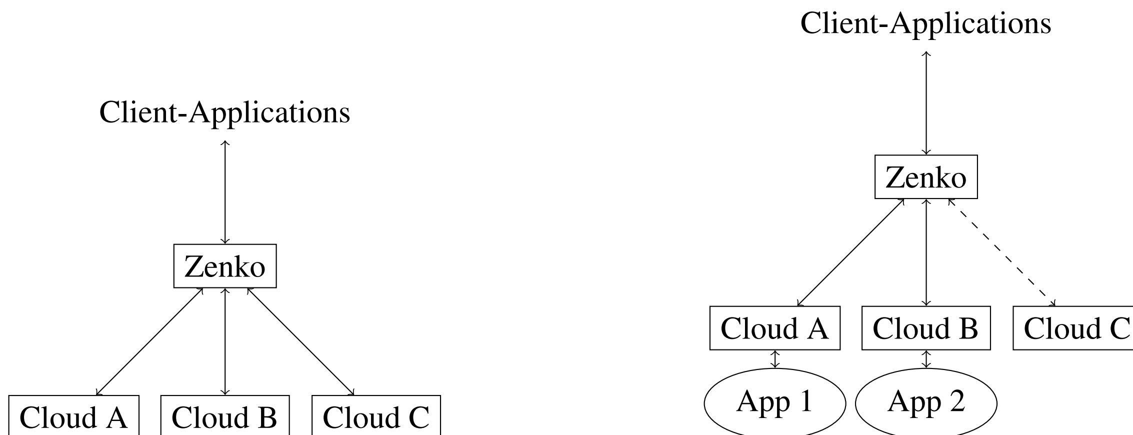

Figure 1.1: Deployment scenarios for Zenko and pre-indexing.

corresponding updates to metadata attributes need to be propagated to Zenko. This leads to high network usage in the case of write-heavy workloads and potentially stale search results, as write operation are only eventually delivered to Zenko.

In addition to the difference between in- and out-of-band updates, cloud applications using Zenko have varying characteristics and requirements. For example, different applications may have write- or search-dominated workloads, some may require low search latency while others always consistent search results. Addressing these needs requires a flexible design that can make performance trade-offs.

This use case aims at introducing a new geo-distributed metadata search design to improve the system's flexibility. The key insight is to allow flexible placement of state (indexes, caches) and computations across a geo-distributed system, instead of on a single data center. Different state placement schemes, for example having a geo-distributed index with partial indexes placed closer to the backend storage systems or closer to the clients at the edge, and the resulting index maintenance and query processing communication schemes, can express different points in the design space of the problem of distributed query processing, allowing the search system to adjust to various application requirements.

1.1 Architecture

For the formal description of this use cases, we consider two different scenarios.

In the first scenario (Figure 1.1 on the left) all traffic goes through the Zenko system, which is deployed in one data center. In this scenario, we have full control over the data and could therefore offer any consistency level to client applications.

In the second scenario (Figure 1.1 on the right), there are applications communicating directly with the cloud systems. Here, Zenko is used to provide additional functionality not offered by all cloud providers. With this setup, we depend on triggers offered by the cloud providers. When data changes in a cloud, Zenko is (eventually) notified about the change and we can update the Indexes. Because triggers are asynchronous, this scenario limits what consistency guarantees we can provide at the Zenko interface. However, this scenario is relevant in practice as it allows to colocate applications in the same cloud as the

```
// store an object
void putObject(ObjectKey key, Binary data, ObjectMetadata metadata)

// delete an object
void deleteObject(ObjectKey key)

// get data and metadata for an object
ObjectWithMetadata getObject(ObjectKey key)

// get all metadata for an object
ObjectMetadata getObjectMetadata(ObjectKey key)

// attach metadata to an object
void putObjectMetadata(ObjectKey key, MetadataKey m, Binary data)

// delete metadata from an object
void deleteObjectMetadata(ObjectKey key, MetadataKey m)

// Search for objects matching the given conditions
// A condition has a metadata key and a range of values to include
Set<ObjectKey> search(List<Condition> conditions)
```

Figure 1.2: API provided by the indexing system.

corresponding storage. Moreover, some customers have legacy applications, that directly work with a cloud and are not yet connected to Zenko.

1.2 API

Applications communicate with the Zenko Multi-cloud provider using an API similar to AWS S3. For the formal model we only consider a subset of the provided functionality, which is related to indexing data. Figure 1.2 shows the API we assume for our model.

The system stores some binary data together with its metadata under an ObjectKey. The metadata is a mapping from MetadataKeys to binary data. Data can be retrieved using the ObjectKey or a search query. We represent a query by a list of conditions, where each condition consists of a MetadataKey and a minimal and maximal value to include in the result. We use the lexicographic ordering on the binary values here. If no condition is given, the search function will return all object keys. If more conditions is given, search will return only the keys that match all conditions.

1.3 Specification

We begin by describing the state of the index based on the invocations of the API (see 1.2). Calls to the API can occur on different pre-computation nodes and are not executed with strong consistency (i.e. with a total order). To specify the application in this setting, we use notations similar to the ones used for the LightKone programming model.

We assume that all calls from the same app to the same node are ordered sequentially and that all calls are ordered in a partial order *vis*. We say that a call is visible to another call, if the latter observes all writes done by the first. Moreover, we assume that there is

a total order *ar* on all calls, which is an extension of *vis* and can be based on timestamps. The semantics of individual API calls will be described based on the set of past API-invocation events E, which are visible at the time and location of the request.

The data of an object with key k is determined by the calls to putObject and deleteObject. We first define $puts(k)$ to be the set of putObject-events, which have not been affected by a deleteObject-event:

$$puts(k) = \{e \in E \mid op(e) = putObject(k,_,_) \land (\forall e' \in E.\ op(e') = deleteObject(k) \to e' \prec_{vis} e)\}$$

The definition of *puts* describes a delete-wins semantics, where a delete-event wins over a concurrent put-event. For concurrent put-events, we use the last-writer-wins policy, which we formalize using the arbitration relation. The following function $data(k)$ describes what data is stored under key k:

$$data(k) = \begin{cases} \bot & \text{if } puts(k) = \emptyset \\ d\big(max_{ar}(puts(k)\big) & \text{otherwise} \end{cases}$$

$$\text{where } d(e) = x, \text{ if } op(e) = putObject(_,x,_)$$

Similarly, we can define the state of the metadata for an object with key k and metadata-key m. Again, we specify that deletes win over concurrent updates. We introduce an auxiliary function *isMetaPut* in order to unify the cases where metadata is changed by *putObject* and *putObjectMetadata*. This also helps us express, that updates on distinct metadata keys are independent. For example, if there are two concurrent operations $putObject(k,d_1,[a \mapsto x_1, b \mapsto y_1])$ and $putObject(k,d_2,[b \mapsto y_2, c \mapsto z_2])$, then reading metadata key a is guaranteed to be x_1, reading c will yield z_2 and only reading metadata key b will yield a result that depends on timestamps.

$$metaPuts(k,m) = \{e \in E \mid isMetaPut(e,k,m) \land (\forall e' \in E.\ isMetaDelete(e',k,m) \to e' \prec_{vis} e)\}$$

$$\begin{aligned} \text{where} \quad isMetaPut(e,k,m) &= \big(\exists M.\ op(e) = putObject(k,_,M) \land m \in dom(M)\big) \\ &\quad \lor\ op(e) = putObjectMetadata(k,m,_) \end{aligned}$$

$$\begin{aligned} isMetaDelete(e,k,m) = op(e) &= deleteObject(k) \\ \lor\, op(e) &= deleteObjectMetadata(k,m) \end{aligned}$$

$$metadata(k,m) = \begin{cases} \bot & \text{if } metaPuts(k,m) = \emptyset \\ md_m(max_{ar}(metaPuts(k))) & \text{otherwise} \end{cases}$$

$$\text{where } md_m(e) = \begin{cases} x & \text{, if } \exists M.\ op(e) = putObject(_,x,M) \land M[m] = x \\ x & \text{, if } op(e) = putObjectMetadata(k,m,x) \end{cases}$$

Using the auxiliary functions *data* and *metaData*, we can now define the result of the queries *getObject* and *search*. For *getObject* we return $\bot$, if no data exists for the object.

Otherwise, we return a pair consisting of the data and a map with all the metadata of the object.

$$res(getObject(k)) = \begin{cases} \bot & \text{, if } data(k) = \bot \\ \big(data(k), (\lambda m.\ metaData(k,m))\big) & \text{otherwise} \end{cases}$$

For specifying the search query, we model the conditions as triples (m,l,h), where m is a metadata key, l is the lowest value to include and h is the highest value. The query *search* then returns the set of all keys, matching all of the given conditions.

$$res(search(C)) = \{k.\ \forall(m,l,h) \in C.\ metaData(k,m) \neq \bot \wedge l \leq metaData(k,m) \leq h\}$$

(a) Consistency levels

The specification above allow different levels of consistency, since we did not restrict the visibility relation. In fact, we plan to deploy our solution in different scenarios that require and/or allow for different consistency levels. In this document, we consider three scenarios:

Scenario 1: Eventual consistency When supporting legacy applications, there might be updates that directly write to an underlying cloud storage, without going through Zenko. In that case, we have to rely on trigger notifications from cloud providers, which typically only provide eventual delivery guarantees. Therefore, we can only provide eventual consistency in the general case. We could provide additional guarantees for the updates that are issued via Zenko, but this would constraint the design space too much.

Therefore, we do not formally restrict *vis* for this scenario. We merely require a best-effort approach – the system should be built to reduce the number of events that are not visible when performing a query.

Scenario 2: Transactional causal+ consistency If we want to allow requests to be performed at different data centers, coordination might be too expensive. Then the strongest suitable consistency model is transactional causal+ consistency, which we choose here. This model imposes three restrictions on the visibility relation:

1. Causal consistency: *vis* is transitive.
2. Session guarantees: All operations performed on the same connection are totally ordered by the session order (*so*). Visibility should include this relation: $e_1 \prec_{so} e_2 \longrightarrow e_1 \prec_{vis} e_2$
3. Transactional guarantees: If operations are performed in the context of a transaction, we want to guarantee atomic visibility: For events $e_1, e'_1, e_2, e'_2 \in E$ with $tx(e_1) = tx(e'_1) = tx_1$ and $tx(e_2) = tx(e'_2) = tx_2 \neq tx_1$ we have $e_1 \prec_{vis} e_2 \leftrightarrow e'_1 \prec_{vis} e'_2$.

Scenario 3: Strong consistency If all requests are made through a single Zenko instance, we can coordinate all requests within this instance and use consensus to enforce a total order on the requests. In this case, *vis* must be a total order and whenever a request e_1 returns before another request e_2 starts it must hold that $e_1 \prec_{vis} e_2$.

(b) Invariants

In Deliverable 2.1 we proposed the following two invariants for the system:

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.

Both conditions are now precisely specified by the equation for $res(search(C))$, which uniquely determines the result of a search based on the given context. In Deliverable 2.1 we allowed (temporary) violations of the invariants. This is covered by the different consistency scenarios described here. A search must always return a correct result with respect to the current visibility of operations, but consistency scenarios 1 and 2 allow operations to be delayed and only become visible at a later time.

1.4 Discussion

We have specified only a subset of the functionality. There are some minor omissions and simplifications regarding the API: Not all options and convenience methods are supported and we did not include support for namespaces in our API. The more important omissions from our model are access control and versioning.

Access control on the database level is typically handled at the granularity of name spaces or on the whole database level. Access control on single objects and metadata entries is usually not handled at the database level and therefore is orthogonal to the specification we gave here. Instead, access control at the object level is typically handled at the application side, for which we can leverage techniques developed as part of work package 6.

Versioning in the context of Zenko and S3 is mainly used to avoid data loss, by giving clients the chance to recover older versions of an object. It would be possible to include version information in search results, so that fetching the data for a key returned by a search would be guaranteed to return matching results. However, it is still unclear if this is a desirable feature, so we did not include it in the specification.

Chapter 5

Stritzinger

1 The RFID-powered conveyor – overview

This use-case targets a typical conveyor belt that moves products being manufactured, uniquely identified with an RFID tag used for identification and to store some manufacturing state. Products are moved from station to station according to a tailored production plan; each station modifies the products, whenever applicable, further evolving the state of their production plan.

Embedded nodes are placed at key positions in the conveyor belt, interconnected in a mesh network, responsible for interacting with (1) the RFID tags, (2) the conveyor (to route the objects), and (3) with the processing stations (to instruct them what to do). We assume that the mesh network connecting these nodes is fast enough to share tag information, but that the network may have partitions, and temporary reconfigurations of nodes may induce slowdowns in the communication.

A concrete scenario is depicted in Fig. 1.1. In this example new objects arrive from the bottom left corner and are being routed between stations to evolve in their production plan, leaving from the top-right corner when done. The full construction plan and its progress is stored in the RFID-tag, and at each decision point (before branching or before entering a station), the corresponding node needs to access this plan and act accordingly.

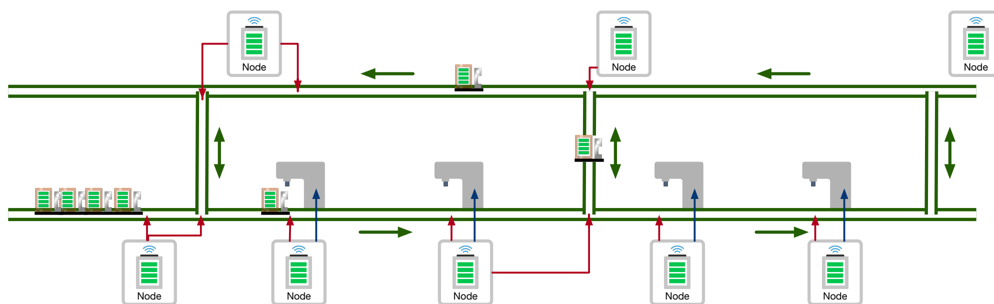

Figure 1.1: Scenario of a conveyor belt. Green arrows denote directions of the conveyor belt, red arrows are communication points between nodes and the RFID tags, and blue arrows capture interactions with the stations.

Our contribution

The key problem we address is the slow speed to write and read these tags, requiring the objects to stop at each decision point to be read before proceeding. A common solution is to move the plan and status from the tag to the meshed network, using the tag solely as an identifier. The key disadvantage of this approach is the intolerance to network splits, and the possible dependency upon a centralised server to collect all data – a single point of failure.

We propose a hybrid approach where the plan and status of each object are kept both at the tag and at the mesh network of nodes. Furthermore, we avoid relying on a centralised server by replicating this information among the nodes. We validate our choice by formalising a concrete approach to manage different versions of the product states, shared among the RFID tags and the network of nodes. This formalisation uses timed automata, providing insights regarding the time overhead that different design decisions can incur.

2 Distributed state over tags and nodes

This section starts by clarifying the assumptions and challenges of this use-case (Section 2.1), followed by an informal description of how the information is stored and synchronised among RFID tags (Section 2.2) and the mesh of distributed nodes (Section 2.3).

2.1 The distributed challenges

We identify four different situations that may occur in our system.

Tag as backup – node has the latest information. This is expected when the network is fully functioning, not partitioned, and data flows faster than physical objects. This means that each tag does not need to be read, and only needs to be fully written at the exit point (so-called *critical write*, possibly requiring the object to stop); in the rest of the system it is enough to be partially written or not written at all.

Tag as fresh sources – tag is the source of information. This is expected when unknown objects are detected, either because they are new, or because they come from a completely disconnected source, or because they were manually modified. Hence, at decision points the node may need to read all or part of the tag to be able to act accordingly – waiting for the network will not help.

Tag as faster source – resulting from network delays. This is expected when nodes are busy with other tasks other than routing, when nodes are being reconfigured, or when data needs to cross network partitions using slower mechanisms. As above, at decision points the node may need to either read the tag, or to wait for the network. We also assume that data in the tag may be dirty in these cases, i.e., only partially written – hence the node must wait for the network information.

Concurrent operations – when a node wants to write information on a more recent tag, even before its newer version is known (either via the network or the tag). This can happen in situations where the tag is faster than the network, the desired section

is still being written (dirty), but the node still wants to modify it; for example, to append a timestamp logging its passage. A naive approach would require stopping the conveyor and wait for the network update before attempting to write.

This work addresses the three top situations, leaving the efficient manipulation of concurrent operations for future work. Our approach is to structure the RFID memory to include version numbers of different sections of the memory, and the degree of confidence over the stored data.

2.2 The tag structure

Each tag contains a unique ID and around 2KB of storage, subdivided into *blocks* of 8 bytes each that define the granularity of reading and writing to the tag. I.e., if a writing operation fails, only the block being written is affected. In a typical scenario, a sensor can read or write 2 to 5 blocks of bytes without requiring the conveyor to stop, depending on the speed of the conveyor. We add additional structure by grouping sequences of blocks into *sections*. Each section consists of:

- **Version number** – counter incremented every time the section is updated;
- **Dirty bit** – set to true if any of its inner blocks started to be overridden but has not finished the process;
- **Blocks** – sequence of blocks of bytes with the actual data;

The tag memory is subdivided into sequences of sections, uniquely identified by their order number. We call *section header* to the version number and the dirty bit, and we call *tag header* to the union of all section headers and the tag identifier. This model will be further simplified to facilitate verification, explained after introducing Timed Automata.

2.3 Replicas of the tag structure at the nodes

Each node contains a local replica of different tags, where sections may have higher or lower versions than the tag. This replica is enriched with meta-information regarding what sections need to be read or written. More concretely, each node contains, for each tag, a list of contiguous sections, each consisting of the following:

- **Version** of the section;
- **Data**, as a list of blocks;
- **To-write labels** to blocks that are pending to be written;
- **To-verify labels** to blocks that are pending to be verified;

Once a tag enters the range of a node, this node starts by reading a single block with its tag header, followed by possible reads or writes based on information from the network. These read or write operations may be interrupted and continued by another node, who will get notified over the network about unfinished operations. However, reading or

writing may be critical – for example, a section may be needed in order to decide how to proceed (critical read), or the node may be the last one interacting with the tag before being dispatched (critical write). We make this more precise, distinguishing 4 different situations when a conveyor **may need to stop** if certain operations are not yet finished.[1]

1. *Critical read* – when a section of the tag is needed before proceeding and the physical tag refers to a more recent information than the node; for example, when key data is needed to make a decision;

2. *Critical write* – when a tag is about to leave the network partition, and it is not up to date with the node information yet.

3. *Read before write* – when new information needs to be added to a section of the tag, but its current version is not known to the node yet, and a more recent version is mentioned in the physical tag.

4. *Write new version number* – when new information needs to be added to a section, the tag needs to be updated to have its most recent version number before proceeding, even if the section itself is not updated.

Observe that the reading operations above (1 and 3) occur only when the network fails to be fast enough, either because of a network partition, or because one of the nodes is misbehaving (e.g., being reconfigured). In these cases, the node may either read the section from the tag, if the relevant section has been successfully written and verified, or it may wait for a network update.

2.4 Optimisations

The proposed model can be further optimised on a scenario basis. For example, one may vary the size of the sections. *Smaller (and more) sections* will mean finer granularity to specify more precisely what is critical; potentially allowing only a small number of blocks to be read before proceeding. *Less (and larger) sections* will mean less version numbers, hence a smaller header and possibly less time spent before starting to read or write relevant data. Based on the scenario, different sections can have different sizes, and more complex approaches involving dynamically changing the sizes of sections could be advantageous.

The size of the blocks, unlike the size of sections, are predefined by the manufacturers of RFID tags and readers and cannot be fine-tuned on a per-scenario basis. Other optimizations can also be made, which we leave for future work. For example, the structure of the header could be compressed when assuming the range of version numbers is small, or if the number of dirty bits set to true is mostly kept very small, or if certain sections are rarely accessed (and may require the read of more than a single header block).

Another interesting direction of future optimisations, aligned with Lightkone, is to follow the same ideas behind eventually consistent mechanisms, by assuming that certain

[1] In practice it is not always possible to stop the conveyor; the implications on the formal model are left for future work.

sections have a more restricted behaviour or require fewer consistency guarantees. For example, if a given section is used only for keeping a write-only log of events, represented as a set of timestamp-action pairs, then a node may update its version even without knowing in advance its previous version (avoiding the stopping case (4) from Section 2.3).

3 Formalisation using Timed Automata in UPPAAL

Recall the notion of Timed Automata and how to analyse systems in UPPAAL (Section 5). This section provides the formalisation using timed automata for a simplified scenario, using a network of communicating automata consisting of:

- **Tag** – one automaton for each product with a tag, describing the possible paths that a product may take between nodes, and the distance between them;
- **Node** – one automaton for each node, interacting with the Tag automata to simulate reading and writing of blocks, describing both the time needed to perform operations and the time a tag stays in the proximity of a node;
- **Cache** – one automaton also for each node, to encapsulate the behaviour regarding the distributed cache, i.e., how to broadcast updates from one node to all neighbours from all network partitions.

These automata synchronise via channel synchronisation, but the content of the tags (both in the physical tag and their replicas in the nodes) and some configurations are stored in shared data structures. The channel-based synchronisation controls the access to these data structures.

The rest of this section analyses a specific scenario in detail. It starts by presenting the configuration parameters of our scenario, followed by the definition of each of the automata, and wraps up with a set of properties that can be verified using this formalisation.

3.1 General configurations

Global parameters and variables are described in a C-like language, including: constants used as parameters that configure the scenario, channels used to synchronise the different automata, dedicated data types, and variables that can be read or modified by any of the automata – used to share more complex data structures. We now describe the values used to model our concrete scenario; by choosing variations of some of these constants one quickly understands the impact such choices have.

- *General parameters*: Number of *Tag* automata ($\mathtt{T} = 2$), of node and associated *Cache* automata ($\mathtt{N} = 6$), of sections in each tag ($\mathtt{S} = 2$), and speed of the conveyor belt ($\mathtt{Speed} = 1$). The speed is used to define bounds on the expected communication time between the tag and the nodes ($19/\mathtt{Speed} \leq \mathtt{TCom} \leq 21/\mathtt{Speed}$) and on the time to write or read a block ($2/\mathtt{Speed} \leq \mathtt{TOp} \leq 4/\mathtt{Speed}$). The time to send over a network partition is bounded by $450 \leq \mathtt{TLag} \leq 550$.

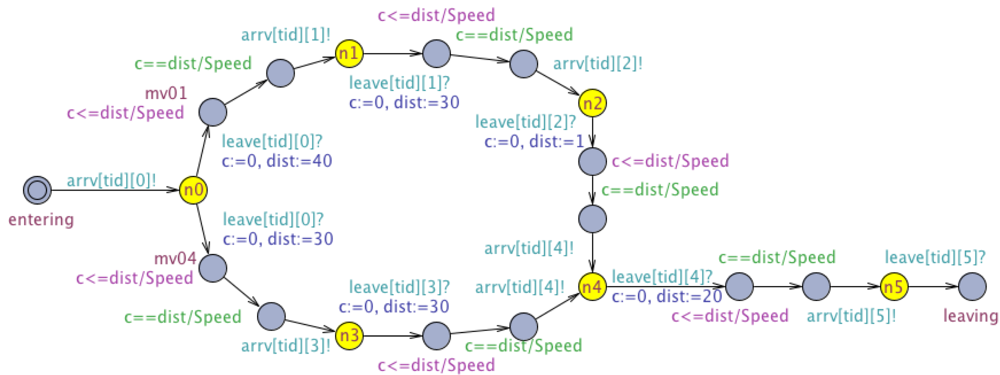

Figure 3.1: Automata of a Tag automaton with ID `tid`.

- *Channels*: For every *Tag* `t`, *Node* `n`, and *Cache* `c` $\texttt{arrv}_{\texttt{t}\to\texttt{n}}$ is a channel from `t` to `n`, $\texttt{leave}_{\texttt{n}\to\texttt{t}}$ from `n` to `t`, $\texttt{tagUpd}_{\texttt{t}\to\texttt{c}}$ from `t` to `c`, and $\texttt{netUpd}_{\texttt{c}}$ from *c* to its associated *Node*. The `arrv` channels are performed as soon as possible (called urgent channels in Uppaal), and the channels `tagUpd` is not restricted to binary synchronisations, allowing any send to be received by exactly the automata ready to receive it (even if none – known as broadcast channel in Uppaal).

- *Data types*: Structures to represent sections of the physical tags, consisting of a `version` number, a data value (integer), and a boolean `dirty` bit; and to represent the node replicas of these sections, consisting of a `version` number, a `data` value, a number of pending blocks to be written (`toWrite`), and a number of pending blocks to verify (`toVerify`).

- *Initial tags*: Our scenario has 2 tags, each with 2 sections. For simplicity, all are initially on version 5 and with non-corrupted data (i.e., the dirty bits are set to false).

- *Node parameters and initialisation*: Each node has a different view of the 2 *Tag*s. Initially, all *Node*s have the 1st section of the 1st tag on version 6 with 2 pending writes and 1 pending read, and the 2nd section on version 4. The exception is node 1 – its 2nd section of the 1st tag is already up to date on version 5. The second tag is assumed to be in an older version for all nodes.

  Furthermore, *Node* 0 has a critical section: the 2nd section of the 2nd tag; *Node* 5 is marked as an exit node, which means it will have to write all pending data; and every node will try to write one block into the 1st section of the 1st tag, except for node 0 which will try to write 2 blocks to the same section.

- *Network partitions*: The network is partitioned into two parts, the 1st consisting of the nodes 0,1,2,3, and the 2nd of the *Node*s 4 and 5.

3.2 Tag Automata

We use one *Tag* automaton for each of the two products with an RFID tag in the conveyor. The automaton for each of these is depicted in Fig. 3.1. This automaton describes the path and distance that the product has to perform. After it passes by node 0 it can either

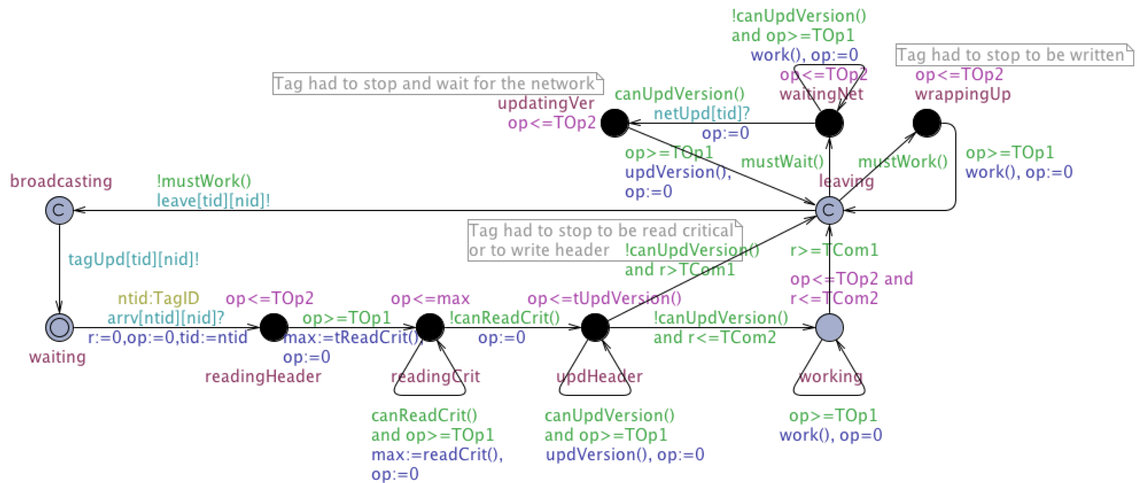

Figure 3.2: Automata of a Node automaton with ID `nid`.

go to nodes 1, 2, 4, and 5 (by this order), or go to nodes 3, 4, and 5. The time constraints capture the time spend while travelling, which depend on the distances between stations, explicitly defined by setting the variable `dist`. For example, when leaving location `n0` the product can transition to location `mv01`, setting `dist` to 40. There it will stay for at least 40 time units, after which the transition $\mathtt{arrv}_{\mathtt{tid}\to 1}$ (written `arrv[tid][1]`) becomes available. Note that this transition is urgent, meaning that it has to be performed as soon as both the *Tag* and the *Node* are ready.

3.3 Node Automata

The automata for nodes describes their interactions with tags, and the time spent during these interactions (Fig. 3.2). Each of the 5 *Node* automata can either be *waiting*, until it receives an `arrv` message, or it can be interacting with a tag, until it allows it to leave by sending it a `leave` message (and immediately broadcasting updates about the tag to the other nodes).

In this specification, the node has the responsibility to control the time the tag stays close to the node. In a traditional run, the tag will leave after some time between `TCom1` and `TCom2`, which is the expected time the tag will stay when passing by the node without stopping. In exceptional cases the tag may have to stop, which can only happen at the locations coloured in black. The 3 bottom black locations correspond to: (1) when the node reads the header (with the tag ID and the meta-data about the sections), (2) when it is reading critical sections with essential information to decide what to do next, and (3) when it is updating the header because some sections will update their version numbers. The three top black locations correspond to when the node needs to wait for the network (and consequently update of the section version – nodes *waitingNet* and *updatingVer*), and when the node is marked as an exit one and must write and verify all pending blocks to the physical tag (node *wrappingUp*).

Note that variables and functions, such as `tid` and `canReadCrit()`, are specified in the companion block of code to the automaton definition, using a variation of a subset of C.

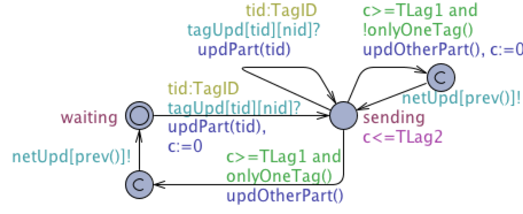

Figure 3.3: Automata of a Cache automaton with ID `nid`.

$\exists\Diamond Tag(0).n5$	Tag_0 can reach node 5.
$\exists\Diamond exists(n : NodeID)$ $Node(n).waitingNet$	The location *waitingNet* is reachable for some node.
$\exists\Diamond exists(n1,n2 : NodeID)$ $Cache(n1).sending$ *and* $Cache(n2).sending$ *and* $n1! = n2$	More than one cache can be sending data at the same time.
$\exists\Diamond$!done *and* $(total \geq 205)$	It can take up to 205 time units.
$\exists\Diamond$done *and* $(total \leq 175)$	It can finish in 175 time units.
$\forall\Box$!done *imply* $(total \leq 205)$	It cannot take more than 205 time units.
$\forall\Box$done *imply* $(total \geq 175)$	It cannot finish in less than 175 time units.
$\forall\Box(deadlock$ *imply* done$)$	It can deadlock only when both tags are leaving.
$\forall\Box!(Tag(0).n1$ *and* $Tag(1).n1)$	Only one tag can be in node 1 at a time.

Table 3.1: Properties verified by UPPAAL for our proposed scenario, where done is a shorthand for $Tag(0).leaving$ *and* $Tag(1).leaving$.

3.4 Cache Automata

Each node has an associated *Cache* automaton, depicted in Fig. 3.3, describing the propagation of tag-related information among the nodes. Each of these *Cache*s interacts with its associated node, receiving requests to broadcast updated information via the `tagUpd` channels, and notifies remote network partitions when updating their information.

The actual propagation of data is realised by modifying a global structure at controlled moments of time, using the functions `updPart` (for an instantaneous update of nodes in the current partition) and `updOtherPart` (to update nodes in remote partitions). Internally it manages a queue of tags pending to be delivered to remote partitions, by specifying that each tag in this queue takes between `TLag1` and `TLag2` to be send.

3.5 Verification

In Table 3.1 we list some useful properties of this scenario that we can verify using UPPAAL [7], where the macro `done` means that both tags left the conveyor (independently if the distributed cache is stable).

These properties show, for example, that both tags can be completed in somewhere

between 175 and 205 time units, depending on the non-deterministic choices. These choices include, going to node 1 or 3 after leaving node 0, taking between `TCom1` and `TCom2` time at a given node, or succeeding to write or fail when the time to leave a node coincides with the time to finish writing the block.

When experimenting with different values of `Speed`, `TCom`, `TOp`, `TLag`, or even when changing the topology of the *Tag* automaton, one can have a quick understanding of the temporal implications. Consider, for example, a scenario where `TCom` is exactly 19 and `TLag` is exactly when the time to perform an operation is exactly 2 or exactly 3, the quickest time to be done is the same: 175 time units. When increasing this value to perform an operation becomes 4, the quickest time to be done becomes 475 (2.7 times more). These reflects the fact that, in the best case, the conveyor can have the same amount of stops when `TOp` is 2 or 3, but it will incur extra stopping time when `TOp` is larger than 3.

Chapter 6

Gluk

1 Use Case: Self-sufficient precision agriculture management for Irrigation

1.1 The story

On a very warm day in the Mediterranean summer, farmer Manolis has to irrigate his citrus fields using the water from his well. Last summer Manolis installed Subsurface Drip Irrigation in his fields. He wakes up very early in the morning to be in the field and start pumping water. This way he has enough water to irrigate his fields before his neighbors, who are also pumping water, drain the underground water supplies.

The heat period this summer is longer than usual, so Manolis has to irrigate his fields very often. Unfortunately, he doesn't have an accurate indication of the moisture levels in the ground throughout his fields. Being afraid of loosing part of the production, he irrigates his fields every 7 to 8 days. One day, Manolis visits his local consultant (agronomist) in order to discuss and plan the production. While discussing, Manolis explained the problems, worries and difficulties he is dealing with regarding the irrigation of his fields. His agronomist then suggested to install the **LightKone Self Sufficient Precision Agriculture** module. This very unexpensive and easy to install technical solution will help him to better plan his irrigation and will therefore save time and money. Manolis will have more freedom, save electricity, and produce higher quality products.

1.2 Scientific Context of the Use Case

We present a zero-touch configuration and autonomous management capable sensor array for precision agriculture with actuators to achieve management goals for irrigation. We apply this use case to irrigation management in the Subsurface Drip Irrigation method. (citrus trees cultivation – but can be applied in every farming activity, indoor as well as outdoor).

Goal: the core management ability must be completely **autonomous** (no need for PC or cloud control) and as **low-cost** as possible (again, no need for PC or cloud connectivity, which can be expensive), and for this it should run on the sensor array itself. Additional management capabilities can be added, which will cost extra, **but they are not essential**

for the correct operation of the system.

Requirements for the management software. The basic management should be done by the sensor array itself. Higher-level management goals can be added by external systems, such as PCs or cloud tools, but such external systems cannot be guaranteed to be connected to the sensor array. Also, we would like the system to be as low-cost as possible: the most essential management should be done on the sensor array itself, without any external costs. **This gives modularity for the farmer:** he pays only for what he needs, and Internet connectivity is not needed for basic management abilities.

Requirements on the sensor array. In order to achieve this, the requirements on the sensor array are that there should be (1) basic computation ability in the sensor nodes, and (2) basic communication ability between sensor nodes (for example, Wifi or Zigbee), with normal reliability of these nodes as provided by off-the-shelf hardware.

Given these requirements, the software we develop using LightKone technology should be able to perform reliable basic management (24/7) despite problems in the sensor array (nodes going down, communication being unreliable).

(a) LightKone Innovation

We will use LightKone technology to provide reliable computation and communication ability despite unreliable nodes and communication. We will present a proof of concept using Lasp-on-GRiSP and Yggdrasil. Lasp provides a reliable replicated key/value store that runs with very little computational resources, on top of a communication layer, Partisan, that ensures reliable communication despite highly unreliable connectivity (using hybrid gossip). Basic connectivity is provided by Yggdrasil underneath Partisan. We will extend Lasp with a simple task model that stores the management software in the Lasp store itself (which is possible because of higher-order nature of Erlang), and performs periodic computations, storing results in the Lasp store. GRiSP provides native Erlang functionality running with low power, with basic processor power and memory and wireless connectivity. GRiSP also provides Pmod sensor interfacing to provide the sensor and actuator capabilities.

GRiSP nodes can be powered by solar batteries. 100% uptime is not required because of the Lasp redundancy. Occasional problems in individual nodes are solvable by periodic reboot of individual nodes. This will not hinder overall system operation because Lasp replication and Partisan hybrid gossip are designed to survive such problems.

Management policy control is provided by a connection to the sensor array, either by PC or cloud, which the farmer can do at any time. This connection does not need to be continuous or reliable. The management will continue to work even if the connection is not established for several days or more.

1.3 Description

Until now: The water is pumped out from the well (or other water source) and transmitted to the polymer tubes. The water usually is used to irrigate multiple fields. However, every field has different characteristics (soil, area, etc.) and the irrigation should be adapted taking account these parameters. For example, other piece of land needs more water and another piece of land needs less water. In order to avoid under-watering, the farmers

Figure 1.1: Irrigation plan example.

usually irrigate more times than necessary. **Problem:** Waste of water, waste of energy (electricity for the pump), drainage problem (since many farmers irrigate at the same time and the water in the underground water dump is not enough for everyone). In the image 1.1 a typical irrigation scenario is depicted. The polygons in the **red line** are the fields, the **red pin** is the well and pump that irrigates the indicated fields, and the **yellow pins** are the neighboring wells. The irrigation procedure is taking placing following only empirical and observation rules. E.g. we irrigate for 2 hours and then we repeat every 10 days in the summer period. Sometimes either the citrus trees need more water because of extended high temperatures or less water due to lower temperatures.

In general, one of the main drawbacks of the SDI systems is that water applications may be largely unseen, and it is more difficult to evaluate system operation and water application uniformity. System mismanagement can lead to under-irrigation, less crop yield, quality reductions, and over-irrigation. The latter may result, among others, in poor soil aeration and deep percolation problems.

Using the self-sufficient management system, the field could be divided into clusters (e.g. as installed the polymer tubes) and accordingly the LightKone self-sufficient nodes will be distributed, containing the management unit, sensors and actuators. This way the farmer could divide his fields into zones, and when a zone is sufficiently irrigated (retrieved value from the sensors), the actuators will stop the water flow into specific parts of the tubes. The remaining part of the field that still needs water will still be irrigated.

The image 1.2 presents a potential installation of the nodes and gateway in the field. The **blue box** marked with the letter G is going to be the gateway and through it the farmer or the agronomist could occasionally provide the management rules through a PC. This is

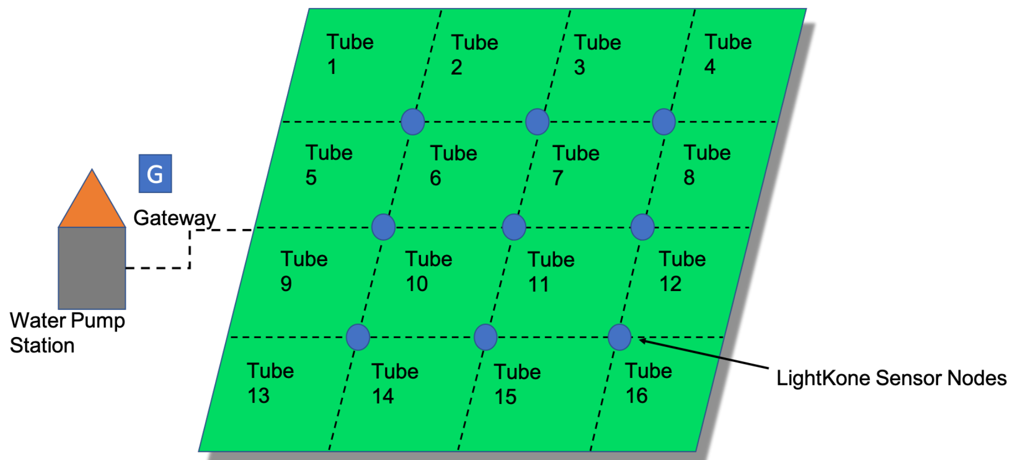

Figure 1.2: Potential installation of the self-sufficient irrigation system

not connected to the internet/cloud. The **black dash lines** are the polymer tubes installed in the field and on top of them, close to each tree, there is the drip station. The **blue circles** are the LightKone nodes including sensors and actuators controlling the flow of the water. (This placement is only made for demonstration purposes. The actual placement will be taking place according to the specific requirements of every field and cultivation). In that way the field is divided into clusters where its cluster can be isolated from the others in terms of water flow.

The self-sufficient system could in a low-cost manner allow the farmer seamlessly to perform irrigation as needed by receiving values from sensors in the soil and controlling actuators to start/stop the irrigation procedure to specific rows of the tubes and accordingly part/cluster of the field.

Using the above installation and the farmer could manage the **fertilizer flow** into his field in the same way.

Another benefit of the self-sufficient system is that it could also act as a frost protection method in the winter. A common problem in the winter time for the cultivation of citrus farms is the frost. The anti-frost protection is very expensive, and the farmers cannot afford to install it an every piece of field. In the winter time when the frost occurs (0 to -3) it affects the production. A method to prevent this is to turn on the irrigation system.

In traditional implementations there is only one central thermostat, usually in the pumping station. Since the pumping station is away from the fields it does not give accurate situation of the status. (The temperature in the pumping station could be over 0 Celsius, however in specific part of the farm could be below 0 – depending on the wind, etc.). Using the self-sufficient system, we can have the temperature values from the sensors on the spot and take the decision to activate the irrigation.

(a) Actors

- **LK Node.** Responsible for local management and decision making, data aggregation, communication among LK nodes and to the LK gateway.
- **LK gateway.** Responsible to updating the management policy to LK nodes and

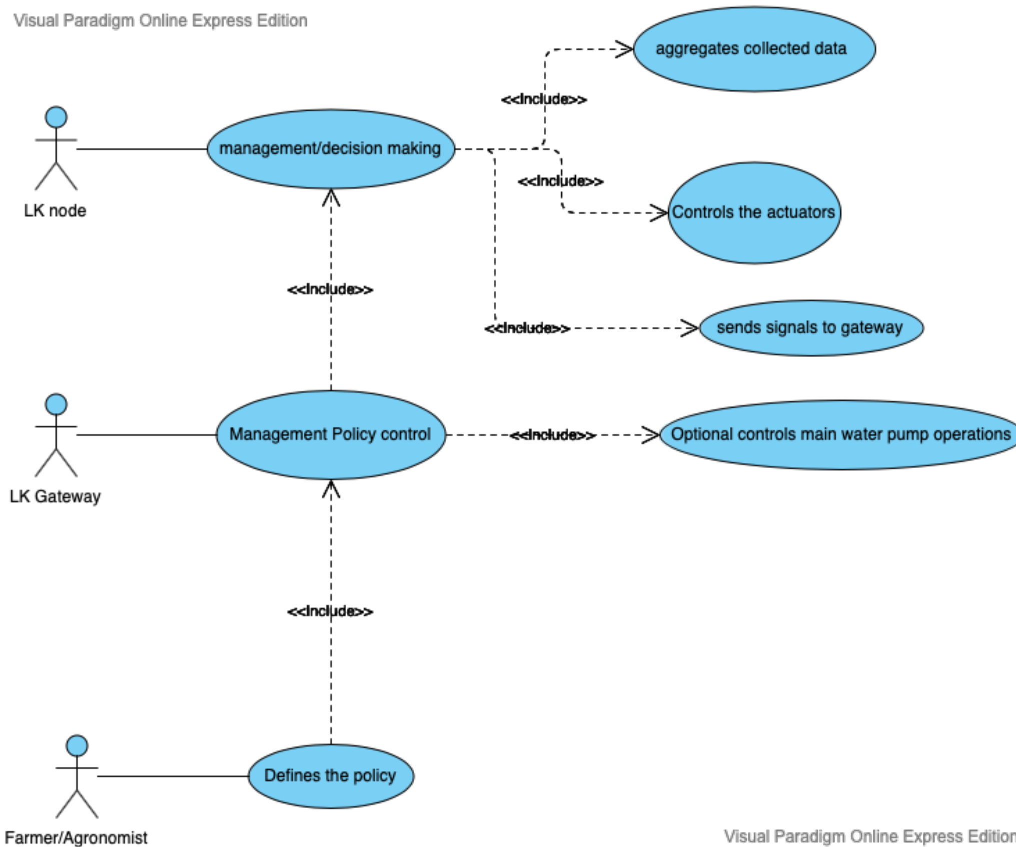

Figure 1.3: Use case actors diagram for Self-sufficient agriculture management use case

eventually controlling the main water pump.

- **Farmer/agronomist.** Defines the policy and updates it through a PC to the LK gateway.

In the image 1.3 you can see the related use case diagram involving the above actors.

(b) Requirements

In the table 1.1 the Use Case requirements and their priority are summarized.

1.4 Return of the Investment

Pain reliefs: LightKone Self sufficient precision agriculture management for irrigation reduces the cost of energy in the water pumping, reduces the stress of the farmer to start/stop the irrigation system (save the production), saves the crop production (increases the profit), improves the quality of the crops, reduces the farming costs (fertilizers, extra manpower, maintenance costs), reduces the environmental footprint.

In the following lines we are providing an estimation of a potential scenario involving financial figures in order to present a level of measurement, among the others, about the cost savings that such an innovative solution can offer.

Table 1.1: Use Case requirements

Id	Requirements	Rationale	Priority
PA1	Zero touch configuration	The farmer should be able to install the system, remove/add nodes on his farm by himself without providing any programming or network setup skills. Every farmer should decide by himself the deployment of the nodes and therefore his own installation depending on his specific requirements.	High
PA2	Actuation	In presence of certain conditions, e.g. the soil moisture level reach above a certain threshold, the actuator, such as solenoid valves, should close and stop the water flow.	High
PA3	Autonomous core management	The core management ability must be completely autonomous and for this it must run on the sensor array itself.	High
PA4	Extendibility	Additional higher-level management abilities can be added as an extra service in a more scalable solution, by external systems e.g. using the cloud and internet connection.	Low
PA5	Policy Control	Management policy control is provided by a connection to the sensor array, either by PC or cloud, which the farmer can do at any time. This connection does not need to be continuous or reliable.	High
PA6	One-time management policy	The policy must continue to work even if the connection is not done for several days or more.	High
PA7	Autonomous operation	No internet connectivity is necessary for the sensor array to operate. The basic management of the sensor array should take place even when some nodes going down or even when the communication gets unreliable.	High
PA8	Periodic computation of the rules	Data filtering threshold to detect anomaly of sensors. If the sensed data is out of range of acceptable values, then it will be discarded. The mean value of the samples will be accepted as an input for actuation decision.	High
PA9	Source Integrity	Communicating end points should be able to verify the identities of each other to ensure that they are communicating with the entities who they claim to be.	High
PA10	Replay Protection	A sensor node can store a data packet and replay it at later stage. The replayed packet can contain a typical sensor reading (e.g. a soil moisture reading).	High
PA11	Data Integrity	No intermediary between a source and a destination should be able to undetectably change secret contents of messages.	Low
PA12	Confidentiality	Messages that flow between a source and a destination could be easily intercepted by an attacker and secret contents are revealed.	Low
PA13	Data clustering	Use of clustering algorithms to identify homogeneous areas for better management of the agricultural land.	Medium
PA14	Autonomous power abilities	The nodes should be enabled with renewable sources of energy (e.g. solar panels) to recharging the node battery	Medium
PA15	Low maintenance cost	The farmer should be able to replace a node by himself in case the node is damaged.	Low

The scenario that we will take account would be for a **one season cultivation** (e.g. citrus cultivation).

In our scenario we will make the following assumptions:

- We will consider the pricing of the LightKone nodes as if they were produced industrially and not experimentally as it is now. (Certified hardware, low hardware mass production cost). In this sense we will define an entry price per LightKone node of 50 Euros.
- We consider that 5 nodes per acre will be installed, including the gateway.
- We will take account as a field of 10 acres. We consider that the **total cost** for installing the LightKone **will be 3.000 Euro** (2.500 Euro hardware costs plus 500 Euro overhead costs).
- The irrigation period lasts from May to October (6 months season).
- The water costs are calculated as per hour cost (approximately 10 Euro per hour). (this depends on the country and the water billing system, usually for agricultue billing is per hour).

In order to calculate the cost benefit, and taking into account the benefits mentioned above, we are following the next clues:

- The net (in the pocket) production money per acre is 700 Euro (This is a pessimistic approach.).
- The improved production (more crops and not losing crops) by proper irrigation could be 15% (realistically it could be more, but we are taking a neutral scenario approach).
- We save 20 hours of water cost (per irrigation period) for a total of the 10 acres (it could be more, but we follow the pessimistic approach).
- We estimate a cost reduction of 50 Euro per acre for other costs (fertilizers, maintenance, time savings, energy costs, other).

Those numbers result in earnings per acre of 140 Euro plus 200 Euro as a total for 10 acres, per season, from the water savings.

In the case of a farm of a total of 10 acres, it can be estimated that within 2 years the farmer will have earned the investment cost in terms of money. Additionally the quality of the land, trees and crops is improved, resulting in a higher production from year to year.

1.5 Agriculture Market Potential

In this paragraph we will succinctly present the market figures for precision agriculture applications in the fog computing era. We will take into account the report commissioned by the Openfog consortium entitled *"Size and Impact of Fog Computing Market" (conducted by the "451 research" company at the end of 2017)*. Here we will present

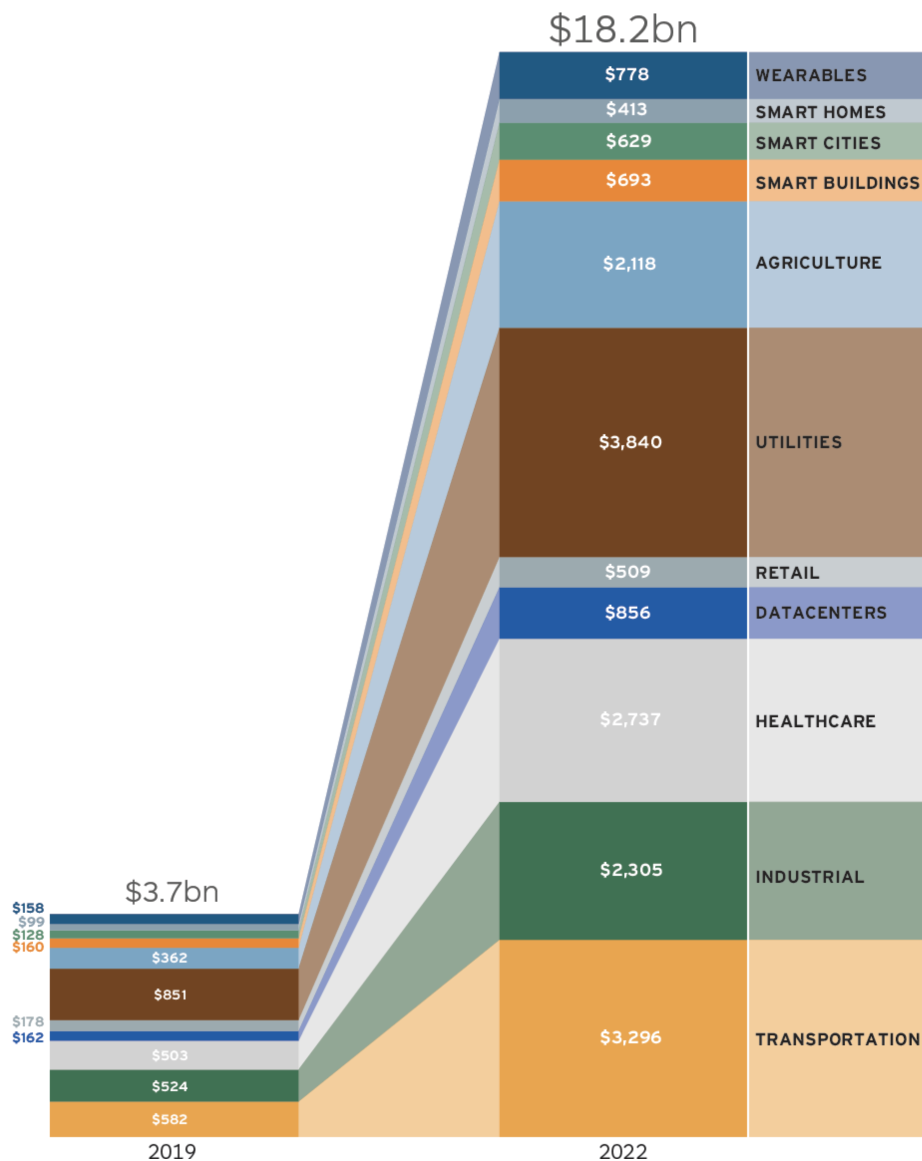

Figure 1.4: Size and Impact of Fog Computing Market

only specific key findings. In the vertical market analysis, we observe that the agriculture market is one of the most promising developing markets in the fog computing paradigm. The agriculture market is projected to reach a market share of 2,118 billion dollars out of the 18,2 billion of the total market and is projected to be one of the five most promising markets, as you can see in 1.4.

For further information see Open Fog Consortium Report.

1.6 State of the art

The development of Wireless Sensor Technologies applications in precision agriculture makes it possible to increase efficiency, productivity and profitability, while minimizing unintended impacts on wildlife and the environment in many agricultural production sys-

tems. Efficient water management is a major concern in many crop systems. WST have a big potential to represent the inherent soil variability present in fields with more accuracy than the current systems available. Thus, the benefit for the producers is a better decision support system that allows them to maximize their productivity while saving water. Also, WST eliminates difficulties to wire sensor stations across the field and reduces maintenance cost. Since installation of WST is easier than existing wired solutions, sensors can be more densely deployed to provide local detailed data. Instead of irrigating an entire field in response to broad sensor data, each section could be activated based on local sensors. Till now there exist numerous implementations of wireless sensor networks for irrigation purposes. The majority of the implementations are using internet connectivity, cloud storage, decision support systems and sophisticated GUIs. Some indicative implementations are presented in [20], [12], [23], [17], [13], [14], [24], [10].

Furthermore, new developments using distributed computing have been released. In [14] a precision agriculture system is presented using a distributed architecture based on a user-centric approach. These systems, however, are also based on cloud usage and external communication interfaces. [18].

Self-organizing network/s (SON) was promoted by the Third Generation Partnership Project (3GPP) and Next Generation Mobile Networks (NGMN). SON has been introduced since in 3GPP Release-8, Release-9 and currently included in Release-10 framework as an excellent solution that promises improvements and market potential for future wireless networks. A system which is self-organized may not have any external or central control entity, but the controlling mechanisms are distributed and localized among the entities within the system.

The academic literature has dedicated significant effort to SON algorithms, providing smart solutions to optimize network operator performance, expenses and users' experience. An extensive literature review of SONs is presented in [18] where comparative tables of the fulfilled and upcoming challenges are presented. A long list of literature is referred and a reference to EU projects that are working on this area is also listed.

Significant work has also been done on EU projects level. Project Figaro [2] aims to increase water productivity in major water-demanding crops and develop a cost-effective decision-support irrigation platform. Meanwhile, the FOODIE [1] project has put in place a cloud-based platform to host both spatial and non-spatial agricultural data while the IoF2020 [3] project has developed new solutions to better integrate 'Internet of Things' (IoT) technologies into agricultural processes.

In the literature review we can hardly find a self-organizing solution with zero touch configuration and autonomous managing , for precision agriculture purposes and more specifically for irrigation management. It is generally agreed that sensor/actuator edge networks are too unreliable to do their own management, so that gateway nodes (PCs) or a cloud connection is necessary. In this use case we will test and present a platform that increases the resilience of sensor/actuator edge networks so that they are able to reliably execute basic management tasks directly on the edge nodes themselves.

2 Abstract Modeling

The operation of a smart agriculture system can be modeled as a discrete sequence of events in time. Each event occurs at a particular instance of time and marks a change of state in the system. A state represents the system in a given moment of its evolution. Such an evolution occurs by processing an ordered sequence of events. Additionally, each event is tagged by a timestamp that specifies the simulated time at which it has to be processed. In a basic scenario the events are represented by the regular data messages sent by sensor nodes and/or actuators. An advanced model of the system could also consider events generated by the system users, e.g. farmers, service providers, third-party developers, etc.

We will use a Discrete Event Simulation (DES) [11] approach in order to create a representative model of the system, test it under various use-case scenarios and collect statistical data. Such a simulation will not only support the design and implementation of the smart agriculture system, but will also help to maintain and improve it in the future.

DES relies on several main components, namely:

- A collection of *state variables* that describe the state of the modeled system. The state of the system changes whenever an event occurs.
- An ordered list of *events* that are processed during the system evolution. An event is described by the *time* at which it occurs and its *type*. The time of event can also be represented as an interval, providing a start and end time.
- A *global clock* that represents the current time in a simulated system. In DES the time is not continuous, but rather it jumps over events as the simulation proceeds. This fact allows us to run simulations much faster as compared to alternative modeling techniques.

3 A Use-Case Example: Irrigation Control System

This deliverable analyses formally automated irrigation in the context of smart agriculture. This example does not describe the autonomous management irrigation system presented above. It is an example that has been chosen to demonstrate formalization techniques. The goal of such system is to take control of the soil moisture at the field. A network of distributed sensors measures the soil moisture periodically and transmit the measurements to a central controller. This controller then processes the measurements received and acts accordingly: powers on or off the water pump responsible for adjusting the amount of water given to the plants.

3.1 Problem Description

The developed irrigation system has to avoid large delays to adjust the water pump, confirm that the water pump is always turned on when the soil is dry and the opposite – when the soil is wet. The system needs to be autonomous and ensure stability. It relies on both edge node and the cloud backend for a DES controller logic. The controller has to be

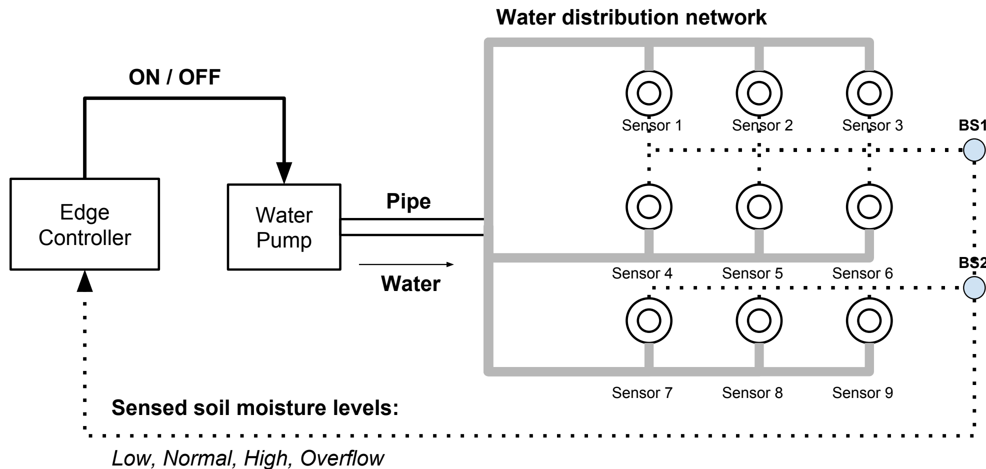

Figure 3.1: The architecture of automatic irrigation system.

able to perform its function while in online (cloud processing) and offline modes (local processing).

As shown at Figure 3.1 an irrigation control system consists of two subsystems, a water pump and a water distribution network. The water distribution network has multiple soil moisture level sensors distributed across the field and a pipe that connects to the water pump. The water pump is controlled by an ON/OFF switch managed by the edge controller. It fills the water distribution network through the water pump. The sensors periodically measure the soil moisture and send their results to the edge controller via the nearest Base Station (BS).

3.2 DES Model of the Irrigation System

Representation of the system in the DES formalism is the first step in the design of a discrete event controller whose objective is to keep the soil moisture level between low and high. We first identify the key events occurring in the system (see Figure 3.2:

- Commands to power on (”pw?ON”) and power of (”pw?OFF”) the water pump.
- The actual start (”w?WSTART”) and stop (”w?WSTOP”) of the water flow.
- Detection of the soil moisture level to be low (”ls?L”) or high (”hs?H”) and switching the pump on/off.

Figure 3.2 shows a DES model of the irrigation system. As compared to representation of the system architecture above at Figure 3.1 we specifically focus on the water pump, sensors and the controller.

The initial state of the system is INIT. The state of the water pump includes the variable ”lev”, which measures the water level and can have one of the following values:

- Low = abnormally low moisture level,

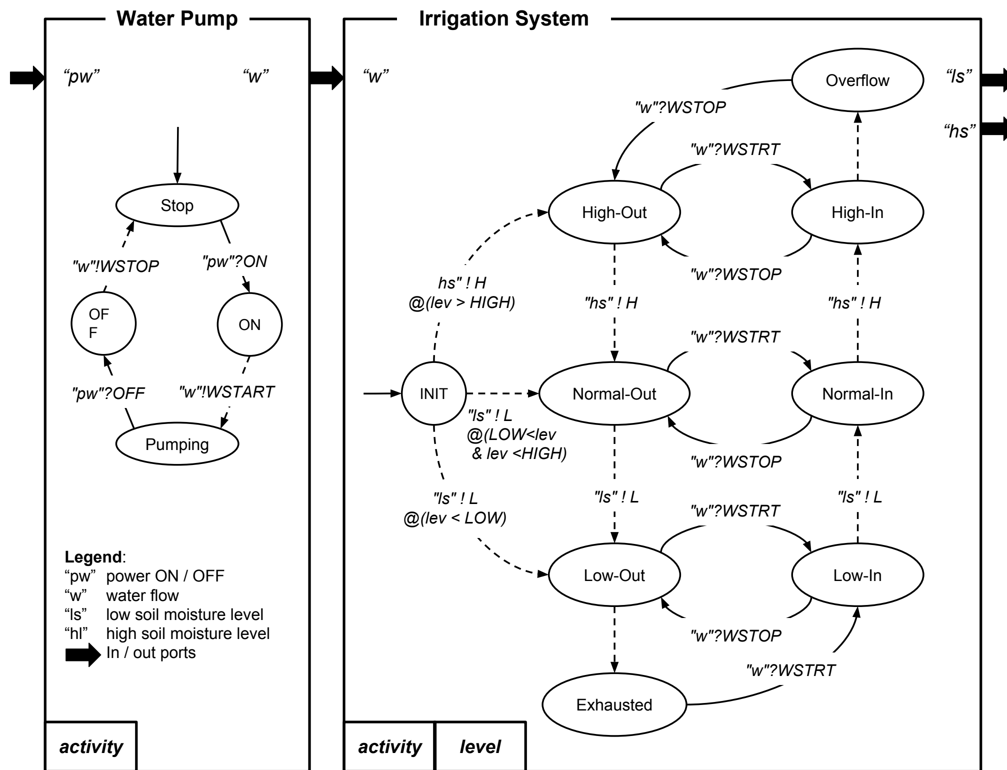

Figure 3.2: State trajectory of the irrigation system.

- Normal = normal moisture level,
- High = abnormally high moisture level.

The input water flow is represented by:

- Out = stopped (pump is not active, and the water is consumed or evaporates). At Figure 3.2 it is shown as state trajectory High-Out → Normal-Out → Low-Out.
- In = flowing (pump is active and the water flows in the irrigation system). At Figure 3.2 it is shown as state trajectory Low-In → Normal-In → High-In.

With water flowing in, the state of the system transitions from Low-In to Normal-In and to High-In moisture levels, generating corresponding sensor signals. If the water continues to flow and the moisture level goes beyond the High level, the system reaches the Overflow state. When the water pump is not active, the water is naturally consumed by the field plants and evaporates during the day. The state of the system transitions from High-Out to Normal-Out and to Low-Out. These transitions will trigger the corresponding sensor signals. If the pump or controller brake, the water flow will stop, and the system will eventually reach the state Exhausted.

The input port of the pump is *"pw"* (power) and the output port is *"w"* (water). If the water pump is turned on *("pw"?ON)*, it starts pumping water through the water pipe *("w"?WSTRT)*. If it is turned off *("pw"?OFF)*, it stops pumping and the flow of water will be stopped *("w"?STOP)*.

Consequently, the pump has four states:

- Stop = passive state,
- ON = transient, issues a start output,
- Pumping = active,
- OFF = transient, issues a stop output.

The two components, pump and irrigation system, are connected through the port *“w”* (pipe). As Figure 3.2 shows, there is just one input port *“pw”* (power ON/OFF) and two output ports for the high and low moisture levels.

3.3 Controlled State Trajectory

The next step is to get a global state transition diagram (GSTD) from the system representation. Let a state *s* in the GSTD be *(P.s, I.s)* where *P.s* denotes a state of the water pump and *I.s* a state of the irrigation system. We formulate the desired state path, denoted K, by the constraints:

1. From any global state (x,y), the state *(Stop,High-Out)* will eventually be reached, where x and y mean any state of the pump and irrigation system respectively. In other words, an active and working irrigating system will always reach the state when the irrigation is complete, and the pump is turned off.
2. $\exists$ no *x* such that *(x, Exhausted)* or *(x, Overflow)*. These are safety constraints.
3. Hysteresis: If *(x,Normal-In)* evolves to *(y,Normal-In)*, then *x* and *y* must be different.

Now we can extract a controlled state trajectory from GSTD and the control objectives. Figure 3.3 features the desired state trajectory of the irrigation system that was built by combining the components showed at Figure 3.2. Initially, the stated is changed to *(Stop, Low-Out)* after a few internal transitions. Then the pump should be *ON*, which is followed by a concurrent event *“w”#WSTRT* (*“w”!WSTRT* and *”w“?WSTRT*).

If the moisture level reaches the high level, the pump should be OFF and water flow should be stopped. Naturally, when the moisture level drops again, the pump should be turned ON again and the process repeats. This cycle of state paths satisfies the three previous objectives.

The next step is to check the controllability of the desired state path. The system must withstand sensor failures and, as a result, avoid reaching overflow or exhausted states. The system controller therefore needs to control the states trajectory transition carefully to avoid the corner cases. Since every successive pair of states is connected by an internal transition with unconditional input, the irrigation system should be strongly controllable. To check this, consider Figure 3.4 in which the transition marked with 1 occurs from *(Stop, High-Out)* to *(Stop, Overflow)*. This is an internal transition which results from the remaining water in the pipe continuing to flow into the system after the pump is turned off. This transition will occur if the moisture sensor is broken or disconnected. However,

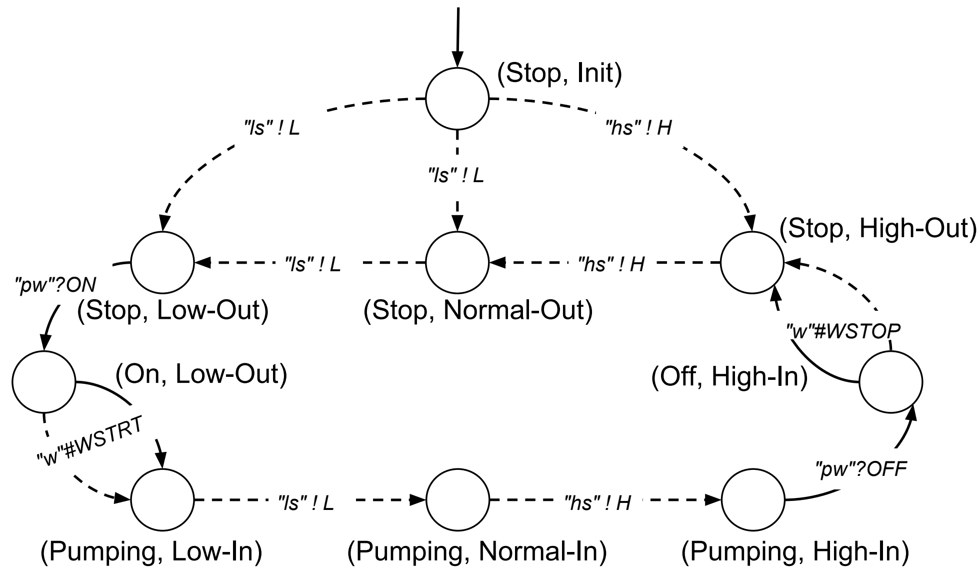

Figure 3.3: Desired state trajectory of the irrigation system.

this internal transition is not on the desired path which requires *(Stop,High-Out)* to be followed by *(Stop, Normal-Out)*, i.e., for the water to start going into the soil after the influx ceases without first overflowing. Moreover, there is no external transition that can rectify the situation since the pump has already been turned off. A similar internal transition, marked 2, occurs when the water is not coming to the system despite the pump being turned back on, due to the transport delay in the pipe.

We see, that the system is in fact weakly controllable. However, it can be made strongly controllable by utilizing the redundancy of the sensors and thus improving the fault-tolerance of the system. Additionally, the controller should consider the time it takes for a water to reach the sensors.

3.4 Deriving a Discrete Event Controller

At this point we have represented a system as a coupled DES, obtained the state transition behavior and interpreted the informally stated controlled objectives as a desired path in the state space of the resultant. This path can be strongly controllable (assuming fault-tolerant sensor deployment) so an associated state trajectory exists. Now, we derive a discrete event controller from the state trajectory using the inverse DES transformation.

Before the transformation, we can reduce the states of the desired state trajectory. The left part of Figure 3.5 is obtained by state reduction from the trajectory in Figure 3.3. A discrete event controller is obtained by inverse DES transformation of the desired reduced state trajectory. The right part of Figure 3.5 shows the DEC after the transformation. Note that the transformation is straightforward and intuitive.

The two components, the controller, and the irrigation system are coupled through ports that have the same name to form a hierarchical coupled DES, shown in Figure 3.6. The dynamics of the resultant design follow the desired state trajectory since the controller

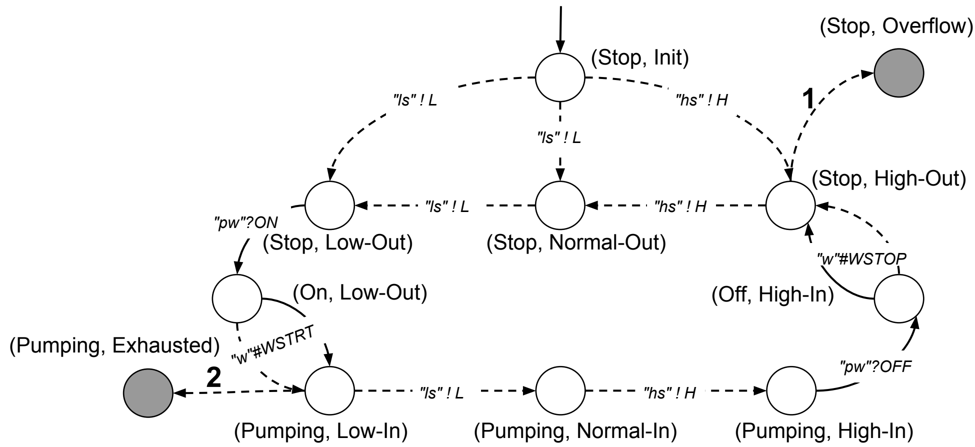

Figure 3.4: Possible state trajectory with high sensor failure.

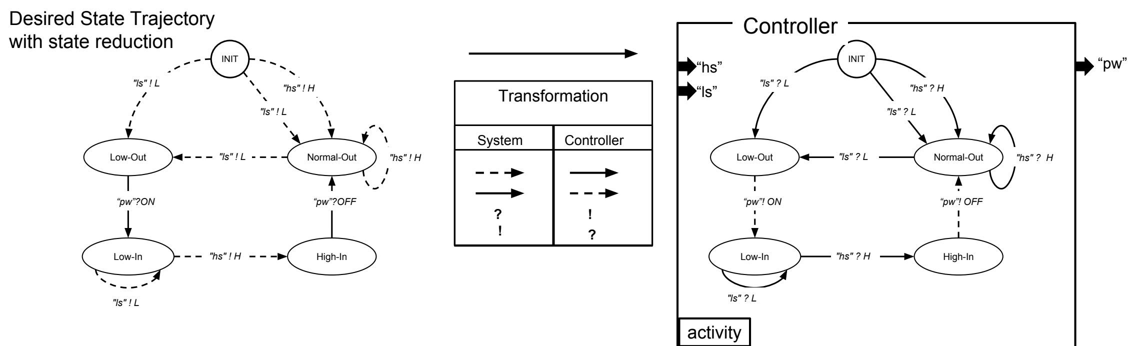

Figure 3.5: Design of a DES controller.

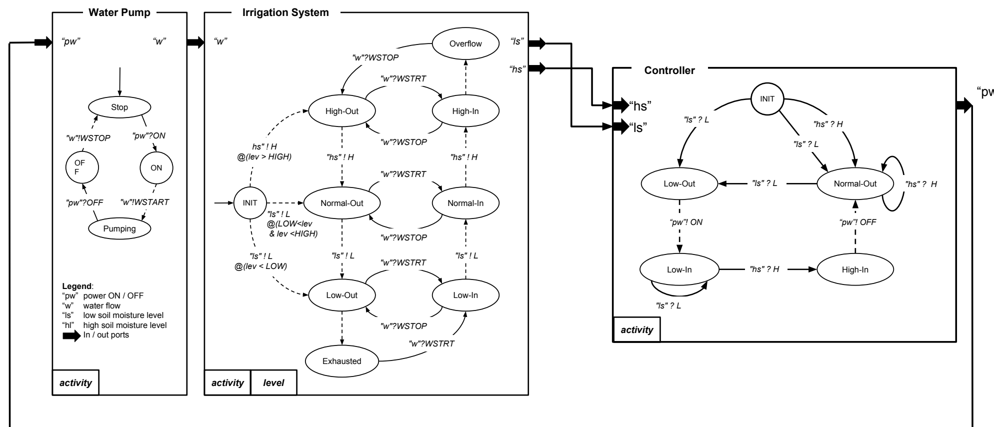

Figure 3.6: Complete irrigation system coupled with the controller.

and the irrigation system, once started in corresponding states, maintain this correspondence forever.

4 Conclusion

By representing an irrigation system as a state trajectory graph, we can verify that desired constraints are met. This is, for instance, important to account for high sensor failure. We can also run a DES simulation to completely explore the space of all the possible states. This can greatly simplify system testing and evaluation, as certain system states are hard to reach with manual testing. We also consider exploring other use cases and different sensors in upcoming months, checking the model failure, and including state probabilities or real time simulations.

Chapter 7

SysML Requirement Diagrams

In this chapter all requirements for the individual projects are summarized and visualized in Requirement Diagrams as standardized in the Systems Modeling Language (SysML) [21].

1 Introduction

SysML is a graphical modeling language for system engineering applications. It is a derivation from UML 2. From SysML we use the Requirement Diagram, which helps visualise system requirements and their relations. Here we give a short explanation of the elements we use from it.

The only block type we use in the diagrams is ”Requirement”. This block type has a title and several attributes. There is a ”Text” attribute which is used for further descriptions if necessary. The ”kind” attribute can have the values ”Functional”, ”Performance” and ”Interface”. The attribute ”verifyMethod” can also only have predefined values which meaning is less intutive as the former attribute which are defined as follows:

”Inspection” is the nondestructive examination of a product or system using one or more of the five senses (visual, auditory, olfactory, tactile, taste). It may include simple physical manipulation and measurements.

”Demonstration” is the manipulation of the product or system as it is intended to be used to verify that the results are as planned or expected.

”Test” is the verification of a product or system using a controlled and predefined series of inputs, data, or stimuli to ensure that the product or system will produce a very specific and predefined output as specified by the requirements.

”Analysis” is the verification of a product or system using models, calculations and testing equipment. Analysis allows someone to make predictive statements about the typical performance of a product or system based on the confirmed test results of a sample set or by combining the outcome of individual tests to conclude something new about the product or system. It is often used to predict the breaking point or failure of a product or system by using nondestructive tests to extrapolate the failure point.

The attribute ”Status” can have the values ”Proposed”, ”Approved”, ”Rejected”, ”Deferred”, ”Implemented”, ”Mandatory” and ”Obsolete” and is meant to track a requirement through the project lifetime.

There are various types of relations between requirements which are depicted as arrows with labels such as a derivation 1.1, a satisfaction 1.2 and refinement 1.3. The arrow direction for ”derive” and ”refine” is a bit counterintuitive, please refer to the example figures for their precise meaning.

There can be also unlabeled dashed arrows which stand for a not further specified dependency. Dependency means a relationship between two requirements where changes in one of them similarly affects the other.

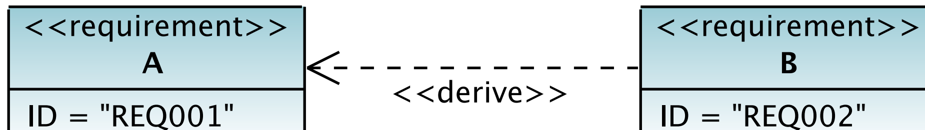

Figure 1.1: The derive relationship shown here means that a requirement B is derived from A.

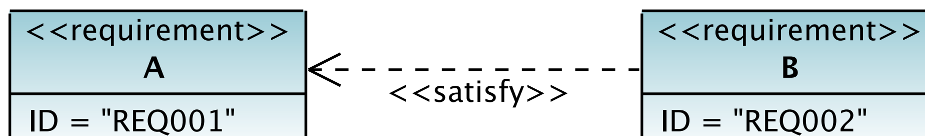

Figure 1.2: The satisfy relationship shown here means that the requirement or implementation B satisfies requirement A.

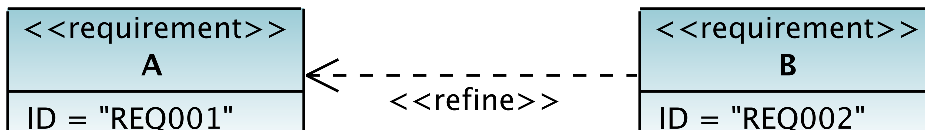

Figure 1.3: The refine relationship shown here means that a requirement B is a refinement of A.

2 Requirements Analyis of Use-Cases

2.1 Requirements Analysis: UPC

Figure 2.1 depicts the requirements for the Guifi.net usecase. To get overview of the network state, the status of the individual network nodes must be collected and stored into a distributed database. To overcome the current shortcomings, the system shall be adapted to the usecase of a big mesh network with unreliable links. This means load must be distributed among monitoring servers, data must be replicated and stored in a distributed database, failures must be detected, and the system must work without any manual intervention. The assignment component should detect and handle inconsistencies in the distributed database and merge them.

2.2 Requirements Analysis: Scality

The requirements for the scality project are depicted in figure 2.2. The system should give access to cloud storage via an S3-like API and store the metadata in a seperate and replicated database using MongoDB. The system should be geo-distributed, scalable and reliable towards outages. Data are immutable. Different consistency levels shall be supported. It should still be possible to write to the backend storage out-of-band.

2.3 Requirements Analysis: Stritzinger

Figure 2.3 shows the requirement analysis for the No-Stop RFID use case from Stritzinger. We started at the toplevel requirements (depicted in pink):

- Allow concurrent writes with last writer wins semantics
- Communication with Maufacturing Procees
- Data of different RFID Tags is independent
- Localize Workpiece
- Mesh Like Network Topology
- Persistence of information
- Runs on existing RFID reader hardware
- Store Processing Information on each Workpiece
- RFID tags shall stop as little as possible

And refined and derived more detailed requirements and proposed implementation detais from those (depicted in aquamarine).

2.4 Requirements Analysis: Gluk

In 2.4 you can see the current state of the requirement analyis for the new use case of Gluk as described in Chapter 6 in this document. Due to time constraints we were only able to show the toplevel requirements from the use case. During the implementation and evaluation this diagram will be furthe refined.

<<requirement>> **Overview of Network State**
ID = "REQ001"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Collect Status from Nodes**
ID = "REQ002"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Read/Write Access**
ID = "REQ008"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Data Replication**
ID = "REQ006"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Redundancy of Monitoring Servers**
ID = "REQ003"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Load Balancing**
ID = "REQ004"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Detect Server Failures**
ID = "REQ005"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **No Manual Intervention**
ID = "REQ007"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **GuifiDB**
ID = "REQ015"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Distributed Database**
ID = "REQ014"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Provide Information about available Monitoring Servers**
ID = "REQ010"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Write Assignment to Distributed Storage**
ID = "REQ011"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Assignment Component reads Monitoring Assignement from Storage**
ID = "REQ012"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Provide Info about Network State and Devices**
ID = "REQ017"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Support initial Assignment**
ID = "REQ016"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Aggregate or Merge Datasets**
ID = "REQ013"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

<<requirement>> **Assignment of Nodes to Monitoring Servers**
ID = "REQ009"
kind = "Functional"
verifyMethod = "Test"
status = "Mandatory"

Figure 2.1: Requirement Analysis UPC Usecase

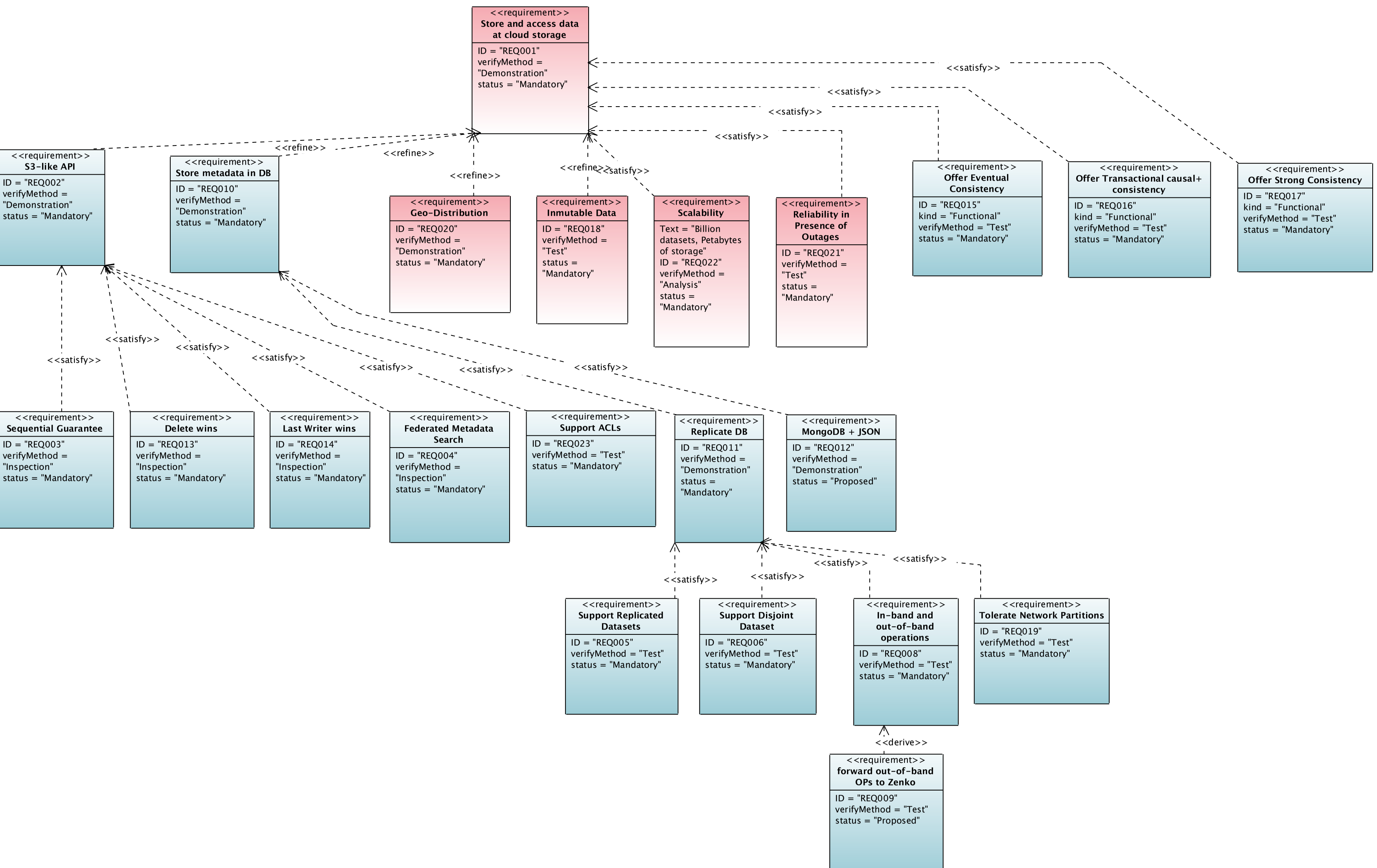

Figure 2.2: Requirement Analysis Scality Usecase

Figure 2.3: Requirement Analysis Stritzinger Usecase

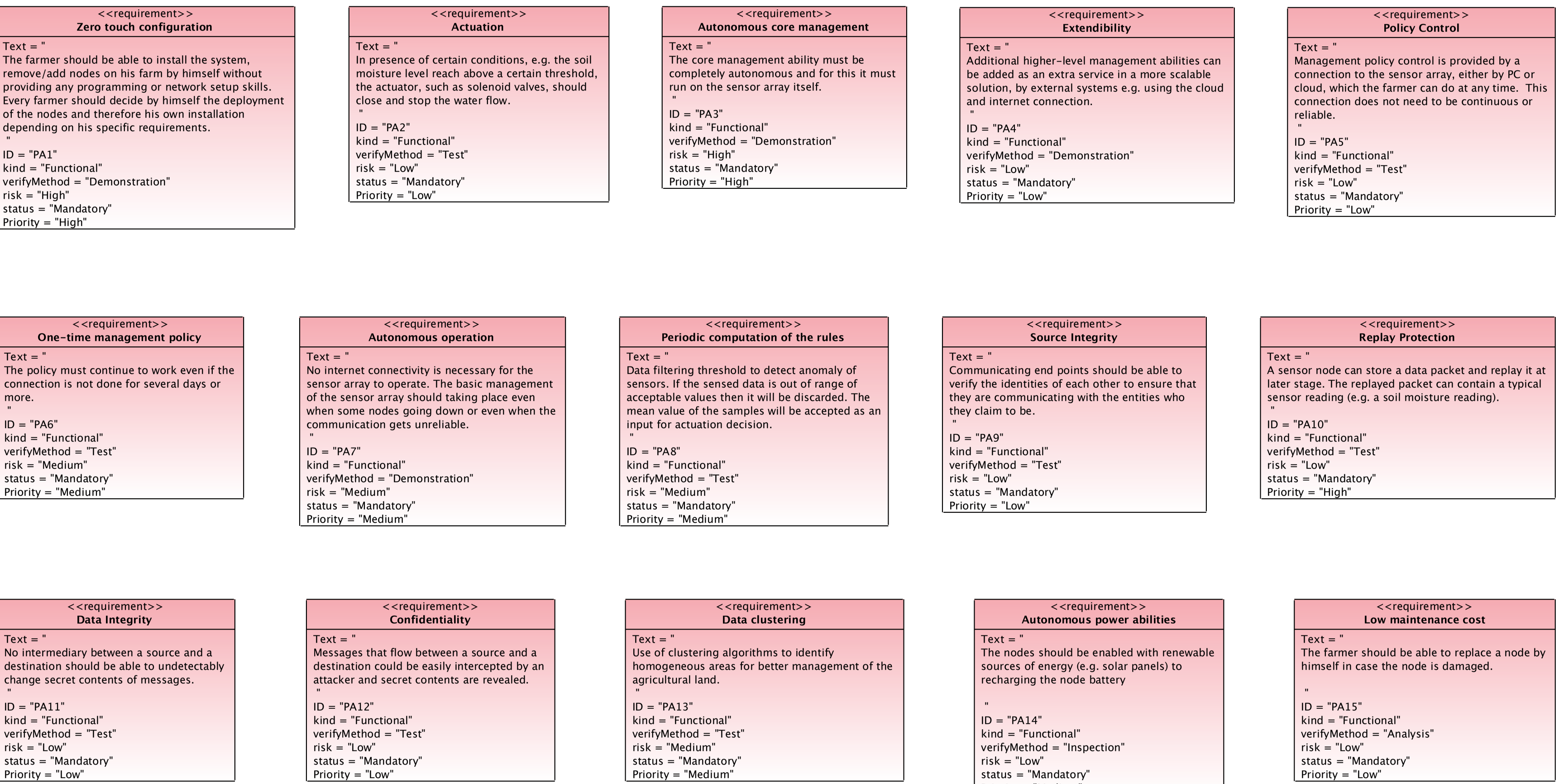

Figure 2.4: Requirement Analysis Gluk Usecase

Bibliography

[1] Farm Oriented Open Data in Europe. http://www.foodie-project.eu.

[2] Flexible and Precision Irrigation Platform to Improve Farm Scale Water Productivity. http://www.figaro-irrigation.net/home/en/.

[3] Internet of food and farm 2020. https://www.iof2020.eu.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[5] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *IEEE 34th Symposium on Reliable Distributed Systems (SRDS 2015)*, pages 31–36, Sept 2015.

[6] Roberto Barbuti and Luca Tesei. Timed automata with urgent transitions. *Acta Informatica*, 40(5):317–347, 2004.

[7] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on Uppaal. In *SFM*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

[8] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.

[9] Sebastian Burckhardt. *Principles of Eventual Consistency*, volume 1. now publishers, October 2014.

[10] Robert Coates, Michael J Delwiche, Alan Broad, Mark Holler, Richard Evans, Lorence Oki, and Linda Dodge. Wireless sensor network for precision irrigation control in horticultural crops. *American Society of Agricultural and Biological Engineers Annual International Meeting 2012, ASABE 2012*, 3, 01 2012.

[11] George S Fishman. Principles of discrete event simulation.[book review]. 1978.

[12] J. Gutiérrez, J. F. Villa-Medina, A. Nieto-Garibay, and M. Á. Porta-Gándara. Automated irrigation system using a wireless sensor network and gprs module. *IEEE Transactions on Instrumentation and Measurement*, 63(1):166–176, Jan 2014.

[13] Mehmet Isik, Yusuf Sönmez, Cemal Yilmaz, Veysel Özdemir, and Ercan Yılmaz. Precision irrigation system (pis) using sensor network technology integrated with ios/android application. *Applied Sciences*, 7:891, 09 2017.

[14] Francisco Javier Ferrández-Pastor, Juan Manuel García-Chamizo, Mario Nieto-Hidalgo, and José Mora-Martínez. Precision agriculture design method using a distributed computing architecture on internet of things context. *Sensors*, 18:1731, 05 2018.

[15] Leslie Lamport. The tla home page. https://lamport.azurewebsites.net/tla/tla.html.

[16] Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[17] Carlos D. Moreno-Moreno, María Brox-Jiménez, Andrés A. Gersnoviez-Milla, Mariano Márquez-Moyano, Manuel A. Ortiz-López, and Francisco J. Quiles-Latorre. Wireless sensor network for sustainable agriculture. *Proceedings*, 2(20), 2018.

[18] Jessica Moysen and Lorenza Giupponi. From 4g to 5g: Self-organized network management meets machine learning. *CoRR*, abs/1707.09300, 2017.

[19] Chris Newcombe. Why amazon chose tla+. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39. Springer, 2014.

[20] Tamoghna Ojha, Sudip Misra, and N Raghuwanshi. Wireless sensor networks for agriculture: The state-of-the-art in practice and future challenges. *Computers and Electronics in Agriculture*, 118, 10 2015.

[21] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.5, 2017.

[22] Paul Regnier, George Lima, and Aline Andrade. A tla+ formal specification and verification of a new real-time communication protocol. *Electronic Notes in Theoretical Computer Science*, 240:221–238, 2009.

[23] Luis Ruiz-Garcia, Loredana Lunadei, Pilar Barreiro, and José Robla. A review of wireless sensor technologies and applications in agriculture and food industry: State of the art and current trends. *Sensors (Basel, Switzerland)*, 9:4728–50, 06 2009.

[24] Amin Ullah, Jamil Ahmad, Khan Muhammad, Mi Lee, Byungseok Kang, Oh Beom Soo, and Sung Baik. A survey on precision agriculture: Technologies and challenges. 12 2017.

Appendix A

Source code

1 Guifi.net TLA+ Specification

MODULE *MonitoringSystem*
EXTENDS *Naturals*, *FiniteSets*, *TLC*

The CONSTANT section defines constant parameters.

CONSTANTS

DEVICES,	set of devices
SERVERS,	set of servers
MinRATE,	monitoring "rate" for each device
MaxCAPACITY,	monitoring "capacity" for each server
MaxFAILURE,	allowed server failures

ASSUME

$\land$ DEVICES $\neq \{\}$	at least one device
$\land$ SERVERS $\neq \{\}$	at least one server
$\land$ MINRATE ≥ 1	device's monitoring rate is at least 1
$\land$ MAXCAPACITY ≥ 1	each server has to monitor at least a device

The number of failures, restricted by the servers monitoring capacity does not lead to an invariant violation.

$\land$(*Cardinality ((SERVERS)) - MaxFAILURE) * MaxCAPACITY*
$\geq$ (*Cardinality ((DEVICES) * MinRATE)*

The VARIABLES section defines constant parameters.

VARIABLES

monitorisation,	*monitorisation* relation between servers and devices
crashed,	set of crashed servers

Record type describing the monitorisation of a device by a server.

$Monitor \triangleq [server : SERVERS, device : DEVICES$

The type-correctness invariant, indicating the possible values that can be assumed by the variables. Variable *monitorisation* describes a global shared database relation is represented by a set of records [server,device], *e.g.*, $monitorisation = \{[srv1, dev1], [srv2, dev1], [srv3, dev2], \ \ldots\}$

$TypeInv \triangleq$
$\quad \land monitorisation \in \text{SUBSET } Monitor$
$\quad \land crashed \in \text{SUBSET } SERVERS$

Function devices returns the devices monitored by $server - srv-$ given a $monitorisation$ relation $-pairs$

$devices(srv, pairs) \triangleq \{x.device : x \in \{p \in pairs : p.server = srv\}\}$

Function servers returns the servers that monitor a $device - sdev-$ given a $monitorisation$ relation $-pairs$

$servers(dev, pairs) \triangleq \{x.server : x \in \{p \in pairs : p.device = dev\}\}$

The initial predicate. Non deterministic assignment of devices to servers.

$Init \triangleq$
$\quad \land monitorisation = \text{CHOOSE} pairs \in \text{SUBSET } Monitor :$
$\qquad \land \forall d \in DEVICES : Cardinality(servers(d, pairs)) \geq MinRATE$
$\qquad \land \forall s \in SERVERS : Cardinality(devices(s, pairs)) \leq MaxCAPACITY$
$\quad \land crashed = \{\}$

Figure 1.1: TLA+ specification of module *MonitoringSystem* (I).

THE ACTIONS

The next-state action is the disjunction of the three actions revoke, assign, and crash.

$revoke(srv, dvc) \triangleq$

Preconditions:

$\land\ [server \mapsto srv, device \mapsto dvc] \in monitorisation$

$\land\ Cardinality(servers(dvc, monitorisation)) \geq MinRATE + 1$

Update state variables with the new monitorisation pair:

$\land\ monitorisation' = monitorisation \setminus \{[server \mapsto srv, device \mapsto dvc]\}$

$\land$ UNCHANGED $\langle crashed \rangle$

$assign(srv, dvc) \triangleq$

Preconditions:

$\land\ srv \notin crashed$

$\land\ [server \mapsto srv, device \mapsto dvc] \notin monitorisation$

$\land\ Cardinality(servers(dvc, monitorisation)) < MaxCAPACITY + 1$

Update state variables by removing the monitorisation pair:

$\land\ monitorisation' = monitorisation \cup \{[server \mapsto srv, device \mapsto dvc]\}$

$\land$ UNCHANGED $\langle crashed \rangle$

Action to be refined so that server failure and reassignment of its devices does not occur simultanoeusly. This might lead to temporary invariant violation for the devices monitored by $-srv-server$.

$crash(srv) \triangleq$

Preconditions:

$\land\ srv \notin crashed$

$\land\ Cardinality(crashed) < MaxFAILURE$

Update state variables by assigning the devices being monitored by $server - srv -$ to other (non-crashed) servers.

$\land\ crashed' = crashed \cup \{srv\}$

$\land$ LET $devs \triangleq devices(srv, monitorisation)$

$\quad assignment \triangleq \{p \in monitorisation : p.server = srv\}$

$\quad live \triangleq SERVERS \setminus crashed'$

$\quad reassignment \triangleq$ CHOOSE$pairs \in$ SUBSET$[server : live, device : devs]$:

$\qquad \land \forall d \in devs : Cardinality(servers(d, pairs)) \geq MinRATE$

$\qquad \land \forall s \in live : Cardinality(devices(s, monitorisation)) \leq MaxCAPACITY$

IN $monitorisation' = (cmonitorisation \setminus assignment) \cup reassignment$

Figure 1.2: TLA+ specification of module *MonitoringSystem* (II).

Invariant: All devices are monitored by at least *MinRATE* devices.

$Resilience \triangleq \forall d \in DEVICES :$
$\quad Cardinality(servers(d, monitorisation)) \geq MinRATE$

Invariant: The number of failures, restricted by the servers monitoring capacity allows devices to be monitored by at least *MinRATE* devices.

$LimitFailures \triangleq$
$\quad (Cardinality(SERVERS) - MaxFAILURE) * MaxCAPACITY$
$\quad\quad \geq (Cardinality(DEVICES) * MinRATE$

Invariant: The number of devices monitored by each servers does not exceed the allowed capacity.

$LoadBalancing \triangleq \forall s \in SERVERS \setminus crashed :$
$\quad Cardinality(devices(s, monitorisation)) \leq MaxCAPACITY$

Invariant: Crashed servers do not monitor any devices.

$Crashed \triangleq \forall s \in crashed : devices(s, monitorisation) = \{\}$

The specification's next-state action.

$Next \triangleq \exists s \in SERVERS, d \in DEVICES :$
$\quad revoke(s, d) \lor crash(s)$

The system invariants.

$Invariants \triangleq \Box(Resilience \land LoadBalancing \land LoadBalancing \land Crashed)$

The complete specification.

$Spec \triangleq$
$\quad \land Init$
$\quad \land \Box[Next]_{\langle monitorisation, crashed \rangle}$
$\quad \land TypeInv$
$\quad \land Invariants$

Figure 1.3: TLA+ specification of module *MonitoringSystem* (III).