Master Thesis

Design of a development platform to monitor and manage Low Power, Wide Area WSNs

J.J. Schutte, University of Twente
j.j.schutte@student.utwente.nl

June 27, 2019

Abstract

The recent explosion of Low Power Wide Area (LPWA) WSN devices has raised interest in perceiving the Quality of Service (QoS) provided to and by such applications. Current QoS solutions do not respect LPWA-specific considerations, such as limited resources and extreme scale. This study has set out to research an appropriate solution to QoS monitoring and management that does concern these considerations. This is achieved by establishing a development platform focused on LPWA QoS. The platform consists of two chief concepts. The first of which is a distributed stream processing architecture. The architecture backbone is based on Apache Storm and provides scaffolding for different classes of stream transformations, which guides users in implementing their monitoring applications. The second artefact is a model capable of captivating resources and calculating the performance of a system, considering different modes of operation of that system. The proposed development platform is validated by implementing an instantiation of it, based on an actual, commercial on-street parking application. Though the study shows some deficiencies still present in the solution, its results demonstrate it as an applicable and feasible aid in constructing scalable applications capable of QoS monitoring in LPWA WSNs.

Keywords: Wireless Sensor Networks, Internet of Things, LPWA, Quality of Service

Committee

This thesis was supervised and examined by:

prof.dr.ir. M. Aksit	University of Twente, Formal Methods and Tools
dr. N. Meratnia	University of Twente, Pervasive Systems
R. Boland	Nedap N.V., Identification Systems

Contents

1 Introduction **4**
1.1 Domain overview . . . 4
1.2 Challenges in monitoring QoS in LPWA . . . 5
1.3 Current State of the Art . . . 7
1.3.1 QoS protocols . . . 7
1.3.2 QoS platforms . . . 8
1.3.3 Deficiencies in current state of art . . . 9
1.4 Contribution of this Thesis . . . 11
1.4.1 Goal . . . 11
1.4.2 Research questions . . . 11
1.4.3 Approach . . . 12
1.5 Thesis organization . . . 13

2 Background **14**
2.1 Context of the project . . . 14
2.2 Commonality/variability analysis . . . 16
2.3 Distributed computation technologies . . . 17
2.3.1 Monolith vs. micro-component . . . 17
2.3.2 Apache Storm . . . 18
2.3.3 Message brokers . . . 19
2.3.4 Distributed processing . . . 20
2.4 Quality of Information of WSN data . . . 21
2.5 Constraint programming and solving . . . 22

3 Design of WSN monitoring platform architecture **25**
3.1 Objective of this chapter . . . 25
3.2 Conceptualization of the problem domain . . . 25
3.3 Requirements for the proposed software platform . . . 27
3.4 Evaluation of the solution domain . . . 28
3.4.1 Solution decisions . . . 30
3.5 Design of the software platform . . . 31
3.5.1 Micro-component architecture . . . 31
3.5.2 Scaffolds for micro-components . . . 34
3.6 Demonstration by example case . . . 37
3.6.1 The example case . . . 37
3.6.2 Application of the platform . . . 38
3.7 Discussion of the proposed software platform . . . 40

4 Resource Distribution Model **43**
4.1 Objective of the model . . . 43
4.2 Conceptualization of the problem domain . . . 43
4.3 Requirements for the proposed model . . . 45
4.3.1 Requirements . . . 45
4.3.2 Justification of identified requirements . . . 46
4.4 State of the art of the solution domain . . . 46
4.4.1 State of the art . . . 46
4.4.2 Evaluation of the solution domain . . . 47
4.4.3 Choices of employed solutions . . . 48
4.5 Design of the Resource Distribution Model . . . 48
4.5.1 Demonstration by example case . . . 51
4.5.2 Computing a valid, optimal model assignment . . . 51
4.6 Discussion of the proposed model . . . 53

5 Proof-of-concept validation by case study **57**
5.1 Context of the case study . . . 57
5.1.1 Background . . . 57
5.1.2 Conceptualization of the monitoring application . . . 58
5.2 Validation method . . . 60
5.2.1 General approach . . . 60
5.2.2 Claims . . . 60
5.2.3 Bounds . . . 62
5.3 Implementation of the WSN monitoring application . . . 62
5.3.1 Design and Implementation . . . 62
5.3.2 Adapting the application . . . 67
5.4 Results & Evaluation . . . 69
5.5 Conclusion & Discussion . . . 72
5.5.1 Conclusions . . . 72
5.5.2 Discussion . . . 73
5.5.3 Limitations and recommendations . . . 75

6 Conclusion & Discussion **77**
6.1 Conclusions . . . 77
6.1.1 Platform architecture . . . 77
6.1.2 Resource Distribution Model . . . 79
6.2 Discussion . . . 80
6.3 Future work . . . 83

1. Introduction

1.1 Domain overview

Wireless Sensor Networks (WSNs) have received large amounts of research the past decades. However, this mainly resulted in isolated ad hoc networks. With both the size of WSN's and the amount of networks increasing, the deployment of multiple networks in the same geographical area for different applications seemed increasingly illogical. Therefore, recent endeavours have attempted design networks and protocols in order to create a general, ubiquitous internet for automated devices and sensors: the Internet of Things (IoT). A specific recent development in IoT has focussed on the field of Low Power Wide Area networks (LPWA). These networks serve devices that communicate over large distances with very limited computational and communication resources [1]. They therefore entail low data rates, low radio frequencies and raw unprocessed data.

These extremely restrictive requirements entail that a regular wireless internet connection does not suffice, as it is not optimized for the extreme resource limitations of LPWA WSN applications. Multiple corporations are developing and deploying exclusive wide area networks for low powered devices. Examples of these networks are Narrow-Band IoT [2], LoRaWAN [3] and Sigfox [4]. These networks are deployed and operated by telecom providers and allow instant connectivity by incorporating a SIM or proprietary network connectivity module. As a consequence large-scale LPWA applications are moving from node-hopping and mesh network strategies to operated cell networks [5, 6]. Because of the aforementioned reasons the number of connected devices has exploded in the recent years. Estimations vary but a consensus established from multiple sources predict about 15-30 billion connected devices in 2020 [7, 8, 9, 10]. This would imply that by 2020 the number of connected IoT/WSN devices will have surpassed the number of consumer electronic devices (e.g. PC's, laptops and phones) [10].

Both the explosion of devices, entailing explosion of data, and the shift to shared operated cell networks implies a great stress on monitoring sensor applications. While relatively small sized applications on proprietary networks allow for a best-effort approach, the convolution of many large applications on a shared network requires knowledge of the performance provided by the application. The term coined for this is Quality of Service (QoS). QoS parameters such as application throughput, service availability and message drop allow the description of the performance state of a system or application [11]. It is therefore paramount for a commercial application to have its QoS metrics observed.

The notion of QoS in a networked application is not a novel concept. It has been a research and industry paradigm for as long as commercial applications have existed. Consequently, many forms of QoS monitoring and management exist for regular internet and networking applications. However, these methods do not transfer well to the field of WSN and IoT, as will become apparent in this section. This presents a vacancy that requires exploration. Access to such QoS solutions will improve the maturity and operational feasibility of commercial, large-scale IoT applications.

The remainder of this introductory chapter will determine some of the key challenges which differentiate QoS monitoring of regular networks and wireless sensor networks. The next section will deliberate some key obstacles in the current state of art of monitoring Quality of Service in LPWA Wireless Sensor Networks. Subsequently, it will be deliberated why existing solutions cannot provide for the QoS monitoring needs of LPWA applications. After which the succeeding section will introduce the proposed approach to design a development platform for applications to deal with these challenges and capture the QoS in WSN's.

1.2 Challenges in monitoring QoS in LPWA

Three key challenges were identified that fundamentally complicate QoS measurement and management in LPWA networks and applications. These challenges affect the applicability of conventional QoS mechanisms to the field of IoT and WSN.

Technical limitations of end-devices

The first challenge of LPWA applications are the previously mentioned extreme resource constraints [1, 12]. For example, LPWA devices are expected to communicate on a network shared by a vast amount of nodes, diminishing the individual connectivity resources. As a consequence, uplink communication is regularly aggregated over time and transmitted opportunistically. Therefore, back-end applications are required to facilitate irregular and infrequent reporting intervals from sensor nodes. Additionally, an LPWA device is required to perform for a certain amount of time, typically at least 10 years [13, 14, 15], on a finite battery energy supply. Therefore, there are no resources to spare for expensive auxiliary processes [16]. Consequently, devices usually send low-level auxiliary data, instead of intelligently derived values. The burden of calculating high level information is then deferred to be computed in-network (edge computing) or at the back-end application server.

Additionally, evolution of sensor device software is far more restrictive then evolution of back-end application's software. Firstly, because of the long lifetime of devices, it can occur that services based on modern day requirements need to be performed by decade old technology. Secondly, most LPWA networking protocols do not require devices to retain a constant connection in order to save energy (duty cycling) [13, 14, 16, 17]. Instead, the devices connect periodically or when an event/interrupt occurs. This entails that devices are not updated *en masse*, but individually when a device wakes up. As this requires additional resends of the updated code it consumes more connectivity resources in the network.

For these reasons LPWA sensor applications often employ a *"dumb sensor,* *smart back-end"* philosophy. Consequently, the computations are deferred to the network, back-end or cloud [18, 19]. The problem however with deferring the computations further to the back-end is that more and more computations have to be performed centralized. This requires the back-end to be extremely scalable because more tasks need to be performed as more devices are added to the application [20, 21, 22].

IoT QoS is different

Aside from the low-level information sent by the large amount of devices, QoS in WSNs is distinctly different from classical client-server QoS. Often QoS in a client-server application can be measured at the server. QoS monitoring in a cloud environment may require some aggregation of data, but even then the number of data sources is relatively limited. Large WSN applications require data aggregation by default. As the Quality of Service provided by the application can only be ascertained by calculations based on auxiliary data collected from a huge number of devices. This concept is known as Collective QoS [23] and comprises parameters such as collective bandwidth, average throughput and the number of devices requiring replacement. As this information eventually requires accumulation on a single machine in order to determine concrete values, aggregation of expansive volumes of auxiliary sensor data must be performed intelligently as not to form a congestion point or single point of failure.

However, device level information is still required alongside of collective QoS [24]. If a device is not performing according to expectations of a predetermined strategy, it is required that this is mitigated or informed. This introduces a second distinction to classical QoS: multi-level monitoring and reporting. Conventionally, only the QoS provided by the server(s) running an application is of interest. However, in a wireless sensor environment, monitoring of parameters on different levels is required. Examples of these monitoring levels are single sensor, the application as a whole or analysis per IoT cell tower or geographic area. This requirement entails data points of different levels of enrichment, calculated from the same raw sensor data.

The final distinction in IoT monitoring is the dynamic nature of WSN applications [18]. Firstly, an IoT monitoring application needs to be prepared for devices added to the network and dropping out of the application [25]. As a collective QoS parameter is based on a selection of devices, the monitoring application must support adding and remove devices from the equation. Additionally, diverse deployment of nodes causes them to behave differently. Therefore, QoS procedures should account for the heterogeneity exhibited throughout the WSN [16].

In conclusion, IoT QoS management will require a flexible and dynamic method of resource parameter modelling. Additionally, this process should be able to be applied to a high influx of sensor date. This monitoring technique should be able to calculate both lower level (single sensor) and higher level (application) resource distribution.

Movement to operated cell network

A final challenge in contemporary QoS monitoring of LPWA applications is the earlier recognised increasing trend of shared, telecom-operated cell networks [13, 14]. Though it makes IoT connectivity more efficient because many applications can be served by a single network infrastructure, it effects complications to the QoS. Firstly, Many applications will be competing for a shared scarce amount of network resources. When other applications consume a large portion of the resources, due to poor rationing or event-bursts, your application suffers and cannot provide the expected QoS.

Secondly, by out-sourcing the network infrastructure, control over the network is lost. Though beneficiary to the required effort, some important capabilities are conceded. For example the network can no longer be easily altered in order to suit the needs of the application. Additionally, auxiliary data can not be extracted from the network and edge computing is not an option, again deferring the burden of aggregating QoS data entirely to the back-end.

Finally, the telecom operator will require adherence to a Service Level Agreement (SLA). Though this ensures a certain service provided to an application and prevents other applications of consuming extraneous resources, it also requires close monitoring of applications. A breach of the SLA may cause fines or dissolving of a contract. Therefore, strict adherence to the SLA parameters is necessary and timely proactive intervention is required, if the limits of the SLA are threatened to be exceeded [26].

To summarize, outsourcing the management of the network infrastructure to a professional telecom provider aggravates the need for exact and real-time curtailment of digital resources, while simultaneously impeding the ability to do so in the network itself. This will need to be remedied by adapting the parts of the WSN architecture within the domain of control, i.e. the sensor devices and the back-end application. Because of the earlier proposed concerns and challenges this increased responsibility will be mostly attributed to the back-end application.

1.3 Current State of the Art

The previous section illustrated some key challenges in measuring and determining QoS in WSNs. This section will deliberate on some known QoS protocols and existing monitoring solutions. It will conclude by arguing why the current state of the art does not provide a suitable solution for the previously identified challenges.

1.3.1 QoS protocols

The first well known protocol often employed for QoS monitoring is SNMP [27]. SNMP provides a formalized, device-independent addressing scheme to request key device and networking data points. Additionally, it allows application developers to specify custom addressable data points. Though SNMP does not feature command and control capabilities, the information obtained by it can be used to configure and control an application by other means.

A protocol that does feature such command & control capabilities is Integrated Services (IntServ) [28]. This protocol negotiates a resource allocation in the network per data flow. This allocation is then permeated throughout the network domain and retained until the data flow has ended. It provides hard QoS guarantees within the network, but at a severe preparation cost and overhead.

A more cost-efficient QoS protocol is Differentiated Services (DiffServ) [28]. This protocol does not require resource negotiation and instead identifies differentiating traffic classes. Depending on the determined class, the data will enjoy specific benefits such as priority handling or increased network resources allocation. Though the QoS guarantees provided by this protocol are softer than that of IntServ, it also generates vastly less overhead.

The former protocols are all general application networking protocols. Though there are proposals for IoT-specific QoS monitoring frameworks. A promising solution is presented by R. Duan et al [29]. This framework aims for an automated negotiation procedure between node, network and back-end layers in order to deliberate a reporting level that compromises the monitoring needs with the available resources and device capabilities. In this manner it can offer the greatest benefit to QoS without considerably impacting it negatively.

1.3.2 QoS platforms

Aside from protocols managing QoS there also exist some IoT platforms that are capable of (or enable) some form of QoS monitoring. This section will detail three of them and how they curtail the posed challenges or are invalidated by them.

PTC ThingWorx

PTC ThingWorx [30] is a proprietary IoT PaaS solution developed by PTC. It is a full-scale cloud platform offering many prepackaged IoT support services. The focus of this platform is on rapid application design, development and deployment. The aim of the ThingWorx team is to offer the ability to develop IoT applications without coding and instead device an application by only using the ThingWorx application interface. This simplifies the development cycle and shortens time-to-market [31]. Though it is capable of monitoring the performance of an application, the focus of the platform is on application development and data management. Therefore, employing it for performance monitoring only might be a disproportionate approach, especially considering that ThingWorx is a paid platform. Additionally, only using a small section of the platform's functions might lead to installing bulky, cumbersome agents in sensor devices. This will potentially unnecessarily consume resources of a constraint device. Aside from the previously mentioned extravagances, sources report that ThingWorx has scalability problems [32].

Cisco Jasper Control Center

Cisco has extended its Jasper cloud platform and has optimized it for several IoT markets. This extension includes a product specifically designed for LPWA IoT applications named the Control Center for NB-IoT [33]. It is specifically designed for SIM-connected (LTE) device connectivity management [34]. It accomplishes this through Cisco's proprietary network hardware and partnerships with mobile operators that incorporate data extraction end-points in their devices. Jasper therefore focuses on data and information obtained from network nodes and edge computation instead of communicating with actual end-devices. This decreases the burden on resource constraint devices and alleviates the challenge posed by the movement to provider operated cell networks. However, in doing so it neglects information that can only be acquired by node inspection.

Jasper Control Center allows the usage of business rules for information extraction and actuation, and can employ outbound communication channels (e.g. email or SMS) for alerting purposes. In addition it includes API's for more complex further analyses. Jasper Control Center is a proprietary SaaS solution which can be procured in packages. However, the basic packages seems to only include minimal functionality and more advanced functions such as rule-based automation and third-party API access are sold in separate additional packages [33]. Finally, Jasper Control Center can report on a few Collective QoS parameters (e.g. data usage, number of reports received), but it has been reported that Jasper lacks in analytic functionality [34].

Nimbits

Nimbits [35] is an open-source cloud data logger and analysis PaaS. It employs a rule-based engine to filter, log and process incoming data. Additionally, rules can be defined to instruct the engine to report alerts via external communication channels. It operates by defining data points to which sensors and servers can write and read data [31, 36]. Devices can do so by employing a Nimbits client or via HTTP API's. It has been reported that Nimbits can communicate via the light-weight MQTT protocol [36], but documentation demonstrating this is lacking. It therefore appears that Nimbits lacks the considerations required for resource constraint LPWA devices.

Nimbits is not primarily intended as a QoS monitoring platform, but can be configured as such by regarding auxiliary QoS data as primary data of a dedicated QoS monitoring application. However, after analysing Nimbits's design of data points, Nimbits seems to be most appropriate for applications with a small pool of distinct sensor types. Establishing and managing data points for a colossal amount of devices of equivalent data types, as a monitoring job will often encompass, rapidly becomes a cumbersome effort to automate.

1.3.3 Deficiencies in current state of art

QoS protocols

The protocols described in Section 1.3.1 are unfortunately not applicable to the LPWA WSN domain. Firstly, SNMP generally operates according to a master-slave architecture [37] which requires slaves (sensor nodes) to remain on-line permanently, or at least regularly [38]. This demand is invalidated by the resource restriction complication featured in LPWA applications [12]. This can be partly alleviated by proxying the sensor devices by a proxy that is less resource constrained. This would however come at the cost of a lack of real-time data or delayed response times [39]. Therefore, a more appropriate solution would be to employ a client-initiated approach. Furthermore, SNMP and related protocols consider end-to-end QoS. As discussed in Section 1.2, WSN application monitoring must consider both end-to-end and Collective QoS. Therefore, even if SNMP is employed, further processing is required.

	ThingWorx	Cisco Jasper Control Center	Nimbits
LPWA specific[1]	✗	✓	✗
QoS monitoring focus	✗	✗	✗
Open-source	✗	✗	✓
Device-level inspection	✓	✗	✓
Extreme scalability	✗	✓	✗

Table 1.1: Comparative analysis of IoT QoS monitoring platforms

Though IntServ's hard QoS guarantees are powerful, the overhead required to establish these flows is far too imposing [40, 41]. Since LPWA only sends small message payloads, the heavy per flow negotiation data will easily exceed the payload data. With LPWA's limited resources in mind this cannot be considered as an efficient solution. Conversely, DiffServ does not feature this immense overhead cost. However, application of the protocol is complicated by the movement to commercial network operators, as it would require them to implement a class-based allocation system in their networks. The previously mentioned inhibitions are potentially aggravated by local net neutrality laws. Though this was not a concern in privately operated proprietary networks, in universal Internet of Things extreme networks severe net neutrality laws may prohibit priority treatment of data flows based on their source, destination or content [42]. This implies that the required QoS guarantees cannot feasibly or legally be (fully) provided by a commercial Internet of Things network provider and in-network protocols.

Furthermore, both IntServ and DiffServ consider only network QoS, therefore they lack the level of inspection to report or consider the state of limited resources in end-devices. This deficiency also troubles IoT-specific QoS protocols. Most efforts are focussed towards efficient and effective networking in order to facilitate increasing data-rates. These protocols disregard important device metrics, such as node lifetime and sensor measurement accuracy, which are paramount to determining the health and performance of an IoT application. Finally, though the protocol of R. Duan et al [29] does feature this level of inspection, the details require further implementation to fully complete the protocol. Since the field of IoT is relatively young, no such IoT-specific QoS procedures have matured to a uniform and universal internet standard. From the preceding it is concluded that contemporary general purpose or IoT-specific QoS protocols cannot provide for an adequate in-network solution. Instead, this obligation is imposed on the back-end and the end-devices.

QoS platforms

An assessment of the discussed platforms and their applicability to the field of LPWA is depicted in table 1.3.3. It shows that these platforms are all lacking in some important considerations. These platforms are either not conceived with a focus on LPWA's severe resource constraints, a primary focus on resource and QoS monitoring or the extreme scale of contemporary WSN applications [31, 32, 34, 43].

[1] I.e. constrained by resource limitations

These deficiencies make the existing monitoring platforms insufficient solutions for monitoring and controlling large-scale LPWA IoT applications. This implies that the technologies are either inapplicable or require a composition of these technologies. This complication of the technology stack could be acceptable for a key function of an application, but not for an auxiliary monitoring processes. As not to complicate a software product which does not enjoy the main focus of development efforts it would be beneficiary to have a versatile platform which enables development of a single monitoring and management application [44]. The preceding concludes a vacancy in the current state of the art. The remainder of this chapter will be devoted to how this vacancy is proposed to be absolved.

1.4 Contribution of this Thesis

The preceding sections have demonstrated that LPWA-specific challenges leave a deficiency in WSN QoS monitoring and management which contemporary QoS management solutions cannot absolve. This section will proposition how the deficiency in the current state of affairs is aimed to be abridged. First, the overall goal of this thesis will be clearly stated. After which, the goal will be explicated into distinct research questions. Finally, the general approach to absolve this deficiency will be covered shortly.

1.4.1 Goal

The goal of this study is to research and develop a development platform providing capabilities of measuring and monitoring QoS parameters of LPWA WSN applications. This platform will be devised to overcome the challenges identified in Section 1.2. To reiterate, these core challenges are: the deference of processing to the back-end due to restricted processor capabilities and obscuration of the network, and the unique QoS challenges in WSN networks such as multi-level abstractions and aggregation of massive amounts of multi-sourced snapshots. The platform to be designed will enable development of support applications that process auxiliary IoT data. This data is raw and low-level, but is enriched by the platform by associating streaming data with data obtained from relevant data sources and aggregating streaming data to infer higher-level information. This information can be exported for reporting and visualization purposes, can alter the state of a system (single sensor, group of sensors, entire application, etc.) and can cause alerts to be dispatched for immediate intervention.

1.4.2 Research questions

To accomplish the goal set out for this study the following question require answering.

RQ1 What are the key data transformations and operations that are performed to process and enrich (auxiliary) data streams produced by WSNs?

RQ2 How to design a platform that facilitates the identified WSN data streams, transactions and operations?

RQ3 What is the appropriate level of abstraction for a WSN monitoring platform, such that

- the platform is applicable to monitoring a large domain of WSNs,
- provides for minimal development effort, and
- supports evolution of the application.

RQ4 What are the challenges regarding scalability in a WSN data stream processing platform?

RQ5 How can these challenges be overcome?

RQ6 What are the key concepts regarding modelling and calculation of QoS parameters?

RQ7 How can the state of a system with variable behaviour be modelled?

RQ8 How can the optimal system behaviour be determined, in accordance with its state?

The listed research questions feature a focus that is twofold. The first point of focus is the design and development of an abstract, scalable streaming platform for IoT data enrichment. The associated questions are RQ1–5. It concerns the appropriate abstraction of a platform combatting the challenges in iteratively refining low-level sensor data to high-level information with business value and scalability due to the vast amount of data generated by the WSN. The second focal point concerns the representation and processing of information depicting the state of a system under investigation. This entails capturing some data points produced by sensor devices or intermediary processes, calculating the derived parameters from those measurements and producing a decision in accordance with the model's values and set rules. This focal point is represented by research questions RQ6–8.

1.4.3 Approach

With the goal and research questions defined, The general method intended to accomplish this goal will be clarified.

As the previous section mentioned, the research questions can be divided into two categories: The design of the platform and modelling the distribution of resource and QoS parameters. The approach is therefore to research these individually before integrating the efforts into one resulting software development platform. First, the design of a processing platform architecture will be explored. This platform endeavours to compete the challenge of immense influx of data. Additionally, it will feature multi-stage calculation and enrichment in order to provide for the need of multi-level QoS processing and reporting. Subsequently, a modelling method capable of captivating the distribution of resources and interconnectivity of QoS will be researched. This model will again take into account the multi-level modelling needs in accordance with the identified challenge. Additionally, it will combat the challenge of enriching deferred low-level data to high level usable information by allowing transformations of resource parameters.

Both individual points of focus — i.e. the processing platform and the resource model — will be devised, designed and developed according to the following schedule. First, the problem domain of the to be designed artefact will be explored. This will be performed with a commonality/variability analysis (Section 2.2). This analysis allows the determination of the appropriate level of abstraction. This analysis will result in a list of requirements for the solution to adhere to. With the requirements defined, the state of the art of the solution domain will be explored to identify viable technologies and their deficiencies, before selecting the best applicable technologies. With these technologies identified they will be adapted and the intended artefact will be designed and developed. To concretize the application of the designed artefacts, an instantiation based on a hypothetical use case will be provided. This instantiation will assist in comprehending the abstract concepts offered by both the platform and the resource model. Ultimately, the devised solution will be evaluated and discussed by paralleling them to the set requirements and some additional concepts and criteria.

Finally, the conceived model will be incorporated in the larger context of the developed platform architecture. Once the two concepts have been compounded into a single solution, the challenges it claims to combat will need verifying. A proof-of-concept validation study will be performed by applying the platform to a real-world commercial LPWA WSN on-street parking application developed and operated by the Dutch company Nedap N.V. This will be achieved by providing a prototype implementation of the constructed platform. By examining the development process and the resulting solution, the validity of the designed artefact(s) will be investigated. The three metrics the implementation will be evaluated on are the applicability, ease of implementation and adaptability of the implementation. The first is validated by whether a satisfactory implementation for the case can be instantiated. Should such an instantiation be achievable, the level of abstraction and utility offered will be evaluated according to the code required to realize that instantiation. Finally, should the development platform provide adequate means for separation of concerns, evolution of the instantiation should prove facile. This capacity for evolution will be validated by hypothesizing three simple adaptations to the context or requirements of the applications. If the asserted flexibility is provided, these changes should be able to facilitated with minimal, localized changes in the application. Ultimately, the validation study will be concluded with a summation of the obtained results and conclusions, and their implications to future development and research.

1.5 Thesis organization

The remainder of this thesis is structured as follows. Chapter 2 will briefly elaborate on some background concepts required for the understanding of this thesis. Chapter 3 will depict the design of the proposed distributed architecture for the QoS monitoring platform. In Chapter 4 the proposed model capable of calculating the state and optimal performance of a system will be discussed. The two aforementioned artefacts will enjoy preliminary validation in a proof-of-concept study in Chapter 5. Finally, the thesis will be concluded in Chapter 6, which will discuss the efforts and results of this study, and will provide suggestions for continued research.

2. Background

2.1 Context of the project

First, this section will scope the efforts the project. This will be achieved by two analyses. Firstly, the set of target applications will be described in abstract concepts. Secondly, the efforts will be focussed be defining the stakeholders that are affected be an implementation of the intended monitoring platform.

Defining the set of applications

As stated before, the concrete group of target applications for the QoS monitoring platform is WSN and IoT applications. However, the group of applications can be defined more conceptually by specifying and parametrizing the data emitted by them and expected after processing. For the purpose of scoping, an implementation-agnostic view will be taken regarding the intended platform. This brings the focus to intended inputs, expected outputs and their contrasts, without assumptions of the internals of the platform.

Firstly, there is the issue of *individual information capacity*. Individual messages presented to the platform contain very little individual capacity for information. Some information can be extrapolated from it, but only about the device that emitted it and at the exact moment the measurements were taken. Though, for example, detection of failure of a single node is an important task, it has little impact on the application at large if this application concerns thousands of sensors. This immediately identifies a second feature of the emitted data, in that it is extremely multi-source. The data originates from an incredible amount of distributed devices. This entails that, though the measured data points from similar devices describe similar data, the aggregation of data from these sources is not a trivial task [21]. Not only is a series of data temporally relevant, it is also related across the plain of geographically distributed sensor devices. Finally, the huge amount of devices and the dynamic nature of sensor networks and IoT induces a high variety of scale. Therefore, any back-end application — main or auxiliary — should anticipate and provide a sufficient potential for scalability. Conversely, the outcomes of the platform are considered. The platform is expected to output a relatively small amount of high-information actions, alerts and reports. The high-information consequences are contrary to the low-information capacity of individual device messages. Likewise, the moderately small number of output responses/events contradicts the immense influx of data-messages into the platform. This entails that somewhere in the application the data is transformed and condensed.

The transformation from low individual information capacity to high information messages can be achieved through three means. The first is enrichment, which uses outside sources to annotate and amend the data in a device measurement message (e.g. device location data extracted from a server-side database) [45]. The second is transformation, which takes raw low-level data points and performs calculations on them to transpose it to higher-level information (e.g. combining location data and time to calculate the speed of an object) [46]. The third method is data aggregation and reduction. This method joins and merges related data points across several — and often vast amounts of — input messages to formulate a single output message containing a few data points, depicting some collective parameters of the domain [46]. Again, the reach of this domain can be temporally, geographically, etcetera. The first two methods operate on individual data entries emitted by sensors. Hence, they can be easily parallellized and are thus incredibly scalable [47]. However, the aggregation implies an eventual reduction into a single snapshot on a single machine. This introduces possible single points of failures or congestion, and if adequate precautions are not taken scalability is lost.

To summarize, the input data is characterized by *low individual information value*, *multi-source* and *extremely high volumes*. Conversely, the output is characterized by a *finite* number of *high information value* whose data processing will require *scalable data enrichment and aggregation*. These will be the parameters of the scope of applications observed by the platform and the successive applications the platform will serve.

Stakeholder analysis

Another approach to scope the efforts is by identifying the stakeholders of the platform. This will be performed by analogy of the Onion Stakeholder Model [48]. This model divides stakeholders in consecutive layers, ordered by the degree of interaction and benefits received from the product. For this stakeholder division both the platform to be developed and potential future implementations of it will be considered as the **Product**. Intuitively, this project definition would result in a two level product in the model, with the platform as core and the group of all instantiations as the first layer around it. However, since this analysis focusses on human stakeholders, it will be treated as a single instance in the application of the model. A visual representation of the application of the onion model is given in Figure 2.1.

The first layer of the model directly encasing the product is **Our System**. It encompasses the designed and developed product (i.e. the platform and its instances) and the human parties that directly interface with the product. The first group of these stakeholders is the *Employee Developing and Maintaining* implementations of the platform. They interact directly with scaffolding and frameworks provided by the core platform. Some explanations of the onion model place developers in the outer layer of the model (the wider environment), since after development they no longer interface with the product unless they remain involved in a maintenance capacity. However, developers of a platform instantiation interact with the framework directly provided by the core platform. Therefore, their importance will be emphasized by placing them in the system layer of the model. The second role in the system layer is the *Normal Operator*. These operators receive information from the product directly and interact with subsequent systems and operational support employees to effect change. More specifically, this entails changes to the application under investigation or reports regarding the long-term performance of the application intended for managers and employees higher up in the organization.

The second layer of the model is the **Containing System**. It contains stakeholders that are heavily invested in the performance and benefits of the product, but do not interact with it directly on a regular basis. Two such stakeholder roles were identified. The first is the *Support and Maintenance Operator* of the application observed by the platform. A stakeholder analysis of the application under investigation would place these operators in the first layer of the model. However, since they do not (necessarily) directly interface with the support platform, they are placed in the second layer of the model for this analysis. They are however heavily invested in the performance and results of the platform, since identified problems and deficiencies can direct their efforts toward maintaining and improving their own application. The second role in this layer is the *Sales Person* of the application under investigation. Again, this regards a sales person of the application under investigation, not of the support platform. The task of a sales person is to convince potential clients to employ a developed product. Performance guarantees are an important part of a sales pitch held by this stakeholder. Therefore, employees of sales departments benefit hugely from known, concrete and stable QoS metrics.

The third layer of the model is the **Wider Environment**. This final layer contains stakeholders that do not sentiently interface with the product and are not heavily or conscientiously interested in its execution or performance, but are affected by it to some degree. The first stakeholder role in this category is the *Financial Benefactor*. This entity is not heavily invested in the development and daily routine of the system, but does benefit financially from it. This role applies to investors, companies and other business units that are not concerned with the technical upkeep of the product, but do benefit from the gained revenue or cost-efficient measures provided by the product. Closely related with this is the *Political Benefactor*. This benefactor does not directly reap monetary benefit from the solution, but does gain political benefit from it. This can apply to both stakeholders in public office or private business by improving their position in their respective markets. The final stakeholder is the *General Public*. Members of the public do not interface with the platform in any capacity, but can benefit heavily from it. For example, many WSN and IoT applications are deployed in smart city management [49] and industry4.0 [50]. Though deployment of dependable IoT technologies in these fields require initial investments, in the long term these technologies can improve efficiency, reducing costs and prizes. Therefore, guaranteed uptime and low resource usage can benefit the consumer, without them realizing it. Though the benefit to singular consumers is relatively small, due to the huge size of the public at large this amounts to an incredible benefit.

2.2 Commonality/variability analysis

In order to design for the problem domain it will require conceptualization. The problem domain(s) will be conceptualized by means of a commonality/variability analysis (C/V analysis). Whereas this analysis is usually performed during

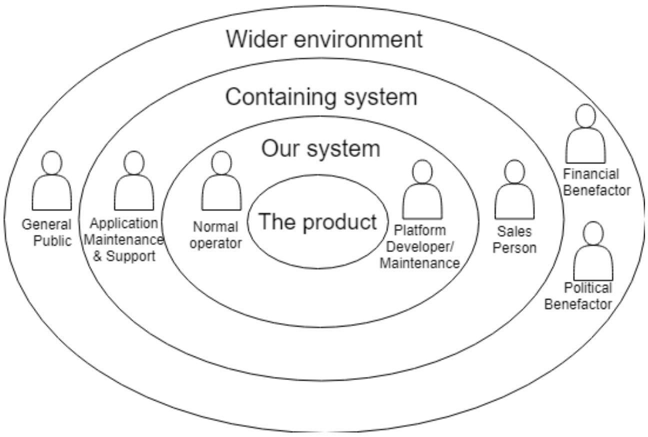


Figure 2.1: Visual depiction of application of onion stakeholder model

the process of system decomposition in product line engineering, it can also be employed to identify common and varying concepts in a problem domain. This analysis identifies the commonalities (invariants) that may be assumed fixed and may be depended upon and the variations in the problem domain which will need to be accounted for by the solution.

J. Coplien et al [51] describes the process of a commonality/variability analysis in five steps.

1. Establish the scope: the collection of objects under consideration.
2. Identify the commonalities and variabilities.
3. Bound the variabilities by placing specific on each variability.
4. Exploit the commonalities.
5. Accommodate the variabilities.

The performed conceptualization of the problem domain will mostly focus on step 2 in which a list of common definitions, shared commonalities and variabilities will be provided. Also, steps 4 and 5 will be combined by formulating a list of requirements for intended solution, based on the identified commonalities and accounting for the found variabilities.

2.3 Distributed computation technologies

This section will discuss some distributed technologies and concepts that will be evaluated and used during the design of the development platform (Chapter 3).

2.3.1 Monolith vs. micro-component

The first decision to make is the high-level architecture to adopt. The first option for which is to implement the platform as a monolithic software system. The benefit of such a system is that it keeps the solution as simple as can be. This is reflected by a famous proverb of Edsger Dijkstra: "Simplicity is a prerequisite for reliability" [52]. This simplicity entails a better understanding of the product by any future contributor or user, without the need to consult complex, detailed documentation. However, monolithic software products have been known to be difficult to maintain. The reason for this is that code evolution becomes more difficult as development progresses and changes and additions are made to the code base. Additionally, monolithic software systems are notoriously difficult to scale and balance [53].

Converse to the monolith is the micro-component architecture. It consists of a multitude of smaller components that are functionally distinct. These components communicate to one another through a underlying message distribution system. By functionally encapsulating the application into distinct modules, an inherent separation of concerns is achieved. This in turn reduces entanglement and improves the application's capacity for evolution. Micro-components are more flexible than monoliths, allow for better functional composition, are easier to maintain and are much more scalable [53]. Additionally, distributed cloud computing solves some of the tenacious obstacles in IoT's, such as the constraint computational and storage capacity [54].

2.3.2 Apache Storm

Apache Storm is a micro-component streaming library especially designed for scalability and separation of concerns. It achieves distributed computation by partitioning the stages of computation. It separates stages of computation in distinct processors performing a portion of the global process. These processors are composed into a topology. This topology specifies which processors communicate to which other processors using Storm's inherent message broker. By breaking up the computation, different stages can be distributed among machines and duplicated if required. Processors are specified and executed completely separately and communicate to one another with messages. This messaging is provided by an internalized messaging system and handles are provided by the platform in order to emit and receive messages.

The Storm platform consists of three chief concepts.

Spouts
: Nodes that introduce data in the system,

Bolts
: Nodes that perform some computation or transformation on data, and

Topology
: An application-level specification of how nodes are connected and messages distributed.

A topology can be configured such that a spout/bolt can emit messages to any other bolt. However, some remarks must be made. Firstly, though spouts/bolts can be connected to multiple bolts, each connection must be specified as an explicit one-to-one mapping. This is converse to many other distributed messaging architectures, in which components subscribe or produce to an addressed channel (topic) that acts as a shared message buffer. Secondly, though the topology is distributed among a cluster, the application is initiated as a single program on the master node. Consequently, the entire application topology must be specified before run-time and the topology cannot be altered or attributed during execution. Such alteration will require a redeployment of the topology and reexecution of the application.

2.3.3 Message brokers

By employing a micro-component architecture (without an inherent messaging system), a communication technology for components to communicate to each other is required. This approach employs a service to which producers write messages to a certain topic. Consumers can subscribe to a topic and subsequently read from it. This obscures host discovery, since a producer need not know its consumers or vice versa. The routing is instead performed by the message service. The following will explore the two widely used message broker services in the industry.

RabbitMQ

RabbitMQ [55] is a distributed open-source message broker implementation based on the Advance Message Queue Protocol. It performs topic routing by sending a message to an exchange server. This exchange reroutes the message to a server that contains the queue for that topic. A consumer subscribed to that topic can then retrieve it by popping it from the queue. Finally, an ACK is returned to the producer indicating that the message was consumed. The decoupling of exchange routers and message queues allows for custom routing protocols, making it a versatile solution. RabbitMQ operates on the *competing consumers* principle, which entails that only the first consumer to pop the message from the queue will be able to consume it. This results in an *exactly once* guarantee for message consumption. This makes it ideal for load-balanced micro-component applications, because it guarantees that a deployment of identical services will only process the message once. It does however make multicasting a message to multiple consumers difficult.

Apache Kafka

Conversely, Apache Kafka [56] distributes the queues itself. Each host in the cluster hosts any number of partitions of a topic. Producers then write to a particular partition of the topic, while consumers will receive the messages from all partitions of a topic. Because a topic is not required to reside on a single host, it allows load balancing of individual topics. This does however cause some QoS guarantees to be dropped. For instance, message order retention can no longer be guaranteed for the entire topic, but only for individual partitions. Kafka, in contrast to RabbitMQ's competing consumers, operates on the *co-operating consumers* principle. It performs this by, instead of popping the head of the queue, a pointer is retained for each individual consumer. This allows multiple consumers to read the same message from a queue, even at different rates. The topic partition retains a message for some time or maximum number of messages in the topic, allowing consumers to read a message more then once. Ensuring that load-balanced processes only process a message once is also imposed on the consumer by introducing the notion of consumer groups. These groups share a common topic pointer, which ensures that the group collectively only

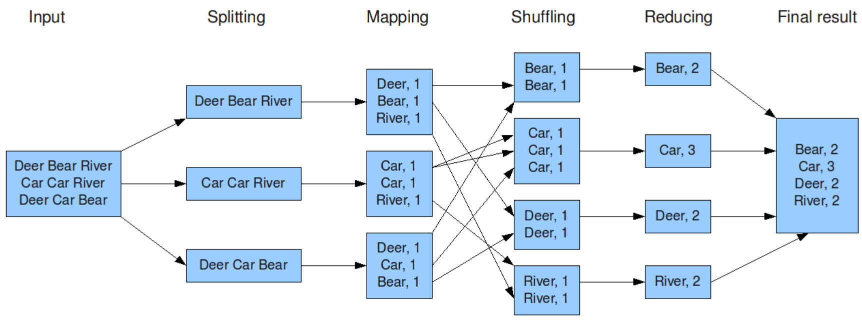


Figure 2.2: The overall MapReduce word count process [58]

consumes a message once. This process does not require an exchange service, so Kafka does not employ one. This removes some customization of the platform, but does reduce some latency. Lastly, Kafka does not feature application level acknowledgement, meaning that the producer cannot perceive whether its messages are consumed.

2.3.4 Distributed processing

MapReduce

MapReduce [57] is a distributed computing framework. It operates by calling a *mapper* function on each element in the dataset, outputting a set of key-value tuples for each entry. All tuples are then reordered and grouped as sets of tuples with a common key. The key-value sets are then distributed across machines and a *reduce* function is called to reduce the many individual values into some accumulated data points. The benefit of this framework is that the user need only implement the *map* and *reduce* functions. All other procedures, including tuple distribution and calling the mapper and reducer, are handled by the framework. An example of the algorithm on the WordCount problem is illustrated in Figure 2.2.

Though the ease of implementation is very high and the technology is very useful, the algorithm has proved to be comparatively slow. The reason for this is that before and after both the map and reduce phase the data has to be written to a distributed file system. Therefore, though highly scalable, the approach suffers from slow disk writes [59]. Finally, MapReduce works on large finite datasets. Therefore, the data streams must be processed into batches in order for MapReduce to be applicable.

Apache Spark (Streaming)

Apache spark [60] is an implementation of the Resilient Distributed Dataset (RDD) paradigm. It employs a master node which partitions large datasets and distributes it among its slave nodes, along with instructions to be performed on individual data entries. Operations resemble the functions and methods of the Java Stream package [61]

Three sort of operations exist: narrow transformations, wide transformations and actions. *Narrow transformations* are parallel operations that effect individual entries in the dataset and result in a new RDD, with the original RDD and target RDD partitioned equally. Examples of such functions are *map* and *filter*. Because these transformations are applied in parallel and partitioning remains identical, many of these transformations can be performed sequentially without data redistribution or recalling the data to the master. *Wide transformations* similarly are applied on individual dataset entries, but the target RDD may not be partitioned equal to the original RDD. An example of such a transformation is *groupByKey*. Since elements with he same key must reside in the same partition, the RDD might require reshuffling in order for computation to complete. Finally, Actions, such as *collect* and *count* require the data to be recalled to the master and final calculation is performed locally, resulting in a concrete return value of the process. RDD's provide efficient distributed processing of large datasets, that is easy to write and read. However, careful consideration must be given to the operations and execution chain in order to avoid superfluous dataset redistribution [62].

Additionally, the framework does not require disk writes as MapReduce does. Instead, it runs distributed calculations in-memory, thereby vastly improving the overall calculation speed. This does however raises a reliability issue, because if a slave node fails, its state cannot be recovered. Such occurrences are resolved by the master by replicating the part of the dataset from the intermediate result it retained and distributing it among the remaining slave nodes. Because the sequence of transformations is deterministically applied to each individual entry in the dataset any new slave node can continue calculations from the last point the state was persisted [63].

Finally however, Apache Spark suffers the same deficit as MapReduce and is performed on finite datasets. Therefore, streams need to be divided in batches in order to perform calculations. Fortunately, such a library exists: Apache Spark Streaming [64]. It batches input from streams on regular intervals and supplies it to a Spark RDD environment. The time windows can be as small as a millisecond. Therefore, it is not formally real-time, but can achieve near real-time stream processing [65].

2.4 Quality of Information of WSN data

In WSNs and IoT applications there is the concept of Quality of Information (QoI). QoI describes parameters depicting quality attributes of information presented by and derived from a system. It is especially applicable to WSNs as they present raw low-level data which is then highly processed by subsequent applications. Therefore, the concept of QoI will be employed to validate and evaluate the processing architecture presented in chapter 3. V. Sachidananda et al [66] identify the following attributes describing Quality of Information.

Accuracy The degree of correctness which provides the level of detail in the deployed network. It is the value which is the close imitation of the real-world value.

Precision The degree of reproducibility of measured values which may or may not be close (accurate) to real-world value.

Completeness The characteristic of information which provides all required facts for user during the construction of information.

Timeliness An indicator for the time needed when the first data sample is generated in the network till the information reaches the target application for decision making.

Throughput The maximum information rate at which information is provided to the user after raw data collection.

Reliability The characteristic of information, in which information is free from change or no variation of information from the source to the end application.

Usability The ease of use of information that is available after raw data collection has undergone processing and can be applied to the application based on user's evolvable requirements.

Certainty The characteristic of information from the source to the sink with desired level of confidence helping the user for decision making.

Tunability The characteristic of information, where the information can be modified and undergo processing based on user's evolvable requirements.

Affordability The characteristic of information to know the cost for measuring, collecting and transporting the data/information. It is the expensiveness of information

Reusability The characteristic of information, where the information is reusable during its lifetime or as long as it is relevant.

2.5 Constraint programming and solving

Chapter 4 will employ the concept of constraint programming and constraint solvers. The concept of constraint programming encompasses modelling a problem by means of a collection of correlated variables and associated value domains. The relations between variables are captured in a list of constraints. The problem is then solved by finding assignments for each variable with respect to their domains that conform to the specified constraints.

An example of a problem modelled as constraint problem is a Sudoku. The model will be a list or matrix of integer variables, with each entry having a domain $\{V_i | 1 \leq V_i \leq 9\}$. The associated constraint would then be $V_1 \neq V_2$ for every combination of entries (V_1, V_2) in the same row, column or 3-by-3 grid.

Several methods exist in order to solve a combinatorial constraint problem. The first and simplest is to perform a brute force search over the solution space. This would produce the Cartesian product of the domains of all variables ($\prod_{i \in I} D_i$) and test them against the constraints. Candidate solutions are rejected until a valid composition of variable assignments is found. This is however a very inefficient procedure as it has to search though the entire search space without optimization. For large combinatorial problems this search space grows exponentially. For instance, for the sudoku example with 20 values filled in, the solution space has a size of $9^{61} (\approx 1,6 \cdot 10^{58})$.

A more efficient search algorithm is presented by backtrack-search. Whereas the brute force approach assigns every variable a value and then checks its

Backtracking
Input: A constraint network R and an ordering of the variables $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

0. (Initialize.) $cur \leftarrow 0$.
1. (Step forward.) If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, $cur \leftarrow cur + 1$. Set $D'_{cur} \leftarrow D_{cur}$.
2. (Choose a value.) Select a value $a \in D'_{cur}$ that is consistent with all previously instantiated variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$ (x_{cur} is a dead-end), go to Step 3.
   (b) Select a from D'_{cur} and remove it from D'_{cur}.
   (c) For each constraint defined on x_1 through x_{cur} test whether it is violated by $\vec{a}_{cur-1}$ and $x_{cur} = a$. If it is, go to Step 2a.
   (d) Instantiate $x_{cur} \leftarrow a$ and go to Step 1.
3. (Backtrack step.) If x_{cur} is the first variable, exit with "inconsistent". Otherwise,set $cur \leftarrow cur - 1$. Go to Step 2

Listing 2.1: Algorithm for backtrack-search[67]

validity, the backtrack-search algorithm operates on a subset of the variables assigned. By incrementally assigning values to variables it performs a systematic Depth First Search through the search space. If a partial assignment is determined to violate the set of constraints, the algorithm will reject the entire remainder of that branch of the search tree. In this manner the algorithm optimizes failing variable assignments by attempting to identify them earlier. For the example of the sudoku solver this entails that an assignment of a 3 to a position adjacent to another square with a 3 will immediately halt the exploration of that branch of the search tree, without the need to consider subsequent variable assignments. It will instead backtrack through the tree by rolling back assignments and attempt a different assignment. The full algorithm for backtrack-search is given in listing 2.1.

The backtrack-search algorithm can be improved upon further by implementing constraint propagation. This technique attempts to prune invalid variable values from the domain before they are assigned by the backtrack-search algorithm. For example if a square in the sudoku is assigned a three, then the effect of this assignment will be propagated by pruning the number 3 from the domains of every entry in the same row, column or 3-by-3 grid. This eliminates inconsistent options that would violate the constraints before they would be assigned. Additionally, the concept of local inconsistency can be extended to variable domains without requiring any assignment. For example, given two variables V_1 and V_2 with domains $D_1 = \{1, 2, 3\}$ and $D_2 = \{2, 3, 4\}$ and the constraint $V_1 \geq V_2$, then the values 1 and 4 can be pruned from D_1 and D_2

respectively. For they are inconsistent with any of the values in the opposing domain and can therefore never validate the constraint [67, 68].

3. Design of WSN monitoring platform architecture

This chapter will detail the process taken in order to device the platform and its architecture. This will be accomplished by first exploring the general problem domain. Subsequently, the design of the proposed platform and its implementation will be deliberated by identifying the available supporting technologies, clarifying the adaptations made to those technologies and explaining further implementation details. The chapter will be concluded by discussing the advantages, limitations and considerations of the proposed solution.

3.1 Objective of this chapter

Large sensor applications send immense amounts of low-level raw monitoring data that requires capturing, enriching and processing. Individual snapshots of raw data will contain very little information. However, when accumulated, these snapshots contain the potential from which meaningful conclusions can be derived. These decisions range from single sensor scale to the sensor application as a whole. The raw data is enriched by combining and analysing datasets of similar, related data, in order to achieve a higher degree of information.

The objective of the efforts described in this chapter is to conceive a software platform that enables software developers to construct their own sensor application monitoring system. The intention to achieve this is by devising a generic application backbone and base building blocks for developers to compose and extend.

3.2 Conceptualization of the problem domain

In this section the problem domain will be investigated in order to eventually determine the requirements for the model. This will achieved by performing a commonality/variability analysis (C/V analysis) of the problem domain, as described in Section 2.2. The analysis consists of three concepts:

- The definitions that will be used in the analysis and the remainder of this chapter,
- the common features shared by all elements in the problem domain and which may be assumed as established concepts, and

- the variations that appear between aspects of the problem domain for which must be accounted for in the proposed solution.

Definitions

Firstly, some key terms will be defined that will be used in the analysis and the remainder of this chapter.

Platform
: The monitoring platform to be designed.

Application
: The application that is being investigated by the platform.

Snapshot
: A message containing a collection of data points indicating the state of a system on a certain instant.

Source
: An entity emitting a snapshot. This can be a physical end-device, external service or an process internal to the platform.

Consequence
: An action effected by the platform based on the analysis of one or more snapshots.

Commonalities

With the definitions established some common features shared by each application in the problem domain will be identified next. These commonalities may be presumed during the design of the platform and grants a scope to the design efforts.

C1.1 The group of target applications involves an enormous amount of sensors, which entails a high throughput of snapshots requiring analysis by the platform.

C1.2 As mentioned in the definitions, data is captured in snapshots. These represent the (partial) state of the application as measured or determined at a certain point in time. These snapshots can be used for both input of the platform as for representing intermediary computation states.

C1.3 The parameters and values of a snapshot, and therefore consecutive derived values, may be considered fixed. Parameters can only change by outputting a new snapshot, not during evaluation of the current one.

Variabilities

Finally, the variety within the problem domain will be explored. As the purpose of the solution is to process information, the analysis will mostly focus on the variations in the domain of data and information produced by applications. The solution should provide proficient adaptability in order to account for these variabilities. This will be ensured by captivating these variations in requirements.

V1.1 The first variety encountered is the variation in Quality of Information (QoI). As described in the Background chapter (section 2.4), there are many parameters characterizing the QoI of data. A snapshot or collection of snapshots can vary on any combination of them.

V1.2 Secondly, there is the information base on which conclusions are made. The first conclusion basis is elementary:

(a) Single snapshot. (e.g. a sensor requiring maintenance)

The second identified analysis is based on a large amount of low-information snapshots [69], of which two types are identified:

(b) Multiple sequentially relevant snapshots from a single source (longitudinal), used to analyse tendency of parameters. (e.g. a sharp continuous increase in bandwidth used may indicate future capacity issues.)

(c) Many multi-source snapshots without individual significance (lateral). E.g: while the individual bandwidth usage of sensors may be of little interest, knowledge of the average and total bandwidth usage of the system may be warranted.

V1.3 The possible consequences by the platform have a large range of implementations and cannot be fully anticipated. However, though the exact implementation of consequences can never be anticipated exactly, some groups of consequences can be identified.

(a) Build a model for reporting purposes. In order to generate reports some high-level information data points need to be calculated based on large datasets. These data points are then exposed either by an in-memory component with an API or by persisting it to intermediary permanent storage.

(b) Analyses which invoke immediate responses to the application or a command & control service administrating the application.

(c) Alerting or reporting according to a specified rule. When this user defined rule is met or violated an alert is sent to a maintenance operator or auxiliary system.

The final variety is the scale of the application. It has already been established that the platform will operate on applications of very large-scale, i.e. thousands of sensors. However, given a thousand as lower bound, the upper bound is still uncertain. Therefore, the size of the application is still uncertain and differing degrees of size require different computational needs.

V1.4 The scale of large wireless sensor applications varies wildly. This yields for both the number of devices in the application and the rate at which a device emits snapshots.

3.3 Requirements for the proposed software platform

In this section the requirements of the proposed platform will be described, in accordance with the variability identified in the previous section.

R1.1 The platform should enable the capture and transformation of snapshots.

R1.2 The platform should enable processing of a single snapshot.

R1.3 The platform should enable processing of a window of homogeneous snapshots.

R1.4 The platform should enable processing and aggregation of an enormous amount of snapshots.

R1.5 The platform should enable implementation of a wide range of consequences. It should at least provide for these anticipated types of consequence:

- report building,
- application feedback, and
- alerts of behavioural violations

R1.6 The platform should be scalable in order to support any large amount of inputs

Justification

This section will be concluded by justifying the identified requirements according to the earlier performed C/V analysis. The formal traceability between the requirements, commonalities and variability is listed in table 3.1

Requirement	Variability	Commonality
R1.1	V1.1	C1.2, C1.3
R1.2	V1.2a	
R1.3	V1.2b	
R1.4	V1.2c	
R1.5	V1.3	
R1.6	V1.4	C1.1

Table 3.1: traceability table for justification of requirements

The first requirement (R1.1) regards the definition and concepts of snapshots and is based on the commonalities and the variation in Quality of Information (Section 2.4). As illustrated by the traceability table, the following three requirements (R1.2–R1.4) closely correlate with the three varieties identified in V1.2. Requirement R1.5 attempts to captivate the variability described in V1.3. This variation is captured in a single requirement as opposed to differentiating them as for V1.2. This is because the possible consequences are not limited to the identified consequence groups. Therefore, they are grouped into one abstract requirement. Lastly, the final requirement considers the scale of the target applications. This regards both the amount of devices in the target application as the frequency they send their snapshots.

3.4 Evaluation of the solution domain

This section will explore the solutions and supporting technologies that are offered. These technologies have been described in Section 2.3. First, the base

	RabbitMQ	Kafka
Speed	+	++
Scalable	+	++
Multi-cast	×	✓
Multiple reads	×	✓
Acknowledged	✓	×
Delivery guarantee	✓	×
Consumer groups	✓	✓
Order retention	Topic-level	Partition-level
Consumer model	Competing	Cooperating

Table 3.2: Summary comparison of RabbitMQ and Kafka

architecture type will be considered of the platform, as it is the most fundamental decision to be made. Continuing, options for supporting technologies will be explored. The section will be concluded by examining some distributed computing technologies. These technologies should enable data-intensive computations by distributing them over a cluster, as to provide the required scalability.

Architecture and execution platform

Though a monolith presents the simplest software solution, it severely lacks the flexibility which enables software evolution and scalability of input capacity. Since this would invalidate requirement R1.6, a distributed micro-component architecture will be employed instead. Storm is especially suited for the purpose of this study since it was designed for interconnected micro-components. By employing Apache Storm, both the distributed computation environment as the means of data distribution are obtained, simplifying the technology stack.

However, the built-in messaging mechanism is completely internalized, complicating integration with auxiliary processes. Tasks such as data injection, platform monitoring and data extraction for debugging, processing or reporting by third-party programs and stakeholders will require an exposing mechanism. Additionally, Storm requires bolt connections to be explicitly defined at start-up. This causes two disadvantages: Firstly, a single process cannot be updated or reconfigured without restarting the entire topology. Considerations should therefore be made on when to update the system and when to delay rolling-out an updated version. Secondly, the bolts are connected pair-wise. This is in contrast to most conventional publish/subscribe communication platforms (such as Kafka and RabbitMQ). These systems decouple the producer and consumers and instead write and read to addressable communication channels (topics). Storm allows reading and listening on streams of a certain topic, but the connection still needs to be explicitly specified. This is cumbersome, but should be able to be overcome.

Message brokers

A comparative summery of both discussed message broker technologies is given in table 3.2. From this comparison the first apparent difference is the approach taken to consumer strategies. Kafka allows messages to be read multiple times, both by different consumers or the same consumer, whereas RabbitMQ allows messages to be consumed only once. Secondly, Kafka's lower-level replication provides increased scalability and speed [70]. However, it does so at the cost of some functional benefits such as order retention, guaranteed delivery.

Distributed computing

As specified by requirement R1.4, a means of processing large volumes of data is required. This is accomplished by aggregating a large number of snapshots into a distinct smaller amount of snapshots (often singular) with a higher-degree of information. In order to accomplish this a scalable means of computation is required (requirement R1.6)

Firstly, the MapReduce paradigm and platform seem very useful to the platform. In the early exploration phase it quickly became apparent that there were many use cases where one might want to extract accumulated snapshots per individual sensor or grouped by cell tower. This approach also allows to compensate for devices sending at different rates. These devices would be overrepresented in the population if they were not normalized. By first grouping and averaging the messages per device, it can assured that every device has the same weight in the analysis.

```
1 // assumes initial RDD with lines of words = lines
2 JavaRDD<String[]> wrdArr =              lines.map(l->l.split(" "));
3 JavaRDD<String> words =                 wrdArr.flatMap(arr -> Arrays.toList(arr));
4 JavaRDD<String, Integer> pairs =        words.mapToPair(x->(x,1));
5 JavaRDD<String, Integer> counts =       pairs.reduceByKey((a,b) -> a+b);
6 Map<String, Integer> result =           counts.collectAsMap();
```

Listing 3.1: MapReduce example of Figure 2.2 in Spark RDD.

It is interesting to note that the MapReduce framework can easily be reproduced in Spark. This is achieved by calling the *mapToPair* and *reduceByKey* routines subsequently. To illustrate this the MapReduce procedure of Figure 2.2 is implemented using Apache Spark in Listing 3.1. Please note that the intermediate assignments of the RDD are not required. RDD operations can be chained after one another, but intermediate assignments have been used to better illustrate the steps taken. Also note that the first three steps are be performed fully parallized since they are all narrow transformations. Only line 5 (wide transformation) and 6 (action) require RDD redistribution.

Finally, the preceding has shown the two platforms to be functionally similar. However studies have shown Apache Spark to perform better on non-functional metrics, such as execution speed and scalability [59]. Additionally, by employing Apache Spark Streaming, a means of batching input streams is innately provided.

3.4.1 Solution decisions

Apache Storm was chosen for a distributed component environment and messaging system. The reason for this was primarily that Storm was conceived with this type of real-time streaming micro-component application in mind. The spouts and bolts provide the perfect building blocks to design an iterative information refinement application with separation of concerns in mind, while the built-in streaming mechanism provides for the distribution needs. However, the lack of exposure for third-party integration and the tedious process of specifying each and every component connection will have to be accounted for.

Though Storm contains the means for large-scale snapshot aggregation, it will not be employed for it. Instead, the data aggregation will be supported by Apache Spark Streaming. The reason for this is that studies have shown Apache Spark to be upto 5 times faster than both MapReduce [59] and Storm [71]. Spark does however have a larger latency, due to collecting batches of data instead of processing them real-time. This however should not cause a significant problem since the envisioned use case is for timed analysis jobs on very large amounts of input data, in order to detect collective tendencies of the system under investigation. For this scope of application the latency issues of Apache Spark do not impose a large deficiency.

Apache Kafka will be employed to facilitate external communication of the platform. The reason for this is its speed and greater scalability. Additionally, but to a a smaller degree, this was chosen because of Kafka's ability to multicast messages. This will allow multiple auxiliary processes to eavesdrop on the proceedings of the platform. With the decision for Kafka comes another benefit, as the Spark Streaming library contains adapters for Kafka allowing direct connection to it. Therefore, data can simply be emitted to a Kafka topic and consumed by a Spark Streaming process. The greatest deficiency of Kafka, being the lack of topic-level order guarantee, is not of grave importance. The hindrance can be overcome by including timestamps or sequence numbers in the passed messages. Moreover, the Spark calculations most likely will not require order retention. The reason for this is that most computations will contain of a *reduce* step, which requires the reduction operation to be both associative and commutative. Therefore, the message order is disregarded.

3.5 Design of the software platform

The preceding technologies will be adopted by composing them using adapters and abstracting the solutions. The internal implementation details are shielded by abstracting the technologies, simplifying implementation by the user. Some scaffolds for bolts will be provided, intended for different types of data flows and data reductions. Additionally, these technologies are very abstract since they were intended for many unspecified usages. However, the (to be developed) platform and group of target applications feature some known commonalities, which were previously considered variations. Therefore, some functions can be implemented which were originally intentionally left unspecified. This will reduce the implementation effort required, again simplifying usage of the platform.

3.5.1 Micro-component architecture

The remainder of this section will explain what adaptations to the previously discussed technologies have been made.

Apache Storm

The bulk of the processor (micro-component) construction, execution and messaging tasks of the platform will be performed by Apache Storm. However, as mentioned before, the process of specifying a processor topology in Storm is a cumbersome process due to the necessity of interconnecting each and every process individually. Therefore, cross-connecting M producer components with N consumers requires $M \cdot N$ explicitly specified connections. This is contrasted by technologies that employ topic based channels in which M producers write to a channel to which N consumers are subscribed, requiring but $M + N$ connections to be specified. To this end, a topology builder was developed which enables topic based streaming. The builder will automatically connect the specified components according to the topics they are subscribed to. In this manner a component and its connections can be specified with but a few instructions, as demonstrated in listing 3.2. Note that the complexity of the topology does not impact the amount of code needed, as the code complexity is solely depended on the number of components and not how they are interconnected.

```
topologyBuilder.declareBolt(new UserDefinedProcessor("pname"))
    .subscribeAsConsumer("sensor_input_channel")
    .declareAsProducer("debug_channel", "output_channel");
```

Listing 3.2: Declaration of a processor and communication channels

Since Storm allows processes to be duplicated for load-balancing purposes, it employs some methods of controlling which duplicated process worker will consume which snapshot. The two chief methods are supported by the platform. The first method is the *shuffle grouping*. It is the simplest channel specification and does not offer any guarantees on which process worker will consume the snapshot. It is therefore described as receiver-agnostic. However, this lack of guarantee will not effect most tasks since most will be stateless data processors. The second supported stream manipulation method is the *field grouping*. It is used for processors that do retain a state or somehow require similar snapshots to always be processed by the exact same worker. A simple example of this is a processor that counts the number of snapshots received for each sensor in a WSN. If it cannot be guaranteed that all snapshots of a sensor S are always processed by the same worker W, one worker might count 40 snapshots and another would count 60 of them. This requires another singular processor that accumulates those counts in order to derive an accurate snapshot count. Therefore, it is possible to specify a set of fields which will consistently determine which worker will consume a snapshot. In the developed platform this is specified at topic level. Again, to prevent repeated declarations. Therefore, each snapshot emitted to such a channel is required to include all fields specified in the field grouping of that channel.

Finally, though the abstractions and encapsulations of the Storm platform are believed to simplify implementation efforts, it could still be useful to an implementer to inject their own native Storm bolts or spouts. This might be due to reusing earlier defined bolts or requiring more control of a process than the abstraction offers. To this end, the developed topology builder encapsulates the topology builder provided by the Storm Java library. As a consequence, the topology builder provided by the platform, upon calling the *build()* function, will

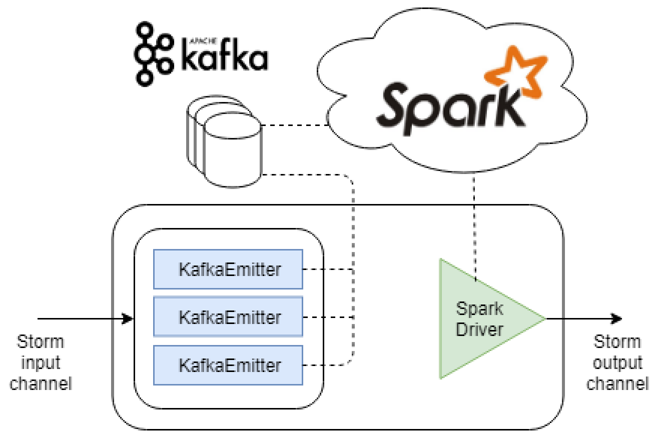


Figure 3.1: Graphical depiction of the distributed accumulator process

return an instance of *org.apache.storm.topology.TopologyBuilder*. This allows last-minute injection of self-specified native storm processes, before ultimately generating the Storm topology with that builder.

Incorporation of Apache Spark Streaming

As identified in by requirement R1.4 there is a need to condense the information of enormous amounts of (individually) low-information snapshots into a diminished number of high-information snapshots. Additionally, the large amount of input snapshots and the assertion that the platform should be scalable (requirement R1.6) entails that a scalable data accumulator should be made available.

As specified in section 3.4.1 Apache Spark Streaming was chosen for this task. However, this causes an earlier identified problem: a direct incorporation of Apache Spark in Apache Storm is difficult. In order to solve this inoperability of interfaces it was decided to device a process that functions as an adapter between Storm and Spark. This adapter employs Apache Kafka, for which Spark does provide interfaces, to pipe snapshots obtained from Storm channels. Snapshots are then read from a Kafka channel and batches of snapshots are fed to Spark RDD computations. Once the cloud computations have concluded the data is returned to the Storm environment and aggregated snapshots are eventually forwarded to consecutive processes. This is achieved by deploying two Storm components. Firstly, a specialized Storm bolt named *KafkaEmitter* is deployed. This process simply consumes Storm messages and forwards them to a Kafka channel. Secondly, a Storm spout is deployed which acts as a Spark driver program. This bolt contains the instructions for the distributed computation of the Spark cloud and results of the cloud computations will be returned to it. A graphical representation of this process is depicted in Figure 3.1.

Two interesting remarks should be made, as apparent from Figure 3.1. Firstly, The KafkaEmmitter can be replicated in order to prevent it being a point of congestion in the topology. Secondly, the fact that two distinct components (KafkaEmitter and SparkDriver) are present is encapsulated by the topology builder. Developers need only declare an implementation of the distributed accumulator processor (acting as Spark driver node) with the appropriate Storm and Kafka channels. The builder will then deploy a KafkaEmitter (or several) and the specified accumulator. This simplifies deploying the processor and obscures the internals by appearing as a single component.

3.5.2 Scaffolds for micro-components

With the high-level architecture and technologies established, the component scaffolds that are provided to application developers by the platform will be described. First, the base functions shared by all components will be described, before discussing them more in depth individually.

Common functionality

Firstly, the components contain all functionality and information required to emit snapshots to subsequent components. A developer need only package the information in a snapshot consisting of key-value pairs and specify to which stream a snapshot must be emitted. The component then uses the information it received during the building of the topology to route the snapshot to all receivers subscribed to receive it. This not only implies routing the snapshot towards the correct component but also the correct component worker according to the defined field grouping.

Secondly, all components contain a base implementation of the *prepare()* method. This method can be implemented to instantiate some properties that cannot be instantiated in the object's constructor. The reason that some properties cannot be instantiated in the constructor is that Storm processes (spouts and bolts) adhere to a prespecified execution order. The component is:

1. created by one of its constructors,
2. transmitted to one of the worker nodes of the Storm cluster,
3. further instantiated using the *prepare()* method, and
4. executed according to its specification.

The reason for this course of action is that step 1 is performed on the Storm master node, before distributing the functional object over the cluster. Therefore, during step 2 the object and its members need to be serializable. Non-serializable members are consequently instantiated during step 3, after the object has been transferred and before functional execution. The *prepare()* method thus can be used to instantiate certain non-serializable properties.

Spout

This process mirrors the Apache Storm spout and is the component that introduces snapshots to the network. This component typically contains a handle to some external data source such as a database, API or auxiliary streaming technology. The reason for such a specific processor for this is the special execution cycle it has compared to a Storm bolt. Bolts execute with interrupts. They halt their execution until a new snapshot is available. However, a spout runs on an infinite-loop (until termination), continuously executing the method *nextTuple()*. This method polls, retrieves and emits snapshot depending on the origin of the source.

SingleMessageProcessor

This component is the most basic scaffold and closely resembles a Storm bolt. It however contains some additional functionality that improve its usability. It receives a snapshot and performs computations or analyses on it, before emitting new, enriched snapshots. Its typical use is for transformations of individual snapshots. As noted before, this component requires implementation of a singular method: *runForMessage(Message m)* which will be called for each snapshot received by the component.

HistoricBufferedProcessor

The HistoricBufferedProcessor resembles the SingleMessageProcessor in that it consumes single snapshots, but instead it processes or analyses an ordered series of relevant snapshots, called a *window*. This is performed by retaining an in-memory buffer to which new snapshots are amended and is periodically filtered on relevance. This component can for example be used to determine recent trends in system parameters. The methods that require implementation for this component are *runForBuffer(List<Message> l)*, which is run every time the buffer is updated, and *cleanBuffer(List<Message> l)* which implements how and which elements should be pruned from the buffer, should they lose their relevance.

DatabaseBufferedProcessor

From a processing perspective the DatabaseBufferedProcessor is similar to the regular HistoricBufferedProcessor. It analyses a buffer of snapshots in order to emit a snapshot containing accumulated or averaged knowledge based on its input snapshots. However, rather then keeping an in-memory buffer of snapshots it maintains a connection with an database. This allows for buffered processing of snapshots that is not performed regularly, thereby not superfluously occupying memory resources.

To keep the component applicable to many database implementations and query languages it was chosen not to instil a specific database connection. Instead, a developer is offered scaffolds to stepwise implement the intended behaviour with an actual database connection. This scaffolding contains the methods processing the buffer (*runForBuffer(List<Message> l)*) and purging the buffer (*cleanBuffer(List<Message> l)*) as included in the HistoricBufferedProcessor. Aside from those functions it specifies function end-points for storing a new snapshot into the database and for fetching the relevant buffer from the database, respectively named *persistMessage(Message m)* and *fetchBuffer(Message m)*.

DistributedAccumulatorProcessor

This component aggregates large amounts of laterally relevant snapshots. Laterally relevant entails that the snapshots describe similar data points, but have little sequential relevance. The input for this process is a large amount of (individually) low-information snapshots. Conversely, the goal of the processor is to emit some high-information snapshot. An example of its usage is combining thousands of snapshots from sensors in order to obtain some collective application-level performance parameters. To accomplish the aggregation of these enormous amounts of data the accumulator principle described in section 3.5.1 is employed. By means of the method *runForRange(JavaRDD<Message> rdd)* this component offers implementers a reference to the Spark RDD which contains all the snapshots collected during a prespecified time period. The implementer can then use this RDD reference to sequentially manipulate and aggregate the collection of snapshots. Keeping proper parallelization in mind, this distributed component can perform data enrichment tasks on enormous batches of streaming data.

AccumulatorProcessor

This component closely resembles the function of the above described DistributedAccumulatorProcessor, but is executed locally rather than on a cloud cluster. The purpose of this processor is tasks that would otherwise require the distributed accumulator, but whose limited scope be run in-memory on a single worker node. This could be a viable solution for applications that either run the accumulator task often enough or do not collect excessive amounts of snapshots. For these class of applications a locally executed accumulator task should prove sufficient and inclusion of such a components eliminates the base requirement of a Apache Spark cluster to be deployed in order for the platform to be executed, since the DistributedAccumulatorProcessor is the only component that employs it. It should however be noted that not deploying an accumulator in distributed mode could introduce a bottleneck in a Storm topology since the accumulator cannot be duplicated or load-balanced.

The processor was modelled after the MapReduce paradigm [57] to guide its implementation. An implementer need only specify a *map*, *reduce* and *collect* step. The exact methods to implement for this are:

map(Message m) : String
: Computes the key for a key-value snapshots.

reduce(String key, List<Message> l) : Message
: Reduces sets of key-value pairs grouped by key determined in the map step.

collect(Map<String,Message> m) : Map<String,Message>
: Collects the key-message pairs emitted by a reduce step. The return value of this method is a map of snapshots indexed by the Storm topic on which it should be forwarded.

Please note that the result of the reduce step is a set of snapshots. It is therefore possible to chain multiple map-reduce steps sequentially, as long as the sequence is concluded with a single collect step.

ResourceDistributionModelProcessor

The final component is the ResourceDistributionModelProcessor. This processor is a special instantiation of the SingleMessageProcessor that analyses inbound snapshots according to a prespecified Resource Distribution Model (RDM). This model will be discussed in detail in Chapter 4. In contrast to all other processors, this processor is not just a scaffold. Instead, it executes completely automatically, requiring only an instantiation of an RDM and a specification of which model variables to output on which Storm channels. The processor then automatically provisions the input variables of the model, calculates the derived values and outputs the requested values as specified.

3.6 Demonstration by example case

This section will demonstrate an example of a composition of the specified components to a hypothetical WSN application. First, the hypothetical example case used in this chapter (and the next) will be described shortly. Following, the platform will be explained according to a hypothetical application to that case.

3.6.1 The example case

Before describing the example case, some context must be given. This case may sometimes seem oversimplified and nonsensical, but it does provide an elementary example to illustrate all facets of the solutions without overcomplicating the case. This case is expressly not intended to demonstrate the validity or utility of the proposed solutions. For that purpose, an application to a more complex real-world case will be performed in Section 5

The proposed case encompasses an enormous network of low power devices sensing for meteorologically anomalous events. These sensors perform measurements on a regular interval and transmit the measurements to a cell tower to be forward to a back-end application for further processing. For the best results devices should measure and transmit as much as possible. However, since these sensors are not very powerful and employ a limited power supply they will require pacing.

The behaviour of the sensors is typified by two parameters: the sensing interval and transmission interval. Intuitively, it can be stated that shortening either or both of the intervals will result in more fine grained reporting, but will increase the power consumption of the device. Additionally, over time several types of sensors have been deployed with different power sources. Therefore, a sensor's power consumption over a given time needs to be restrained in accordance with the specification of its power source and expected life-time. Additionally, sensors in areas of high interest will require a shorter polling interval, as instructed by the back-end application, to gain the most precise information. Finally, given that the sensor performs the adequate amount of measurements and does not consume more power than it is specified to use, it should measure and report as much as permitted.

As for what requires monitoring, the most interesting metric is the measurement rate averaged over all sensors. Additionally, it is required to pro-actively monitor the trend of the total bandwidth used by the sensor application. The reason for this is that a constant rise in data rates may ultimately violate the data rate limits agreed upon with network service providers.

To summarize, a sensor must:

- not consume more power then it is allowed according to its battery specification,

- measure at least as much as is specified according to the area of interest it is in, and
- generally try to measure and report as much as is allowed by the previous two requirements.

Additionally, the following information must be reported by the application:

- The average polling rate, and
- whether the data rate of the sensor application rises consistently for a certain amount of time.

In order for the server to determine the intended behaviour of the device and calculate the level of service provided by the application, the following data regarding a sensor device is provided to the monitoring application:

- the required measurement rate,
- the maximum power provided by the power source,
- the measurement rate of the sensor device, and
- the bandwidth used by the sensor

Each of these data points stipulates the behaviour of a single sensor at a certain instant. Notice that some data points are normally inferred from raw basic data by auxiliary processes (e.g. required measurement rate). For simplification of the demonstrations these processes are omitted and these parameters are presumed known as a message is introduced into the monitoring application.

3.6.2 Application of the platform

A graphical representation of the topology for the example implementation is depicted in figure 3.2. As the figure makes apparent, the application encompasses a large number of sensor devices. These devices regularly send their status information to the monitoring application via some external communication technology (e.g. Apache Kafka). These snapshots are introduced into the topology by *SensorSpouts*. These spouts have been duplicated in order to accommodate the large amount of sensors which might send a sudden burst of data. The snapshots are then forwarded to the *SensorProcessors* which have been provisioned with a Resource Distribution Model. This model consumes the measured parameters of the input snapshot and uses them to further calculate all the parameters which can be derived from the inputs, according to the specified model. This model also determines the optimal Resource Utilization Model (RUM) for this sensor device. Should no valid model composition be found this is reported to the *NoRumActuator* which forwards a log message to the *Reporter* component. The *Reporter* will delegate the message to the correct reporting/alerting mechanism, outside of the topology.

Should the current mode of operation be determined not to be optimal, the *SensorProcessor* will report to the *ChangeRumActuator*. The *ChangeRumActuator* will report requests for change to an entity outside of the topology of the application. The actuator has been implemented as a DatabaseHistoricProcessor. The reason for this is that it will recollect the last few snapshots it received for this sensor and will only actually change the mode of operation of

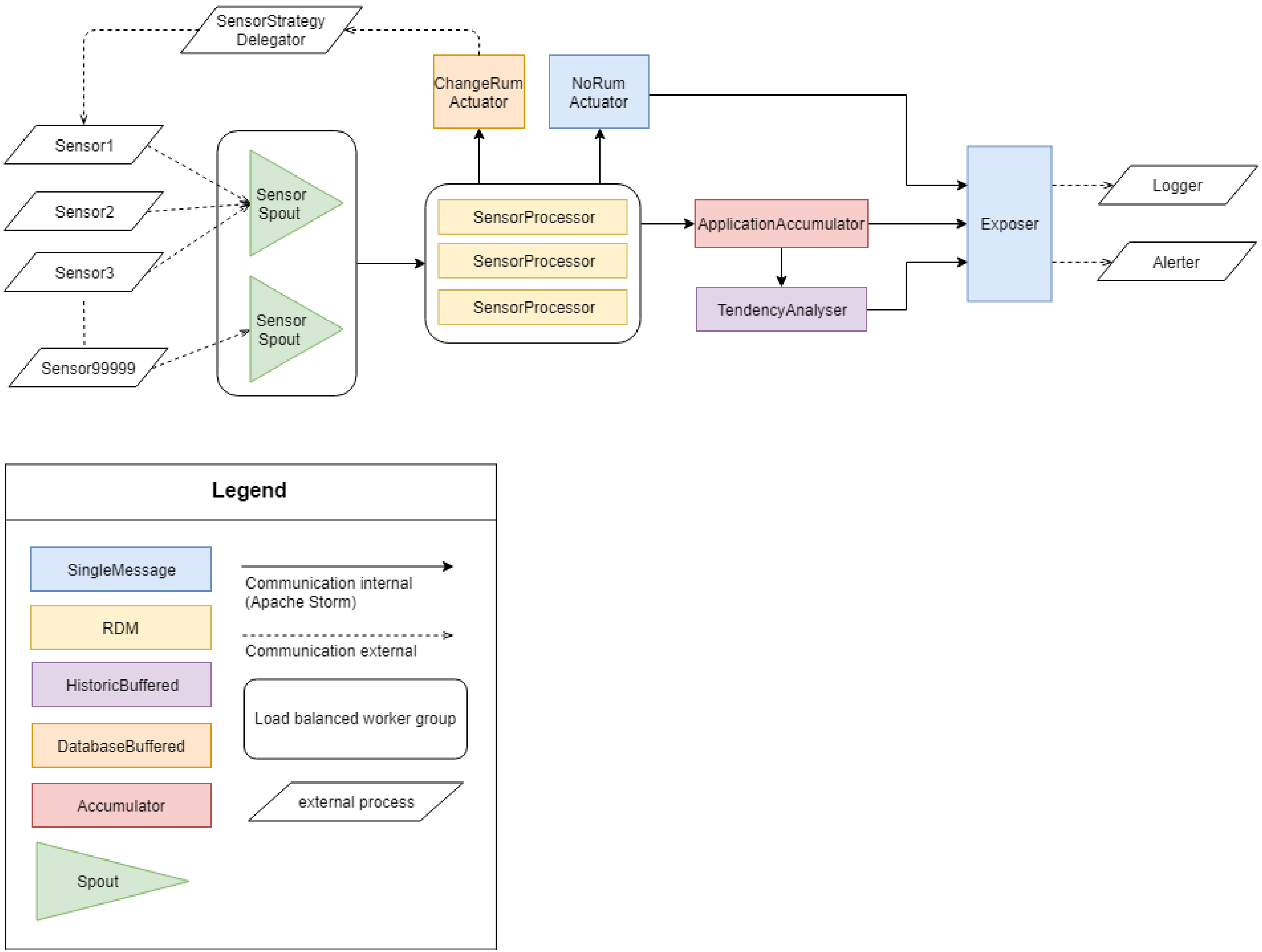


Figure 3.2: Example topology of a platform implementation according to the example case

the sensor if it is consistent with the last few snapshots received. This eliminates superfluous communication with the sensor device caused by sporadic behaviour. Alternatively, this component could have been implemented as a BufferedHistoricProcessor. However, a sensor is expected to send monitoring data only a few times per day and changes of operation will occur even less. It would therefore make little sense to keep a buffer of the last snapshots sent for each and every sensor in-memory. Additionally, this would have required a field grouping in case the component were to be load-balanced in order to enforce that the request for change of a particular sensor always be sent to the correct worker.

The final transformation to be performed is to infer application-level intelligence from the low-level sensor statuses. This is performed by the ApplicationAccumulator which collects data for a certain time period and calculates some high-level data points, such as the measurement rate of the application averaged over its sensors, the total throughput and how many devices are performing on which RDM. This information is forwarded to the *Reporter* which will make it available for visualization performed outside of the topology. Additionally, the accumulator sends its aggregated snapshot to a *TendencyAnalyser* which keeps a sequence of the total bandwidth used during previous time windows. Should this total consistently rise over a period of time, an alert will be sent by the reporter, as specified by the alerting requirements listed in section 3.6.1.

3.7 Discussion of the proposed software platform

This section will evaluate the design of the monitoring platform.

Satisfaction of requirements

The first order of business is whether the proposed design satisfies the earlier stated requirements. The message-passing micro-component architecture provides the basis for snapshot transferral and transformation as stated in requirement R1.1. Furthermore, the requirements R1.2, R1.3 and R1.4 are satisfied by the inclusion of the *SingleMessageProcessor*, *BufferedProcessors* and *AccumulatorProcessors*, respectively. Finally, the last two requirements regarding the size of the applications in the problem domain and entailing scalability of the solution have been decisive for certain choices of the supporting technologies. For example, it is reflected in the employment of cloud processing technology Apache Spark. From the aforementioned arguments it is concluded that every requirement is represented and met in the design of the platform.

Completeness with respect to QoI attributes

The goal of the platform is to process and enrich data. It is therefore rational to evaluate the appropriateness and completeness of the platform by considering the information processing capabilities it offers. This section thusly evaluates the platform's completeness by demonstrating that the platform only positively impact the Quality of Information (QoI) of the input data. This entails that the

QoI is improved or retained, but never lost as data passes through a platforms topology. This will be achieved by arguing the QoI parameters which were enumerated in Section 2.4 of the Background.

The first consideration of QoI is regarding the processing of data by the platform and affects the precision, completeness and usability of information. Firstly, *precision* and *certainty* are obtained by employing the HistoricProcessors. By averaging measurements, anomalies are mitigated and the reported value closely approaches the norm of the measurements. Provided that the accuracy of the measurements is sufficient, this improved precision should consistently yield a measurement near the actual value. Secondly, the *Usability* of information is improved as data moves throughout the topology. To illustrate this a thought experiment is proposed, using the example topology listed and described in section 3.6 and a batch of raw data emitted during a certain time window. Before the data enters the platform it contains the potential to calculate the average throughput offered by the sensor application during that time window. However, this data point is not present explicitly. This process is performed by an implementation of the platform and the resulting information is offered for further processing or visualization. This demonstrates that the platform can facilitate usability for information by calculating and producing ready-for-use values. It should however be noted that the *completeness* of the information is greatly reduced during this process. To illustrate, from the average application throughput the throughput for individual devices can no longer be determined. For this reason, and others which will become apparent, committing the raw data to storage before processing is recommended.

The second class of QoI attributes regards the processing efforts, expressed in time and costs. As the relevance of information degrades as time progresses, timely processing is paramount. *Timely* execution is achieved by providing a scalable distributed solution. This ensures that, regardless of the intense information *throughput*, the calculations can be performed in near real-time. Notice that only near real-time is claimed, since Apache Spark collects records during a time window and performs calculations in batches. However, the time window of such a batch can be set arbitrarily small for fine-grained processing. Thereby it does not impact the timeliness significantly. However, adverse to this gained timeliness is a decreased *affordability*. In order to incorporate these distributed cloud technologies a cluster of machines and increased development resources will need allocation. When the solution does not require this degree of scalability this poses an undue burden. Therefore, locally deployable alternatives to these distributed processors are also provided. Implementations of the platform are therefore offered a trade-off between timeliness and cost.

Lastly, are the *tuneability* and *reuseability* of the information. Firstly, the data can be duplicated among different communication channels which allows differentiating calculations to be performed on the same data. Secondly, in order to facilitate evolution of end-user demands the platform has been designed with separation of concerns in mind. This allows continuous reconfiguration of the platform to be performed with reduced occurrence of concern entanglement. By redeploying the topology the same raw information can be used to facilitate updated user demands. This is also another reason to store the raw data before processing it.

Some final remarks should be made on the analysis. Firstly, the platform cannot offer any improvement or retention of information *accuracy*, as it is solely determined by the method and quality of data measurement. Secondly, it should be noted that the platform does not assure preservation of any of these claim, since an implementation of the platform can violate any guarantee made. It can only be claimed that the platform does not impede any of the parameters and offers the means for developers to develop applications that do guarantee it.

Ease of adoption

A second point of focus is the ease of adoption provided by the platform itself. It is asserted that low-level implementation details of Apache Storm and Spark are effectively obscured. This was achieved by offering some abstract components that require implementation of only a few methods. This obscuration entails a clearer programming interface to an implementer, as stated by the façade software design pattern [72]

Secondly, the provided topology builder facilitates easy and fast building of a Storm topology. It does so by providing context-aware topology and process instantiation, and topic based communication subscription and emission. As mentioned before this allows M producers and N consumers connected by a single topic to be connected with complexity $\Theta(M + N)$, instead of the complexity $\Theta(M \cdot N)$ which would be required without the concept of topics. These assertions will be formally validated in Chapter 5.

Technology stack

Another issue to contemplate is the technology stack required for the platform. As mentioned in section 3.4.1, Apache Storm was chosen as chief enabling technology. The main reason for this is that it offered most of the features required and would reduce the technology stack. However, by employing Apache Spark for distributed data aggregation, two additional cloud technologies are introduced. Spark itself and Kafka which is required in order to be connected to a Storm topology. However, the inclusion of a distributed aggregation is necessary in order to keep the computations scalable. Additionally, the speed and efficiency arguments raised in section 3.4.1 justify the deployment of these additional technologies. Finally, when this scalability is not required Apache Spark and Kafka clusters can be executed locally on a single machine. This would still enjoy benefits from process parallelization, without requiring cluster deployment. Finally, Spark and Kafka may be omitted entirely, if permitted by the snapshot influx, as a non-distributed accumulator is also included in the platform.

4. Resource Distribution Model

4.1 Objective of the model

The aim of the Resource Distribution Model (RDM) is to captivate the distribution, conversion and restrictions of resource parameters in a system. The suggested target usage of these models is to allow automated analysis and optimization of the system under investigation. Therefore, a detailed model with explicitly defined entities and relations is required. Only then can the model be employed by automated tools and algorithms. The research questions related to this chapter are RQ6–RQ8.

This will be performed by first exploring the problem domain. With the definitions and concepts of the problem domain identified, a list of requirements for the proposed model will be composed. With these requirements in mind, the contemporary resource modelling solutions will be explored and evaluated on the applicability to the requirements. Afterwards, the adaptations to the selected technologies will be explained. Subsequently, the conceived model will be described in detail. To assist the understanding of the model, it will be exemplified by application to the described example case (Section 3.6.1). This chapter will be concluded with an evaluation of the proposed modelling technique.

4.2 Conceptualization of the problem domain

This section will investigate the problem domain in order to eventually determine the requirements for the model. Again, this will be achieved by performing a commonality/variability analysis (Section 2.2) of the problem domain, determining the definitions, common features and variations in the problem domain.

Definitions

First, some terms that will be used throughout the C/V analysis and the remainder of this chapter will be stated. Following that, the common features and variations in the problem domain will be examined.

Resource
: A measurable/calculable parameter of a system

Resource constraint
: A constraint imposed on a resource due to scarcity.

Component
: A physical or hypothetical entity that can consume or produce a resource

Resource Utilization Model (RUM)
: A model depicting how much of a resource is produced or consumed by a component. Each instance of such a model is internalized by a single component.

Resource Distribution Model (RDM)
: A model depicting how components are interconnectively connected by resources. This global model encompasses all Resources, Components, their relations and behaviour

QoS parameter
: Particular resource parameters that are indicative of the level of service provided by a system.

Commonalities

Following the definitions, the commonalities that appear throughout the problem domain will be asserted. These assumed features allow the efforts to be focussed and allows a more expressive specification of assumed concepts.

C2.1 A resource can be produced or consumed by multiple components.

C2.2 A component can produce or consume multiple resources.

C2.3 Resources are scarce, i.e. the amount produced must exceed the amount consumed.

C2.4 Resources are correlated and can be converted into one another (many-to-many).

C2.5 Resource parameters can be used to objectively compare functionality of a system.

Variabilities

With the commonalities established, the variabilities in the problem domain will be considered. These variations cannot be specified expressively in the model. Instead, they require proper abstraction, to be implemented when a instantiation of the model is achieved.

V2.1 Though all use cases agree on the above commonalities, not all resources, components, constraints and interconnection that can occur can be predicted [41].

V2.2 Resources of a system can be modelled on a micro-scale or macro-scale.

- A micro-scale (e.g. a single sensor device) comprises concrete, palpable parameters.
- A macro-scale (e.g. an entire WSN application) comprises derived, theoretical parameters.

V2.3 A system can have a variety of resources as QoS indicators [41].

V2.4 Short-term resource usage (e.g. interval of seconds) requires a different granularity than long term resource usage (e.g. interval of days).

V2.5 Some resources are directly measurable and thus known at a certain moment of measurement. However, some resources are derived and calculated using other resource values.

V2.6 Resource values can differ depending on a system's measured state.

V2.7 Resource values in a system differ depending on a specific operational. strategies.

V2.8 Given a system's state some operational strategies are better suited than others.

4.3 Requirements for the proposed model

With the common and variable features of the problem domain established, the following section will formulate a list of requirements that need to be incorporated in the solution. First, a full list of the identified requirements will be provided, before justifying them according to the C/V analysis of the previous section.

4.3.1 Requirements

R2.1 The model should represent resource distribution in a system

R2.2 Resources should be able to be transformed into other resources (many-to-many)

R2.3 The model should account for the fact that the value of a resource can originate from different sources. The identified sources are the following:

- **constant** a predefined value specified on development time (e.g. initial battery capacity),
- **measured** a value specified as observed on run time (e.g. percentage of battery capacity left),
- **calculated** derived from measured values (e.g. runtime left),
- **variable** any value or a calculation depending on specific system function (e.g. power usage).

R2.4 Each model should have one, and only one, resource that is associated with a heuristic QoS function.

R2.5 The model should contain constraints that describe the limitations of bounded resources.

R2.6 Given a resource distribution model, constant-valued resources and measurements, for each combination of values for variable resources, a value should be able to be evaluated for each calculated resource

R2.7 Given a calculable resource distribution model (R2.6), a set of resource constraints and an optimizer function; an optimal, valid appointment for each variable resource value should be able to be solved efficiently.

Variety	Requirements
V2.1	R2.1, R2.3, R2.5
V2.2	R2.1, R2.3
V2.3	R2.2, R2.4
V2.5	R2.2, R2.3
V2.6	R2.3
V2.7	R2.3, R2.6
V2.8	R2.4, R2.5, R2.7

Requirement	Commonalities
R2.1	C2.1, C2.2
R2.2	C2.4
R2.3	
R2.4	C2.4, C2.5
R2.5	C2.3
R2.6	C2.4
R2.7	C2.3, C2.5

Table 4.1: Justification of requirements by variety and commonalities

4.3.2 Justification of identified requirements

Table 4.1 demonstrates how the proposed requirements account for the determined variety, based on the observed commonalities. Most requirements can intuitively be traced to the variety it strives to restrain. An exception is requirement R2.4, which states that one resource is used to optimize the QoS. This is seemingly contradicted by V2.3, which states that multiple resources can be indicative of the level of QoS. This is however explained with use of C2.4. This commonality states that resources can be transformed into one another. It can therefore be inferred that it is possible to transform multiple QoS markers into a single optimizable, derived resource according to some heuristic QoS indicator function.

Evidently omitted from the justification table is variation V2.4. This is due to that a this variety has far-reaching consequences for the implementation of the model. Therefore, a choice has been made to focus on modelling of resource distribution during large time intervals. This choice will elaborated in section 4.4.3.

4.4 State of the art of the solution domain

This section will explore the current techniques and technologies in the field of resource modelling. First, state of the art of the field will be identified, before evaluating their applicability according to the established requirements. Finally, the choices made before adapting the technologies in the next section will be declared and defended.

4.4.1 State of the art

Work regarding modelling resource distribution has been performed in several studies. Elementary examples of such research are the studies of Ammar et al [73]. Through their efforts they laid the ground work for representing entities interconnected by shared resources. This UML-based model was one of the first examples of such a representation using formal models. Another example of early research is the study performed by Seceleanu et al [74]. This study focussed on modelling resource utilization in embedded systems using timed state machines. The transitions in these automata are attributed costs to model the consumption of resources for transitioning to a state of residing in one.

Resource consumption and performance over time can then be calculated and analysed according to the paths taken in this model.

A continuation of this work was performed by Malakuti et al [75]. They combined the methods of the previous authors by provisioning the modelled system components with their own state machines. These state machines model the resources and services that are offered and required by the components. By analysing these component models as composite state machines, model checking tools can be used to analyse and evaluate the performance of the investigated system as a whole.

4.4.2 Evaluation of the solution domain

These efforts have produced methods of representing components connected by shared resources. Especially the notation of Malakuti et al [75], which is both intuitive and descriptive. Therefore, this notation will be adopted. However, the models in these studies focus on components that are self-aware of their resource usage and performance. Instead, the interest is in off-site analysis of interconnected resources and accumulated performance of a composite system. Alternatively, the focus will therefore be more centred around the concept of resources. It is concerned how production and consumption of a resource is interconnected. Components serve as secondary elements, merely specifying how these resources are connected and converted into other resources. Therefore, a resource-centred adaptation of this framework might be more suitable.

Secondly, there is the issue of how to represent Resource Utilization Models (RUM), the model for variable behaviour of components. Previous studies have used timed automata to represent behaviour cycles [74, 75]. This allows for automated tools to calculate a runtime schedule in high degree of granularity. However, the high level of granularity comes at the cost of efficiency. When the time interval for the automata is shortened, entailing higher granularity, then solvers require additional computational resources and time to execute. This might enforce a complication on resource constraint devices or applications that require the solver algorithm to run many times for a multitude of devices. Additionally, it must be considered that a model contains multiple components specified by RUMs. A composition of such related automata explodes the search space for the composite automaton, reducing the feasibility of calculating them effectively.

An alternate approach is to model the RUM as a set of static parameters. A component then has multiple RUM's representing different modes of execution. This is achieved by averaging the behaviour for that mode of execution, which would otherwise be modelled by a single timed automaton. This comes at great cost of granularity, since the RUM's now only describe a few static, predefined periodical behaviours. However, it significantly reduces the complexity of the search space. For this approach timed automata are no longer a suitable technology, since the element of time intervals has been eliminated. Instead, the problem is a pure decision problem. The problem to be solved is to find a suitable RUM for each modelled component.

The search space of a decision problem can be explored with a simple brute force search, exploring all options and compositions. However, more effectively, combinatorial problems can often be solved with constraint solvers. The problem is easily transposed to a constraint problem with the RDM as model, resource constraints as constraints and the RUM's as variables for the components. With the many solution strategies described in 2.5 available for different types of problems, a suitable solver should be able to be found or devised.

4.4.3 Choices of employed solutions

With careful consideration the following choices for the solution implementation have been made. For modelling it was chosen to adapt the framework of Malakuti et al [75] by emphasizing on resources and introducing some new features. The components will still exist in the model, but will merely serve the function of connecting resources to one another. Another adaptation is the existence of multiple RUM's for a component, which allows injection of different methods of operation.

As for how to model the RUM, it was chosen to reduce the complexity of the system by modelling resource usage with fixed mathematical functions. Modelling changeable behaviour is subsequently achieved by providing a component with multiple RUM, detailing different operational strategies. The strongest advocate for this choice is the fact of the focus for this study: large WSN applications. In a WSN monitoring platform the task of determining optimal device function will need to be performed repeatedly for many sensor devices. Additionally, devices in most large-scale LPWA applications only communicate data a limited amount of times per day (at most a few hundred)[76, 77]. Therefore, high granularity is not of grave importance because the feedback-control cycle is not that short.

The fact that a component can have more than one mode of operation and the choice of static parameters for those functions, makes constraint solvers most suitable as means to solve the model. However, the search algorithm will be complemented to conclude not only the valid compositions but the optimal solution, given some heuristic function.

4.5 Design of the Resource Distribution Model

This section will be dedicated to detailing the resulting model. First, the general modelling concepts will be described, before focussing on specific modelling entities. The section will be concluded with an examination of how the optimal RUM configuration of the model is proposed to be deduced.

As stated, resource distribution is modelled by extending the model by Malakuti et al [75]. The chief adaptations to the model are:

1. RUM's with static resource values,
2. the existence of multiple RUM's for a single component,
3. the inclusion of a single explicitly defined optimised resource, and
4. constraints defining valid resource interconnectivity:
    (a) implicit constraints enforcing availability: $R_{offered} \geq R_{consumed}$
    (b) additional explicit constraints specified by developer

A graphic representation of the adapted meta-model is given in figure 4.1. In essence the model is a collection of *Resources* and *Components*. Each of these

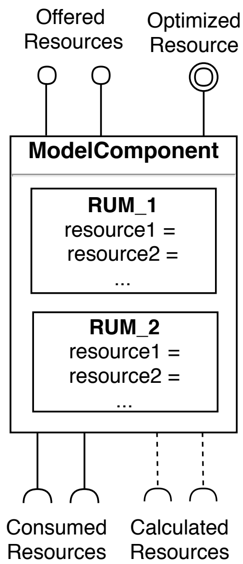


Figure 4.1: Notation of an RDM component with RUM's

resources can be connected to components by means of a *ResourceInterface* and a *ResourceFunction*

Resource

A resource is an entity describing a parameter of a system. This can be a measured parameter (e.g. battery capacity or throughput), but can also describe a derived parameter (e.g. service time left). Each resource is identified by it's name and has a unit associated with it.

ResourceInterface

Resources are interfaced with through ResourceInterfaces. A ResourceInterface can be one of three types:

Offer Indicating that the component produces an amount of the resource,

Consume Indicating that the component consumes an amount of the resource,

Calculate A special consume relation. This interface supplies 100% of the offered resource, without formally consuming any amount. This relation is used to further calculate with the offered value, without it impacting the constraints of the resource

Each interface has a value specifying the amount of the resource produced or consumed by the component. This value is repeatedly set and evaluated at runtime by executing a ResourceFunction. By aggregating the interfaces of a resource the amount of the resource produced and consumed can be computed and analysed.

ResourceFunction

The value of a ResourceInterface is determined by a ResourceFunction. It consists of a function that takes a double array and an array of resource identifiers as argument, and has a double as result. Runtime solvers or engines will then fill the input array, in accordance with the resource identifiers, in order to execute the function. ResourceFunctions are compactly instantiated using lambda expressions and VarArgs. E.g.:

```
ResourceFunction totalServiceTime = new ResourceFunction(
    (x)->x[0]+x[1], "yearsServed", "yearsLeft"
);
```

Component

Any entity producing, consuming and converting a resource is represented by a component. A component can therefore be a physical entity such as a radio module or a battery, or a hypothetical entity such as a QoS calculator executing a heuristic function. A component possesses a ResourceFunction of each Resource it is connected to.

A specific subtype of the Component is the ModelComponent. This class inherits all functionality of the ordinary Component. However, its ResourceFunctions are specified by its RUM's. Each RUM describes the parameters during one mode of operation of the component. This allows runtime analysis of variable behaviour as effect of different performance strategies.

To model and evaluate the intended behaviour of the model a set of *Requirements* and an *Optimizer* are introduced.

ResouceConstraint

A resource can have a number of constraints that limit the possible values of variation for that resource. The standard inherent requirement for every resource is the *OfferConsumeGTE* requirement which enforces that the amount produced needs to be greater or equal than the amount consumed. Additional requirements *OfferConsumeEQ* and *RangeRequirement* are specified, that respectively require the exact amount offered to be consumed and the amount offered or consumed to be within certain bounds. Finally, the abstract class *Requirement* can be extended by a developer to specify any tailored requirement.

Optimizer

The Optimizer is introduced to ascertain the heuristic score of an RDM with a valid RUM configuration. The Optimizer is an extended class of Resource of which exactly one must exist in an RDM. The optimizer takes the evaluated offered amount of this resource and calculates a score. This score is a value on a comparative scale on which a higher value implies a more optimal performance. Specified are the *MinMaxOptimizer* which evaluates that the amount offered must have a minimal or maximal value and the *ApproxOptimizer* which evaluates that the resource must have an amount offered as close to a specified value as possible. However, custom implementations of the Optimizer can be made.

RdmMessage

Finally, to supply the model with the state of the system under investigation the RdmMessage is posed. The RdmMessage is provisioned using values measured from the system and injected into the model, after which the appropriate resource functions are evaluated accordingly. Technically, a simple mapping from a resource identifier to a measured value would suffice for this purpose, but this mapping is wrapped in an object to support future evolution of the model.

4.5.1 Demonstration by example case

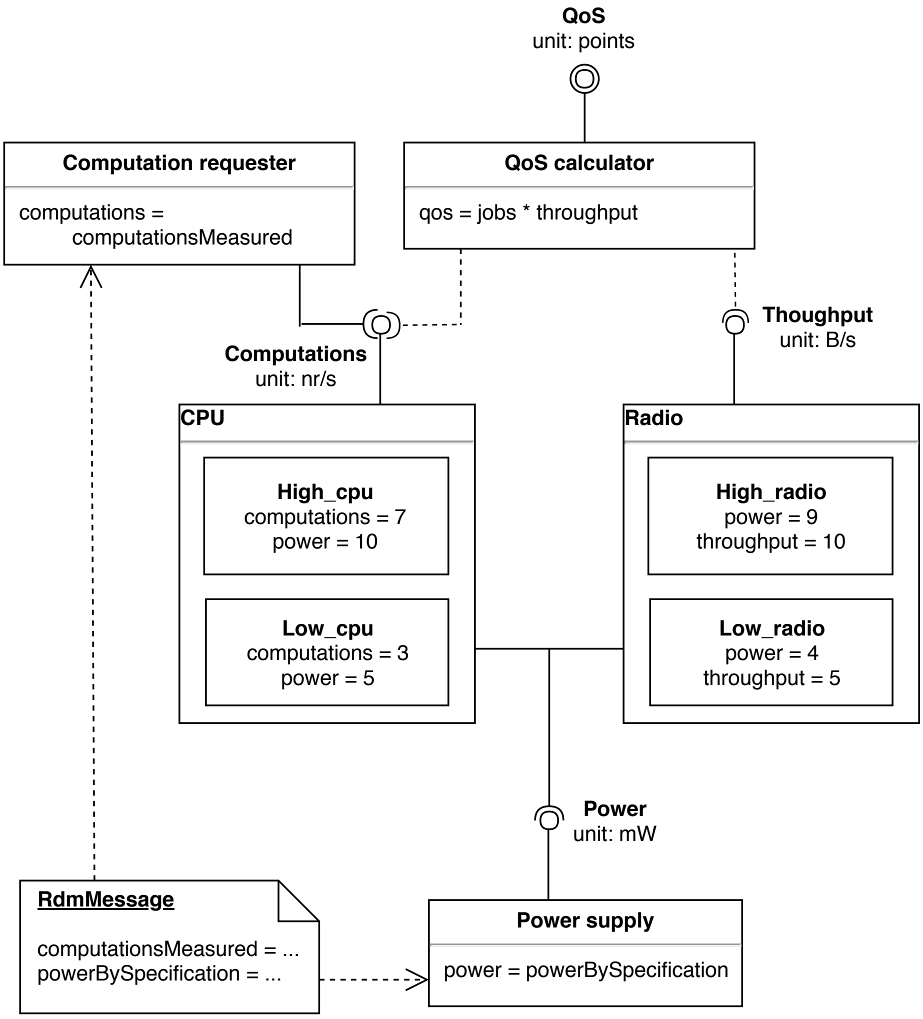
To illustrate the application of this model, an example of an instantiation of the model is provided in Figure 4.2. This instantiation is again based on the example case described in Section 3.6.1. This depiction contains a power supply (battery) which emits a resource 'power', measured in milliwatts. The actual value of this variable is instantiated based on the input message (illustrated by dotted arrow). The reason for this is that, as described earlier, specifications of power supplies vary in the example case. This power is consequently consumed by the device's CPU and radio module. This entails an implied resource constraint c_1, which enforces that the joint power consumption of the CPU and radio may not exceed the power produced by the power supply. Both the CPU and Radio can run on a high or low performance model, with the high models having aggravating consequences for the power consumption and the offered number of measurements and throughput respectively. The amount of measurements per second offered by the CPU is subsequently consumed in full by the *Measurement requester*. This component simulates a resource request on the sensor devices and imposes a requisite on the minimum amount of measurements performed and offered by the CPU, as formulated by constraint c_2. The requested value is determined by a parameter supplied by the input message.

Finally, both the amount of measurements and bandwidth provided are supplied to the *QoS calculator*. It uses these resources to calculate a singular value depicting the level of QoS provided by the model instantiation. This value is used to determine the optimal variable composition given a set of valid models. In closing, emphasis should be given to the interfaces of the QoS calculator. These interfaces are not regular *consume* relations but *calculate* relations. This entails that the QoS calculator has full knowledge of the amounts offered, without affecting the consumption of those resources. This ensures that the behaviour of the QoS calculator has no influence on the validity of the model by impacting constraint c_2.

4.5.2 Computing a valid, optimal model assignment

With the model well established and exemplified, its resolvability requires attention. Requirement R2.7 yields that solving the model is to find a composition of RUM's such that:

1. each ModelComponent has exactly one RUM associated with it,
2. all resource constraints are satisfied, and
3. the optimizer function of the optimized resource has the highest value.



Constraints:
$c_1 : power_{power\_supply} >= power_{cpu} + power_{radio}$
$c_2 : measurements_{cpu} >= measurements_{measurement\_requester}$

Optimize:
$max(QoS)$

Figure 4.2: Example instantiation of the Resource Distribution Model according to the example case.

The first and second requirement imply constraint solvers as a highly applicable technology, since they are effective in finding a valid solution for a constraint decision problem. However, the third requirement entails that not just any valid solution is requested, but the *optimal* valid solution. In order to achieve that, every valid solution to the problem needs to be considered and compared how they rank heuristically. This entails an exhaustive search approach through the entire search space of RUM compositions. However, constraint solver paradigms can be used to efficiently traverse that search space.

This is performed by employing backtrack-search. A simple brute force search would calculate all RUM compositions (Cartesian product) and for each composition the full model is provisioned and evaluated. Instead backtrack-search iteratively selects a component and one of its models. It will then not provision the entire model, but inject only the selected model in the chosen component. Subsequently, given the current state of the model, the variables which can be resolved are assigned a definite value. After which, the resource constraints are evaluated. Given a partial model assignment, any constraint can have one of three statuses:

- satisfaction,
- failure, or
- unresolved

for all subsequent assignments of unprovisioned components.

If a constraint evaluates to *satisfied* it will be pruned from the constraint set and will not evaluated for the remainder of this branch of the search tree, for it is known to always succeed. If a constraint is *unresolved* it is kept, since its status is not certain for every future state. If even a single constraint *fails* the remainder of that branch of the search tree will never be valid. Therefore, the algorithm backtracks through the tree by partially rolling back model assignments. A different model is then selected for the same component or a different component entirely and the algorithm is repeated. This ensures that validated constraints are not reevaluated and invalidated search tree traversals are detected early.

By rapidly detecting unsatisfactory options in the search tree, large portions of the tree can possibly be eliminated. An example of the application of this algorithm on the previously illustrated example (Figure 4.2) is given in Figure 4.3. This example is executed based on an RdmMessage with values $\{measureRateRequired = 5,\ powerBySpecification = 16\}$. This application demonstrates that using this algorithm eliminates a significant portion of the search tree. This is due to early detection of constraint failure in the *CPU=high_cpu* branch of the tree.

4.6 Discussion of the proposed model

This chapter will be concluded by endorsing some of the choices that were made for the proposed model.

Behaviour as static RUM's

As stated before it was chosen to use a static representation of resource utilization in Resource Utilization Models. This was chosen in order greatly reduce

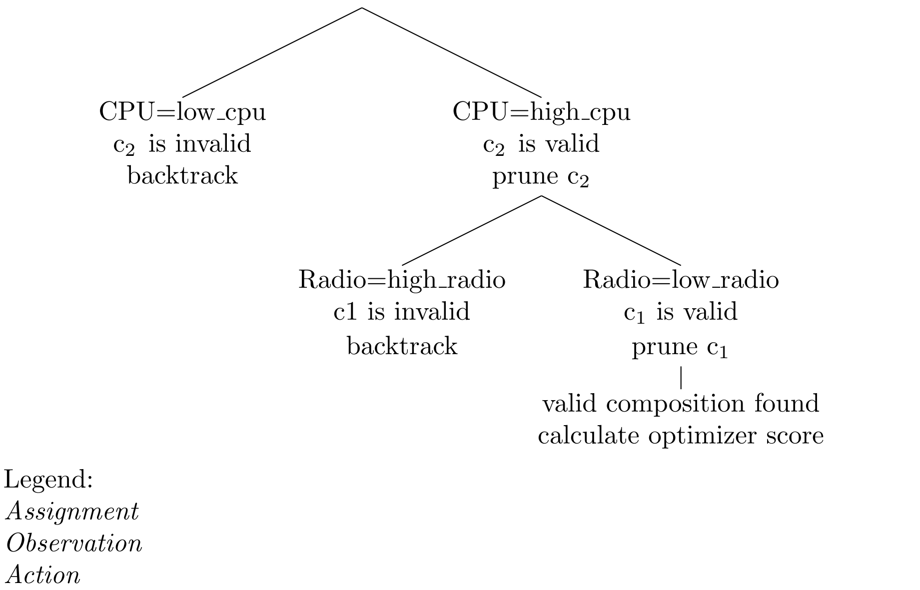


Figure 4.3: Application of backtrack-search on RDM of Figure 4.2

the complexity of the search problem, which allows the model to be evaluated within a reasonable amount of time. This was decided after early experiments with timed automata. In this experiment a minimal system with one component with three RUM's was modelled. When computing the behaviour of the model using time intervals of one week over a life span of ten years, it took over one minute to calculate the optimal traversal of the automaton. Granted, this was performed on a laptop machine and not a high-powered server. When deployed on a machine with elevated computational resources the time to calculate will be reduced. However, this is counteracted by the fact for a WSN application this calculation needs to be repeated for thousands of sensors. When this performance is compared to that of the static models, which can evaluate more complex models within seconds, timed automata must be eliminated as viable solution for real-time analysis. However, this does not eliminate automata entirely. Automata can still be used to model the fine grained runtime behaviour a system in order to abstract generalized static RUM's from it.

Solver libraries

When developing this solution a decision was made to implement a custom constraint solving algorithm, instead of employing existing libraries such as Choco Solver [78] or OptaPlanner [79]

The Choco Solver is a powerful solver which not only employs backtrack-search, but also constraint propagation to eliminate failing search paths before assigning them. However, while powerful, it has only limited support for real intervals [80]. Additionally, it proved very difficult to convert the user-specified models and ResourceFunction's arithmetic expressions to the modelling mechanism of the solver. Requiring a developer to either input the model and calculations in the complex modelling mechanism of the Choco Solver or for a translator to be developed that compiles the user-defined model to Choco Solver code.

Another examined library is the OptaPlanner [79]. The OptaPlanner is a modelling framework for constraint problems and excels in use cases involving planning and resource allocation. It also enables object injection which would be greatly suitable for injecting RUM's into components. However, the OptaPlanner is strictly a constraint modelling framework and does not employ advanced solving techniques developed in the field of constraint programming. It performs a brute force depth-first search over the search space and executes a single code block which evaluates all constraints. It consequently cannot reduce the search space by eliminating failing branches and redundant constraints. Therefore, it lacks the means to solve the problem efficiently.

Finally, developing a custom solver allowed incorporation of domain knowledge into the search algorithm, further reducing overhead. This reduces the comparative benefit of employing a constraint solver library and eventually led to the development of a custom solver implementation.

Constraint propagation

A technique in constraint solvers mentioned before is the concept of constraint propagation [68]. Constraint propagation explores the search space in the same manner as backtrack-search. However, for each variable assignment V_1 all other variable domains are preventatively reduced by pruning all variable assignments V_2 that are incompatible with V_1. For example in the example of Figure 4.2: if *CPU=Low_CPU* is initially assigned, *Radio=High_radio* is pruned immediately, because it would require more power than is actually produced. This eliminates inconsistent variables without the need of assigning them, thereby reducing the search space even more effectively than native backtrack-search. This is easily implemented with integer/real variables that are interconnected with constraints. However, in the model the variables are not integer/real domains, but objects with integer/real variables and functions. This doesn't make constraint propagation impossible, but does complicated it.

Secondly, the interconnected nature of resources can impede the benefits received from constraint propagation. To illustrate this consider the following complex example: resource R is connected to a set of producers P and a set of consumers C, for each the amount produced or consumed is variable. The amount produced or consumed by any component x is denoted by R_x. The availability constraint (i.e. amount produced must exceed the amount consumed) on R can then be written as:

$$\sum_{p \in P} R_p \geq \sum_{c \in C} R_c$$

Which entails for any consumer $c1 \in C$:

$$R_{c1} \leq \left(\sum_{p \in P} R_p - \sum_{c2 \in (C - c1)} R_{c2} \right)$$

In order to be able to prune any value from the domain of consumer *c1*, all producers must be assigned. Only then can a concrete upper bound be determined[2]. This requires the search to be already at least $|P|$ levels deep, reducing the part of the tree possibly eliminated. Even then, only values can be pruned for which:

$$R_{c1} > \sum_{p \in P} R_p$$

Which might not be many since a single consumer must consume more of a resource than produced by all producers combined, in order for the constraint to fail. When other consumers get a value assigned pruning becomes more effective, but this requires even more variable assignments.

To conclude, the part of the tree that is eliminated with constraint propagation is limited. For any variable to be pruned the algorithm must already be halfway into the search tree. Furthermore, the chance that a value is eliminated halfway in the tree is very small. Therefore, no further effort was made to incorporate constraint propagation or other look-ahead strategies in the solver.

[2]Future assignments of the other consumers may be disregarded since they will never raise the upper bound for R_{c1}, only lower it.

5. Proof-of-concept validation by case study

This chapter will attempt to validate the applicability of the platform to the field of LPWA QoS monitoring and management. This will be performed by designing and developing a prototype monitoring application, based on the proposed platform. This will not be performed on the afore-used hypothetical example from section 3.6.1. Instead, it will be performed on an actual, commercial car parking sensor application.

Firstly, some background will be given on the sensor application to be monitored. Next, the goals, claims and methodology of the study will be declared. With the goal and means stated, the experiment will be performed by realizing a prototype monitoring application. As the actual implementation details are auxiliary, they will not be examined in detail. However, the implementation will be described superficially to contextualize the validation efforts. After implementation, the results of the validation study will be presented and their implications deliberated. This chapter will be concluded with a discussion on the results, conclusions and limitations of the study.

5.1 Context of the case study

5.1.1 Background

The case the development platform will be applied to is to the Nedap Identification Systems smart parking application: SENSIT. Nedap [81] is a Dutch company based in the city of Groenlo. They produce hardware and software integrated products for a plethora of industries, such as retail, health-care and smart city management. The department Identification Systems [82] focusses on the latter category. They develop solutions for detection, identification and physical access management of people and vehicles. This is performed by employing a series of self-produced hardware products such as RFID tags, sensor devices and cameras, with accompanying software products and platforms.

SENSIT smart parking application

The SENSIT [83] smart parking application is devised by Nedap Identification Systems to monitor city on-street parking. It employs a huge amount (up to thousands per location) of affordable LPWA sensor nodes. Each individual parking spot is equipped with one of these sensors to determine its occupation.

To determine changes in occupation, each sensor is equipped with an infra-red and magnetic induction sensor. Should a change in occupation be detected, a message containing the measured sensor deltas is sent to the back-end application. This granular approach to smart parking allows the SENSIT application to monitor and visualise the occupation of individual parking spaces in a lot, garage or even across cities.

In order to communicate with the back-end the sensors employ wireless technology. Previously, the sensors were connected to sinks using a proprietary network of relay nodes and sinks. However, the recent proliferation of large-scale cellular IoT networks has caused Nedap to shift towards these technologies. This allows large numbers of sensors to a single cell tower, without the need of deploying and managing a network of relay nodes for new sensor deployments. Additionally, the efforts of managing and maintaining the network are outsourced to professional operators. To connect the sensors to the internet the *Narrow-band Internet of Things* technology was determined to be most suitable. New SENSIT sensors are therefore equipped with u-blox [84] NB-IoT radio modules to connect them to operated cell networks.

5.1.2 Conceptualization of the monitoring application

This section will describe and scope the context of the QoS monitoring application to be developed. First, the input for the application, as emitted by the WSN application under investigation, will be examined. Subsequently, the characteristics of the expected outcomes of the application to be prototyped will be discussed.

Sensor data signature

The sensor devices send a message with key performance indicator (KPI) data alongside every data message it sends. Alternatively, it will send one of these messages periodically if no data messages are sent for 12 hours. When computed universally, a message rate was determined of about 15 messages per sensor per day. However, a specific per sensor analysis yields a message rate of between 10 and 50 KPI information messages on average per day, with some outliers for more active sensors which can reach up to 250 messages per day on a regular basis.

The data sent by the sensor contains some typical networking data points, such as source IP address, source port, source device ID, message sequence number and a timestamp. Additionally, the message contains a hexadecimally encoded string describing the KPIs collected by the u-blox radio module. The data collected by the u-blox module contains mostly data points depicting the signalling functions of the radio module. Such KPIs include the signal-to-noise ratio (SNR), signal quality (RSSI), Extended Coverage Level (ECL) and more. Additionally, the KPI information includes some physical attributes of the radio module. Attributes such as the module's uptime, number of restarts and temperature.

The ordinary data plus the u-blox KPI data are contained within 128 Bytes of data. Considering the messaging rate of a typical sensor yields an imposed per sensor footprint on bandwidth of ±1–6 KiB/day for the majority of sensors with outliers of ±KiB/day for extremely active sensors.

At this moment only a few nodes equipped with the NB-IoT technology have been deployed. Consequently, a large-scale test bed for the to be prototyped monitoring application does not exist. Therefore, a simulated sensor environment has been devised to test the prototype application for contemporary and near-future smart parking applications. This simulation is based on data signatures and values observed over a half year period emitted by the few nodes that have been deployed.

QoS monitoring needs

In collaboration with Nedap Identification Systems a list of requirements for the outcomes of the prototype was compiled. These consequences are to be effected by the prototype application, based on input from (simulated) sensors. However, the actual implementation of the prototype is secondary to this chapter, since the primary goal is to evaluate choices made for the underlying development platform. Therefore, a comprehensive, formalized requirements document has not been included in this thesis. However, the features required of the monitoring application to be developed will be described shortly, in order to contextualize the implementation efforts of the prototype.

The consequences the application must effect are classified into three categories. The first of which is sensor feedback. This entails commands sent to sensors to alter its execution strategy, based on observations made in the monitoring application. This can be based on individual sensor data, historic sensor data or higher-level data snapshots (e.g. sink level). An example of such feedbacks are to decrease data rates to guarantee a predetermined minimum sensor lifetime or due to poor cell connectivity. This functionality is currently not present in the Nedap sensors, but is intended in the future. Therefore, it will be implemented into the simulation environment to test the command & control capabilities of the platform.

The second type of effect to be caused by the application is instant alerting. The primary use case for this kind of consequence is when physical maintenance is imminently required in the application or its network. Detectable causes of when this might be warranted have been deliberated with Identification Systems and examples include:

- A long term drop in coverage level which might indicate permanent obstruction of signal.
- Extremely high temperature readings indicating an electrical malfunction.
- Unusually long periods of inactivity or, conversely, extreme data bursts indicate a rouge node not executing according to a valid strategy.
- Calculations estimating node lifetime determining a node needs replacing.

The last type of consequence is reporting. The goal of this is to inform technicians, managers or clients on the general operation of the WSN application. This comprises two types of reporting. The first is *periodical reporting*. Periodical reporting will primarily focus on business goals such as long term performance metrics, compliance to service level agreements of both service providers and clients, and prospected short-term maintenance efforts and costs. The other type of reporting is *real-time reporting*. This is useful to technicians monitoring the performance of an application during its runtime. Use cases include monitoring the number of incoming events, latencies of sensor devices and sinks, environmental conditions (such as weather and temperature) and which sensor strategies currently are deployed. Notice that the real-time aspect of this type of reporting does not require events to be reported instantaneously since for such statistics a per second or minute update suffices.

5.2 Validation method

This section will detail the approach taken for this preliminary validation study. First, the general approach of the study will be listed. After which, the claims to be examined will be detailed, along with the specific methodology employed to test them. The section will conclude with a short discussion on the scope and bounds of the study.

5.2.1 General approach

In order to ascertain whether the level of abstraction of the platform can facilitate the needs of the intended monitoring application for SENSIT, a prototype implementation will be designed and constructed. The expected outcome is an instantiation of the platform that serves the QoS processing needs of Nedap Identification Systems.

The possible existence of such an instantiation demonstrates that (at least for this use case) the level of abstraction is low enough to expose the full functionality that is required (**applicability**). In order to validate that the level of abstraction is low enough, but not too low (**usability**), the program instructions required for the platform instantiation will be considered. These required instructions should not be more then the instructions required for a hypothetical monolithic implementation, supposing the level of abstraction is not too high (applicability claim). Finally, the **adaptability** of the platform and its instantiations will be evaluated by introducing some minor changes to the features and requirements of the monitoring application. It will then be hypothesized what the consequent changes to the platform implementation are. Should the appropriate level of abstraction have been chosen, it should prove uncumbersome to adapt the topology to these novel conditions.

From a business perspective, the most interesting parameter to express these these efforts would be the time required to develop. However, this parameter is extremely subjective as it heavily depends on the level of skill of the developer and its familiarity with the technology. Therefore, the effort will primarily be quantified by the code required, expressed in number of instructions required to construct a monitoring application built by adoption of the platform.

5.2.2 Claims

The cardinal claim investigated is that the appropriate level of abstraction was chosen in the design of the development platform. This entails that the provided collection of components can be adapted to suit a plethora of purposes and target applications. Conversely, the level of abstraction is not that low-level that implementation requires unnecessarily large development efforts because basic procedures require repeated implementation. This claim mirrors research question RQ3, which asks "What is the appropriate level of abstraction for a WSN monitoring platform [...]". This claim is explicated into three sub-claims.

Applicability

Intuitively, the first criterium regarding the level of abstraction is that the platform features a level of abstraction low enough to facilitate the implementation of the monitoring application for SENSIT. I.e. the platform's abstraction does not obfuscate key functionalities which would require reimplementation of formerly present features. This seems an obvious and trivial demand, but without stating it, any subsequent criterium is pointless. More formally, the platform should enable an instantiation which enables iterative enrichment and aggregation of information. At multiple stages of the consequential iteration the application should be able to generate outputs such as alerts and reports for auxiliary processes and systems.

Usability

The second criterium to be validated is that the level of abstraction is not too low. Though the platform should enable an instantiation according to the needs of Nedap Identification Systems, it should do so with minimal development effort. A level of abstraction that is too low requires application developers to repeatedly implement functionality that, due to their frequent nature, should have been provided by the platform itself. This criterium seems similar to the first, but the metrics determining their attainment are measured differently. Therefore, they will be regarded as two separate claims.

These development efforts will be expressed in the number of code instructions required to realize the implementation. Since an absolute benchmark was difficult to ascertain, the upper bound of permissible number of code instructions is established relative to the amount of instructions necessary for a functionally similar monolithic implementation. Should a larger code-base be determined, this entails a level of abstraction that is too low and requires (repeated) implementation of procedures that should have been provided by the platform itself. For the construction of the topology it was chosen to allow at most 4 operations for every component in the platform topology. The criterium of 4 operations per component originates from an assertion made in Chapter 3.

Adaptability

The final criterium employed to validate the appropriate level of abstraction is that the platform facilitates convenient adaptation of a realized platform implementation. This validation will be performed by introducing or changing a minor feature (e.g. new input type, altered reporting requirement). Should the appropriate level of abstraction have been chosen, it should prove uncumbersome to adapt the topology to these novel conditions. For the adaptability of the application provided by the platform, it was determined that minor new features and requirements should require not more than:

- a localized rearrangement of the model/topology, and
- introduction or major change of at most two components.

For all cases, small changes are allowed to the components interfacing with the altered component(s) in order to produce or consume information supplied to or emitted by the altered interface. Additionally, very minor, consistent changes are allowed to be made to other components. The reason for this is that often a change or introduction of a data point requires that data point to be propagated throughout the topology.

The rationale for these allowances is that the modularization provided by the platform should prevent entanglement of concerns and therefore minor changes should cause localized effects. There is however a possibility that (especially new features) require a change in several components since its the functionality was not previously present. Therefore, minor consistent changes are allowed to those components in order to forward the new functionality. Finally, the reason for the allowance of a major change in two components is that often computation and analysis of a data point is separated into distinct components due to separation of concerns. Therefore, a changed requirement will often require a change in both components.

5.2.3 Bounds

Before executing the validation study, the bounds and limitations of this validation study will need to be considered. The first glaring limitation of this study is that it is extremely limited in scope. The platform will only be implemented for a specific WSN application and this study will therefore not state the platform to be appropriate for the entire set of applications that was determined in Section 2.1 of the background chapter. Instead, this study will at most affirm the platform as a proof-of-concept for WSN application QoS monitoring.

The second limitation worthy of notion is that, aside from only regarding a single WSN application, it will also run on a simulation of that application. As mentioned before, this is because the NB-IoT-incorporated sensor devices of the SENSIT application have only recently started deployment. As a consequence, a test bed of significant scale is presently not available. However, simulating a full future deployment of the application enables easy adaptation of the WSN application under investigation, in terms of both scale and functionality. This allows to not only test for intended regular behaviour but also for extreme and niche conditions. Additionally, the simulated environment allows for easy temporal manipulation, which enables the simulation to be accelerated, halted and repeated.

5.3 Implementation of the WSN monitoring application

5.3.1 Design and Implementation

In this section the design for the platform instantiation for Nedap SENSIT will be detailed. First, a top-down look at the entire topology will be taken. After which the functionality of the individual components will be described shortly. Finally, the instantiation of the Resource Distribution Model used to compute the state of sensors will be illustrated.

Application topology

The designed topology is depicted in Figure 5.1. This figure shows the processing to be divided into three stages. In the first stage raw-information snapshots are enriched and normalized. In doing so it improves the information potential and accuracy of the data in the snapshot. The second stage concerns sensor level analysis and management. It calculates the state and resource consumption of the devices, and it includes some services that alert if a sensor exhibits abnormal behaviour or long term deviations of its ordinary parameter margins. The final stage concerns snapshot accumulation in order to extract high level information and adaptations. This stage diverges into three distinct accumulator paths. The top path performs accumulations of snapshots based on the sensor group ID. It reports on data rate violations (as agreed upon in SLA's) and recalculates the share of the data each sensor within a sensor group is allowed to consume. The middle execution path concerns the cells served by nodes. It alerts if a node switches cells more then an allowed amount during a period. The bottom accumulates all snapshots in order to report on the current state of the application as a whole.

The description of the application topology will be concluded by shortly describing the functions of the individual components.

Sensit spout
: Reads sensor snapshots from a Kafka channel and introduces them into the topology.

Translator
: Translates the sensor information from hexadecimal string to key-value pairs.

Nuancer
: Averages the data points received from a sensor to eliminate abnormalities. It does so by keeping a record of the last seen messages for each sensor node in an SQL database.

Attributor
: Enriches the snapshot with some data points not present in the sensor but known by back-end services.

Sensor RDM processor
: Processes the enriched information from the snapshot and calculates the optimal operational device strategy.

Switch RUM buffer
: Buffers the switch strategy messages to prevent superfluous, erratic feedback to the sensors. Doesn't switch strategy on first report, only if a switch is requested over an extended period.

Single message analyser
: Calculates whether the sensor parameters, as calculated by the RDM processor, are within the allowable margins.

Budget recalculator interface
: If the message rate of a sensor is high enough will initiate an immediate budget recalculation. If message rate is low it is allowed to be accumulated over some time to reduce the number of database updates.

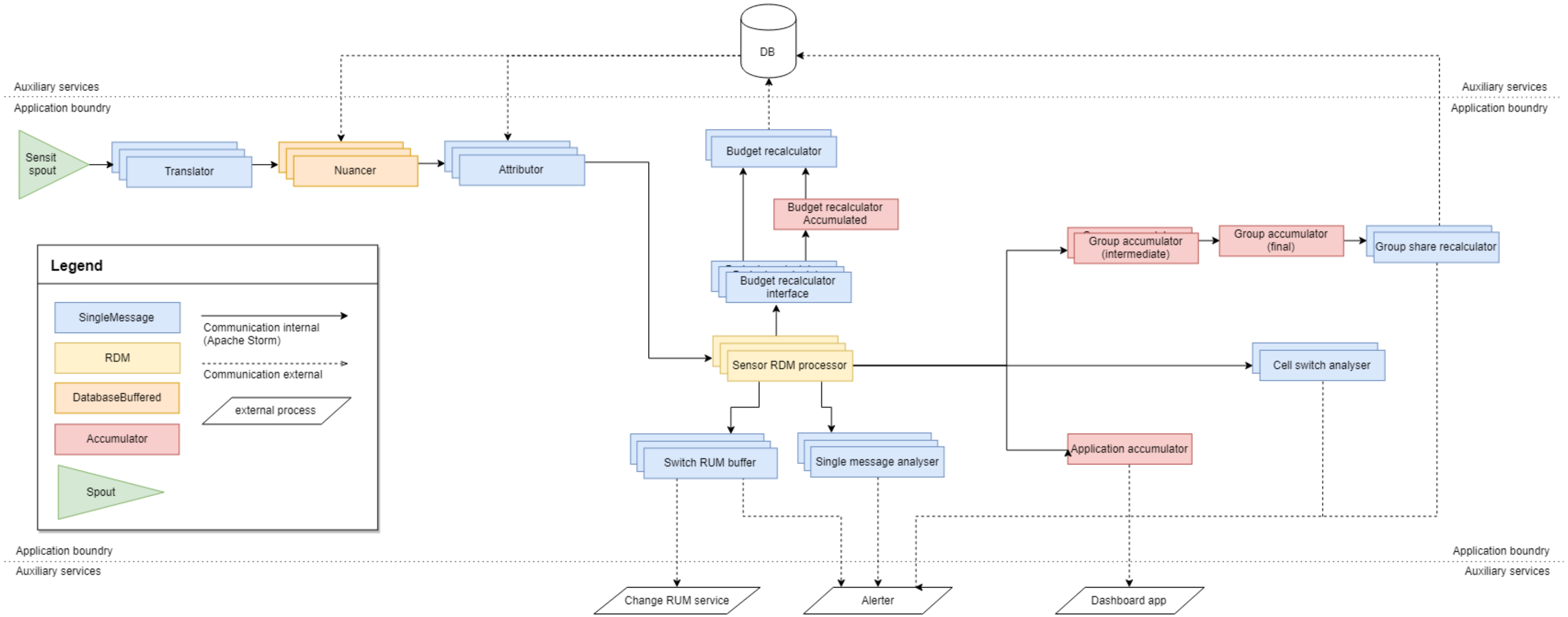


Figure 5.1: Topology of the monitoring application for the Nedap Identification Systems SENSIT WSN application

Budget recalculator accumulator
: Accumulates budget recalculation snapshots and prepares them for batch update.

Budget recalculator
: Executes (batch) budget recalculation.

Group accumulator
: Accumulates snapshots by sensor's group ID. Because this is performed on a weekly basis, this is performed two-stage as not to cause a large data build-up over time.

Group share recalculator
: Recalculates the share of the sensor group's resources each sensor is allowed to consume, based on the data used by each node over a one week period.

Cell switch analyser
: Analyses and reports if a node switches between cell towers more then is allowed.

Application accumulator
: Accumulates the information emitted by the application in order to be presented on an application dashboard.

A final remark on the application design is on the interfaces it provides. The application's inputs and outputs are received from and provided to Apache Kafka channels. This allows actual services to be easily swapped in and out with test services (even at runtime)

Sensor Resource Distribution Model

The Resource Distribution Model proposed in Chapter 4 is employed to model the state, behaviour and strategies of the sensor. The resulting model is depicted in Figure 5.2.

The model takes a few parameters based on the sensor state measurements, such as its current ECL and message rate, and its history, such as its runtime, data already used and messages already sent. Additionally, the model receives some data points on the availability of resources such as the allowed number of messages during a time period (called the budget) and the allowed data usage for that sensor. The model then computes the runtime the sensor has left, current data and budget consumption and the future message rate.

The variable behaviour of the *MessageRateDeterminer* is curtailed by two constraints on the model. The first is that the *budgetLeft* produced by the *BudgetProvider* must exceed the amount that is consumed by the *RemainingCalculator*, which is ultimately derived from the allowed message rate for the sensor device (*messageRateFuture*). Similarly, the same holds for the *dataLeft* produced by the *DataProvider*. Should multiple RUM assignments evaluate as valid, the most optimal one is decided to be the one which provides the highest message rate.

Preliminary experiments with resource consumption models have shown that, when a scarce resource is involved, a device will act differently in the beginning than in the end of its life-cycle. The reason for this is that in the beginning the models will instruct the device to operate on a strategy that will

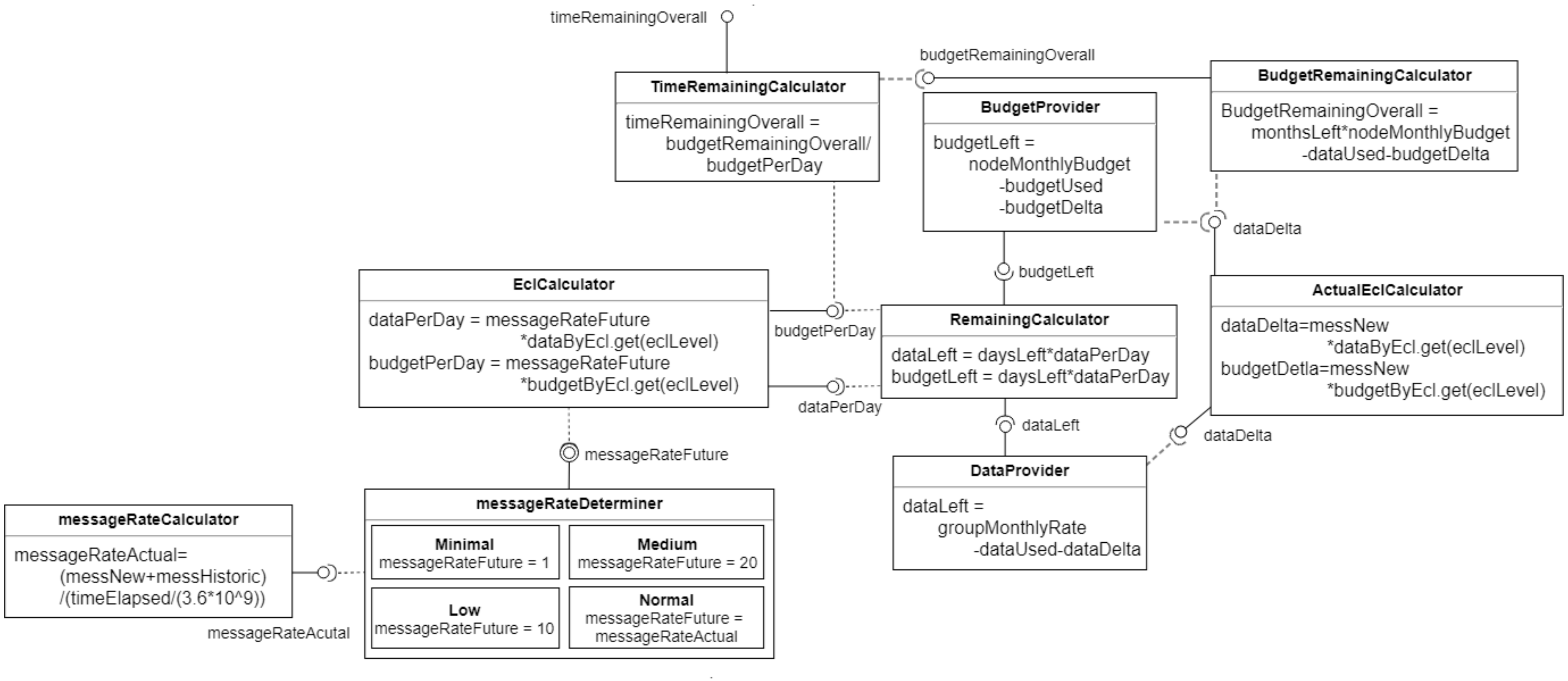


Figure 5.2: Resource Distribution Model for a sensor in the Identification Systems SENSIT WSN application

consume less resources then it is allowed on average. Then, when it has saved up enough of that resource, it is allowed to spend it on a strategy that consumes more than that average. To mitigate this effect it was decided to recalculate the available resources on a monthly basis. This way there is still such a cycle, but its period is far shorter and the effect will be much less and much more regular overall.

5.3.2 Adapting the application

This section will be concluded by deliberating some hypothetical adaptations in order to investigate the adaptability of the platform.

Nuancer local

The first change introduced is the constraint for the *Nuancer* to not require a database connection. Reason for such a requirement could be to reduce latency or to eliminate capacity issues caused by employing an SQL database.

This can be achieved by exchanging the current *DatabaseBuffered* Nuancer implementation with a *SingleMessageProcessor*. This processor keeps an in-memory cache of the last snapshots it has encountered, grouped by node and ordered by timestamp or sequence number. For each incoming snapshot the following sequence of actions is taken:

1. determine node by ID,
2. add snapshot to the node's buffer,
3. prune out-of-scope snapshots from the buffer,
4. calculate average of remaining buffered snapshots, and
5. emit averaged snapshot

This sequence of actions is similar to how the Nuancer operates in the current topology, but it eliminates the database connection in favour of a local buffer of snapshots. Unfortunately, by shifting to a local buffer the scaffolding provided by the *BufferedComponent* can no longer be employed. The reason for this is that the component with local buffer (as currently implemented) operates on a single global buffer, instead of a buffer per node.

Finally, it must be noted that in requiring the snapshots to be cached locally, a large burden is forced upon the memory of the machine/container running the component. Should the application serve a large amount of nodes and snapshots are collected within a large window of interest, the data kept in-memory can rapidly reach large sizes. This can be alleviated by replicating this component to the point that individual memory requirements of workers are within manageable parameters. Alternatively, the memory issue can be evaded by persisting and reading snapshots to local files. This introduces some latency due to disk IO — however far less than database communication does — but can immensely reduce the number of records in the active cache at any time.

New sensor data encoding

As mentioned, the auxiliary performance data of the sensor is received as an encoded hexadecimal string. For this case, a new a new hypothetical type of

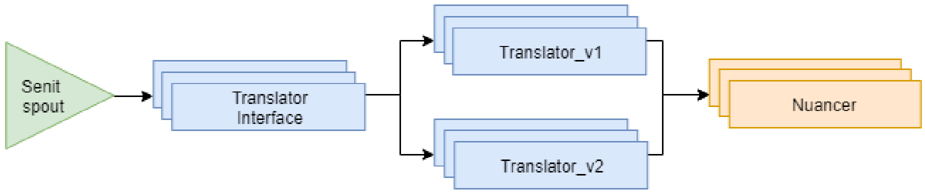


Figure 5.3: Updated partial topology for new data encoding

sensor is introduced. This sensor equipped with a different radio module, which encodes its KPI data slightly differently. It is emphasized that the actual data collected and emitted by the sensors is not changed significantly. As this would entail a major change in how computations need to be performed. Though deliberated as a hypothetical, this case simulates a real future scenario. Since the aim is for a node lifetime of at least 10 years, it is very conceivable that wireless sensor technologies improve and change during that time frame. Since physical replacement of the large volumes of deployed nodes is unprofitable for both Nedap and its clients, this new technology should be supported in tandem with the old sensor types.

This change in the sensor environment can be accommodated by introducing a second *Translator* component specifically intended for the new data format. This component is executed independently of — and in parallel to — the original Translator. How to ensure that a snapshot is processed by the correct translator, will depend on how the new data stream is supplied to the application. This hypothetical will consider the most complicated input option, where the old and the new style snapshots are emitted on a single input channel. An interface component is introduced to split the singular input stream into two. This component performs a superficial inspection of the snapshot and forwards it to the correct Storm channel based on some discerning feature (e.g. data format or type identifier). Though technically this inspection could be performed by the *SensorSpout*, separation of concerns compels a separate component for this purpose. Ultimately, both translators uniformly emit their translated snapshots to a common Storm channel for further processing. The resulting partial topology is illustrated in Figure 5.3

Alert on long-term ECL drop

For the final case, the functionality of the application is extended by introducing a new outcome for the application. The added requirement is the detection of long term drop in ECL level. Such a drop could signify a (possibly alleviable) obstruction placed between the sensor and sensor sink (cell tower). Moreover, should several geographically related sensors report such a disruption, drastic actions cannot be ignored. In the topology this can easily be achieved by extending one of the existing components.

Formally, the *CellSwitchAnalyser* would be most suited for this purpose, since it is already historically aware due to retaining a list of cell towers per sensor. Though the component would obviously require renaming. This functionality is provided by keeping a list of ECLs reported by each sensor node.

When the sightings are inconsistent or do not feature a drop, the list is pruned. When the list's size surpasses a set threshold — i.e. a consistent ECL drop has occurred — an alert is sent to the alerter. This is easily implemented since the CellSwitchAnalyser already features alerting functionality. Finally, this change does not require changes to interfacing components, since the ECL level is already present in the snapshot emitted by the *SensorRdmProcessor*.

5.4 Results & Evaluation

The results of the study will be reported in accordance with the three sub-claims and discussed under their own three headings: applicability, development effort and adaptability.

Applicability

The sensor model was found to be adequate for modelling the behaviour of the SENSIT sensors. The modular design proved very useful for expositioning the different resources and how they were interconnectively calculated and distributed. Unfortunately (for the purpose of this study), the sensor did not feature a large variety of resource metrics specifying its configurable behaviour and therefore the model only featured one configurable component. Additionally, after accumulation of the application-level parameters by the *ApplicationAccumulator* the accumulated parameters needed no further transformations and the WSN application did not feature application-level configuration needs. Therefore, the Resource Distribution Model was only employed on the sensor-level.

The result of the applicability investigation with regard to the distributed topology is that the platform suffices as development platform for the purposes of Nedap Identification Systems. The provided building blocks enable the implementation of a functional application and provide functional abstraction of the specifics of the underlying technologies. During implementation of the application it was noted however that the platform does not provide an efficient way of buffering and processing snapshots grouped per node, cell tower, etc. This functionality could easily be provided by implementing the *BufferedProcessor* with multiple buffers. A mapper function introduced to the processor will then determine which buffer a snapshot will be added into. The existing filter, sort and execution methods will then be performed on these buckets individually, providing a mechanism of grouped computations.

However, such functionality currently is not present, this absences was easily compensated for and was found to be only a minor inconvenience. This issue singularly was not sufficient to invalidate the applicability criterium. Therefore, it is stated that the applicability criterium holds.

Usability

Specifying the sensor model and application topology could be performed within the set parameters. As claimed, each component requires but four actions to be introduced to the topology. These actions are:

1. create component,
2. declare component,

3. subscribe consumer channels, and
4. declare output channels.

However, the internal code of the topology components, which actually performs the calculations and computations, required more code that a monolithic alternative would. While the actual number of lines of code was only a little higher than then its monolithic counterpart, the computations and transformations performed on those lines was far more then would be necessary in a monolithic application. These discrepancies will be deliberated on further in Section 5.4.

This was also reflected in the time required to develop this prototype. It was initially expected that the instantiation could be constructed within 40 man-hours. However, this eventually took twice as many hours. Of that time about 15% was spent designing, 35% developing and 50% debugging the application[1]. The breakdown of the time spent yields that it took an enormous amount of time to debug and adapt the components after its original design and implementation. The chief reason for this was found to be the loose coupling between components. The components are completely disjoint and the snapshot variables they share require custom serialization between components and are accessed with string identifiers. This entails that it is excessively easy to implement a component pair with mismatched coupling. This is since inappropriate variable access due to misspelled identifiers can occur very easily and is not detected by code checkers and compilers of conventional programming tools. Subsequently, when the variable is accessed successfully, the value often requires deserializing into the correct primitive or object type. This again introduces a possible point of failure due to misparsing and miscasting, since the compiler cannot detect the actual object type before executing the application.

The introduction of snapshot struct objects (POJOs) is proposed to alleviate both above mentioned problems. These objects contain the variables of the snapshots passed between components. However, in contrast to loosely coupled key-value bindings, these bindings are explicitly defined in both type and identifier. They can therefore easily be serialized and deserialized by common serialization mechanisms. This would alleviate the need for developers to continually specify custom serialization. By providing direct access to the correctly parsed variables in the snapshots it will reduce the code base by a huge amount. Additionally, by providing a mechanism to directly access the properly parsed variables, the number of possible instances where mismatching, misparsing and miscasting can occur is reduced. Thereby eliminating several points of possible failure which have proved problematic. Combined, this increased traceability and automated (de)serialization should have a noticeable, positive effect on the amount of required code. Consequently, it will reduce the time spent debugging and reworking the application and thus the development time as a whole.

To illustrate this benefit, two simplified code snippets from the *SensorNuancer* are presented. One which does not employ structs (Listing 5.1) and one which does (Listing 5.2). From these examples it is clearly observable that by employing well-defined, serializable structs, the instructions required are reduced. Additionally, it reduces the chance of mismatching variable identifiers by eliminating string bindings.

[1] All hours spent after fully constructing and first execution of the application are pooled into the latter category

```java
public void runForMessagesHistoric(LinkedList<IOMessage> history) {
    Map<String, String> args = new HashMap<>();
    long first = Long.parseLong(
        history.getFirst().getVars().get("TIMESTAMP"));
    long last = Long.parseLong(
        history.getLast().getVars().get("TIMESTAMP"));

    List<Integer> ecls = new LinkedList<>();
    for(IOMessage m : history){
        ecls.add(Integer.parseInt(m.getVars().get("ECL_LOCAL"));
    }
    int normalizedEcl = normalizeEcl(ecls);

    args.put("MILLIS_ELAPSED", Long.toString(last-first));
    args.put("ECL_LOCAL", history.getLast().getVars()
                                                    .get("ECL_LOCAL"));
    args.put("ECL", Integer.toString(normalizedEcl));
    publish("SENSOR_NORMALIZED", new IOMessage(args));
}
```

Listing 5.1: Simplified fragment of *SensorNuancer* without structs

```java
public void runForMessagesHistoric(LinkedList<NStructIn> history) {
    NStructOut output = new NStructOut();
    long first = history.getFirst().getTimestamp();
    long last = history.getLast().getTimestamp();

    List<Integer> ecls = new LinkedList<>();
    for(NStructIn struct : history){
        ecls.add(struct.getEclLocal());
    }
    int normalizedEcl = normalizeEcl(ecls);

    output.setMillisElapsed(last-first);
    output.setEclLocal(history.getLast().getEclLocal());
    output.setEcl(normalizedEcl);
    publish("SENSOR_NORMALIZED", output);
}
```

Listing 5.2: Simplified fragment of *SensorNuancer* with structs

Finally, it was noted that after initially specifying the topology and models, reworking them proved to be frustrating. The difficulty was mainly in locating the instantiation and declaration of a component in the code that builds the topology. The reason for this is that it constantly requires a developer to transition from a two-dimensional graphic image of the model or topology to builder code which is one-dimensional (lines of code). This mental transition can be avoided by eventually developing graphic development tools that allows a developer to conceive a topology by drawing a graphical model of components and resources. The appropriate computational code can then later be introduced into the components. By doing so, a developer would only need to concern themselves with one depiction of the topology instead of two.

Adaptability

Finally, the necessary adaptations to the existing application for each hypothetical case are summarized in Table 5.1. The table depicts that all three scenarios conform to the set criteria. All minor changes to the requirements context were incorporable with the existing application by introducing or changing at most two components. Additionally, the adaptations required either no changes to the topology or only small, localized changes. Incidentally, these scenarios required no changes to the components interfacing with the changed or introduced components.

Summary	Components		Topology changes	
	new	changed	none	only local
Nuancer local	0	1		✓
New sensor encoding	2	0		✓
Alert ECL drop	0	1	✓	

Table 5.1: changes required per adaptation scenario

5.5 Conclusion & Discussion

This chapter will be concluded by contemplating the outcomes. Firstly, the conclusions drawn from the performed study will be stated. Secondly, the validity of the study and therefore the conclusions drawn from it will be discussed. This chapter will be concluded by deliberating the limitations of this preliminary validation study.

5.5.1 Conclusions

The main conclusion to draw from this initial validation study is that it indicates the development platform to be a practical tool to develop a functional WSN monitoring application. The distributed application architecture provides a functional separation of concerns and the provided component scaffolding provides curtailment of most types of data streams and distributions. Secondly, the explicit Resource Distribution Model provides a useful exposition of how resources within a system are interconnected, calculated and utilized. Additionally, the explicit nature of the model allows unknown variables to be computed in accordance with the model's constraints and optimal behaviour. This study has shown that, for the purpose of the Nedap Identification Systems SENSIT application, the monitoring solution can be constructed within the set parameters for required development effort, with the exception of the required implementation of component's internals. Additionally, the provided capability for separation of concern allows for rapid software evolution with respect to minor changes to the monitoring application's requirements or context. There are however some small deficiencies and issues to be solved in order to also make the platform more practicable.

The first main issue to be resolved is the inclusion of functionality to buffer snapshots grouped by some parameter(s) of those snapshots. The second issue regards the inclusion of structs (POJOs) used to communicate between components. These structs can be automatically serialized and deserialized and they increase the traceability of data points between components. This will reduce the code and time required for development. It might be argued that these structs themselves will introduce new code to the application. However, these objects are easily generated by conventional code generators. This approach will therefore reduce the overall development effort required. As the components will no longer be disjunct, but coupled by these objects, it will reduce the time spent debugging the application significantly.

Secondly, the inclusion of a graphical model/topology editor will remove the disjoint between graphical design documents and actual implementation. This will further reduce the development effort as a developer is no longer required

Characteristic	Value
Message payload	<256 Bytes
data rate	±1.6 KiB/day/node[2]
node lifetime	>10 years
node costs	<5 USD
Network infrastructure	Star topology (cellular)

Table 5.2: Characteristics of typical LPWA WSN applications

to transition constantly between two representations of the developed artefacts.

5.5.2 Discussion

To solidify the validity of this study, some contending issues must be addressed.

Representativeness of the SENSIT application

The first issue of contention is the applicability of the study. For any assertion to be relevant to the field of LPWA WSN it must be demonstrated that the SENSIT application is representative and conforms to the characteristics for LPWA WSN applications. Table 5.2 lists the typical LPWA WSN device characteristics, as reported by multiple sources [2, 13, 77, 85, 86, 87].

From the table summation and the application parameters stated in Section 5.1.1 it is concluded that the SENSIT application conforms to the typical features of LPWA WSN applications. Intuitively, the node costs and lifetime, 5 USD and 10 years respectively, match the parameters typifying LPWA applications. Additionally, SENSIT's new NB-IoT network technology features the typical cellular star topology. More importantly, the LPWA data signatures encompass the data signatures featured by the SENSIT application. The 128 Bytes per message are well contained within the typical maximum of 256 Bytes. Finally, supposing a message rate of 15 message per day and a payload of 128 Bytes per message yields a daily average per sensor data rate of about 1.9 KiB. Though the actual daily message rate of a node can vary wildly, as do the general bounds for individual network technologies, the averaged rate conforms to the approximate per sensor data rate typical of LPWA WSN applications.

Threat of over-abstraction

As mentioned, the current state of the development platform features some deficiencies. Should these deficiencies be absolved and the new functions provided, the level of abstraction is raised. Therefore, it must be ensured that the level of abstraction is not raised to the point that the applicability claim (sub-claim 1) is invalidated. For the inclusion of a *MappedBufferedProcessor* this concern is trivial as it provides an abstraction but, as it is extends to the platform, it does not obfuscate any underlying functionality. In selecting or implementing a serialization mechanism, note should be taken that it can transform every innate or user-specified datatype. Provided this concern is considered, a higher level of

[2] Objective message/data rate bounds are difficult to obtain, since different technologies prioritize varying limiting factors (message rate, data rate, energy consumption, etc.)

abstraction is provided, but no functionality is lost. Finally, the to be included graphical modelling/development interface should allow definition, specification and interconnectivity between all components provided by the platform. To this end, it is urged that the graphical interface is included in the platform instead of developed alongside the platform as a separate project. Separate project development will inherently lead to the development of the graphical interface trailing the development of the main platform and possible diverging of goals and requirements. If curtailment of all the above mentioned concerns is guaranteed, the level of abstraction can be raised to an appropriate level while safeguarding the applicability claim.

Developer skill level

A final point of contention regarding the validity of this study is the subjectivity of the executor. The experiment was performed by a subject with full knowledge of the internals of the development platform. Though this allows for rapid development and exploration of the capabilities of the platform, it possibly undermines the conclusions made on required development effort. Reason for this is that the actual subject may be over-qualified with regard to a representative developer of a QoS monitoring application. Therefore, care must be taken that the general development effort is not underestimated. The likelihood of such an underestimation will be deliberated in this section.

Firstly, the construction of Resource Distribution Models will be deliberated. Though this study does not assert bold claims regarding the required effort of constructing such models, the relative impact of a reduced skill level to the effort required can be predicted and discussed. Though a model instantiation may seem daunting, it is actually constructed using only a few concepts. A model consists of *Resources* and *Components* computing, consuming and producing these resources. Subsequently, components are connected to resources by an interface of type *Calculates*, *Consumes* or *Produces*. The only issue complicating this depiction is the *ModelledComponent*, which contains multiple utilization models with a resource interface for each resource interfaced by the component. However, these interfaces are instantiated and act equal to the regular component-resource interfaces. Therefore, understanding of one carries over to the other. Finally, specifying the intended model may prove challenging to less familiar developers. This is due to the nature of the formula specification of resource interfaces. These formulas are very formalized to enable automated computation and evaluation of instantiations. These interface formulas take an array as input containing all input values required to compute its output. Consequently, a list of resource identifiers is provided to the function, specifying the resources to be inserted at each index of the input array. In doing so it provides a compact specification for these formulas. However, it also allows for construction of invalid, incalculable or semantically incorrect models. Therefore, clear and indubious instructions will be provided to guide future developers.

Continuing, the consequences to the application topology are considered. Firstly, the internals of the topology components are plain Java code. Therefore, the level of familiarity has a negligible effect to implementation of the internals. Secondly, the suggested introduction of a formalized and automated (de)serialization mechanism will only aid an uninformed developer, since it provides a clear handle to the implementer, obfuscating the cumbersome details of the underlying communication platform. Additionally, the construction of the application topology with the provided builder was concluded to be specifiable by four instructions per topology component. The skill level of the application developer/designer has no impact to this required number of instructions, since the provided *TopologyBuilder* contains no actions aside these four instructions for a component: create component, declare component, subscribe to channels, declare as producer to channels.

Finally, it is argued that an unskilled implementer will gain more from the platform then the acquainted subject which performed this study. This is asserted due to the limited number of component types that require understanding. The platform only features three different types of components, with at most two variations per component (e.g. distributed/local computation or database/local buffer). Additionally, the scaffolding provided will help developers in specifying more complicated components. For example, the *DatabaseBufferedComponent* requires implementations for abstracted methods that subsequently *add to*, *fetch* and *filter* the buffer managed by the database. This sequence specification guides a developer in implementing the intended behaviour of the buffer. Therefore, it is argued that a less skilled developer will gain more benefit from the platform, relative to his/her skill level.

5.5.3 Limitations and recommendations

Though this validation study demonstrates the platform to be a useful tool, it must be regarded as a proof-of-concept. This study only regarded one sensor application and therefore the results might be accidental and the evidence provided by them is anecdotal. Though the preliminary results do indicate the platform to be a useful tool for WSN QoS monitoring, general statements are not allowed to be asserted unequivocally regarding the general applicability of this tool to the field of WSN applications. For such conclusions to be asserted, much more validation on a more varied base of applications is required.

A second shortcoming of this study is that the SENSIT wireless sensor application did not feature the complex cases to fully explore the capabilities of the Resource Distribution Model. Previous chapters have claimed that the Resource Distribution Model should be applicable at multiple stages of information processing (e.g. sensor, per cell, entire application). However, as mentioned before, there was no case for post-accumulation processing or sensor configuration based on application-level parameters. Therefore, no RDM was employed in the latter stage of information processing. Therefore, the model is claimed to be applicable at sensor level for the SENSIT application. In order to assert the model as a general solution, more research should be performed on sensor applications that do feature more complex application-level processing or configuration.

Procedurally, this study also features a large limitation and therefore so do the conclusions drawn from it. The limitation to the study is that it was not designed as a blind study. As the application instantiation of the platform was developed by a developer with full knowledge of the validation criteria and intimate knowledge of the internals of the development platform. In order to fully and objectively assert the conclusions of this study the experiment must be repeated more formally with impartial subjects. These subjects must be able to repeat the experiment's process without knowledge of the parameters of the study, without familiarity of the platforms internals and only the provided documentation of the platform and its exposed APIs.

However, this eventual full-scale study should not be performed until the latter stages of platform development and validation. The reason for this is that it is far more resource-efficient to discover initial deficiencies and issues with small case studies, as described in this chapter. Only when these studies no longer yield suggested improvements to the platform should the scope be focussed towards more expensive, formalized studies.

6. Conclusion & Discussion

In this final chapter the thesis will be concluded. This will be performed by first revisiting the research questions and answering them as well as permitted by the results of the study. Subsequently, some remaining issues will be debated. Finally, some areas of exploration will be suggested for continued research.

6.1 Conclusions

This section will endeavour to resolve the research questions posed in Section 1.4.1 in order, each under its own heading. The first five of which, regarding the development platform detailed in Chapter 3, will be answered in the next subsection. The remaining three, regarding Resource Distribution Model of Chapter 4, will be resolved in the following.

6.1.1 Platform architecture

Stream transformation types

From the analyses posed in Section 2.1 of the background and the commonality/variability analysis of Chapter 3, it is derived that in broad terms the input for the QoS determination process is a high influx of low-level, raw data describing the condition and performance of sensor devices. From this data a number of high-information output parameters requires derivation in order to cause concrete effects. From this proposition it is firstly concluded that the input data is transformed in order to enrich the data from raw to higher-level information.

Secondly, the data is aggregated to further raise the level of information and increase the accuracy of the information. This aggregation can be performed across two dimensions: laterally or longitudinally. Lateral aggregation entails collecting similar data obtained from different sources to determine high-level information of a state across a large domain (e.g. geographically). Conversely, longitudinal aggregation encompasses data from a single source (a device or intermediary process), but which is collected over a period of time. Such analyses can be used to infer higher-level information such as trends or to improve the accuracy and confidence in measured or computed parameters.

The specified processes can have one or some snapshots as output, whereby *some* is defined as a countable, fixed amount which does not increase as the number of inputs increases. To simplify the abstraction, *one* will be included into *some*. The preceding concludes two types of information processing streams: one-to-some (transformation) and many-to-some (aggregation).

The preceding discussion ignores two types of processing streams. The first of which is the many-to-many relation. It is omitted because it can be simulated by parallelized one-to-one processes. It therefore only serves as an abstraction of the actual processing. The second stream type omitted is the one-to-many. Since the information-potential of data cannot be increased by splitting it apart, it will only produce copies of the information. Because only actual information processing streams are of interest, such copying has also been disregarded.

Platform design

The classified types of data streams are accounted for in the development platform by providing a micro-component architecture. This platform allows for the specification of processors which communicate with one another through the Apache Storm platform. The developed platform scaffolds processors for the identified data streams and provided builders enable rapid development of application topologies. There are three chief types of processors, however variants exist[3]. These chief processor types are:

SingleMessageProcessor
: Takes a single snapshot and emits one or some snapshots.

AccumulatorProcessor
: Takes a large amount of laterally related snapshots emitted by many sources and computes some high-information snapshot(s).

BufferedProcessor
: Takes a sequence of longitudinally related snapshots and averages them to attain trend information or increase the accuracy of the measurements.

These components are abstract instantiations of the general Apache Storm *Bolt* object. This allows them to easily be integrated into a Storm topology while providing convenient abstract scaffolds that aid application developers.

Level of abstraction

Chapter 5 evaluated the level of abstraction of the platform on three concepts: applicability, development effort and adaptability. As a prototype monitoring application for Nedap Identification Systems was able to be designed and developed, it was concluded that the applicability was sufficient, at least for this preliminary validation study. The adaptability of the platform was also demonstrated to be sufficient by efficiently devising three hypothetical adaptation to the developed system.

However, it was shown that the platform was too low-level, as it did not provide a convenient communication mechanism. Instead, the mechanism relied too heavily on the innate key-value messaging system of Apache Storm. Consequently, this required repeated parsing and casting, and did not provide easy access to data points within snapshot messages. This can be alleviated by introducing struct-based messaging (POJOs). This would eliminate the need for casting and provides hard-typed bindings for data points in messages. This would eliminate many points of failure and improve the time required for development and debugging.

[3]local vs. external storage or local vs. distributed computation

Threats to scalability of capacity

From the identified stream traffic types, two threats to the scalability of input capacity are identified. Firstly, should the amount of input devices increase, every task in the application must be performed more and more often. This will eventually approach the computational and memory limitations of the hardware executing the application. Therefore, tasks must be able to be distributed among servers to ensure parallelized execution. If disjunct computations can be parallelized individually, there is still a second issue. This issue is that aggregation must eventually be performed centralized in order to emit an aggregated snapshot. Therefore, it must be provided that aggregation can be executed distributively until the data volume has been reduced sufficiently for a single machine to finish the aggregation.

The developed monitoring platform attempts to account for these challenges. However, due to compatibility issues with the chosen cloud execution platform, no capacity tests or benchmarks could be performed to confirm the scalability. Therefore, the scalability claims will be defended theoretically in Section 6.2: Discussion.

6.1.2 Resource Distribution Model

Key concepts of QoS modelling

The model was conceived to capture the key concepts in modelling QoS. The commonality/variability analysis of Chapter 4 determined these concepts to be interconnected resource parameters which eventually determine some resource(s) indicative of the degree of QoS provided by the system. This is represented in the model as Resources interconnected by Components. These Components determine how one resource is converted into another. Meanwhile, the Resource object enables the portrayal of the multiplicity of a shared resource.

Modelling variable behaviour

By abstracting the conversion of resources into components, the modelling of variable behaviour is also facilitated. This is achieved by equipping some components with multiple Resource Utilization Models (RUM). These models can be interchanged to analyse differing modes of operation and evaluate the implications this has on the state, validity and performance of the system.

Calculating optimal behaviour

The final research question regarding the RDM inquires how the optimal behaviour of the system, considering the current state of the system, can be determined. For this purpose the constraint programming paradigm has been employed. Since the model solution essentially features a constraint model with some entities with variable behaviour, constraint solving is tremendously applicable. The provided back-track model solver iteratively attempts to assign components with RUM's and systematically searches for valid model compositions. All valid compositions are subsequently ranked according to the QoS they provide and the optimal solution is chosen. In order to objectively compare the QoS provided by solutions it was decided that a model should have one

— and only one — optimizable QoS parameter. If a model features multiple QoS parameter, this is achieved by applying an all-encompassing heuristic QoS function to those parameters, netting a universal QoS indicator.

6.2 Discussion

This section will discuss some remaining questions that might have been raised by this thesis.

Is the platform as scalable as proposed?

As mentioned, due to compatibility issues with the chosen cloud infrastructure, no tests could be performed validating the scalable capacity of the platform. However, the scalable capacity can be hypothesized by regarding the features of the supporting technologies. The requirements for scalability, as identified in Section 6.1, are:

1. disjunct computations can be parallelized individually, and
2. aggregation can be performed distributively, at least up until a point where the data volume is reduced sufficiently for a single processor.

The first demand is innately present in Apache Storm. A bolt can be executed by multiple tasks on multiple workers. This entails that if the processes are completely disjunct, scalability is attained by assigning more parallel workers to the process. Furthermore, by employing a field grouping it can be assured that similar snapshots are always processed by the same worker, which can ensure dependable parallel execution of stateful processes.

For aggregation the platform also enables scalability. Firstly, as the DistributedAccumulatorProcessor is implemented as an Apache Spark Streaming application, it enjoys the scalability guarantees offered by Spark. Furthermore, even the regular AccumulatorProcessor can be composed in such a manner that it first accumulates partitions of the input set, before accumulating those intermediary results. This can be achieved by subsequent map-reduce steps, as is provided for by the AccumulatorProcessor. For the BufferedProcessor scalability is less of an issue since it receives its input from a single source. Therefore, issues only arise when that source increases its emission tremendously. However, should such an issue arise, the performance of the processor can be increased by keeping an internal state aside from its buffer. Incoming snapshots are "added" to this state and out-of-scope snapshots are "subtracted" from it, which eliminates repeated scanning of the entire buffer.

Does Apache Storm need another scaffolding layer?

The the platform was conceived in a specific top-down order:

1. conceptualized the problem domain,
2. decided on micro-component architecture,
3. specified the required micro-components,
4. implemented the components, and finally

5. integrated the components with Apache Storm.

Originally, the search for a supporting technology was mainly for its core messaging system and execution environment. Therefore, as a byproduct of this approach, some advanced features of Apache Storm had been overlooked. As a consequence, the scaffolding layer provided by the platform is very close to the Storm functionality.

One Storm feature that approximates the added functionality of the platform is stream windowing. This considers a range of input messages of a certain length or duration. This window is subsequently moved and input in the window is supplied to a processor. This could provide for the base functionality of the BufferedProcessor. However, the scaffolding provided for the BufferedProcessor enables context aware control over the buffer, since the processor can inspect the entire buffer when pruning values. In contrast, the windowing of Storm can only prune values based on the timestamp or buffer length. Additionally, the windowing of Storm keeps the window in-memory, which becomes an issue for high influx processors or long windows. Therefore, the DistributedAccumulatorProcessor and the DatabaseBufferedProcessor attempt to resolve this by employing Apache Kafka/Spark and databases respectively.

Finally, it might be argued that if the additions of the platform are as beneficial as is claimed, they would have been integrated into Apache Storm already. However, This is contradicted by the fact that for this application an explicit scope has been ascertained. Firstly, the platform was designed with a focus on calculating and monitoring QoS of WSN applications specifically, whereas Apache Storm is devised for streaming applications in general. Secondly, research regarding the first research question has yielded a specific taxonomy of the stream and processing types that should be regarded. Finally, a specific implementation language was chosen for the platform: Java. This allows the platform to profit from certain language specific benefits that are disregarded in Apache Storm to become language-independent. To summarize, by regarding domain knowledge a more specific adaptation of Apache Storm was able to be constructed.

Why aren't Apache Storm's fault tolerant measures incorporated?

The platform enjoys the innate service fault tolerance of Apache Storm, i.e. if a service fails it will be automatically restarted. However, Storm also features methods to (partially) ensure fault tolerance on a data level. These features are not present in the developed platform due to the relatively late decision for Apache storm, as prescribed in the previous subsection. These measures and the impact of their deficiency will be deliberated shortly.

The first of these methods is message acknowledgement. Storm tracks messages throughout the topology. This allows messages to be replayed at the spout if processing fails somewhere in the topology. While this is a powerful function, there are some considerations to be made regarding it. Firstly, employing this will definitely result in non-sequential streams. This is caused when a windowed/buffered processor fails processing and every message in its window/buffer is replayed when the time-out occurs. Though any WSN streaming application should arrange for incidental out-of-sequence messages, a failing windowed/buffered processors causes a burst of out-of-sequence messages. The possibility of replays also changes the conditions of the message delivery system. Whereas originally it guaranteed at-most-once processing, with replays it guarantees at-least-once processing with no upper limit to the number of replays. This is exacerbated by the fact that in LPWAs the raw data is emitted by a fallible technology [25]. Most LPWA applications employ a best-effort delivery guarantee to back-end applications that does not account for messages dropped in the network [20]. Therefore, formally the entire application will feature no processing guarantees, as a message can by processed anywhere between zero and many times.

Another measure is stateful processors and checkpointing. Storm allows processors that keep an internal state and persist that state to remote storage periodically. Then if the processor fails, its state can be recovered. This could be used for a stateful variant of the AccumulatorProcessor. However, for the BufferedProcessor to persist its state may become very data-intensive. The reason for this is that, even if it would keep an internal state, it must keep a list of in-scope input messages. It requires this list in order to "subtract" snapshots from its internal state when they become out-of-scope. If that list is very large, periodically persisting it to remote storage may become a problem.

It is important to note that the preceding considerations do not invalidate acking and checkpointing. However, it does present that these measures alone do not guarantee fault tolerance. In order to assert such guarantees, careful considerations must be made regarding the application's topology and implementation. However, even with close consideration 100% data fault tolerance may not be attainable. These features should be incorporated into the platform eventually. However, for the above-mentioned reasons no priority has been given to it yet.

Applicable field of applications

The final issue that will be addressed is the general applicability of the platform. The goal of this study was to design and devise a general platform that would enable the development of a QoS monitoring and management application for LPWA WSN applications. Though a concept platform has been developed, its proof-of-concept validation was only performed on one application. It would therefore be an overstatement to assert the platform's general applicability to LPWA applications based solely on the validation study.

Instead, this assertion is based on the analysis of types of data streams and reductions present in LPWA QoS determination. It was determined that these streams can be categorized as one-to-some, lateral many-to-some and longitudinal many-to-one. Furthermore, a system (at an abstract level) was identified to consist of correlated resources. How these resources are interconnected and calculated may depend on the current operation strategy of the system under investigation. Additionally, given the resource parameters of a system, an operational strategy may or may not satisfy the specified resource constraints. Finally, the performance of a system under certain operational strategies can be compared according to some resource parameters.

These concepts were all present in the case to which the platform was applied. Furthermore, all these concepts could be captivated and processed by the platform implementation. Therefore, it is stated that, under the assumption that the aforementioned concepts are the key identifying features of LPWA

WSN application QoS, the developed platform is a viable solution for LPWA WSN QoS monitoring and management. Whether the presumed concepts are indeed the key identifying features for this class of applications requires further investigation.

6.3 Future work

Though the platform appears promising, there is further work to be done. In this conclusive section some envisioned areas that require further exploration will be suggested.

Obtaining accurate Resource Distribution Models

The proof-of-concept study has shown that the RDM is a powerful tool to calculate the state and performance of a system based on some input measurements. However, in order to perform these calculations, an instantiation of the model must first be realized. This can only be done if all the relations, formulas and adjustable behaviour required to model the behaviour are known. Therefore, even though the model is powerful, obtaining an accurate realization of it can prove laborious. Therefore, efforts should be made to research how these hypothetical models can be extracted from genuine systems. Some research areas of interest would be extraction through formal statistical analysis tools or machine learning. Examples of such approaches using formal tools has been presented in the works of S. te Brinke et al [88, 89], which attempt to extract fine-grained, explicit models (state machines) from actual software and hardware components. It is suggested that efforts are made to investigate the application of such approaches to extract the more coarse models employed by the developed platform.

Suggested improvements

Chapter 5 identified some deficiencies in the abstraction of the development platform. The first of which is the introduction of a strongly typed messaging system. This is required to obfuscate the cumbersome serialization that is required in distributed systems. By introducing such a scheme it becomes vastly easier to precisely and adequately access a data point in a snapshot. The second feature to introduce is a visual representation and editor of the concepts of the platform. This goes for both the Resource Distribution Model and the platform topology builder. Such a GUI will integrate the visual representation of the design process and the programmed representation of the functional artefact into a single visual and functional entity.

Better incorporation of Apache Storm

A method that can verify the implementation of the platform's scaffolding layer is by reperforming a part of this study with a slightly different methodology. The proposed platform architecture was conceived by a method that subsequently:

1. conceptualized the problem domain,
2. decided on micro-component architecture,

3. specified the required micro-components,
4. implemented the components, and finally
5. integrated the components with Apache Storm.

An alternate approach could be to retain the conceptualization of the domain, decisions for a micro-component architecture, functional specification of the micro-components and choice for Apache Storm (steps 1, 2, 3 & 5). However, instead of building the components first and then incorporating a Storm's messaging system, a more bottom-up implementation approach can be employed. This approach would consider the advanced features provided by Apache Storm and build the components upon them. The predicted outcome for this study is a leaner scaffolding layer that better enables the advanced features offered by Storm.

Another possible outcome for this suggested study is that Apache Storm does not require a scaffolding layer to better enable the development of an LPWA QoS monitoring application. Such a conclusion does not trivialize this study however. For this case, the assumption is that a solely Storm-based approach can simulate the features of the developed platform. Then, by the transitive property, the conclusions of the performed validation study also hold for this Storm-based approach. However, it is presupposed that such a conclusion is highly unlikely, since the improvements suggested by the validation study (POJO messaging system, graphical topology editor) are also lacking in Apache Storm.

Further validation

The final recommendation for continued research is to solidify the claims of the validation study by reperforming it with slight alterations. For this continued research three sub-directions are identified.

Firstly, the validation study can be performed on a wider base of LPWA WSN applications. Preferably, this would be performed after the known deficiencies are absolved. Broadening the scope of applications cements the claim that the platform is an applicable development platform for LPWA QoS monitoring in general. Alternatively, it allows for more deficiencies to surface.

The proof-of-concept study shows the conceived Resource Distribution Model as a functional solution. It was shown to captivate the resource distribution of a micro-scale system (i.e. sensor device). Additionally, it provided for an automated mechanism of determining the optimal behaviour of the modelled configurable system. Furthermore, by employing constraint solver paradigms the valid instantiations of the system's behaviour can efficiently be determined. After which, the optimal operational strategy can be calculated. Though the model has shown to be practical at micro-level, the validation case did not feature the complexity that required high-level modelling (e.g. groups of devices or whole application) or convoluted configurable behaviour (i.e. multiple components with varying behaviour). Therefore, the study needs to be performed on WSN applications that do feature more complicated systems to be modelled by the Resource Distribution Model. This entails both models with more convoluted configurable behaviour and applications that require higher-level modelling of its QoS parameters. Such studies should establish the (un)necessity or (in)feasibility of the model at such levels.

The final area of continued research is to have the validation study be carried out by software engineers with limited familiarity of the proposed development platform. Such blind studies should give more objective insight into the usability of the platform with regards to the general population of software developers.

Bibliography

[1] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015.

[2] R. Ratasuk, N. Mangalvedhe, Y. Zhang, M. Robert, and J.-P. Koskinen, "Overview of narrowband IoT in LTE Rel-13," in *Standards for Communications and Networking (CSCN), 2016 IEEE Conference on*, pp. 1–7, IEEE, 2016.

[3] "LoRa Alliance." `https://www.lora-alliance.org/`. [website] accessed 10 April 2018.

[4] "SigFox." `https://www.sigfox.com/`. [website] accessed 10 April 2018.

[5] N. Al-Falahy and O. Y. Alani, "Technologies for 5g networks: Challenges and opportunities," *IT Professional*, vol. 19, no. 1, pp. 12–20, 2017.

[6] M. H. Alsharif and R. Nordin, "Evolution towards fifth generation (5g) wireless networks: Current trends and challenges in the deployment of millimetre wave, massive mimo, and small cells," *Telecommunication Systems*, vol. 64, no. 4, pp. 617–637, 2017.

[7] J. Rivera and R. van der Meulen, "Gartner says the internet of things will transform the data center," *Retrieved August*, vol. 5, p. 2014, 2014.

[8] L. Columbus, "Roundup Of Internet Of Things Forecasts And Market Estimates, 2016." `https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016/#490e967e292d`, 2016. Forbes, [online] accessed 10 April 2018.

[9] S. Lucero *et al.*, "IoT platforms: Enabling the Internet of Things," *IHS Technology white paper*, 2016.

[10] "Ericsson Mobility Report: june 2016," *Ericsson white paper*, 2015.

[11] G. White, V. Nallur, and S. Clarke, "Quality of service approaches in iot: A systematic mapping," *Journal of Systems and Software*, vol. 132, pp. 186–203, 2017.

[12] E. Borgia, "The Internet of Things vision: Key features, applications and open issues," *Computer Communications*, vol. 54, pp. 1–31, 2014.

[13] "Narrowband IoT: The Game Changer for the Internet of Things," *T-Mobile white paper*, 2017.

[14] "Narrowband-IoT: pushing the boundries of IoT," *Vodafone white paper*, 2017.

[15] R. S. Sinha, Y. Wei, and S.-H. Hwang, "A survey on LPWA technology: LoRa and NB-IoT," *ICT Express*, vol. 3, no. 1, pp. 14–21, 2017.

[16] S. Tarannum, "Energy conservation challenges in wireless sensor networks: A comprehensive study," *Wireless Sensor Network*, vol. 2, no. 06, p. 483, 2010.

[17] R. C. Carrano, D. Passos, L. C. Magalhaes, and C. V. Albuquerque, "Survey and taxonomy of duty cycling mechanisms in wireless sensor networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 181–194, 2013.

[18] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[19] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, ACM, 2012.

[20] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005.

[21] L. Wang and R. Ranjan, "Processing distributed internet of things data in clouds," *IEEE Cloud Computing*, vol. 2, no. 1, pp. 76–80, 2015.

[22] F. Samie, L. Bauer, and J. Henkel, "IoT technologies for embedded computing: A survey," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, p. 8, ACM, 2016.

[23] D. Chen and P. K. Varshney, "QoS Support in Wireless Sensor Networks: A Survey.," in *International conference on wireless networks*, vol. 233, pp. 1–7, 2004.

[24] "Getting ready for IoT monitoring." `ttps://www.iot-now.com/2016/11/17/55128-getting-ready-for-iot-monitoring/`. [website] accessed 14 may 2019.

[25] W. Wang, S. De, R. Toenjes, E. Reetz, and K. Moessner, "A comprehensive ontology for knowledge representation in the internet of things," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pp. 1793–1798, IEEE, 2012.

[26] G. Cicotti, L. Coppolino, R. Cristaldi, S. D'Antonio, and L. Romano, "Qos monitoring in a cloud services environment: the srt-15 approach," in *European Conference on Parallel Processing*, pp. 15–24, Springer, 2011.

[27] W. Stallings, *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[28] J. Gozdecki, A. Jajszczyk, and R. Stankiewicz, "Quality of service terminology in IP networks," *IEEE communications magazine*, vol. 41, no. 3, pp. 153–159, 2003.

[29] R. Duan, X. Chen, and T. Xing, "A QoS architecture for IOT," in *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pp. 717–720, IEEE, 2011.

[30] "PTC ThingWorx." `https://www.ptc.com/en/products/iot/thingworx-platform`. [website] accessed 1 may 2019.

[31] B. Nakhuva and T. Champaneria, "Study of various internet of things platforms," *International Journal of Computer Science & Engineering Survey*, vol. 6, no. 6, pp. 61–74, 2015.

[32] P. P. Ray, "A survey of IoT cloud platforms," *Future Computing and Informatics Journal*, vol. 1, no. 1-2, pp. 35–46, 2016.

[33] "Control Center for NB-IoT." `https://www.jasper.com/control-center-for-nb-iot`. [online], accessed 1 may 2019.

[34] "The Forrester Wave: IoT Software Platforms, Q4 2016," *Forrester white paper*, 2016.

[35] "Nimbits." `https://www.nimbits.com/`. [website] accessed 1 may 2019.

[36] "Nimbits MQTT server." `https://github.com/nimbits-admin/nimbits-io/wiki/Server-Setup:-Installing-Mosquitto-MQTT-Server`. [online] accessed 19 April 2018.

[37] H. Lindholm-Ventola and S. Bilhanan, "CoAP-SNMP Interworking IoT Scenarios," *Tampere University of Technology, Department of Pervasive Computing. Report*, vol. 3, 2014.

[38] H. Hui-Ping, X. Shi-De, and M. Xiang-Yin, "Applying snmp technology to manage the sensors in internet of things," *Open Cybernetics & Systemics Journal*, vol. 9, pp. 1019–1024, 2015.

[39] M. Savić, "Bridging the SNMP gap: Simple network monitoring the internet of things," *Facta universitatis-series: Electronics and Energetics*, vol. 29, no. 3, pp. 475–487, 2016.

[40] M. Amadeo, C. Campolo, J. Quevedo, D. Corujo, A. Molinaro, A. Iera, R. L. Aguiar, and A. V. Vasilakos, "Information-centric networking for the internet of things: challenges and opportunities," *IEEE Network*, vol. 30, no. 2, pp. 92–100, 2016.

[41] B. Bhuyan, H. K. D. Sarma, N. Sarma, A. Kar, and R. Mall, "Quality of service (qos) provisions in wireless sensor networks and related challenges," *Wireless Sensor Network*, vol. 2, no. 11, p. 861, 2010.

[42] J. Domzal, R. Wójcik, and A. Jajszczyk, "Qos-aware net neutrality," in *2009 First International Conference on Evolving Internet*, pp. 147–152, IEEE, 2009.

[43] M. Zdravković, M. Trajanović, J. Sarraipa, R. Jardim-Gonçalves, M. Lezoche, A. Aubry, and H. Panetto, "Survey of Internet-of-Things platforms," in *6th International Conference on Information Society and Techology, ICIST 2016*, vol. 1, pp. 216–220, 2016.

[44] O. Said, "Performance evaluation of wsn management system for qos guarantee," *EURASIP Journal on Wireless Communications and Networking*, vol. 2015, no. 1, p. 220, 2015.

[45] A. Moraru and D. Mladenić, "A framework for semantic enrichment of sensor data," *Journal of computing and information technology*, vol. 20, no. 3, pp. 167–173, 2012.

[46] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, "Wrangler: Interactive visual specification of data transformation scripts," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3363–3372, ACM, 2011.

[47] T. Dasu and T. Johnson, *Exploratory data mining and data cleaning*, vol. 479. John Wiley & Sons, 2003.

[48] I. Alexander and S. Robertson, "Understanding project sociology by modeling stakeholders," *IEEE Software*, vol. 21, no. 1, pp. 23–27, 2004.

[49] H. Arasteh, V. Hosseinnezhad, V. Loia, A. Tommasetti, O. Troisi, M. Shafie-Khah, and P. Siano, "Iot-based smart cities: a survey," in *2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)*, pp. 1–6, IEEE, 2016.

[50] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *Journal of Industrial Information Integration*, vol. 6, pp. 1–10, 2017.

[51] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and variability in software engineering," *IEEE software*, vol. 15, no. 6, pp. 37–45, 1998.

[52] E. W. Dijkstra, "Essays on the nature and role of mathematical elegance (1)," 1977.

[53] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and ulterior software engineering*, pp. 195–216, Springer, 2017.

[54] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "On the integration of cloud computing and internet of things," in *2014 International Conference on Future Internet of Things and Cloud*, pp. 23–30, IEEE, 2014.

[55] "RabbitMQ." `https://www.rabbitmq.com/`. [website] accessed 11 may 2019.

[56] "Apache Kafka." https://kafka.apache.org/. [website] accessed 11 may 2019.

[57] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[58] "Hadoop MapReduce deep diving and tuning." https://www.todaysoftmag.com/article/1358/hadoop-mapreduce-deep-diving-and-tuning. [website] accessed 11 may 2019.

[59] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.

[60] "Apache Spark." https://spark.apache.org/. [website] accessed 11 may 2019.

[61] "Java API for Stream interface." https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html. [website] accessed 11 may 2019.

[62] "Apache Spark: RDD Programming Guide." https://spark.apache.org/docs/latest/rdd-programming-guide.html. [website] accessed 11 may 2019.

[63] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[64] "Apache Spark Streaming." https://spark.apache.org/streaming/. [website] accessed 11 may 2019.

[65] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pp. 423–438, ACM, 2013.

[66] V. Sachidananda, A. Khelil, and N. Suri, "Quality of information in wireless sensor networks: A survey," *ICIQ (to appear)*, 2010.

[67] R. Dechter and D. Frost, "Backtracking algorithms for constraint satisfaction problems," tech. rep., Technical Report, 1999.

[68] F. Van Harmelen, V. Lifschitz, and B. Porter, *Handbook of knowledge representation*, vol. 1. Elsevier, 2008.

[69] J. Jin, J. Gubbi, T. Luo, and M. Palaniswami, "Network architecture and QoS issues in the internet of things for a smart city," in *Communications and Information Technologies (ISCIT), 2012 International Symposium on*, pp. 956–961, IEEE, 2012.

[70] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pp. 227–238, ACM, 2017.

[71] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pp. 1789–1792, IEEE, 2016.

[72] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[73] H. H. Ammar, V. Cortellessa, and A. Ibrahim, "Modeling resources in a uml-based simulative environment," in *Proceedings ACS/IEEE International Conference on Computer Systems and Applications*, pp. 405–410, IEEE, 2001.

[74] C. Seceleanu, A. Vulgarakis, and P. Pettersson, "Remes: A resource model for embedded systems," in *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 84–94, IEEE, 2009.

[75] S. Malakuti, S. te Brinke, L. Bermans, and C. Bockisch, "Towards modular resource-aware applications," in *Proceedings of the 3rd international workshop on Variability & Composition*, pp. 13–18, ACM, 2012.

[76] R. Ratasuk, B. Vejlgaard, N. Mangalvedhe, and A. Ghosh, "NB-IoT system for M2M communication," in *Wireless Communications and Networking Conference (WCNC), 2016 IEEE*, pp. 1–5, IEEE, 2016.

[77] D. Sjöström, "Unlicensed and licensed low-power wide area networks: Exploring the candidates for massive iot," 2017.

[78] "Choco Solver." `http://www.choco-solver.org/`. [website] accessed 12 may 2019.

[79] "OptaPlanner." `https://www.optaplanner.org/`. [website] accessed 12 may 2019.

[80] J.-G. Fages, G. Chabert, and C. Prud'Homme, "Combining finite and continuous solvers towards a simpler solver maintenance," in *The 19th International Conference on Principles and Practice of Constraint Programming*, pp. TRICS–13, 2013.

[81] "Nedap N.V.." `https://nedap.com/`. [website] accessed 21 may 2019.

[82] "Nedap — Identification Systems." `https://nedap.com/markets/identification-systems/`. [website] accessed 21 may 2019.

[83] "Nedap SENSIT." `https://www.nedapidentification.com/nl/producten/sensit/`. [website] accessed 21 may 2019.

[84] "U-blox: SARA-N3 series." `https://www.u-blox.com/en/product/sara-n3-series`. [website] accessed 12 may 2019.

[85] "A comprehensive look at Low Power, Wide Area Networks," *LinkLabs white paper*, 2016.

[86] B. Vejlgaard, M. Lauridsen, H. Nguyen, I. Kovács, P. Mogensen, and M. Sørensen, "Coverage and Capacity Analysis of Sigfox, LoRa, GPRS, and NB-IoT," in *Proceedings of the IEEE 85th Vehicular Technology Conference, Sydney, Australia*, pp. 4–7, 2017.

[87] G. Margelis, R. Piechocki, D. Kaleshi, and P. Thomas, "Low throughput networks for the IoT: Lessons learned from industrial implementations," in *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pp. 181–186, IEEE, 2015.

[88] S. te Brinke, S. Malakuti, C. Bockisch, L. Bergmans, M. Akşit, and S. Katz, "A tool-supported approach for modular design of energy-aware software," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1206–1212, ACM, 2014.

[89] S. t. Brinke, C. Bockisch, L. Bergmans, S. Malakuti, M. Akşit, and S. Katz, "Deriving minimal models for resource utilization," in *Proceedings of the 2013 workshop on Green in/by software engineering*, pp. 15–18, ACM, 2013.