[frontpage]

February 5, 2019

Contents

1. Introduction

1.1 Domain

1.1.1 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 area for different applications made less and less sense. 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).

The recent development in IoT has manly focussed on the field of Low Power Wide Area networks (LPWA). These networks serve devices that communicate over large distances with limited computational and communication resources. 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 LWPA IoT applications.

The scientific progression in the field of IoT has in turn spaked recent commercial interests. Multiple corporations are deveoping and deploying wide area networks for low powered devices. Examples of these networks are Narrow-Band IoT[?], LoRaWAN [?] and Sigfox [?]. These networks are deployed and operated by telcom providers and allow instant connectivity by activiating a SIM or network connectivity module. As a consequence large scale LPWA applications are moving from node-hopping and mesh network strategies to operated cell networks [?]. Because of the aforementioned reasons the number of connected devices has exploded in the recent years. Estimations vary but a concensus taken from multiple sources predict about 15-30 billion connected devices in 2020. This would imply that by 2020 the number of connected IoT devices will have surpassed the number of consumer electronic devices (e.g. PC's, laptops and phones) [?].

Both the explosion of devices and 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 state of the application. The term coined for this is Quality of Service. QoS parameters such as application throughput, service availability and delivery guarantee allow the description of the state of a system or application.

1.1.2 Challenges of monitoring QoS in IoTs and WSNs

Though the concept of QoS is well understood, the exist challenges in measuring and determining QoS in WSNs.

Technical limitations

The first challenge of LPWA applications is the aforementioned extreme resource constraints. As a LPWA device is required to perform for a certain amount of time (typically at least 10 years [?]) on a finite, bounded battery energy supply, there are no resources to spare for expensive auxiliary processes. Therefore, devices usually send low-level auxiliary data, instead of intelligently derived values. The burden of calculating high level information is therefore deferred to be computed in-network (edge) or in the back-end.

Additionally, evolution of sensor device software is far more restrictive then evolution of server software. Firstly because of the long life-time of devices, it can occur that services based on modern day requirements need to be performed on decade old technology. Secondly, most LWPA networking protocols do not require devices to retain a constant connection, in order to save energy[?, ?, ?]. Instead the devices connect periodically or when an event/interrupt occurs. This entails that devices cannot be 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 this reason LPWA sensor applications often employ a *"dumb sensor, smart back-end"* philosophy. Again deferring the computations to the network or the back-end.

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 as more jobs need to be performed as more devices are added to the application.

Why IoT QoS is different

Aside from the low-level information sent by the large amout of devices, QoS in WSNs is distinctly different from classical client-server based QoS. Often QoS in a server-based applicaiton can be measured at the server. QoS may need aggegation when the service is run on a cloud environment, but even then the number of data sources is relatively limited. Large WSN applications require data aggegation by default. As the level of service provided by the application can only be assertained by calculation based on temporal auxiliary data collected from the devices. This concept is known as Collective QoS [?] and comprises parameters such as collective bandwidth, average throughput and the number of devices that require replacement. As this information eventually requires accumulation on a single machine in order to determine singular values, aggregation of expansive amounts of auxiliary sensor data must be aggregated intelligently as to not form a congestion point or single point of failure.

Alongside of collective QoS we still require device level information. If a device is not performing according to expectations of the predetermined strategy, it is required that this is notified. This introduces a second distinction to classical QoS: multi-level monitoring and reporting. Usually we are only interested in the QoS provided by the sever(s) running our application. However in a wireless sensor environment we require monitoring parameters on different levels. 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. An IoT monitoring application needs to be prepared for devices added to the network and dropping out of the application is prone to change of scale and devices are prone to failure and replacement. As a collective QoS parameter is based on a selection of devices, the monitoring application must support adding and remove devices from the equation.

Movement to operated cell network

A final challenge in contemporary QoS monitoring of LPWA applications is the earlier recognised increasing trend of commercial telecom operated cell networks. Though is makes IoT connectivity more efficient because many applications can be served by a single network infrastructure, it does pose some difficulties to 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 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, deferring the burden of aggregating QoS data entirely on 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 neccecary and timely proactive intervention is required, if the limits of the SLA are threatened to be exceeded. [**?**]

Summation

In conclusion, The tendency to defer LWPA computations challenges the computing capabilies of centralized solutions. Additionally measuring and controlling QoS in Wireless Sensor networks is [very] different from measuring and controlling QoS in resource abundant networks. Both because of the resource constraints and the fact that the QoS characteristics in WSN's is [definitively] different from the characteristics in conventional networks and applications. Finally, by [outsourcing] the responsibility of network management the control and [observability] of those networks is also lost.

[To] this purpose we will [attempt] research the applicability and design of a WSN QoS platform. This platform should address the issues of scalability and limitations of end-devices and in-network processing. It should be noted however that we will not address the issues of end-device resource restiction and network [obscuration] directly, only the [challenges] it imposes on QoS monitor and control.

1.2 State of affairs

Several platforms exist that are capable monitoring and controlling IoT applications to some degree [?]. However all are lacking in some of the 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 [?].

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 would be acceptable for a key function of an application, but not for an auxiliary monitoring processes. As to not complicate a software product which does not enjoy the main focus of development efforts it would be beneficiary to have a single platform which enables it's development.

1.3 Goal

The goal of this study is to research and develop a single development platform capable of measuring and monitoring. This platform will enable development 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.

1.3.1 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 on (auxiliary) data streams generated 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, and
- allows for the highest ease-of-implementation?

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 we model the state of a system with variable behaviour?

RQ8 How can we determine the optimal system behaviour in accordance with its state?

From the listed research questions we find a focus that is twofold. The first point of focus is the composition and development of an abstract, scalable streaming platform for IoT data enrichment. The associative 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. 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.

1.4 Approach

Before [embarking] on the [following]of this thesis, the methodology requires clarification. The remainder of this introductory chapter will be [used] to [kaderen] the context of this project. This will be performed by means of an anlysis of the context of the intended solution and its stakeholders. This will allow us to spearhead our research and design efforts.

As the above section mentioned the research questions can be divided into two categories: The platform and modelling resource distribution. Our approach is therefore to research these individually before integrating these efforts into one resulting software development platform. Each point of focus will be devised, designed and developed according to the following schedule. We will first explore the problem domain of the to be designed solution/model. This will be performed with a commonality/variability analysis (Section 2.4.1). This analysis allows us to conceptualize the problem domain which will result in a list of requirements for the solution to adhere to. With the requirements defined the state of the art of the problem domain will be explored to identify viable technologies and their deficiencies, before selecting the best applicable technologies. With these technologies identified we will adapt, design and develop the intended artifact. For design and development we will adopt the iterative development approach of Design Science Methodology[**?**] (section 2.4.2). Ultimately, the devised solution will be evaluated and discussed by paralleling them to the set requirements and some additional concepts and criteria.

Once the two compounds have been integrated into a single solution, the challenges it claims to combat will need verifying. In order to perform initial validation of the developed solution it will be applied to a real-world commercial car park WSN application developed and maintained by the Dutch company Nedap N.V.

1.4.1 Context of the project

Before discussing the research method we employ for the remainder of this thesis we will attempt to focus our efforts by scoping the project. This will be achieved by two analyses. First we will attempt to describe the set of target applications in more abstract concepts. Secondly we will focus our efforts be defining the stakeholders that stand to gain from 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 we can scope the group of applications more conceptually by specifying and parametrizing the data emitted by them and expected after processing, since this will be the input and output data for our platform and its implementations. For the purpose of scoping we will consider an implementation-agnostic of the platform as a black box. In doing so we can focus on the intended inputs and expected outputs, and their contrasts, without concerning the internals of the platform to be designed.

Firstly we have the issue of *individual information capacity.* Individual messages emitted by applications and 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 probably 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. Not only is a series of data temporally relevant, it is also related across the plain of topologically distributed sensor devices. Finally the huge amount of devices and the dynamic nature of sensor networks and IoT induces a high degree of (dynamic) scalability. Therefore any back-end application — main processing or auxiliary support — should anticipate and provide a sufficient potential for scalability. In contrast we have the expectations of the outcomes of the platform. Firstly, the platform is expected to output a relatively small amount of high-information actions, alerts and reports. The high-information consequences directly contradict the low-information capacity of individual device messages. Conversely, the moderately small number of output responses/events contradicts the immense influx of data-messages into the platform. These contradictions in turn affect the required scalability of the platform.

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)[**?**]. 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)[?]. The third method is data aggregation and reduction. This method joins and merges related datapoints accross several — and often vast amounts of — input messages to formulate a single output message containing a few datapoints, depicting some collective parameters of the domain [?]. Again the reach of this domain can be temporally, geographically, et cetera. The first two methods operate on individual data entries emitted by sensors. Hence they can be easily parallellized and are thus increadibly scalable [?]. 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*. This 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 our efforts is by identifying the stakeholders for our platform. We will perform this by analogy of the Onion Stakeholder Model as proposed by [ref][?]. This model divides stakeholders in consecutive layers, ordered by the degree of interaction and benefits received from the product. For the stakeholder division we will consider the product to be both the platform to be developed and potential future implementations of the platform. 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 al the first layer around it. However since this analysis focusses on human stakeholders, we will treat it as a single instance in our application of the model. A visual representation of the application of the onion model is given in Figure 1.1.

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, since developers of a platform instantiation interact with the scaffolding and frameworks directly provided by the core platform, we emphasize their importance 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. For our product this entails changes to the application under investigation or reports regarding the long term performance of the application to be forwarded to 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. We have identified two of these stakeholder roles. The first is the *Support and Maintenance Operator* of the application observed by the platform. If we were to analyse the stakeholders of the application under investigation, these operators would be placed in the first layer of the model. However since they do not (necessarily) directly interface with our support platform, they are placed in the second layer of the model for our product. 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 our 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 type of 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 conciently 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 dayly 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 our platform in any capacity, but can benefit heavily from it. For example, many WSN and IoT applications are deployed in smart city management and industry4.0[**?**]. 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 are relatively small, due to the huge size of the public at large this amounts to a incredible benefit.

1.5 Organisation of thesis

[TODO]

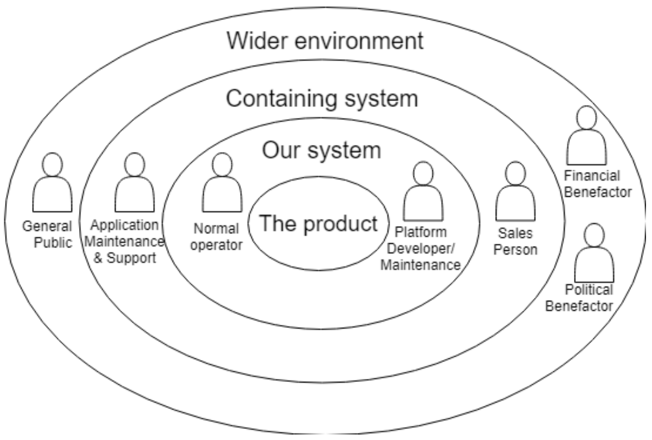


Figure 1.1: Visual depiction of application of onion stakeholder model

2. Background

2.1 Micro-service architecture

2.2 Quality of Service & Quality of Information

2.2.1 Quality of Service in WSN

Existing platforms?

2.2.2 WSN energy conservation methods

2.2.3 Quality of Information of WSN data

Aside from Quality of Service, in WSNs and IoT applications we find the concept of Quality of Information (QoI). QoI [describes] parameters depicting quality attributes of information presented by and derived from as system. It is especially [applicable] to WSNs as they present raw low-level which is then highly processed by subsequent applications. We will therefore employ the concept of QoI to validate and evaluate the processing architecture presented in chapter 3.

[?] identifies 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.

2.3 Constraint programming and solving paradigms

In chapter 4 we will visit the concept of constraint programming and 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 which conforms with the specified constraints.

An example of a problem modelled as constraint problem is an automatic sudoku solver. The model would be a list or matrix of integer variables, with each entry having a domain $\{V_i | 1 \leq V_i \leq 9\}$. The constraint we would have is $V_1 \neq V_2$ 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 our sudoku example for instance we find that if 20 values are predetermined, then our 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 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 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 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 algorthm. For example if a square in the sudoku is assigned a three, then the effect of this assingment will be propegated by pruning the number 3 from the domains of every entry in the same row, column or 3-by-3 grid. This eliminastes inconsistent options that would violate the constraints befere they would be assigned. Additionally, the concept of local inconsistency can be extended to variable domains without reequiring any assingment. For example if we have 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 since they are inconsistent with any of the values in the oppoising domain and can therefore never validate the constraint. [?, ?]

2.4 Design Methods

2.4.1 Commonality/variability analysis

In order to design for our problem domain it will require conceptualization. We will conceptualize the problem domain(s) by means of a commonality/variability analysis (C/V analysis). Whereas this analysis is [usually] performed during 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 common concepts - or invariants - that may be assumed fixed and may be depended upon and the variations in the problem domain which will need to be [captivated] and accounted for by our solution.

[?] 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.

In our conceptualization of the problem domain we will mostly focus on step 2 in which we will provide a list of common definitions, shared commonalities and variabilities. Also, in our approach we will combine steps 4 and 5 by formulating a list of requirements for our solution based on the identified commonalities and accounting for the found variabilities. As the list of requirements depends on invariants and accommodates variabilities it will allow us to design automated solutions.

2.4.2 Design Science Methodology

2.5 Example case

Throughout this [thesis] we will demonstrate our solutions by applying them to a hypothetical case. Though this case may sometimes seem oversimplified and nonsensical, it does provide an elementary example to illustrate all facets of our solutions without overcomplicating the case. This case is expressly not intended to demonstrate the capabilities or utility of our proposed solution. For that purpose, an application to a more complex real-world case will be performed in section 6.

The case we propose 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 we want devices to measure and transmit as many as possible, however since these sensors are not very powerful and employ a limited power supply (e.g. battery) the 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 the amount of electrical power a sensor can use during a given time needs to be restrained in accordance with the specification of its power source and expected life time. Finally, sensors in areas of high interest will require a shorter polling interval, as to gain the most precise information. However, 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 possible.

As for monitoring we are most interested in the measurement rate averaged over all sensors. Additionally we are required to pro-actively monitor the trend of the total bandwidth/throughput of our sensor application. Since a constant rise in data rates may ultimately violate the data consumption 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 we are required to provide the following pieces of information:

- The average polling rate, and
- whether the data rate of our sensor application rises consistently during 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 we state the following data to be provided to our 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 of time. Notice that some data points are normally inferred from raw basic data by auxiliary processes (e.g. required measurement rate). For simplification of our demonstrations we have omitted these processes and these parameters are assumed known as a message enters our monitoring application.

3. Design of IoT monitoring platform architecture

In this chapter we will explain the process taken in order to device our general platform and its architecture. We will accomplish this by first exploring the general problem domain. We will then demonstrate why existing IoT monitoring platforms do not provide the services we require. We will then deliberate the design of our proposed platform and its implementation by identifying the available supporting technologies, clarifying the adaptations made to those technologies and explaining further implementation details. We will then conclude by discussing the success, applicability, disadvantages and deficits of our proposed solution.

3.1 Goal

Large sensor applications send immense amounts of low-level raw monitoring data that requires capturing and enrichment. Individual messages of raw data might contain very little information. However, these messages contain the potential from which meaningful conclusions can be derived, either on single sensor scale or about the sensor application as a whole. This raw data is enriched by combining and analysing datasets of similar, relevant data, in order to achieve a higher level of information. The goal 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. We intend to do this by devising a generic application backbone and base building blocks for developers to extend and compose.

3.2 Conceptualization of the problem domain

In this section we will investigate the problem domain in order to eventually determine the requirements for the model. We will achieve this by performing a commonality/variability analysis (C/V analysis) of the problem domain, as described in section 2.4.1. This analysis consists of three concepts:

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

- the variations that occur between aspects of the problem domain for which we will need to account for in our proposed solution. Each point of variance needs to be accounted for in the requirements to be established.

Definitions

We will start by defining some key terms that we will use 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 external device or an internal process.

Consequence: an action taken by the platform based on the analysis of one or more snapshots.

Commonalities

With the definitions established we will continue to identify some common features shared by each application in the problem domain. These commonalities may be presumed during the design of our platform and grants focus to our efforts.

C1.1 The group of target applications involves a huge amount of sensors ([scale] which entails a high throughput of snapshots sent and requiring analysis by the platform.

C1.2 As mentioned in the definitions data is captured in snapshots. These represent the state of (a part) 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 states.

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

Variabilities

Finally we will explore the variety within our problem domain. As the purpose of our solution is to process information we will mostly focus on the variables in the domain of information. Our solution should provide proficient adaptability in order to account for this variability. We ensure this by captivating these variations in requirements.

V1.1 the first variety we encountered is the variation in Quality of Information (QoI). As described in section 2.2.3 there are many parameters characterizing the QoI of data and QoI can vary on any combination of them.

V1.2 Secondly, there is the information base on which conclusions are made. The identified conclusion bases are:

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

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

(c) Many multi-source snapshots without individual significance. E.g: while the individual throughput of sensors may be of little interest, knowledge of the average throughput 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. Though the exact implementation of consequences can never be exactly anticipated, we can identify some groups of consequences.

(a) Build a model for reporting purposes. In order to generate reports some high-level information data-points need to be calculated based on (possibly multiple sequential) 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) Analysis which invokes an immediate feedback response 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 an employee or auxiliary system.

The final variety is the scale of the application. We have already established that the platform will operate on applications of 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 the devices send data.

3.3 Requirements for the proposed software platform

In this section we will describe the requirements for the proposed platform, 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 single snapshot.

R1.3 The platform should enable processing of a limited 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:

- model building
- application feedback
- rule-based alerts

R1.6 the platform should be scalable in order to support any large amount input devices

Justification

We will conclude this section by justifying the identified requirements according to the earlier performed C/V analysis. A formal traceability between the requirements, commonalities and variability is listed in table 3.1

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

Table 3.1: traceability table for justification of requirements

The first requirements regards the definition and concepts of snapshots and is based on the commonalities and the variation in QoI. As illustrated by the traceability table the following three requirements 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 differenting them (as for V1.2), because the possible consequences are not limited to the identified consequence groups. Ultimately, the final requirement is regarding the scale of the target applications. This regards both the amount of devices in the target application as the frequency the send their snapshots.

3.4 Exploration of the solution domain

In this section we will explore the solutions and supporting technologies that are offered to us. We will first consider the base architecture and backbone of the platform, as it is the most fundamental decision to be made. We will then continue to explore the options for message brokers, as a choice for a distributed architecture almost certainly requires one. We will conclude this chapter by examining some distributed cloud computing technologies that should allow us to perform expensive computations by distributing them over a cluster, as to provide the required scalability.

3.4.1 Architecture basis and execution platform

Monolithic architecture

The first option to implement the platform is a monolithic software system. The benefit of such a system is that it keeps the solution as simple as can be. This is illustrated by a famous proverb of Dijkstra: "Simplicity is a prerequisite for reliability"[?]. 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, because code evolution becomes more difficult as more and more changes and additions are made to the code base[?]. Additionally, monolithic software systems are notoriously difficult to scale up and load balance[?], which violates requirement R1.6. Therefore we will instead adapt a micro-component approach. Micro-component are more flexible than monoliths, allow for better functional composition, are easier to maintain and much more scalable[?].

Apache Storm

Apache Storm is a big data computing library especially designed for separation of concerns. It performs distributed computing by partitioning the stages of computation. By breaking up the computation, different stages can be distributed among machines and duplicated if need be. The Storm platform consists of three chief concepts.

Spouts: nodes that introduce data in the topology,

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

Streams: connect nodes to one another and allows data to be transferred.

The computation is regarded as a directed graph with bolts as vertices, spouts as initial vertices and streams as edges.

Because data is emitted by spouts individually, Storm can achieve real-time processing of large amounts of data. By breaking up the computations into multiple consecutive bolts, Storm allows computations to be spread over a cluster. Additionally Storm allows individual bolts to be replicated and distributed. This lateral distribution prevents the occurrence of bottlenecks in the network due to bolts executing expensive pivotal processes

Storm is especially suited for our purpose since it was designed for microcomponents connected by streams. In contrast, many micro-component platforms focus on components exposing services which are explicitly invoked by other services[?, ?]. By employing Apache Storm we obtain both the distributed computation environment as the means of data distribution, simplifying our technology stack.

Conversely however, the built-in stream distribution mechanism is completely internalized, making integration with auxiliary processes difficult. Tasks such as data injection, platform monitoring and data extraction for 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, we cannot update or reconfigure a single process without restarting the entire system. 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 in tuples. This is in contrast to conventional publish/subscribe communication platforms (such as Kafka and RabbidMQ) which decouple the producer and consumers and instead write and read to addressable communication channels called 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. Though cumbersome, this also grants an advantage. With strong component bindings it should prove more difficult to deploy an invalid architecture due to small mistakes as mistypes or not updating all topic bindings on a refactor.

Micro-component architecture without execution platform

A final option is to employ a micro-component architecture without an execution platform. Instead we would deploy components ourselves and have them communicate using message brokers. This would increase the efforts needed to develop and deploy the platform, but does provide greater control over its execution. Additionally this would alleviate the deficiencies identified for Apache Storm, such as difficult third party integration, cumbersome topology building and lack of run-time reconfiguration.

3.4.2 Message brokers

By employing a micro-component architecture we need to identify a communication technology for components to communicate to each other. This approach employs a service to which producers write messages to a certain topic. Consumers can subscribe to a topic and consequently read from it. This obscures host discovery, since a producer need not know its consumers or vice versa. This routing is instead performed by the message service. The following will explore the two widely used message broker services in the industry: RabbidMQ.

RabbidMQ

RabbidMQ[?] 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 sent 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 multi-casting a message to multiple types of consumers difficult.

	RabbidMQ	Kafka
Speed	+	++
scalable	+	++
Multi-cast	×	✓
multiple reads	×	✓
Acknowledged	✓	×
Delivery guarantee	✓	×
Consumer groups	✓	✓
Retain ordering	Topic level	Partition level
Consumer model	Competing	Cooperating

Table 3.2: Summary comparison of RabbidMQ and Kafka

Apache Kafka

Instead, Apache Kafka [?] 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 example message order retention can no longer be guaranteed for the entire topic, but only for individual partitions. Kafka, in contrast to RabbidMQ's competing consumers, operates on the *cooperating consumers* principle. It performs this by, instead of popping the head of the queue, a consumer retains a counter pointing to its individual head of the queue. 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 pointer, which ensures that the group collectively only 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.

Comparison

A comparative summery of both technologies is given in table 3.2. Following this comparason we have chosen to employ Kafka for our platform. The first observation is that Kafka performs better in non-functional metrics. Sources report Kafka to be 2-4 times faster than RabbidMQ[?] and the partitioned topics allow Kafka to be distribute and scale overloaded channels. Secondly, the cooperating consumer model Kafka is based on allows us to natively multicast messages to multiple consumers, while still being scalable by defining consumer groups. By choosing for Kafka we do however default some features such as producer acknowledgement and topic level order guarantees. As for producer acknowledgement we do not require it, as producers simply send messages into the clear and consumers are required to make efforts that it processes all data.

Using the feature to read messages more than once, we should be able to build a dependable platform. Finally, Kafka cannot guarantee the read order of partitioned topics. We therefore will need to enforce it ourselves in the platform and implementations of it. This can be either done by sorting messages in buffers on some ordered parameter (e.g. timestamp or sequence number) or by not partitioning topics containing order-critical streams.

3.4.3 Distributed computing

As specified by requirement R1.4 we require a means of processing large amounts of data. We accomplish this by aggregating large numbers of snapshots into a distinct smaller amount of snapshots with higher-degree of information. In order to accomplish this we require a scalable means of computation (requirement R1.6)

MapReduce

MapReduce[?] 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, grouped by key as a key-value set tuple. 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 *mapper* and *reduce* functions. All other procedures, including calling the mapper and reducer, are handled by the framework. An example of the algorithm on the WordCount[?] problem is illustrated in Figure 3.1.

The concept of a mapped processor is of a large benefit to our 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 groups of devices sending more data then others. These devices would be overrepresented in the population if we did not account for them sending more messages than others. By first grouping the messages per device ID we can assure that every device has the same weight when we, fore example, calculate summations or averages.

Though the ease of implmenetation is very high and the technology is very appliccable to our platform, the algorithm has prooved 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 by slow disk writes[?]. Finally, MapReduce works on large finite datasets. Therefore we need to manually preprocess stream data into batches in order for MapReduce to be applicable[?].

Apache Spark (Streaming)

Apache spark is an implementation of the Resiliant Distributed Dataset (RDD) paradigm. It entails 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 [?].

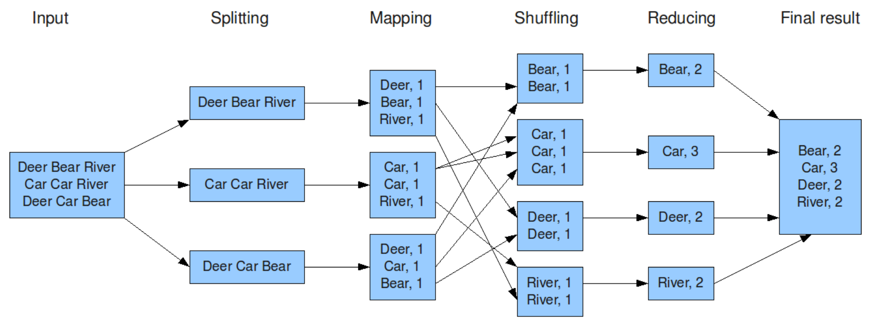


Figure 3.1: The overall MapReduce word count process[?]

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 stays the same, 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 continue. Finally, Actions, such as *collect* and *count* require all data to be recalled to the master and most of the calculation is performed locally, resulting in a concrete return value of the process. RDD's provide an 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 eliminate superfluous dataset redistribution.

```
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 3.1 in Spark RDD.

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

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 it cannot recover it's state. This is 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 that point.[?]

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. In fact a Apache Spark library exists (Apache Spark Streaming[?]) which performs in this manner. It batches input from streams on regular, pre-specified time 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.

3.4.4 Solution decision

For distributed component platform we have chosen to build upon Apache Storm. 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 us with the perfect building blocks to design an iterative information refinement application with separation of concerns in mind, while the built-in streaming mechanism provides the needs for a real-time distributed application. We will however need to account for the lack of expose points for third party integration and the tedious process of specifying each and every bolt connection.

Though Storm contains the means for large scale snapshot aggegation, we will not employ it. Instead we will base our data aggegation on Apache Spark Streaming. The reason for this is that studies have shown Apache Spark to be 5 times faster than both MapReduce[?] and Storm[?]. 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 our envisioned use case is for timed analysis jobs on very large amounts of input data, in order to detect or visualise collective tendencies of the system under investigation. For this scope of application the latency issues of Apache Spark do not impose a large deficiency.

To facilitate external communication of the platform we will employ Apache Kafka. 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 listen in on the proceedings of the platform. With our choice for Kafka comes another benefit, as the Spark Streaming library contains adapters for Kafka allowing direct connection to it. Therefore we can simply emit data to a Kafka topic and connect a Spark Streaming process to it. The greatest deficiancy of Kafka, being the lack of topic-level order guarentee, is not of grave importance. The hindrence 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 reaseon for this is that most computations will contain of a *re-*

duce step, which requires the reduction operation to be both associative and commutative[**?**]. Therefore the message order is of no importance.

3.5 Design of the software platform

We will adapt these technologies by composing them using adapters and abstracting the solutions. By abstracting the technologies we shield the internal implementation details, simplifying implementation by the user. We will provide the implementer some scaffoldings for bolts intended for different types of data flows and data reductions. Additionally, these technologies are very abstract since they were intended for many unspecified usages. Since our platform and group of target applications features some known commonalities, which were considered variations when designing the original technologies, we can implement some functions 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

In the remainder of this section we will explain what adaptations to the previously discussed technologies are made.

Apache Storm

The bulk of the component construction and execution, and streaming services of the platform will be performed by Apache Storm. However, as discussed before, the process of specifying a 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 we have developed a topology builder which enables topic based streaming. The builder will automatically connect the specified components according to the topics they are subscribed to, when executed. In this manner a component and its connections can be specified with but a few lines of code, 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 component 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 messages. The two chief methods are supported by our 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 message. 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 messages to always be processed by the exact same worker. An simple example of this is a processor that counts the number of messages received for each sensor in a WSN. If we cannot guarantee that all messages of a sensor S are always processed by the same worker W, one worker might count 40 messages and another would count 60 of them. This would require another singular processor that accumulates those counts in order to derive an accurate message count. Therefore it is possible to specify a set of fields which will deterministically and consistently determine which worker will consume a message. In our adaptation 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 for that channel.

Finally, though we believe the abstractions and encapsulations of the Storm platform to be useful 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 our abstraction offers. To this end we have chosen our topology builder to encapsulate the topology builder provided by the Storm Java library. This entails that our topology builder, upon calling the *build()* function, will 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 distinct 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 we should make a scalable data accumulator available.

As specified in section 3.4.4 we have chosen Apache Spark Streaming 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 we have chosen to device a process that adopts the adapter software pattern [**?**]. 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 master node. 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 3.2.

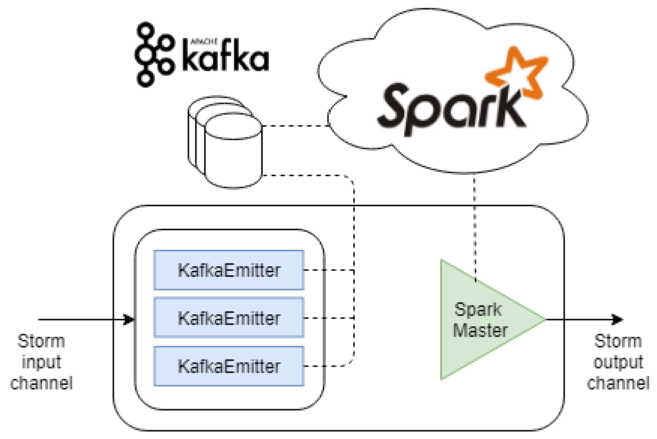


Figure 3.2: Graphical depiction of the distributed accumulator process

Two interesting remarks should be made, as apparent from Figure 3.2. Firstly, The KafkaEmmitter can be replicated in order to prevent it being a choke-point in the topology. Secondly, the fact that two distinct components (KafkaEmitter and Spark Master) are present is encapsulated by the topology builder. Developers need only declare an implementation of the distributed accumulator processor (acting as Spark master node) with the appropriate Storm and Kafka channels. The builder will then deploy a KafkaEmitter (or several) and the accumulator. This makes deploying the processor easier and obscures the internal implementation by appearing as a single component.

3.5.2 Scaffolds for micro-services

With the supporting technologies established we will now describe and deliberate the component scaffolds that are supplied for application developers by the platform. We will first describe the base functions shared by all components, before discussing them more in depth individually.

Common functionality

Firstly, the components contain all functionality and information required to emit new snapshots to consequent components. A developer need only package the information in a message containing key-value pairs and specify to which stream a snapshot should 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(args)* [1] method. This method is used to instantiate some properties that cannot be instantiated in the objects constructor. The reason for this is that all components extend some abstract spout or bolt class of Apache Storm. In the Storm

[1]actual arguments have been omitted due to simplification

platform all spouts and bolts adhere to a pre-specified execution order. The component is:

1. created by one of its constructors,
2. transmitted to one of the slave nodes of the Storm cluster,
3. further instantiated using the *prepare(args)* 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(args)* method thus can be used to instantiate certain user-specified non-serializable properties. However, one should note that overwriting this method also requires invocation of the super method, since the default implementation specifies some non-serializable Storm properties and classes.

Spout

This process is named after to 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 streaming technology. The reason we need a special 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 message is available. However, a spout runs on an infinite-loop (until termination) continuously calling a method *nextTuple()*. This method polls, retrieves and emits messages 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 the ease-of-use. 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 key-value pair received by the component.

HistoricBufferedProcessor

The HistoricBufferedProcessor resembles the SingleMessageProcessor in that it consumes single snapshots, but instead it computes on or analyses a series of sequentially relevant snapshots, called the *window*, sorted by sequence or time. 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 analyse and 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

TODO

DistributedAccumulatorProcessor

This component is used to aggregate large amounts of laterally relevant snapshots. By laterally relevant we mean that the snapshots describe similar datapoints, but have no sequential relevance. The input for this process is a large amount of (individually) low-information snapshots in order to emit some high-information snapshots. An example of its usage is combining thousands of snapshots from individual sensors in order to obtain some collective performance parameters. For the task of accumulating and aggregating these enormous amounts of data we employ the accumulator principle described in section 3.5.1. 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 user-specified 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. A final remark to be made is on the granularity of the batch processing. As stated before [(echt?)] some real-time properties are lost by collection and processing streaming data as batches. This has been partly mitigated by employing the windowing mechanism of Apache Spark Streaming. This mechanism collects data in relatively small sub-RDDs. one or more of these smaller consecutive RDD's are then collected as one larger RDD called the 'window'. This window has a fixed size and slides over the sequence of sub-RDDs. This allows these small batches to be part of several consecutive windows. A graphical representation of this process is depicted in Figure 3.3. By this method it allows for example the analysis of data windows of the past 5 seconds, every one second. Whereas without this mechanism it would only be possible to process the last 5 seconds every 5 seconds or the last second every 1 second. Additionally, this process is very efficient, since the internal windowing mechanism automatically caches the results of the intermediary sub-RDD's. Therefore the entire chain of computations does not need to be recalculated for each windowed operation, only the transformations past the caching of the sub-results.

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 can instead be run in-memory on a single machine. 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

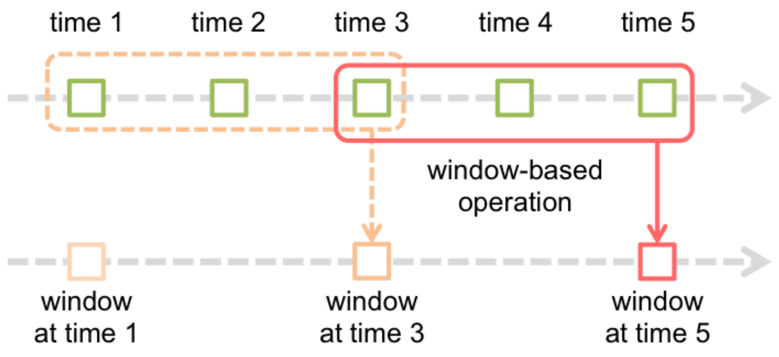


Figure 3.3: Apache Spark windowing mechanism. Source: [?]

Apache Spark cluster to be deployed in order for the platform to be deployed, 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 load-balanced. Load-balancing would require a sequential singular component that combines intermediary results aggregated by the load-balanced workers into an eventually final snapshot

To facilitate the easy implementation of the AccumulatorProcessor the processor was modelled after the MapReduce paradigm. An implementer need only specify a series of MapReduce steps (possibly singular) and an eventual single collect step. The exact methods to implement for this are:

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

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 messages indexed by the Storm topic on which it should be forwarded.

Please note that the return type for the reduce step is a new message. It is therefore possible to chain multiple map-reduce steps sequentially, as long as the sequence is concluded with a collect step.

ResourceDistributionModelProcessor

[TODO]

3.5.3 Demonstration by example topology

In this section we will illustrate an example of the composition of the specified components. For this purpose we will consider the case exemplified in section

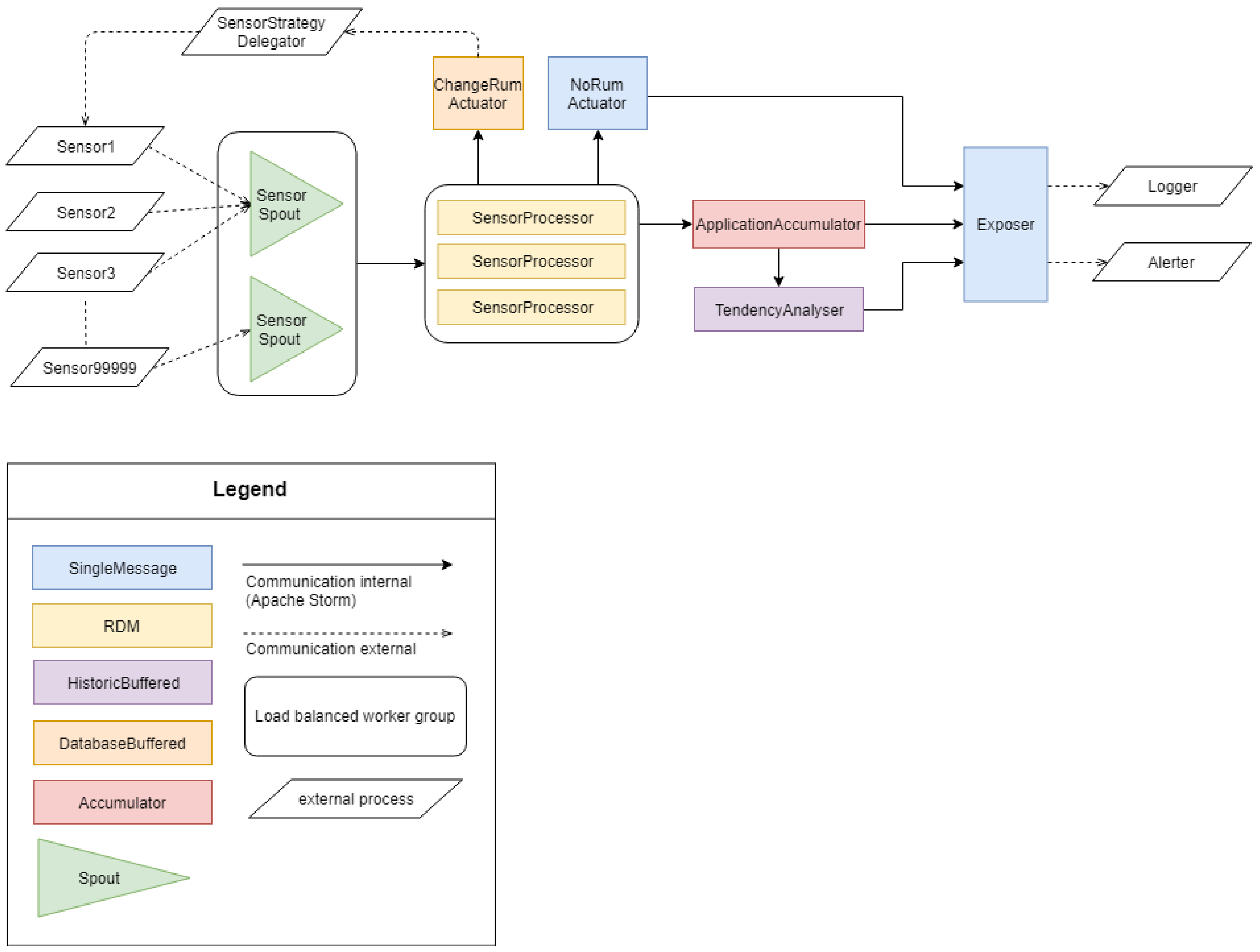


Figure 3.4: Example topology of a platform implementation according to the example case

2.5. A graphical depiction of the topology for the example implementation is found in figure 3.4.

As figure 3.4 makes apparent, the application encompasses a large number of sensor devices. These devices regularly send their status information to our application via some external communication technology (e.g. Apache Kafka). These snapshots are introduced into our 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 mode of operation 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 messages it received for this sensor and will only actually change the mode of operation of the sensor if it is consistent with the last few messages it received. In this manner we can eliminate superfluous expensive communication with the sensor device due to 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 consequent a changes of operation would occur even less. It would therefore make little sense to keep a buffer of the last messages 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 instance.

A 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 during the time windows. Should this total consistently rise over a period of time or over a number of snapshots an alert will be sent by the reporter, as specified by the alerting requirements listed in section 2.5.

3.6 Discussion of the proposed software platform

In this section we will evaluate the design of our monitoring platform.

Satisfaction of requirements

The first order of business is whether the proposed design satisfies the earlier stated requirements. we believe that the message-passing micro-service architecture provides the basis for snapshot transferral and transformation as stated in requirement R1.1. Furthermore, we believe that the requirements R1.2, R1.3 and R1.4 are satisfied by the inclusion of the *SingleMessageProcessor*, *Buffered-Processors* 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 many choices of the supporting technologies and are reflected in our employment of cloud processing technology Apache Spark. From the aforementioned arguments we conclude that every requirement is represented and met in the design of the platform.

Completeness according to QoI attributes

The goal of the platform is to process and enrich data. It is therefore rational to evaluate the appropriateness and compleness of the platform by considering the information processing capabilities it offers. In this section we will thusly evaluate the platforms completeness by demonstrating that the platform not only satisfied our identified requirements, but also does not negatively impact the Quality of Information (QoI) of the input data. By this we intend that the QoI is improved or retained, but never lost as data passes through the platforms topology. We will achieve this by arguing the QoI parameters which were enumerated in section 2.2.3.

The first consideration of QoI is regarding the processing of data by our platform and affects the precision, completeness and ease of use of information. Firstly, *precision* and *certainty* are obtained by employing the HistoricProcessors. By averaging measurements anomalies are mitigated and the measured 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 *ease of use* of information is improved as data moves throughout the topology. To illustrate this we propose a thought experiment using the example topology listed and described in section 3.5.3 and a batch of raw data emitted in a certain time window. Before the data enters the platform it contains all the information potential to calculate the average throughput offered by the entire sensor application during that time window. Otherwise our platform equally would not be able calculate it. However, actually calculating it would involve extracting the correct data-point(s) from each snapshot, calculating device performance, extracting the throughput, averaging it for each device individually and ultimately calculating the average over the entire application. Instead this process is automated by an implementation of our platform and the resulting information is offered for further processing or visualization. This demonstrates that our platform can facilitate ease of use for information by calculating and producing a ready-for-use value. 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, we recommend committing the raw data to storage before processing it.

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. We provide *timely* execution 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 we only claim near real-time, 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 and the windowing mechanism of Spark allows for efficient fine grained processing, so it does not impact the timeliness greatly. However, adverse to this gained timeliness we have 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. We have therefore also supplied the locally deployable alternatives to these distributed processors. Implementations of the platform are therefore offered a trade-off between timeliness and cost.

Lastly, we have the *tunability* and *reusability* of the information. Firstly, the data can be duplicated among different communication channel 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. By caching the data it can be re-fed into an updated topology in order to initialize an application as if it had been running for days.

Some final remarks should be made on the analysis. Firstly, our 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 our platform cannot assure preservation of any of these metrics, 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 adaptation

The first point of focus is the ease of adoptation provided by the platform itself. We believe that by offering some abstract components that require implementation of one or but a few methods, we have effictively obscured the low level implementation details of Apache Storm and Spark. This obscuration entails a clearer programming interface to an implementer, as defined by the *facade* programming pattern. [?]

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. This allows our example topology described in section 3.5.3 can be specified using only [xxx] lines of code.

Technology stack

The second issue to contemplate is the technology stack required for the platform. As mentioned in section 3.4.4 we chose Apache Storm as enabling technology because it offered most of the features required and would reduce our technology stack. However by employing Apache Spark for distributed data aggregation we have introduced two cloud technologies, as Spark requires Apache Kafka in order to be connected to a Storm Topology. We do however hold the belief that the inclusion of a distributed aggregator is necessary in order to keep the computation scalable. Additionally the speed and efficiency arguments raised in section 3.4.4 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, which would still enjoy benefits from process parallelization. Finally Spark and Kafka may be omitted entirely, as a non-distributed data aggregator is also included.

Future work

Finally, our topology-based separation of concern approach allows for visualization of the computations and distribution. The chain of computations can easily be depicted as a directed graph with processors and topics as nodes and processor-topic connections as vertices. Such a topology visualization would for example be very useful for identifying incorrectly or disconnected components. With an even more extensive user interface an editor tool could be device, allowing a topology to be drawn and functional methods to be implemented later. It should be noted that, though promising, the library does not feature such visual user interfaces. However future efforts could be made to facilitate them.

4. Resource Distribution Model

4.1 Objective of the model

The aim of the Resource Distribution Model (RDM) is to comprehend the distribution, conversion and requirements 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 we require a detailed model with explicitly defined entities and relations. Only then can the model be employed by automated tools and algorithms.

This will be performed by first exploring the problem domain. With the definitions and concepts of the problem domain identified, we will compose a list of requirements for the proposed model. With these requirements in mind we will explore contemporary resource modelling solutions and evaluate them on the applicability to our requirements. We will then explain how the selected technologies will be adapted for our purposes. Subsequently, we will describe our model in detail and exemplify how we intend to use the model in order to calculate the optimal performance of a modelled system. We will conclude this chapter with an evaluation of the proposed modelling technique.

4.2 Conceptualization of the problem domain

In this section we will investigate problem domain in order to eventually determine the requirements for the model. Again, we will achieve this by performaing a commonality/variablility analysis (section 2.4.1) of the problem domain, determining the definitions, common features and variations in our problem domain.

Definitions

We will first establish some terms we will be using throughout the C/V analysis and the remainder of this chapter.

Resource: Any measurable/calculable parameter of a system

Resource constraint: A constraint imposed on a resource.

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

Quality of Service (QoS): Parameters which are indicative of the level of service provided by a system.

Commonalities

Following the definitions we will now identify commonalities that appear throughout the problem domain. These assumed features allow us to focus our efforts and allows more expressive specification of assumed concepts.

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

C2.2 A component can produce or offer 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.

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

Variabilities

With the commonalities established we will now consider the variabilities in the problem domain. These variations cannot be specified specifically in the model, but instead require proper abstraction in the model, to be implemented when a instantiation of the model is performed.

V2.1 Though all use cases agree on the above commonalities, we cannot predict all resources, components, constraints and interconnection that can occur.

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) entails concrete, palpable parameters.
- A macro-scale (e.g. an entire WSN application) entails accumulated, theoretical parameters

V2.3 A system can have multiple resources as QoS indicators

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 for a certain moment of measurement. However, some resources are derived and calculated using other resource values. [?]

V2.6 Most resource values differ depending on system's measured state

V2.7 Some resource values/usages differ depending on a specific system function

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

4.3 Requirements for the proposed model

With the common and variable features of the problem domain established we can formulate a list of requirements that need to be incorporated in the solution. In this section we will therfore identify the requirements for the projected model. We will first provide a full list of the identified requirements before justifying them according to the C/V analysis of section 4.2.

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 A model should contain constraints that describe the limitations of interconnected 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.

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 easily 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 (many-to-many). It can therefore be inferred that it is possible to transform multiple QoS markers into a single optimizable, meta-physical resource, according to some heuristic QoS 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

In this section we will explore the current techniques and technologies in the field of resource modelling. We will first identify the state of the art of the field,

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

before evaluating their applicability according to our established requirements. Finally we will declare and defend the choices we made before adapting the technologies in the next section.

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[**?**]. 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 methods and tools. Another example of early research is the study performed by Seceleanu et al[**?**]. This study focussed on modelling resource utilization in embedded systems using timed state machines. The transitions in these automata were attributed resource costs to model the consumption of resources for transitioning to a state of remaining 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[**?**]. 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 (such as UPAAL[**?**]) 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[**?**], which is both intuitive and descriptive. We will therefore continue to use this notation.

however these models are all focussed on components that are self-aware of their resource usage and performance. Instead, we are interested in off-site analysis of interconnected resources and accumulated performance of a composite system. Our focus is therefore alternatively more resource-centred. It is concerned how production and consumption of a resource is interconnected. Components only serve as secondary elements, merely specifying how these resources are converted into other resources. Therefore a resource-centred adaptation of this framework might be more suitable for our problem.

Secondly, there is the issue of how to represent a Resource Utilization Models (RUM)[?], the model for variable behaviour of components. Previous studies [?, ?] have used timed automata to represent behaviour cycles. This allows for automated tools to calculate a runtime schedule in high levels of granularity. However the high level of granularity comes at the cost of efficiency. When we shorten the time intervals for the automata, entailing higher granularity, then solvers require additional computational resources and time to execute. This might force a problem on resource constraint devices or applications that require the solver algorithm to run many times for a multitude of devices. Additionally, we need to consider that a model contains multiple components specified by RUM's. For these models a valid, optimal RUM composition needs to be determined. In this case RUM's might influence each other, which implies that for different compositions of these models, the individual models need to be re-calculated.

An alternative 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 long-term behaviours. However it significantly improves the complexity of the search space. For this approach timed automata is no longer a sensible technology since the element of time intervals has been eliminated. Instead the problem is a pure decision problem[?]. The only 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 2.3 available for different types of problems, a suitable solver should be able to be found or developed.

4.4.3 Choices of employed solutions

With careful consideration the following choices for the solution implementation have been made. For modelling we chose to adapt the framework of Malakuti et al[?], 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 two resources to one another. Another adaptation is the existence of multiple RUM's for a component, which allows injection of different methods of operation and calculation of the optimal system functionality.

As for how to model the RUM, we chose to reduce the complexity of the system by modelling variable resource usage with static parameters. The strongest advocate for this choice is the fact of the focus for this research: large IoT applications. In an IoT 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 IoT applications only send and receive data a few times per day[?]. 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. We will however complement the search algorithm 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 exerting the adaptations made to the previously described modelling efforts. We will first depict how we defined our model in both broad terms and specific modelling entities, followed by how we intend to solve the model by calculating the optimal configuration of variables in the model.

4.5.1 The model

As stated we will model resource distribution by extending the model by Malakuti et al[**?**]. The chief adaptations in our model are:

1. the inclusion of a single explicitly defined optimised resource,
2. RUM‘s with static resource values,
3. the existence of multiple RUM‘s for a single component, 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 can be found in figure 4.1. A complete entity relation diagram for the meta-model can be found in Appendix **??**.

In essence the model is a collection of *Resources* and *Components*. Each of these 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. By aggegating the ResourceInterfaces of a resource the amount of the resource produced and consumed can be collected and analysed.

ResourceInterface

Resources and components are connected through resource interfaces. 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,

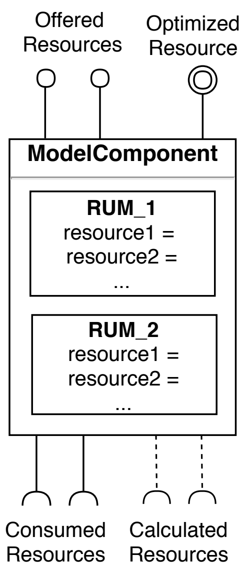


Figure 4.1: Notation of an RDM component with RUM's

Calculate 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. For example a QoS indicator that is "consumed" by a general QoS calculation.

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.

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 special subtype of the Component is the ModelComponent. This class inherits all functionality of the ordinary Component, but its ResourceFunctions are extracted from one of its RUM's. Each RUM describes the parameters during one mode of operation of the components. This allows runtime analysis of variable behaviour as effect of different functionalities.

ResourceFunction

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

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

To model the intended behaviour of the model we introduce a set of *Requirements* and an *Optimizer*.

Requirement

A resource can have a number of Requirements as constraints that limit the possible values of variation for that resource. The standard built-in 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

To ascertain the heuristic score of an RDM with an injected RUM configuration we introduce the Optimizer. 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 solution. 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 again be made by developers.

RdmMessage

Finally, to supply the model with the state of the system under investigation, we pose the RdmMessage. The RdmMessage is provisioned using values measured from the system and injected into the model, after which the appropriate resource values 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.

4.5.2 Demonstration by example case

To illustrate the application of this meta-model, an example of an instantiation of the model can be found in figure 4.2. This instantiation is based on the example case described in section 2.5. In this depiction we can see the 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) since, as described earlier, specifications of power supplies vary in our 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 request value is based on a parameter supplied by the input message. Finally both the amount of measurements and bandwidth provided are supplied to the *QoS calculator* which uses the information 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 validated set of competing 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.

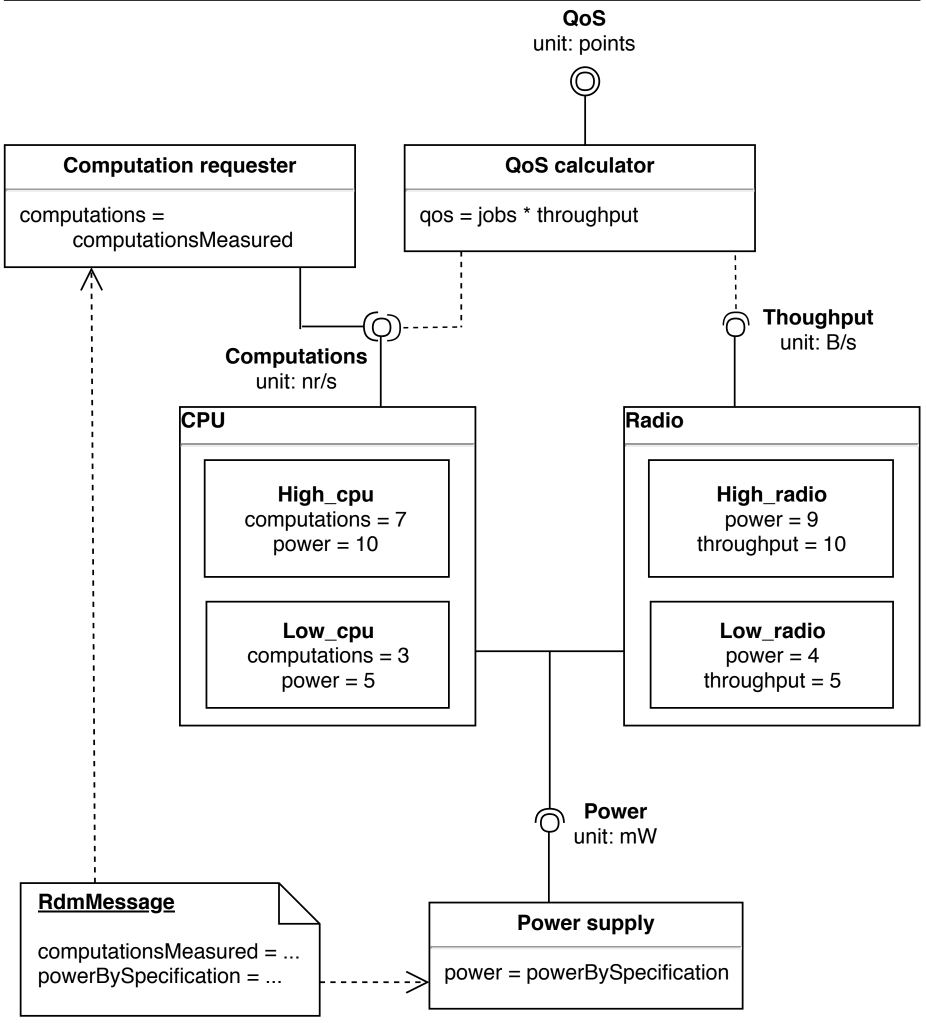
4.5.3 Computing an optimal model assignment

With the model well established we can now try and solve the model. From requirement R2.7 we find the goal of 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.

The first and second requirement imply constraint solvers as an applicable technology, since they are effective in finding a valid solution for a constraint decision problem. However, the third requirement entails that we do not want to find just any valid solution, but the *optimal* valid solution. In order to do that we need to consider every valid solution to the problem and compare how they compare heuristically. This entails a full brute force search approach through the entire search space of RUM compositions. We can however use constraint solver paradigms to preventively reduce the search space as we search through it.

The way we do this is by employing backtrack search. In a simple brute force search we would calculate all RUM compositions (Cartesian product) and for each composition we provision the full model and evaluate it. Instead we will iteratively select a component and one of its models. We will then not provision the entire model, but inject only the selected model in the chosen component. Consequently, we set the values for variables for which we can resolve a definite value, given the current state of the model. We then evaluate the resource constraints. Given an incomplete model any constraint can have one of three statuses:



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

Optimize:
$max(QoS)$

Figure 4.2: Example instantiation of the RDM meta-model according to the example case

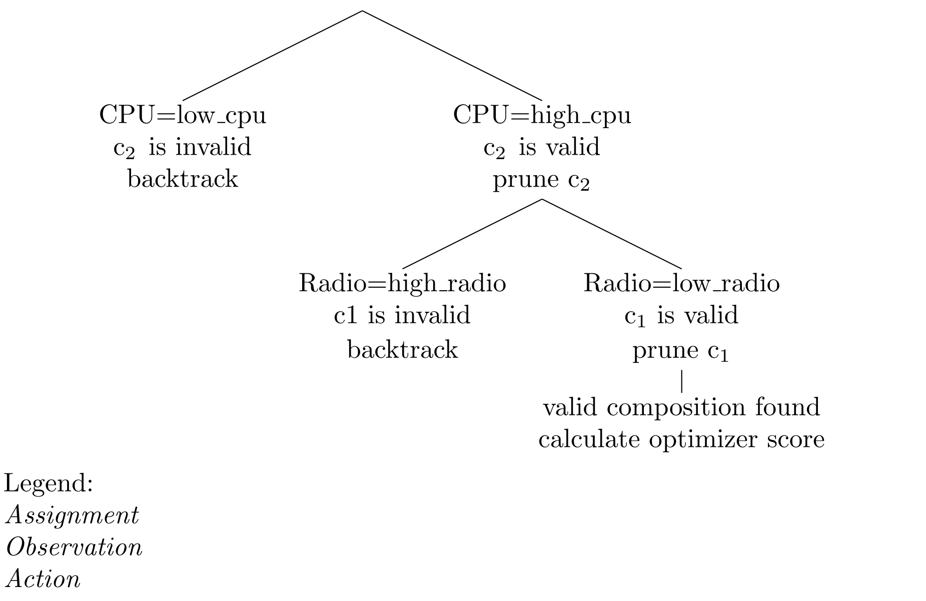


Figure 4.3: Application of backtrack search on RDM of Figure 4.2

- satisfaction,
- failure, or
- uncertain

for all consequent 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, since we know it will always succeed. If a constraint is *uncertain* we keep it, since we do not know its status for each and every future state. If even a single constraint *fails* we know the remainder of this branch of the search tree will never be valid. Therefore we backtrack through the tree by partially rolling back model assignments. We then select a different model for the same component or a different component entirely and repeat the algorithm. This way we do not recheck constraints we already know the state of and do not evaluate paths we know will not satisfy the constraints. The full original algorithm is given in Listing 4.1.

Given that we encounter unsatisfactory options early in the tree, this will possibly eliminate large parts of the search tree. An example of the application of this algorithm on the example previously posed (Figure 4.2) is given in Figure 4.3. This example is executed based on an RdmMessage with values $\{measureRateRequired = 8,\ powerBySpecification = 16\}$. This application demonstrates that using this algorithm, we eliminate a significant portion of the search tree. This is due to early constraint failure detection in the *CPU=high_cpu* banch of the tree.

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.

```
1. (Initialize.) $cur \leftarrow 0$.
2. (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 $D'_{cur} \leftarrow D_{cur}$.
3. (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 $\overrightarrow{a}_{cur-1}$ and $x_{cur} = a$. If it is, go to Step 2a.
    (d) Instantiate $x_{cur} \leftarrow a$ and go to Step 1.
4. (Backtrack step.) If $x_{cur}$ is the first variable, exit with “inconsistent”. Otherwise,set $cur \leftarrow cur - 1$. Go to Step 2
```

Listing 4.1: Algorithm for backtrack search[**?**]

4.6 Discussion of the proposed model

We will conclude this chapter by endorsing some of the choices that were made for our proposed model.

Static model

As stated before we chose to use a static representation of resource utilization by ModelComponents. We chose this in order greatly reduce the complexity of the problem and this allows the model to be evaluated within a reasonable amount of time. We came to this conclusion after early experiments with timed automata. In this experiment we modelled a minimal system with one component with three RUM's. When analysing 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 server with sufficient calculatory resources the time to calculate will be reduced. This is however counteracted by the fact for a WSN application this calculation needs to be repeated for thousands of sensors. When we compare this performance to that of the static models, which can evaluate more complex models (e.g. 3 components, 5 RUM's each) within seconds, we must eliminate timed automata as valuable real-time technology. However, this does not eliminate automata entirely. Automata can still be used to model the fine grained run cycles of parts of a system in order to develop generalized static RUM's.

Solver libraries

When developing this solution we chose to implement the constraint solving algorithm ourselves, instead of employing existing libraries such as Choco Solver[**?**] or OptaPlanner[**?**].

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 powerfull, it has only limited support for real intervals [**?**]. Additionally it proved very difficult to convert the user defined models and arithmetic expressions to the modelling mechanism of the solver. Requiring the user to either input the model and calculations in the complex modelling mechanism of the Choco Solver or for us to develop a compiler to rewrite the easy to write user input to Choco Solver code.

Another examined library is the OptaPlanner. 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 our 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 (Cartesian product of all RUM compositions) running a single code block which evaluates all constraints. It consequently can not reduce the search space by eliminating failing branches and redundant constraints. Therefore it lacks the means to solve the problem efficiently

Finally, the implementation of backtrack search does not differ much from the implementation of depth-first search. Additionally, developing our own solver allows us to incorporate domain knowledge into our custom search algorithm, further reducing the runtime required. This reduces the comparative benefit of employing a constraint solver library and eventually led us to develop our own solver implementation.

Constraint propagation

A technique in constraint solvers mentioned before is the concept of constraint propagation. Constraint propagation explores the search space the 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=High_CPU* is initially assigned, *Radio=High_radio* is pruned 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 our model the variables are not integer/real domains, but objects with integer/real variables. This doesn't make constraint propagation impossible, but does complicated it greatly.

Secondly, the interconnected nature of our problem can impede the benefits received from constraint propagation. To illustrate this consider the following 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 (more must be produced than is 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*, we need to assign all producers in order to determine a reliable upper bound[1]. This requires the search to be already at least $|P|$ levels deep, reducing the part of the tree possibly eliminated. Even then, we are only able to prune the values 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 we may be able to prune values

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

more easily, but this requires even more variable assignments. This problem is aggravated when R_p is a derived value calculated using a number of other resources. Values for all these resources must be known in order to calculate the value of R_p.

To conclude, the part of the tree that is eliminated with constraint propagation is limited since we are already halfway into the search tree and, additionally, 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.

5. Design method

5.1 Adaptation

Application of the Design Cycle to these design artefacts

5.2 Cycles architecture

5.3 Cycles RUM

6. Proof of concept by case study

6.1 Case study

6.1.1 Background

Nedap - Identification Systems

company andere marktgroepen/producten marktgroep

SENSIT [smart parking] application

Tech stack information emitted data signature

6.1.2 Description of the Case Study

Description of the case
simulation scale

6.2 Requirements of the Case Study

6.2.1 Claims

what claims to validate and requirements implied by those claims relate to research questions (RQ2 how to) RQ3 level of abstraction RQ5 overcome scalability issues easy to develop (RQ3) harde eis moeilijk lack of examples measured in lines of code per scalable (RQ5) event burst eis: backup processed within X seconds some-many-huge ('on the fly') measure latency introduce 'single point of failure' -¿ no bottleneck measure congestion

6.2.2 Bounds

only one case. Partial validation, proof of concept simulated environmant (based on acutal data signatures and data rates) current state of sensit not a scale challenge simulation can fast forward

6.2.3 Requirements

functional features not measurable, just attainable non-functional scale ease-of-implementation measurable validation criteria

Justification of the Requirements

6.3 Design and Implementation

6.3.1 Design

6.3.2 Implementation

6.3.3 Equipment

6.4 Results

6.5 Evaluation

6.5.1 Evaluation of Requirements and Claims

6.5.2 Discussion

no validation of intuitiveness only applicablility needs test persons more broad validation needed to validate estimated lines of code per input unit (datapoints, results, etc)

7. Conclusion

7.1 Discussion

why not sensor or edge computing feedback into models (learning models) wider applicability?

7.2 Conclusions