1. Introduction

introductory section

1.1 Problem statement

1.2 State of the art (working title)

Current state of affairs
deficit in current state of affairs

1.3 Approach

1.3.1 Goal

1.3.2 Reserch questions

1.3.3 General methodology

1.4 Organisation of thesis

2. Backgound

2.1 Sensor networks/IoT

2.2 Micro-services

2.3 Variety/commonality analysis

2.4 Design Science methodology

more to add as needed.

3. Design of IoT platform architecture

3.1 Problem statement

3.1.1 Goal

3.1.2 Variety/commonlity analysis

Definitions

Platform: the monitoring platform to be designed.

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

Snapshot: a collection of data points indicating the state of a system on a certain [point in time].

Source: a device (external) or process (internal).

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

Commonalities

C1.1 transform snapshots

Variety

conclusion basis

V1.1 single snapshot. (e.g.)

V1.2 multiple temporally ordered snapshots from a single source. Used to analyse tendency of parameters. (e.g.)

V1.3 many snapshots multi-source without individual significance. (e.g.)

We can never anticipate the exact consequence intended for a [certain] application.

V1.4 The possible consequences by the platform have a large range of implementations.

Though the exact implementation of consequences can never be [exactly] anticipated, we can identify some [common] types for a consequence.

V1.5 build a model for general/global reports. Either by an in-memory component with an API or by persisting it to intermediary permanent storage.

V1.6 analysis invokes an immediate feedback response to the application or a command & control service of the application

V1.7 alerting or reporting according to a specified rule. When this rule is met or violated (user defined) an [intilligente] alert is sent to a [werknemer] or auxiliary system.

The final variety is the scale of the application. We have already established [enforce in paper!] that the platform will operate on applicaitons of large scale, i.e. thousends of sensors. However given a thousend 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.

V1.8 The scale of large [WSN] applications varies wildly. This [geld] for both the number of devices in the application and the rate at which the devices send data.

3.1.3 Requirements

V1.1 The platform should enable the transformation of snapshots.

V1.2 The platform should enable processing of single snapshot.

V1.3 The platform should enable processing of a limited window of homogeneous snapshots.

V1.4 The platform should enable processing of a [large] amount of snapshots.

V1.5 The platform should enable implementation of a wide range of consequences, with a [focus] on a specific set of types:

- model building
- application feedback
- rule-based alerts

V1.6 the platform should be scalable in order to support any large amount of sensor devices.

3.2 State of the art

3.3 Solutions

3.3.1 Architecture basis

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. A [ding] illustrated by a famous [uitspraak] 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 [raadpleeg] 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 V1.6. Therefore we will instead adapt a micro-component approach. Micro-component are more flexible than monoliths. Micro-component systems allow better functional composition, are easier to maintain and much more scalable[?].

[choice microservices platform]

3.3.2 Message brokers

By employing a micro-component architecture we need to identify a communication technology for components to communicate to eachother. 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. It does however introduce some latency and introduces a new upper bound for speed, [since] the system can exchange messages only as fast as the message broker can route them. The [proceedings] will [explore] the two widely used message broker services in the industry.

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 [versitile] solution. RabbitMQ operates [on] the *competing consumers* principle, which [states] 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 guarentees 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.

Apache Kafka

Kafka [instead] distributes the queues itself. Each host in the cluster hosts any number (or none) of partitions of a topic. Producers then write to a particular partition of the topic, while consumers will recieve 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 guarentees to be dropped. For example message [ordering] can no longer be guarenteed throughout 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 [remembers] a counter pointing to its individual head

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

Table 3.1: Summary comparison of RabbidMQ and Kafka

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 applicaiton level acknoligement, meaning that the producer cannot [detect] whether its messages are consumed.

Decision

A comparative summery of both technologies is given in table 3.1. Following this comparason we have chosen to employ Kafka for our platform. The first observation is that Kafka performs better in non-functional [parameters]. Sources report Kafka to be 2-4 times faster than RabbidMQ[?] and the partitioned topics allow Kafka to be distribute and scale [busy] 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 [miss out on] features such as producer acknolidgement and topic level order guarentees. As for producer acknolidgement we do not require it, as producers simply send messages into the clear and consumers are required to make efforts that it [fetches] all data. Using the [ability] to read messages more than once, we should be able to build a dependable platform. Finally, Kafka cannot guarentee the read order of partitioned topics. We therefore will need [enforce] it ourselves in the platform and [implementtaions] of it. This will be eigther done by sorting messages in buffers on some [temporally] ordered prameter (e.g. timestamp or sequence number) or by not partitioning topics [describing] order-critical streams.

3.3.3 Distributed computing

[iets over de accumuator task]

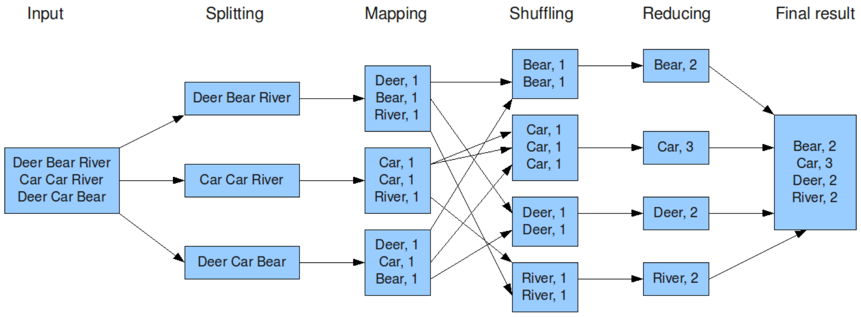


Figure 3.1: The overall MapReduce word count process[**?**]

MapReduce

MapReduce[**?**] is a distributed computing framework developed by [devs]. 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-valueset tuple. The key-valuesets are then distributed across machines and a *reduce* function is called to reduce the many individual values into some accumulated datapoints. 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 a large benefit to our platform. In early talks it quickly became apparrant 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 applicable 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 [**?**].

Three sort of operations exist: narrow transformations, wide transformations and actions. *Narrow transformations* are [item wise] operations that effect individual data entries and result in a new ([changed]) RDD, with the original RDD and target RDD partitioned [hetzelfde]. Examples of such functions are *map* and *filter*. Because these transformations are applied [item wise] and partitioning [are equal], many of these transformations can be performed consecutively without recalling the data to the master. *Wide transformations* similarly are applied itemwize, but the target RDD may not be partitioned [gelijk] to the original RDD. An exapmle of such a transformation is *groupByKey*. Since elements with the same key must reside in the same partition, the RDD might require reshuffeling in order to continue. Finally, Actions, such as *collect* and *count* do require all data to be recalled to the master and calculation is performed locally, resulting in a concrete return value (integer, list, etc.). RDD's [entail] 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 [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 [same] [chain] of transformations is applied to each individual entry in the dataset any new slave can continue calculations from that point.[**?**]

Finally, Apache Spark suffers the same [problem] as MapReduce and is performed on finite datasets. Therefore streams need to be [collected] in batches in order to perform calculations. In fact Apache Spark [has] a library, Apache Spark Streaming[**?**], which achieves [just that]. 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 does achieve near-real-time stream processing.

Apache Storm

Apache Storm is a big data computing library especially designed for seperation of concerns. It performs distributed comuting by [seperting] the stages of computation. By breaking up the computation different stages can be distributed among machines and duplicated if need be. This is [performed] by three chief concepts.

Spouts: nodes that introduce data in the [network],

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 acyclical graph with bolts as vertices, spouts as [start 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 compuations into multiple consequtive 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 [expensive] singleton bolts.

Storm is especially [applicable] to our purpose since it was designed for microcomponents connected by streams. In [comparison] many micro-component platforms [focus] on components exposing services which are explicitly [called] by other services[**?**, **?**]. By employing Apache Storm we [recieve] both the distributed computation environment as the means of data distribution, simplifying our technology stack [check met kafka].

Conversely however, the built-in distrribution 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 stakeholders will require an exposing mechanism. Additionlly, Storm requires bolt connections to be expllicitly specified at startup. This [has] two implicated disadvatages. 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 update.

Secondly, the bolts are connected [one by one]. 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 [device] an invalid architecture due to small mistakes as mistypes and not updating all bindings on a refactor.

3.3.4 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 streaming micro-component application in mind. The spouts and bolts provide us with the [perfect] building blocks to design an iterative application with separation of concerns in mind, while the built-in stream mechanism provide the needs for a real-time distributed application. We will however need to account for the lack of [expose points] and the tedious process of specifying each and every bolt connection.

Though Storm [contains] the means for building a distributed accumulator process, we will not [use] it. Instead we will base our accumulator on Apache Spark Streaming. The reason for htis 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.

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, [we chose it] 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 [great] [incumbrance], being the lack of topic-level order guarentee, is not of grave [importance], because of the before mentioned [fix]. The hindrence cna be overcome by including timestamps or sequence numbers in the messages. Moreover, the Spark calculations most likely will not require order retention. The reaseon for htis is that most computations will consist of a *reduce* step, which requires the reduction operation to be both associative and commutative[?]. Therefore the message order is of no importance.

3.4 Design

We will adapt these technolgies by composing them using adapters and abstracting the solutions. By abstracting the technologies we shield the internal implementation details, simplifying [building] by the user. We will [supply] the user with some scaffoldings for bolts [intended for] different types of data flows and reductions. Additionally, on some [points] these solutions were abstract since they were intended for many [unforseen] usages. Since our [application] features some commonalities which were considered variations when designing the original technologies, we can implement some functions which were intentionally left unimplmeented originally. This will reduce the [implementatory] effort required, again simplifying usage of the platform.

3.4.1 Micro-service architecture

3.4.2 Scaffolds for micro-services

Distributor?

3.5 Discussion

4. Resource Distribution Model

4.1 Requirements

In this section we will investigate the requirements for the RDM. We will achieve this by performing an commonality/variability analysis [?]. This will reveal what the common features are on which we may depend and the variation which we will need to account for.

4.1.1 Commonality/variablity analysis

Definitions

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 of a system.

Commonalities

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

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) 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.

4.1.2 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.

4.1.3 Justification

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

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

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 4.3.2.

4.2 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.

4.3 Solution

4.3.1 Solution options

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 **??** available for different types of problems, a suitable solver should be able to be found or developed.

4.3.2 Solution choices

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 adapt the search algorithm to conclude not only the valid compositions but the optimal solution, given some heuristic function.

4.4 Design

4.4.1 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 **??**. To illustrate the application of this meta-model, an example of an instantiation of the model can be found in 4.2.

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

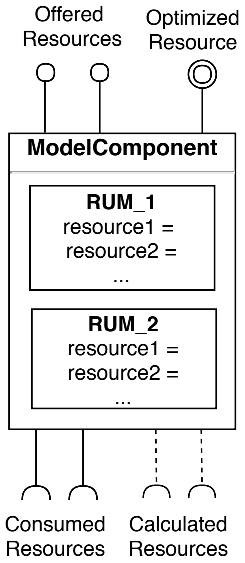


Figure 4.1: Notation of an RDM component with RUM's

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,

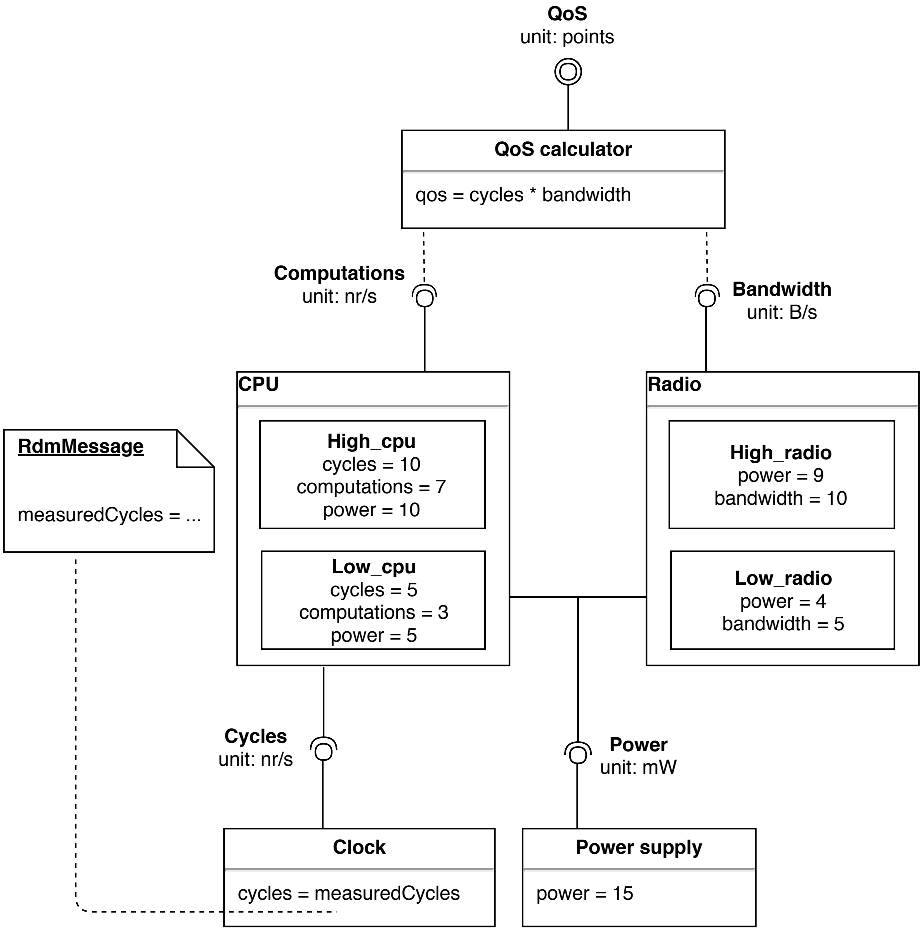
Consume Indicating that the component consumes an amount of the resource,

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.



Constraints:
$c_1 : cycles_{clock} >= cycles_{CPU}$
$c_2 : power_{power\_source} >= power_{CPU} + power_{Radio}$

Optimize:
$max(QoS)$

Figure 4.2: Example instatiation of the RDM meta-model with a CPU and a radio

A special case 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.

4.4.2 Solving the model

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 score.

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 heuristicly. 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:

- 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 application illustrates 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 $\vec{a}_{cur-1}$ and $x_{cur} = a$. If it is, go to Step 2a.
   (d) Instantiate $x_{cur} \leftarrow a$ and go to Step 1.
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[**?**]

4.5 Discussion

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.

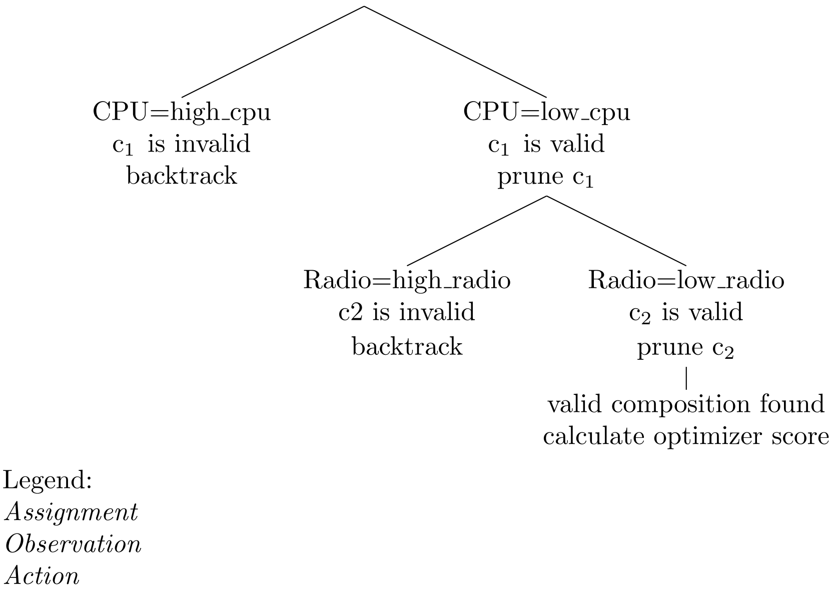


Figure 4.3: Application of backtrack search on RDM of Figure 4.2

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 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.

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

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. Validation by case study

6.1 Claims

what claims to validate and requirements implied by those claims

6.2 Case study

6.2.1 Background

Information of Nedap N.V. and product under investigation (SENSIT)

6.2.2 Description

Description of the case
Justification of the case with regards to the claims

6.2.3 Implementation

6.3 Evaluation

7. Conclusion

7.1 Discussion

7.2 Conclusions