4. 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 exporing the genral 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 furter implementation details. We will then conclude by discussing the success, applicability, disadvantages and deficits of our proposed solution.

4.1 Goal

Large sensor applications send immense amounts of 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.

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. We will achieve this by performing a commonality/variability analysis (C/V analysis) [**?**] of the problem domain. 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 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

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 device or an internal process.

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

Commonalities

C1.1 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.

Variety

V1.1 The first variety we encountered is the basis 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.

V1.2 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 general reports. Either by an in-memory component with an API or by persisting it to intermediary permanent storage.

(b) Analysis invokes an immediate feedback response to the application or a command & control service of the application

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

V1.3 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.

4.3 Requirements for the proposed software platform

R1.1 The platform should enable the 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 of a 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

4.4 State of the art

[TODO] analyse bestaande platforms en deficiencies uit research topics paper

4.5 Exploration of the solution domain

4.5.1 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 platorm 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.

4.5.2 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.

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

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

Table 4.1: Summary comparison of RabbidMQ and Kafka

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 4.1. 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.

4.5.3 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

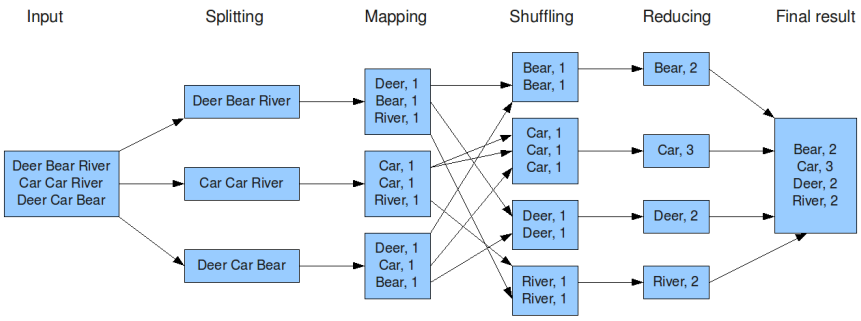

Figure 4.1: The overall MapReduce word count process[**?**]

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 4.1.

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 [**?**].

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 4.1: MapReduce example of Figure 4.1 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 4.1 in Apache Spark using Java in Listing 4.1. 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.

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

4.6 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 imple-

ment some functions which were originally intentionally left unspecified. This will reduce the implementation effort required, again simplifying usage of the platform. [**?**]

4.6.1 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 4.2. 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.

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

Listing 4.2: 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 4.5.4 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 4.2.

Two interesting remarks should be made, as apparent from Figure 4.2. 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.

4.6.2 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

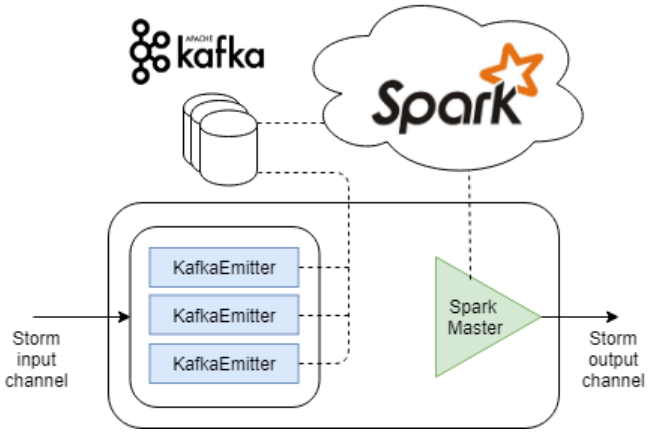

Figure 4.2: Graphical depiction of the distributed accumulator process

platform. We will first describe the base functions shared by all components, before discussing them more in depth individually. The

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)* method[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 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)*

[1] actual arguments have been omitted due to simplification

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.

HistoricProcessor

The HistoricProcessor 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 4.6.1. 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.

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 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

4.6.3 Demonstration by example topology

TODO demonstration by actual case

4.7 Discussion of the proposed software platform

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

Ease of adoptation

The first point of focus is the ease of adoptation provided by the platform. 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 programmign 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 concepts of topics. This allows our example topology described in section 4.6.3 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 **??** 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 4.5.4 justify the deployment of these additional technologies. Finally, when this scalebility 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.

Completeness of the processor components

[Todo] volwaardigheid analyseren aan de hand van QoI metrics.

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.