CHALMERS

Distributed and Online Advanced Metering Infrastructures
Data Validation using Single-Board Devices

Master of Science Thesis in Computer Systems and Networks

Jonas Sandström

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2015

Distributed and Online Advanced Metering Infrastructures
Data Validation using Single-Board Devices

Jonas Sandström

Examiner: Marina Papatriantafilou
Supervisors: Vincenzo Gulisano and Magnus Almgren

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden June 2015.

Abstract

Traditional electrical grids are evolving to information-carrying cyber-physical electrical grids, also referred to as Smart Grids. One of the enablers of this shift is Advanced Metering Infrastructures (AMIs), networks of heterogeneous meter devices that provide information and remote control to energy suppliers. AMIs produce a flow of information carrying the health and power consumption of the infrastructure. This information flow is usually provided at predefined time intervals and concerns large amounts of data. For instance, if one hundred thousand meters are deployed in a city and energy consumption readings for each household are reported every hour, 2.4 million consumption readings per day need to be processed. Electricity suppliers can use this information stream in novel applications, such as real-time pricing and demand-based production. Unfortunately, the correctness of the energy consumption data stream cannot be taken for granted since there are many potential error sources such as faulty devices, wrongly calibrated devices, lossy communication protocols, or fraudulent users, among others. Hence, there is a need for validation before significant decisions are made based on this data. Of importance is that the validation is performed in a real-time fashion with low latency, to deliver up-to-date information. Needed validation may change with the specific AMI or with different error types. Thus, the validation need to have the possibility to consist of a set of rules and be reprogrammable e.g., by adding, removing or modifying existing validation rules. In order to be fast and scalable, with the increasing number of households and finer time granularity, the solution requires the validation to be distributed and parallel. Those specifications can be met by using data streaming. Notice, to accomplish a deployment of such a distributed system in an AMI, Single-Board Computers (SBCs) could be used with low cost and energy use.

This thesis builds a prototype of such a system. It uses data streaming to validate the consumption data. Data streaming is necessary for online analysis. Stream Processing Engines (SPEs) consume the data stream immediately upon arrival by utilizing continuous queries. These continuous queries can be implemented to formulate validation rules, cleansing the consumption data. The implementations can be modelled to handle specific errors, which gives the system customizability. SPEs can process large amounts of data with low processing latency and in distributed and parallel fashion, and thus achieve high throughput. To make the system distributed and AMI deployable, a cluster of SBCs running a SPE will be used. Thus, also keeping the cost and energy usage low.

This thesis show that this is possible with an almost linear increase in processing capacity with each added SBC, i.e. in a nearly perfect scalable way.

Acknowledgement

I would like to sincerely thank my supervisor Vincenzo Gulisano for all the support and guidance, he has offered during the whole project. Big thanks to, my supervisor Magnus Almgren for the constructive feedback, regarding written and oral presentation. I would also like to thank my examiner Marina Papatriantafilou for her guidance during this thesis project.

Jonas Sandström
Gothenburg, June 2015

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Traditional electrical grids are evolving to cyber-physical electric grids. This shift involves the deployment of Advanced Metering Infrastructures (AMIs), networks of heterogeneous device such as Smart Meters (SMs) and Meter Concentrator Units (MCUs) that share information with energy distributors and providers. Energy consumption data produced by the AMIs' devices is known to be noisy, lossy and possibly delivered out of order. Incorrect, out of order or lost data occur due to faulty devices, wrongly calibrated devices, lossy communication protocols, or fraudulent users, among others. In order to cleanse the data before it reaches the utility management the consumption data is preprocessed through data validation analysis. This analysis depends on a set of validation rules, which are depending on the specific functionality of the AMI. If for example it is expected that all values should be at least zero or positive, negative consumption is not expected. Then the data can be filtered by a validation rule that removes all negative values. These validation rules are written by system experts that know the characteristics of the specific AMI and how the values of the consumption data are suppose to be formatted. The amount of consumption data produced by the SMs grows large as the AMI includes each household in a block, district, or town. Energy consumption readings are generated with a predetermined periodicity by heterogeneous devices that uses the same data format. Computing power is required to validate this burst of data, as the utility management relies on real-time data for applications as real-time pricing and demand-based production. A scalable system that can easily create and utilise validation rules is desired.

1.2 Problem

The task is to validate this large quantity of real-time data fast, to be delivered to the utility management with low latency. In order to be able to validate data from e.g. a city of 100000 SMs, for correct real-time pricing, the system needs to be distributed and parallel. Distributed as the network itself is constructed of distributed SMs in the periphery, to be fast in such a setting the validation as well must be distributed. The distributed computational resources needs to be connected to validate the data in parallel, to be as fast as possible. In the context of AMIs the system also need to placeable in the network and have low energy consumption, this can be achieved by using an already deployed device as a MCU or a dedicated one. To cover the need for different validation rules for different AMIs and faults the rules need to be changeable. The rules also needed to validate the data with regard to data from single SMs, correlated data from many SMs or external data. For such a system to be affordable and have decent deployment costs a possible solution would be to use distributed and parallel streaming, running on Single-Board Devices (SBDs). A SBD is dimensionally small and has most of the functionality of a regular computer, built on a single circuit board.

To reach a working prototype the validation must be implemented for streaming, with the properties of distribution and parallelization in mind. This can be achieved by using a Stream Processing Engine (SPE) with those features. A SPE continuously process the input stream of data by running continuous queries, in these queries the needed validation can be implemented. Then the SPE has to be deployed on the

SBD. If high processing capacity is required the devices can be connected into a cluster, by connecting them through a switch. This then requires that the validation is written so that the data stream is partitioned, according to SM. Causing each SBD to only process data from a certain set of SMs. The task is to process the data stream as close to real-time as possible, consequently the evaluation criteria will be throughput and latency of the processed data. The overall challenge is to achieve higher processing with additional SBDs, without distorting the data in this parallel and distributed system. Minor challenges are to measure the evaluation metrics, tweak the system for good performance and to install needed software and hardware, so that they run smoothly together.

1.3 Solution Overview

SMs produces a data stream of consumption measurements. To process the data stream a SPE will be used, this to avoid storage before processing as in a Database (DB) system. The SPE process data by continuous queries defined as Directed Acyclic Graphs (DAGs) of operators. These queries are then executed continuously by the SPE. Because of the query format validation rules can be compiled of standard operators provided by the SPEs. Use of SPEs are popular in online data analysis applications as Twitter, Spotify and Yelp. There are several different SPEs that are built and specialised in specific areas. The validation in this thesis needs to be as fast as possible therefore the SPE must have the possibility to process the stream in a parallel and distributed fashion. For the implementation the thesis use the SPE Storm, which is open source, has a large community, provides parallelization and distribution. Standard operators in Storm are spouts and bolts. Spouts are from where the data are flowing into the DAG, called topology in Storm, and bolts are the computation component. Bolts are used to compose validation rules. Storm has the ability to parallelize and distribute its queries among threads, cores, processors and computers. This results in data processing with high throughput, as well as the possibility to leverage it by adding more processing entities. This thesis use a dedicated computation device, Odroid-U3, in order to be able to test how good the system works. The Odroid is a small Single-Board Computer (SBC), that has low electrical consumption. To take advantage of Storm's parallelization properties and the Odroids collected computation capacity, a cluster of Odroids will be connected.

1.4 Evaluation

The benefits of running data validation on top of the SPE Storm in a cluster of Odroid-U3 SBCs are going to be evaluated. Metrics used to evaluate the system will be throughput, based on tuples/second, and processing latency. Through validation the thesis want to show the scalability property of the system, that throughput increases and latency is retained with every added SBC. The validation rules implemented on top of Storm will be analysed to demonstrate the scalability and added computation power as the SBC cluster enhances. A set of three validation rules will be implemented to analyse the system properties regarding simple single tuple computation as well as computations based on a collected set of tuples.

1.5 Organization of the Thesis

The organization of the thesis is as follows. Firstly, the background concepts needed to understand the rest of the thesis are presented in Preliminaries 2. Work already presented that is related to the subject in this thesis can be seen in Related Work 3. In Design 4 and Implementation 5 the solutions for the problems of this thesis can be seen. The evaluation of the prototype system, in form of throughput and latency, is presented in Evaluation 6. Further improvements and ideas are presented in Future Work 7. Findings amassed during the project is presented in Conclusions 8. The abbreviations used in the thesis can be found in List of Abbreviations 9.

2 Preliminaries

The following sections will provide an introduction to the general system with focus on this thesis' challenges and motivation. The level of abstraction moves from an overall system model of Advanced Metering Infrastructure to the specific techniques used to solve the challenges. Data streaming and SPEs are the techniques used to enable fast validation of electric consumption data. The idea is to create an understanding to enable the reader to later be able to follow the implementation process.

2.1 System Model: Advanced Metering Infrastructure

Electrical power grids are evolving, previously they only transported electricity from plants to end users. Today they also offer applications in for example communication and data processing, which has coined the term Smart Grid. Products that are provided and areas that have been researched are real-time pricing [2], demand and response based consumption [14], costumers' privacy [27], vulnerabilities granted on the basis of remote control [9], intrusion detecting sensors [7] and consumption monitoring for costumers [18]. A smart grid is a communication and monitoring network layer connecting all entities of a power grid from power station to end consumer.

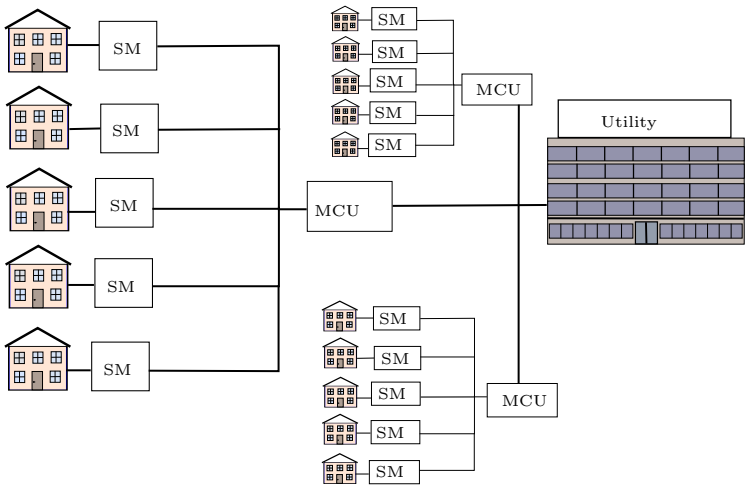


Figure 1: Advanced Metering Infrastructure - with its components SMs, MCUs, and the Utility

Advanced Metering Infrastructure (AMI) refers to a full measurement, collection, and analysing system that can handle different types of products such as water, gas, electricity, and others. Consumption is measured and reported to the utility management. AMIs for electrical consumption consist of four parts Smart Meters (SM), Meter Concentrator Units (MCU), communication network and utility management, which can be seen in Figure 1. In this thesis only the electrical consumption of the households are considered, which removes the possibility for the customers to introduce energy into the power grid. However, if the grid owner allowed it the customers could produce energy by using solar panels, wind power or another energy source and introduce the excess into the power grid. SMs provide electrical consumption readings that are forwarded through the communication network by the MCUs to

the utility management. In reality these networks can be connected in various network topologies such as direct, tree, mesh or others. In direct connection everything is directly connected to the end-point, no interconnection between nodes. While a tree topology has a hierarchy where the number of nodes grows with the layers, a break in connection between the layers could result in lost connection for many nodes. In a mesh topology all nodes are connected and a break in one connection cannot lead to system failure.

A SM's essential purpose is to measure costumers' power consumption and report it to the service provider as close to real-time as possible. In AMIs SMs enable two-way communication, which provide the options of resetting the meter, the internal clock or the measurement interval. The monitoring period for electricity consumption can be fixed or vary over time.

Carrier of the information through the AMI network is the communication network. The network is often a combination of several networks and techniques depending on hierarchy, size and landscape. The entities can be Home Area Network, Neighbourhood Area Network and Wide Area Network. Both wireless and wired communication are used. The key is that the information is delivered in real-time in a secure measure, without the possibility for fraudulent interception.

2.2 Data Streaming

A data stream is an unbounded volatile sequence of data produced over time. Data streams are usually used when an entity produces a huge flow of data that has to be processed with in real-time. Application areas are financial analysis [25], online advertisement metrics [3], fraud detection in continuous data flows [11] and a defence framework against DDoS attacks [8]. These application are all dependant on the data stream processing paradigm, which is an alternative to the store-then-process paradigm of databases (DBs). In an AMI the stream is uniform and consists of *tuples*, which are an ordered list of fields. Each field in the tuple is named and the collection name including all fields is called a *schema*. In this context the schema is a trinity and consisting of the following fields: time stamp, SM ID and energy consumption, see Table 1. A SM produces new tuples with a periodic interval, for example every 12 hours every day all year. Nevertheless, this interval can be altered if for instance the price plan changes, to be calculated each hour instead. The granularity is decided based on the need from the running applications and stipulations by the government. For this thesis the granularity is decided by the data source, a simulator constructed by Richardson et al. [22], where a new measurement is provided each minute.

Description	time stamp	SM ID	energy consumption
Abbreviation	t.s.	id	e.cons

Table 1: Smart Meter tuples' schema

Data streams sequences of tuples are more interesting if they can be related to each other, rather than to extract information from them one by one. In order to build a collection of information, the data needs to be related to something. In DB systems

tables of tuples are joined together by queries, to obtain information from different parts of the DB. Queries in DB systems are only applied on stationary data and used when the operator decides to retrieve the information. In real-time systems such as SPEs a query is in constant operation in main memory as new data arrives. Information changes constantly and the result of the query is not definitive, it alters as time passes.

In DB systems queries are usually defined in a relation based language, such as SQL, where different operators are applied on the tables in the DB. Stream processing usually defines a *continuous query* as a Directed Acyclic Graph (DAG) of operators. A DAG is a graph where the edges between the vertices have a direction and no loops. A continuous query defines between which operators data will flow and in which direction. In general, an operator takes one or more streams of tuples as input, performs some computation, and outputs either one or more output streams of tuples. Stream processing operators are separated into two categories: *stateless* and *stateful*. The differentiating factor is that stateless operators do not maintain a state that evolves accordingly with the tuples being processed. Stateless operators compute a result for each tuple separately; filter, map and union are such operators. Filter discards tuples that does not meet a criteria. Map manipulates the tuples fields. Union merges tuples from from different input streams to one output stream. Stateful operators perform computations on the set of tuples that are in their sliding window at the time. A sliding window is either time-based or tuple-based. The time-based sliding window has a start, end and a slide, which defines how much it will move between each instance. The tuple-based sliding window is defined by the number of tuples to collect and the number to discard between each instance. Parameters used to define a sliding window are WindowType, SizeOfWindow, and SlideAmount. Statful operators are aggregate, join and sort. Aggregate computes functions on a field such as mean and count on the set of tuples in it's sliding window. Join matches tuples in its sliding window based on the value of the tuple fields. Sort produces the tuple with minimum value, based on a specific field, as output. Next is an example of a time-based sliding window and a tuple-based sliding window. In the first example the time-based sliding window collects tuples from the last 5h and slides 2h for every new instance, see Figure 2. In the second example the tuple-based sliding window collects two tuples and discards one for each new instance, see Figure 3.

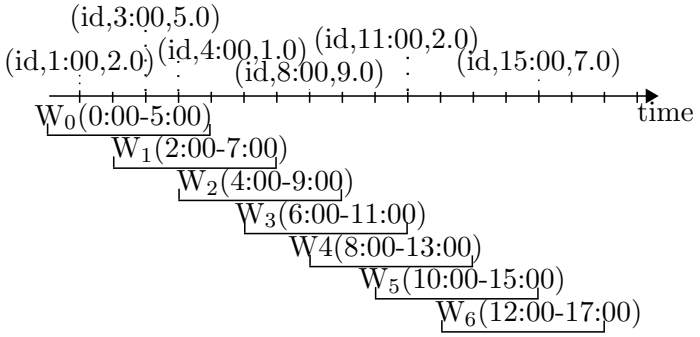


Figure 2: Example of a time-based sliding window with size of 5 hours and advance of 2 hours

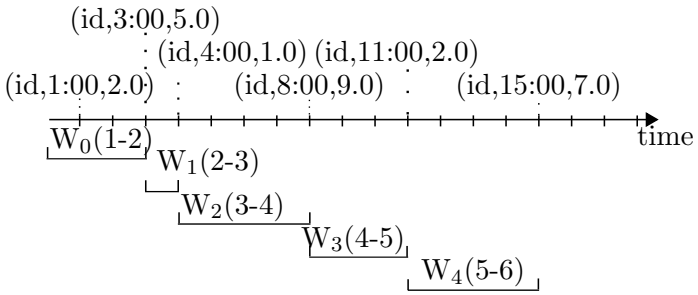


Figure 3: Example of a tuple-based sliding window with size 2 and advance 1

In the following section the operators that are later used by the validation rules in this thesis are presented. While the other common data streaming operators can be found in the Appendix 10.1. When discussing these operators further an abbreviation of the tuple will be used: (id, t.s., e.cons). In order to make further explanation of the validation rules easier to understand and overview, a schematic will be presented. Each new rule has the following schema: operator name(function)(input streams)(output streams), with the abbreviation op.n(func)(in.s)(out.s).

Filter
Filter is a stateless operator that passes or drops a tuple based on the value of a field. The schema for the operator is filter(terms)(I)(O). For instance the operator can be used to filter consumption readings less than 0kWh, negative consumption.
$filter(e.cons \geq 0)(I)(O)$

Map
Map is a stateless operator that manipulates a tuple into one or more outgoing tuples with a different schema. The schema for the operator is map(out.field $\leftarrow$ func(in.field),...)(I),(O), one or more input fields are manipulated by a function and thereby gives the output tuple a different schema. For instance the operator can be used to change the time stamp from seconds to minutes.
$map(id \leftarrow id, t.s. \leftarrow t.s./60, e.cons \leftarrow e.cons)(I)(O)$

Aggregate
Aggregate is a stateful operator that collects a set of tuples to compute functions as mean, median, count, min, max, first or last. The accumulation is either time or tuple based and performed over a sliding window. The schema for the operator is aggregate(WindowType, SizeOfWindow, SlideAmount, out.fields $\leftarrow$ function(in.fields), Group(in.field))(I)(O), each aggregation is done over a sliding window of a certain type. That has a size in time or tuples and that slides a decided amount each time the sliding window advances. A function is performed on the selected input field that spans the size of the sliding window. If the Group function on a input field is used each instance of that field is assigned its own sliding window. For instance the mean consumption of each SM can be calculated with $aggregate(time, 1h, 10min, mean \leftarrow mean(e.cons), Group(id))(I)(O)$, the mean consumption the last hour is calculated every 10 minutes for each SM.

The output fields will be (id,t.s.,mean).

2.3 Stream Processing

SPEs are the entities that run applications to process the data stream, presented in the previous section. Processing is done by running continues quires on the SPEs. The quires are run in main memory. The SPE could interface with a DB, but in this thesis that possibility is not considered. Stream processing has evolved from centralized systems to distributed parallel real-time systems. One of the most used SPEs is *Storm* [26], which is used in this thesis. Following is an explanation of the used terminology. Storm is a SPE that both provides a cluster setup, to distribute the work among the data processing nodes, and an API library, for the implementation of the data manipulation on the incoming stream. Storm claims in the documentation that it can process as many as a million tuples per second, provide scalability, fault tolerance and process each tuple at least once.

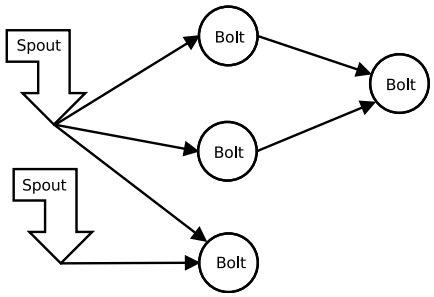


Figure 4: Storm topology - with its components spouts and bolts

SPEs as presented in Section 2.2 process queries consisting of DAGs of connected operators, in Storm these objects are called *topologies*. Data passed through the topology is of the form of a stream, an unbounded sequence of tuples. These tuples are named lists of values, that can be of any object or type. The processing logic entities provided in the topology are *spouts* and *bolts*, as can be seen in Figure 4. A spout is the source of a stream that takes input data and emits a stream of tuples into the topology. A bolt performs calculation or manipulation on the incoming stream or streams and might emit outbound stream or streams. To do advanced stream transformation many bolts are preferred, as the distribution can bring better throughput. In this project's implementation it is in the bolts the data validation is performed. The DAG of spouts and bolts are packaged into a topology that can be deployed in the Storm cluster. A topology can be executed on a single computer in local mode, mostly used for testing, where the cluster explained below is not used. However, to initiate distributed execution of the topology, it has to be injected into a running cluster.

A cluster consists of the following necessary entities *Nimbus*, *Zookeeper* [30], *Supervisor* and *UI* run on one or several computers, as can be observed in Figure 5. Nimbus is the master node daemon which distributes the incoming data among the worker nodes and regulates failures. At each worker node there is a Supervisor daemon that assigns the work to the worker processes, which is the actual data processing

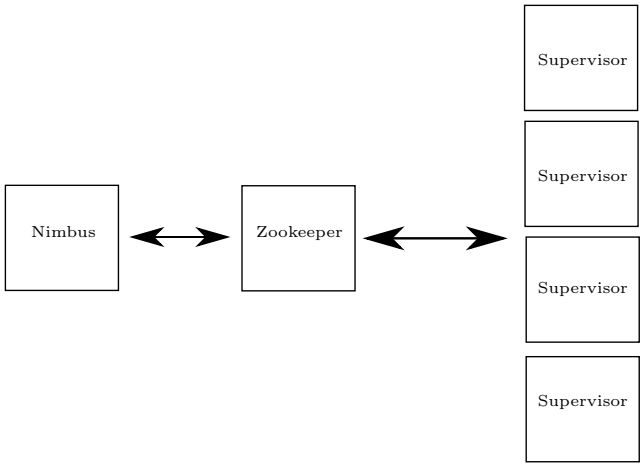


Figure 5: Storm cluster - with the necessary daemon processes

entities. The amount of work that can be performed at each node is bound by the set of threads that can be processed in parallel. Zookeeper is the coordination intermediary between the Nimbus nodes and the Supervisor nodes, it also keep track of the state of the system. System state is kept in order to restart Nimbus and Supervisor nodes without the loss of any data, as Storm claims to process data at least once. UI is Storm's graphical interface that is run at the Nimbus node to give an overview of the system. The cluster can be deployed in local mode, where everything runs on single machine, or in distributed mode, where some or all entities of the Storm cluster are run on different machines, the usage of the data processing system does not change.

2.3.1 Parallelism in Storm

Storm has many parameters that can be adjusted to increase, or if done wrong decrease, performance. To get a grip of how to tweak the parameters the reader need to understand how a topology is processed in a Strom cluster. The entity in Storm that actually performs execution is *tasks*, which executes a bolt or a spout. A task is run by an *executor*, which is a thread spawned by a worker. Each executor runs one or more tasks for the same bolt or spout. A *worker* process runs a subset of one topology and can only run executors for that topology. A topology can therefore be processed by many worker processes across the cluster.

Figure 6 shows the hierarchy between the different processes. There are some observations that can be made for an optimal setup of a cluster. The amount of tasks that is most favourable to run on each machine is dependent on how many threads that specific machine can handle at once. The default setup in Storm is to run one executor for each task, run more and the executor must switch between tasks. This also has to do with the usage of the number of worker processes on each machine, the workers only serve one topology. Hence, to deploy more than one worker per topology per machine will not give better processing capacity. There will only be extra overhead and not more processing power. To get better throughput there are two options either to increase the processing power on thread level or to get more threads. This means that the only solution will be to get more or faster machines

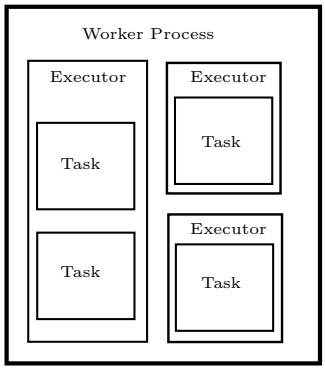


Figure 6: Relationship between tasks, executors and worker processes

to increase the throughput, if all other settings are optimal. However, there are many settings in Storm that can be changed without new hardware, the buffer sizes between components, how many executors and tasks that are allocated to each part of the topology.

3 Related Work

The following section will present previous work related to the subjects of this thesis AMIs, data streaming and data validation. This to show that the subject matter of this thesis is of importance, to cleanse electrical consumption data coming from the SMs before it reaches the utility management. First an overview of research areas with connection to AMIs, in Section 3.1. To give the reader knowledge about the importance and range of the subject area. Thereafter usages of data streaming are presented in Section 3.2, with the intent to present the broad use of the technique used in this thesis. Last in Section 3.3 the benefits of using data validation is discussed.

3.1 Advanced Metering Infrastructures

In AMIs quite a lot of work can be found, as it is an important topic, in the fields of security, privacy, data volumes and consumer applications. The AMIs' beneficial side is the possibility to use the information flow from the SMs to lower electrical consumption. Research has been directed to the area of costumers' electrical consumption, with regards to usage and price. To enable energy awareness and conservation an example real-time energy monitoring application system have been presented by Mikkola et al. [18], to advice the users on their habits. An evaluation of a real-time pricing system is presented by Allcott [2]. The conclusion is that households conserve energy during peak hours and did not increase average consumption during off-peak time. Another approach to minimize energy cost is presented by Guo et al. [14]. The paper focuses on the consumption of a whole neighbourhood including energy storage devices, renewable energy generation and electric utility, where a central entity is responsible to secure enough energy for the neighbourhood. Even if these papers are not about data validation, they are about the data in AMIs and if such data is validated and cleansed it is better for all these systems.

Nevertheless, the AMIs also have some less positive properties that have been researched, which have led to some proposed solutions. Security is an issue related to AMIs, where the information sent through the network system can be altered. It cannot be taken for granted that the network layer is safe. Bartoli et al. [5] identify security problems in the communication and as a solution implements a secure aggregation protocol. Berthier and Sanders [7] have proposed a specification based sensor that monitors the traffic in the AMI, to identify threats in real-time. To ensure that devices run securely they have implemented a set of constraints on the transmission protocol, violations of the specified policy will therefore be detected. Mohammadi et al. [19] proposes another security solution, an AMI intrusion detection system for known and future malicious attack techniques. Their solution is specific to neighbourhood area networks and the paper does not include a concrete implementation. Vulnerabilities are also granted by the remote control capability of SMs, these have been studied by Costache et al. [9]. The scenarios investigated are the frequency property of the grid to cause a blackout and the possibility for an adversary to drive the voltage out of bounds. As the AMI networks grow the data volumes increases, Dieb Martins and Gurjao [10] attack the problem of processing the large data flow from SMs, by dimensional reduction through random projection. The presented calculations are based on offline processing and no details are presented

for a real-time scenario. Another concern is the costumer privacy, that it should not be possible to identify a specific customer's usage in the data flow. Bekara et al. [6] paper also contains measures to keep the privacy in the AMI. Their proposal is an ID-based authentication protocol, but they do not implement it. Tudor et al. [27] investigate the customer datasets to make them more resilient to identification. To keep the customers data secure and private is most important. These papers are not directly related to this thesis' area, but it shows the breath of the subject area AMIs.

3.2 Data Streaming

Closer to this thesis topic, AMIs and stream processing, there is not as much related work, as the subject is narrower. Work that have been presented uses stream processing to implement systems for intrusion detection, real-time pricing, decreasing consumption and data volumes. The point is that data streaming is used in not just this thesis, but in quite a few other projects.

Gulisano et al. [12] proposes an intrusion detection system, METIS, with the same goal of protecting the AMI from attacks into the system from adversaries, as the systems that do not use stream processing. The system is implemented using Storm SPE and the analysis is done in a parallel and distributed system. Storm's parallel and distributed capabilities can also be used in this thesis, as the needed validation would benefit from being processed in a parallel and distributed fashion.

Lohrman and Kao [16] present a stream based system aiming to solve the problem of scalability in smart grid systems, with focus on real-time pricing and monitoring. They present a set of requirements to be able to handle the data volume from the SMs concerning scalability, availability, latency and data management. To address the requirements they propose parallel stream processing in clouds as the solution. The SPE used for the implementation is Nephele [28]. In order to demonstrated the system they implement a prototype real-time pricing application in a private cloud, with a dataset of one million simulated SMs. Each simulated SM communicates via its own TCP/IP connection with a cluster of 19 virtual machines running Nephele on top of the Eucalyptus Private Cloud [21]. With the set of one million SMs they are able to provide price updates every 10 seconds. This thesis will not implement real-time pricing nor deploy anything in the cloud, but it will try to scale the performance by other means. Hence, the scalability properties used by Lohrman and Kao can be taken into account.

Simmhan et al. [24] present a project in the area of smart grids, SMs, customer power consumption and the analysis of such data streams in real-time. To handle the onslaught of data, a cloud platform is utilized to perform stream processing, with the properties of scalability and low latency. A mechanism to throttle the rate at which the SMs produce data is also introduced, to lower bandwidth consumption. However, the paper does not include validation of the consumption data, as this thesis will use. Instead the focus is to decrease the data volume with adaptive rate control, throttling the generation of consumption data.

The stream pipeline used in their paper resembles a continuous query or topology, used in this thesis, and needs a SPE to make it productive. Simmhan et al. uses IBM

InfoSphere Streams [1] as their SPE and deploy it on Eucalyptus Private Cloud [21], with the possibility to later migrate to the Amazon public cloud. They achieve a 50% reduction in network bandwidth used, with the throttle mechanism operative in comparison with a static generation. However, the stream generation does not follow power consumption exactly, it both over and underestimates. This is concluded to derive from the use of tumbling instead of sliding windows, which discards all data in the window every time it moves. This thesis focuses on validation of data not on decreasing generation of measurements, similarities can however be found. In the use of stream processing and the similarity between alerts and validation.

As can be seen from the previous paragraphs there are a lot of usages of data streaming in connection with AMIs and the benefits of using it could be utilized when validating electrical consumption data. These advantages are foremost the ability to handle real-time data and the possibility to scale the system. Data streaming is a mature technique whose strengths have been utilized in other projects such as quality of service in IT systems [17], fraud detection [11], prevention of distributed denial of service [8] and in an advertising system [3]. Application areas for stream processing are very large as can be seen with this sprawling selection.

3.3 Data Validation

Data validation is used in many scenarios where the correctness of data needs to be checked such as data streaming from industrial equipment [29], digital particle image velocimetry [23], stream data management benchmark [4], among others. Validation is common and used in many systems to check the plausibility of data, to then correct or discard it. The paper by Gulisano et al. [13] has the same focus area as this thesis, validation of electrical consumption data produced by SMs in an AMI. The authors show that their system can validate measurements from millions of SMs that provide hourly readings, with a throughput of 1000 to 8000 tuples per second. To cleanse the data a set of validation rules are presented, that filter or interpolates missing consumption data specific to AMIs.

The three validation rules presented rely on the fact that the incoming stream data is of tuple type and has the schema <time stamp, SM id, consumption>. Rule number one is a filter that discards incoming tuples based on a predetermined value. Rule number two collects three hours of consumption data from a specific SM, calculates the mean and discard the tuples that exceeds two times of the mean. Rule number three interpolates missing tuples if the time between tuples from the same SM exceeds one hour. The implementation then utilizes the distribution and parallelization properties of the SPE Storm. Continuously the system can process data from 5-25 million SMs with an hourly measurement interval. Throughput is shown to grow as the batch size of data grows. Latency grows also with the batch size and is affected negatively if the validation rule collects data, but is in the range of milliseconds.

This thesis will build upon the results from [13], but use another set of validation rules. The main difference is the use of hardware, their system consist of a common CPU, while this thesis will use a cluster of Odroid SBCs. The reason is that the

Odroids use less energy and physical space, therefore they are used more easily in an AMI. Another benefit is that the Odroids are cheap in comparison with a common computer. Through the use of an Odroid cluster the capabilities of distributed and parallel stream processing can be better utilized, than when it is used on a single computer.

4 Design

This section introduces how the thesis utilize the data streaming paradigm, through the use of stream operators that consumes the data stream to perform parallel computation. Operators are connected, with a determined direction of the data flow between them, to compose continuous queries. Each continuous query is built by using the operators presented in Section 2.2. These continuous queries are composed of a specific set of operators to construct each validation rule, to correct the data in the stream. The selection of validation rules are based on those presented by Gulisano, Almgren and Papatriantafilou [13]. Their rules are based on a set of Validation, Estimation and Editing (VEE) rules used in meter data managing systems, which collect and store data from AMIs. Validation refers to rules that cleanse data from corrupt values. Estimation refers to rules that can produce missing data. Editing refers to rules that can edit historical data. The validation rules presented in this thesis will cleanse data and interpolate missing data. Validation rule *Fixed Filter* discards all tuples with a negative consumption value, for further explanation see Section 4.1. Validation rule *Varying Filter* discards all tuples that have a consumption value that is less than two times the median, aggregated during a sliding window, see Section 4.2. Validation Rule *Interpolation* interpolates missing tuples into the stream, based on the time between two consecutive tuples, see Section 4.3.

4.1 Validation Rule 1 - Fixed Filter

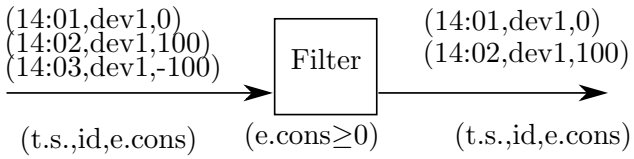


Figure 7: Example of the filtering condition of the validation rule - Fixed Filter

The validation rule Fixed Filter is a filter operation, that has one input stream and forwards an output if the filtering condition is met. This filter rule focuses on the fact that consumption data cannot be negative, a consumer cannot contribute electricity into the power grid, as stated in Section 2.1. There are two options for consumption data either the consumer does not consume any power, 0kWh listed, or the reported consumption is a positive number. To describe each validation rule in a formal manner the definitions presented in Section 2.2 will be used. The formal representation of Fixed Filter uses only one operator, filter. Fixed Filter is defined as $filter(e.cons \geq 0)(I)(O)$, each tuple's electrical consumption value is compared with zero and the condition is met when the consumption is either zero or positive. The rule takes input from one stream and produces output to another. An example of the rule can be seen in Figure 7. The input stream of consumption tuples comes from the left to be validated by the filter rule. That then forwards the tuples that have met the condition of non-negative consumption.

Fixed Filter cleanse the data stream from incorrect values by discarding them, which is the basic property of all validation rules of VEE. Thus, also the more complex validation rules must utilize this property at some point in their continuous queries.

If the data stream needs to be cleansed its tuples' field values have to be compared to a condition value. This condition value can either be fixed, set before the validation is started, or varying, recalculating during the run of the validation. A fixed value is produced by calculation, estimation or observation prior to the start of the validation. A varying value is produced by collecting tuples based on time or number. The collected tuples are then used to produce a value based on one or more of their fields, this value changes as the tuples in the collection is removed and added. In Section 4.2 an example of an extended filter is shown, that does not only rely on a single filter operator.

4.2 Validation Rule 2 - Varying Filter

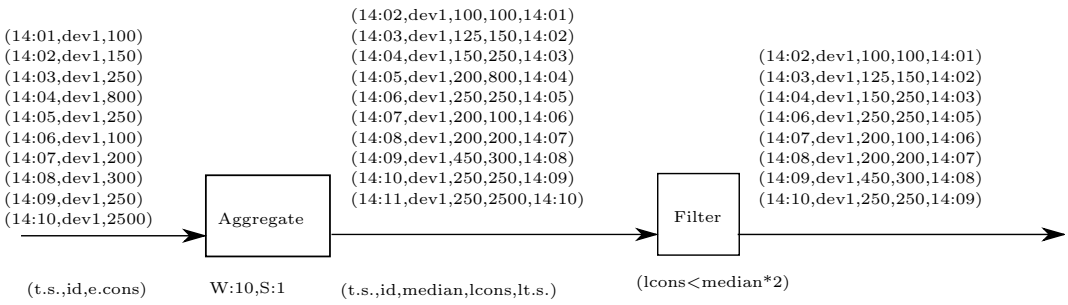


Figure 8: Example of the evolving filter rule - Varying Filter

The validation rule Varying Filter is a filter operation as well, but the condition is not a fixed value. This rule is used to cleanse the data stream from extreme values, which can be induced by faulty SMs, glitches in the communication network or other error sources. The filter condition value is calculated based on median value, as it prevents extreme values to make an impact. If a mean value was used a single extreme value could suddenly change the filter condition, which then would not discard other incorrect values. This could be corrected by using a large set of values to calculate the mean, but that would require the window size to grow and additional memory to be used. To construct the validation rule a time-based aggregate and a filter from Section 2.2 are used. Aggregate are used with a window size of 10 minutes, that slides every minute, while taking input from one stream and producing an output to the filter. The granularity of the electrical consumption data used in this thesis is one minute, to achieve real-time computation the advance of the window is also performed each minute. If the granularity and the advance were not the same an additional stream operator would have been needed. Join could have synchronised the streams, but it would have taken additional computer resources. With each advance of the window the aggregate produces the median value form the tuples' electrical consumption fields. Each SM has its own window as the aggregate operator is grouped-by the field SM id. The tuple produced by the aggregator carries the median value that is used by the filter, as condition value. Output from the aggregator is input for the filter, that then discard all tuples that carries a consumption greater than two times the median. Filter then produces its output to a single stream. An example of the rule can be seen in Figure 8, where the tuples coming from the left are first aggregated and thereafter filtered. The ag-

gregation is shown for a ten minute window, after that the oldest tuple is removed. A result is produced each minute and the set of values that is considered for the median calculation grows up to ten, the size of the window. The formal representation of Varying Filter based on the definitions from Section 2.2 is
$aggregate(time, 10min, 1min, median \leftarrow median(e.cons), lcons \leftarrow last(e.cons),$
$lt.s. \leftarrow last(t.s.), Group(id))(I_{stream})(O_{a1})$
$filter(lcons < median * 2)(O_{a1})(O)$

Varying Filter is an example of the usefulness of aggregation, to be able to calculate new values based on the incoming data. This can be done by either collecting tuples based on the time, for example to produce a metric every hour for the total electrical consumption, or based on the number of incoming tuples, for example to produce an average consumption from one hundred synchronized SMs. This can be expanded to many other situations where the sum, count or other arithmetic operation of a real-time varying number needs to be calculated.

4.3 Validation Rule 3 - Interpolation

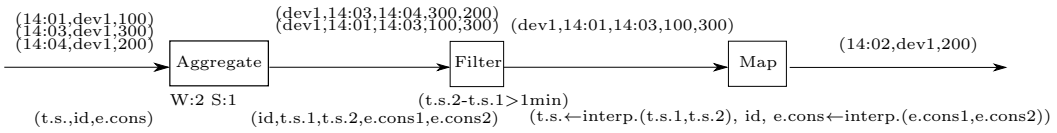


Figure 9: Example of the interpolation rule - Interpolation

Validation rule Interpolation, focuses on missing tuples in the data stream, this can occur because of glitches in the SMs or the network. Tuples are expected to arrive each minute from the SMs, as the granularity of the consumption data is one minute. If a tuple does not arrive in this timespan a new tuple is interpolated into the output stream of the query. In this case the validation rule is constructed of a tuple-based aggregator, a filter and a map operator. The aggregator has a window size of two tuples and slides with every new tuple after the first two, each SM has its own window as the aggregator is grouped-by id. Tuples outputted by the aggregator consists of the two last SM tuples, as can be seen in Figure 9. Filter then compares the two timestamps in the tuple, to determined if they deviate by more than one minute. If the difference is more than a minute the tuple is not discarded. Next in line is the map operator, that calculates the missing tuple's timestamp and consumption value. This is done by an interpolation function, that estimates the values based on the two existing tuples' values. Timestamp for the new tuple is set by calculating the time between the incoming tuples and adding that to the timestamp of the oldest tuple, $(t.s._2 - t.s._1)/2 + t.s._1$. The interpolated tuple's consumption value is calculated as the mean of the two collected tuples' consumption values, $(t.s._1 + t.s._2)/2$. Map also formats the output tuples schema, to correspond with the other tuples in the system. The formal representation of Interpolation based on the definitions from Section 2.2 is
$aggregate(tuple, 2, 1, t.s._1 \leftarrow first(t.s.), t.s._2 \leftarrow last(t.s.), e.cons_1 \leftarrow first(e.cons),$
$e.cons_2 \leftarrow last(e.cons), Group(id))(I_{stream})(O_{aggtuple})$
$filter(t.s._2 - t.s._1 > 1min)(O_{aggtuple})(O_f)$

$map((id, t.s., e.cons) \leftarrow interpolate(id, e.cons_1, e.cons_2, t.s._1, t.s._2))(O_f)(O)$

Interpolation can be used in many areas when something needs to be estimated, based on real-time data. To use real-time data the tuples needs to be aggregated, using a time-based or tuple-based window. The confidence of the estimation grows with the number of data points. With tuple-based windows the specific number of data points are known, but not how much time they span. In contrast, with a time-based window the exact number of tuples are not known, but the time they span is.

4.4 Discussion of Deployment Options

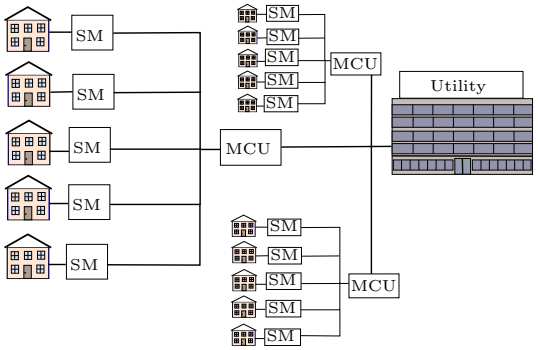


Figure 10: Overview of the deployment options in the AMI

These three validation rules Fixed Filter, Varying Filter and Interpolation can be deployed anywhere in the AMI, Figure 10, at the SMs, MCUs or at the utility head-end, because the data stream is partitioned according to the SMs. Fixed Filter does not consider from which SM the data is coming from, it filters each tuple individually. The two aggregating rules partitions the data according to which SM it comes from, by the use of the group-by functionality. The rules can therefore be deployed anywhere in the AMI and still produce the same result, which means that they are orthogonal to the AMI. The condition of orthogonality is also maintained by the use of data streaming, that partitions the data from a specific SM to always be processed by the same processing entity. In the scope of this thesis, it indicates that additional SBCs will not distort the data processing, but instead increase the capacity. Processing of the validation rules can therefore be done at the utility head-end or at a cluster of distributed SBDs.

5 Implementation

In this section the implementation of the validation rules and the platform used to measure their performance are presented. An overview of the system architecture can be seen in Section 5.1. The validation rule implementation is presented in Section 5.2 with each operator individually, as validation rules are built by connecting these operators together. Lastly in Section 5.2.1 the measurement platform used to measure and later evaluate the system is presented.

5.1 System Architecture

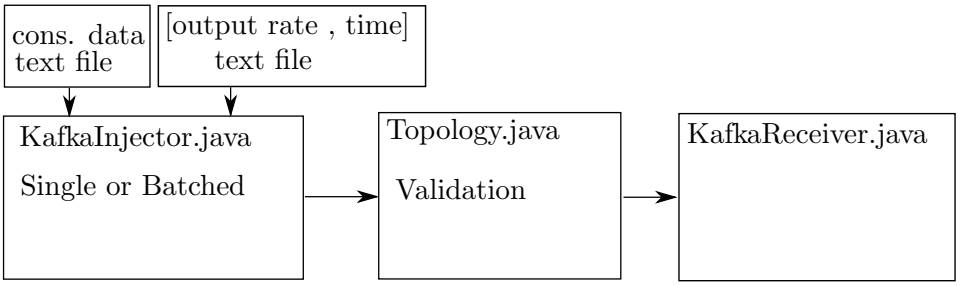


Figure 11: Software system for the prototype

The system architecture for the software system developed for the validation contains not only the validation, but also the injection of tuples into the validation rules and consumption of tuples from them. An overview of the system can be seen in Figure 11. The dedicated injector has been implemented to forward tuples, with electrical consumption values packaged in the schema $(t.s., id, e.cons)$ from Section 2.2, read from a csv file to a *Kafka* [15] broker. Kafka is a queue based message passing system, that is durable and scalable. The consumption values in the text file are from the consumption simulator. Tuples can be sent out either individually or in batches, with batching higher throughput can be achieved. The sending rate of the tuples is controlled by two parameters: injection rate and duration, that can be changed during the run of the program from a text file. The rate control is needed when the saturation of the prototype is sought. The validation rules consumes the tuples from the same Kafka broker that the injector has produced tuples to. After the validation is performed the tuples are forwarded to a different Kafka broker, that the dedicated receiver consumes tuples from. All programs are written in Java, as both Storm and Kafka provide most information for Java development.

The implementation was done with the intent to be modular. Other queue systems than Kafka would also work with the implementation, once they interface with the module for the validation. Consumption data could also be found by listening to a specific port and forward consumption messages received through TCP/IP. The validation rules are implemented in Storm topologies, but with the right interface another SPE would also work.

5.2 Implementation of Data Streaming Operators in Storm

To implement the continuous queries, from Section 4, Storm also uses the abstraction of operators connected by a directed data flow. In this section only the implementation of the used operators will be presented, as the queries are constructed by connecting those. Storm calls its continuous queries topologies. This thesis' three validation rules are using three different Storm operators: filter, tuple-based aggregation and time-based aggregation. Filter is used by Fixed Filter and Varying Filter, Interpolation has the filter functionality implemented in its aggregation. Varying Filter is the only user of time-based aggregation and tuple-based aggregation is solely used by Interpolation. Map's functionality of altering the fields can be performed by all operators in Storm. Both the filter and the tuple-based window operators comes from Storm Trident API, while the time-based window's implementation is based on an operator not provided in the common API.

Filter
Filter is implemented by using the BaseFunction, which operates on individual tuples. It is simple but useful, either it emits the received tuple or not. To implement a conditional filter set a fixed value or compose it from a field of the received tuple.

Tuple-Based Window
The tuple-based window with SM partitioning is implemented by first using a groupBy("id") on the stream, to give each SM its own aggregation window. Then the Trident Aggregate operator is used to collect the two consecutive tuples from each SM. All values are saved in HashMaps, for easy retrieval. The window advances with each new tuple and outputs an extra tuple if the time between the aggregated tuples is more than a minute.

Time-Based Window
The time-based window is also first partitioned with groupBy("id"), to secure a window for each SM. After that the implementation uses AggregateWindow, to correctly slide with time. The size of the window is ten minutes and it advances every minute. It saves the received tuples in HashMaps, that are later discarded when the window size is reached. When the window advances a new output tuple is produced, with the median consumption value for that SM. The median is calculated by sorting the consumption values from the HashMap and thereafter extracting the number in the middle of the list. If the list have an even amount of numbers, the median is calculated as the mean of the two middle numbers.

5.2.1 Measuring Platform

Evaluation of this thesis is based on performance measurements as mention in the Introduction, Section 1.4. In this section the implementation and architecture of the actual measurement system is introduced. The metrics of interest are the throughput, how many tuples/second each validation rule can handle, and the latency, how much time is added in milliseconds by using the validation rule. These metrics will

vary also based on the deployed hardware capacity. The justification to build a measurement system is that it is then possible to get performance measurements for the whole system, the validation topologies in Storm and its input and output queues. It should also be noted that it is possible to use this measurement platform with another SPE to get the throughput rate and latency, as the platform is modular.

Throughput is measured at two points in the system, to compare the actual injection rate to the rate at which the validation rule manages to process tuples. At both points the rate is measured as t/s and saved into two text files. The injection rate is noted just before the Kafka injector sends the tuple. Storm notes its throughput as the rate at which it can clear its input queue, thus the rate is noted just after the tuple has passed the spout.

Latency is measured by comparing the system timestamp at the injection of the tuple to the system time when the validation is done. This is achieved by adding a field for the system timestamp to all tuples. Before the tuple is sent by the Kafka injector it adds the system time, the system time in the tuple field is then subtracted from the system time when the tuple has passed through the validation. The calculated value is saved by the Kafka receiver in a text file. In order to use this method to measure the latency all hardware systems must be synchronized.

6 Evaluation

This section will present the evaluation of the prototype system, by running the three validation rules independently of one another on various numbers of SBCs. Firstly all components that are used to run the tests are presented in Section 6.1. After the introduction the test results for the three validation rules are presented in Sections 6.2, 6.3 and 6.4. Lastly the results are discussed in Section 6.5.

6.1 Evaluation Setup

6.1.1 Data

The electrical consumption data used in the experiments comes from the simulator created by Richardson et al. [22]. Each run of the simulator provides 24 hours of a household's electrical consumption, with a granularity of 1 minute. The parameters for the simulator, as seen in Table 2, are number of residents, weekday or weekend, month of the year, appliance usage and occupancy level. Appliance usage can be simulated or manually altered in a spreadsheet, while the occupancy level of the residents is only possible to simulate.

Parameters	residents	part of the week	month	appliances	occupancy
Variables	1-5	we or wd	1-12	sim. or man.	sim.

Table 2: Parameters of the electrical consumption simulator

The first three parameters were manually chosen and the two last were simulated, at each run of the simulator. In this thesis each simulated household represents one SM. In order to build a dataset the simulation was run 1000 times, to simulate as many separate SMs. However, to accumulate enough data to be able to reach an input rate of 14000(t/s), which was an early estimate of the saturation level of the prototype, the manual simulator would have needed to perform $14000 * 60 = 840000$ simulation runs. This was not feasible, instead the data of the 1000 runs were copied 840 times. The csv file with the consumption data contained 1.2 billion lines, because $24(h) * 60(min) * 840000(simulations) = 1.2 * 10^9$. In order to test the validation rules from Section 4 adjustments to the data were made, to trigger their specific properties. For the filter rule Fixed Filter the consumption data of the injected tuple was switched to a negative value, with a probability of 0.1%. The probability for the random generator was chosen low, so the system would still have to forward 99.9% of the tuples. Since, the evaluation was performed to stress the throughput of the system, a higher percentage would have lower the output rates. When the Varying Filter was tested the data was not manipulated, as the dataset already was divided according to SM. Interpolation is triggered by consecutive tuples with a time difference of more than a minute. The injector removed tuples with the same probability of 0.1%, as was used for manipulation of tuples in the Fixed Filter test. Exploring different percentages is an interesting aspect but not in the scope of the thesis.

6.1.2 Hardware

In this section the hardware test environment will be presented. The prototype consists of four Odroid-U3 connected via a switch, Netgear FS108 with a bandwidth of

10/100 Mbps. Each Odroid-U3 has an Exynos4412 1.7GHz quad-core Samsung processor with a Mali-400 quad-core 400MHz 3d accelerator and 2GB of DDR2 RAM. In order to allow the Odroids to only run and process the validation rules a laptop is also connected to the switch, to function as the server. The laptop has a Pentium Dual-Core T2390 processor and 2GB of DDR2 RAM. How the prototype system is connected can be seen in Figure 12.

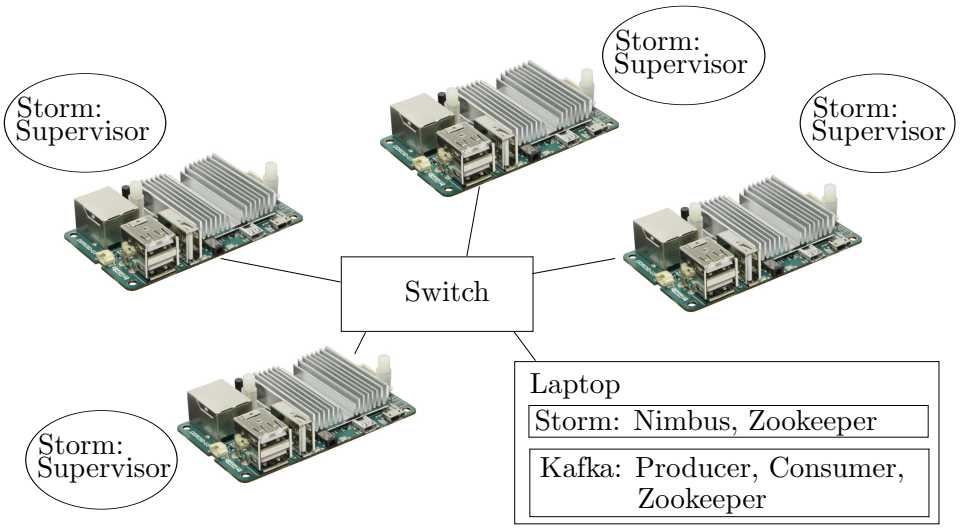


Figure 12: Test environment

6.1.3 Software

Software-wise, the prototype runs both Storm 0.9.2 and Kafka 0.8.2.1. The implementation of the validation rules are done in Storm Trident topologies. These topologies are then processed by the Supervisors run on the Odroids, one on each. As explained in Section 2.3 Storm uses Nimbus to distribute the work and Supervisors to allocate the actual work, with Zookeeper as an intermediary to keep the state of the system. There are two dedicated programs that inject and receive tuples, from the topology. They are based on Kafka's *producer* and *consumer*. The injector reads the consumption data from a text file and can then simulate different input rates, based on what is in the input rate text file. It also keeps track of the injected rate, as this is needed to study the throughput evolution, by saving the metrics in a text file. The receiver takes the outputs from the topologies and monitors the latency, which is saved in a text file to study the latency evolution. Latency and throughput evolution are used to find the maximum capacity of the prototype system. All entities except the Storm Supervisors are run on the laptop. The laptop runs Kafka's producer, consumer and separate Zookeeper as well as Storm's Nimbus and Zookeeper. Storm topologies, Kafka injector and receiver were implemented using Java, as both platforms provide most information about that such implementation. The implementation was performed on an Ubuntu 14.04 LTS system with Java OpenJDK 7.

To evaluate the prototype system throughput and latency are measured. The

throughput is measured at two points in the system, at the injection from Kafka and at the input queue to the Storm topology. Saturation is reached when the rate at which tuples are injected does not match the rate at which they are received by Storm. In order to measure the latency of the system each tuple is timestamped with the system time, of its creation. Hence, the processing latency measured for each output tuple, can be calculated as the difference between the current time and the timestamp of the tuple. All devices' clocks in the prototype system are synchronized through NTP [20], to assure correct timestamps.

6.1.4 Storm Parallelism Setup

In this section the allocation of the available processing capacity will be discussed. The available processing capacity is limited by the number of threads provided by the Odroids, as they perform the execution of the validation rules' topologies. Each Odroid provide as many threads as twice the number of cores, which is 8 threads. Storm assign the number of threads for each part of the topology through the parameter parallelismHint. The parameter specifically assign the number of executors, as presented in Section 2.3.1, that then can run one or more tasks. However, during the experiments one executor per task and thread provided the best results, as each thread only processed one task and no switching was needed. The three topologies that were used in this thesis had the same structure. They took in the tuples from the injector through a OpaqueTridentKafkaSpout, then processed them through the validation rule. Following that, the tuples were processed by rePosition, to be formatted to Kafka messages, and sent out through TridentKafkaStateFactory. During the tests it proved that the input spout needed to have the same parallelismHint as the number of Odroids used, to achieve high throughput. Table 3 shows the optimal distribution of threads among the topology entities, when all four Odroid-U3s were used. In total the Odroids provided 32 available threads.

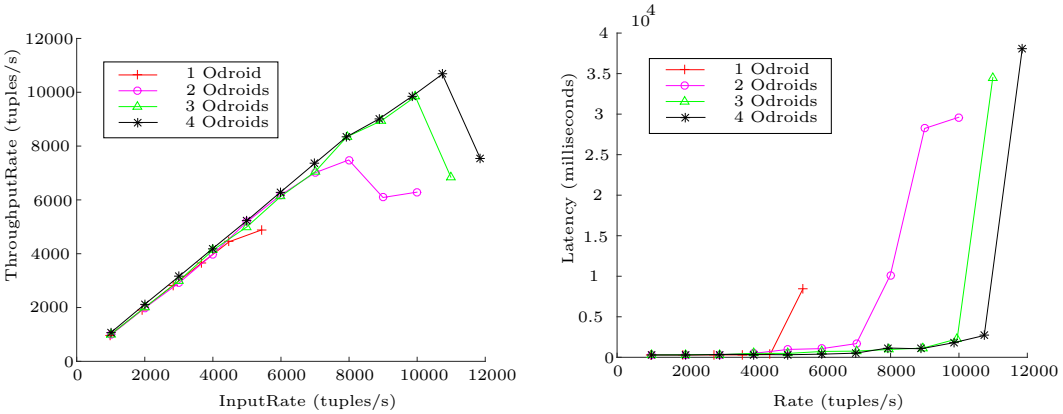
Fixed Filter	OpaqueTridentKafkaSpout		Filter	rePosition	TridentKafkaStateFactory	Total
Nr. Threads	4		1	4	22	31
Varying Filter	OpaqueTridentKafkaSpout	AggregateWindow	Filter	rePosition	TridentKafkaStateFactory	Total
Nr. Threads	4	4	1	4	17	30
Interpolation	OpaqueTridentKafkaSpout	Aggregate		rePosition	TridentKafkaStateFactory	Total
Nr. Threads	4	4		4	18	30

Table 3: Thread distribution in the topologies

The output was most demanding and needed all capacity that was still available. For the validation rules the simple rule that only used a filter, Fixed Filter, coped with one thread. While the more complex rules needed as many threads as the number of Odroids used. The retrieval and forwarding of tuples added a cost in threads, which depended on the validation rule. This thread cost is much higher than the cost for the actual validation. Not all threads could be allocated as this thesis used Storm Trident. Storm interprets the Trident topologies to basic spouts and bolts. That adds extra entities that also needs to be processed by the available threads. Therefore Trident topologies should leave one or two available threads unallocated. The more complex the topology is the more overhead in spouts and bolts are added.

6.2 Validation Rule 1 - Fixed Filter

The first test will run the topology for Fixed Filter from Section 4, that discards all consumption values less than zero. In Figure 13a the increase in throughput for each added Odroid can be seen. The latency does not increase substantially before saturation for the Odroid or Odroids is reached, to then grow exponentially as can be seen in Figure 13b. Throughput grows linearly in the beginning then flats down, opposite of the latency. Each point presented in the graphs, for all validation rules, have been measured at least five times.



(a) Throughput in t/s

(b) Latency in milliseconds

Figure 13: Fixed Filter - Measuerments for one, two, three and four Odroids

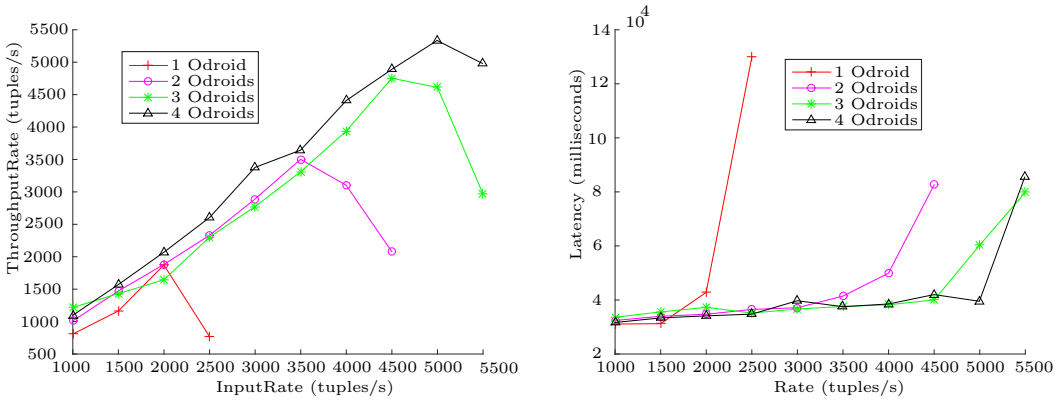
The maximum throughput as measured for each number of Odroids is presented in Table 4, before the latency increases exponentially. The minimum increase in throughput is 700(t/s), for the fourth Odroid.

Number of Odroids	1	2	3	4
Throughput (t/s)	4443	7007	8942	10672

Table 4: Fixed Filter - Maximum throughput for Odroids

6.3 Validation Rule 2 - Varying Filter

The second test will show the performance of Varying Filter from Section 4. This rule discards all electrical consumption values larger than two times the median value. A result is presented each minute and the median is calculated from an aggregation of the last ten minutes. Figure 14 present the same pattern of increasing throughput with added Odroids, as observed in the test of Fixed Filter.



(a) Throughput in t/s

(b) Latency in milliseconds

Figure 14: Varying Filter - Measuerments for one, two, three and four Odroids

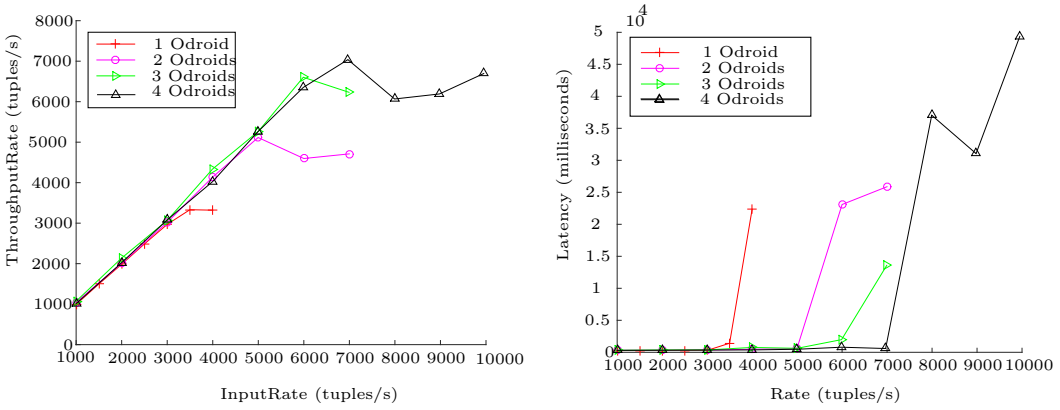
Maximum throughput is achieved when the latency is in the interval between $3 - 6 * 10^4$ milliseconds. After the interval is breached the measured metrics are not reliable, as the time-based window advances every $6 * 10^4$ milliseconds, and thereafter the latency grows almost exponentially. The throughput numbers after the exponential latency increase can therefore not be included in the reported results. The rule produces a result for each SM every minute, which makes it bursty. Between the burst of tuples the records indicates zero latency, the test is badly represented by a mean value calculation. Mean value smears the peaks over the time axis. This behaviour can possibly be avoided by using a Join operator, that continuously updated the filter's condition value. However, that would have allocated additional threads and put further pressure on the hardware's processing capacity. The numbers for the maximum throughput of the test is presented in Table 5. Each added Odroid increases the throughput with at least 500(t/s).

Number of Odroids	1	2	3	4
Throughput (t/s)	1954	3500	4500	5000

Table 5: Varying Filter - Maximum throughput for Odroids

6.4 Validation Rule 3 - Interpolation

The third test show the performance of Interpolation from Section 4. The rule interpolates missing tuples in the stream. If two consecutive tuples from the same SM differentiates by more than one minute, a new tuple is interpolated in the missing timespan. Electrical consumption for the new tuple is calculated as the mean of the two compared tuples. Figure 15 as well follows the pattern of increased throughput with added Odroids, as observed in the tests of Fixed Filter and Varying Filter.



(a) Throughput in t/s

(b) Latency in milliseconds

Figure 15: Interpolation - Measuerments for one, two, three and four Odroids

Table 6 show the increase in maximum throughput with each added Odroid. The additional Odroid increases the throughput by at least 873(t/s).

Number of Odroids	1	2	3	4
Throughput (t/s)	3332	5115	5988	7038

Table 6: Interpolation - Maximum throughput for Odroids

6.5 Summary

The goal set in the Introduction 1 is now reached, it is shown that the prototype is scalable and can run the electrical consumption validation. This while being dimensionally small, keeping the cost and power consumption low, as can be observed in Table 7. In comparison with the paper discussed in Section 3.3 the hardware used by Gulisano et al. [13], 8-core Intel Xeon E5-2650, is expensive, as the price does not include memory and motherboard. To put the maximum throughput reached in the test 10672(t/s) in perspective an illustrative example can be good. If the SMs in a town produces consumption data each hour, a system that can process 10000(t/s) could cover $10000(t/s) * 60(s/min) * 60(min/h) = 36$ million validations/hour. The minimum scaling achieved in the tests with an added Odroid 500(t/s), would cover $500(t/s) * 60(s/min) * 60(min/h) = 1.8$ million extra validations per hour. Gulisano et al. reached similar throughput numbers, 1000-8000(t/s), but their system is not

as deployable in the AMI as it is built on dimensionally larger hardware, that consumes more power. The table below gives an indication of the price and energy consumption differences, which are significant. The SBC cluster can achieve similar throughput at approx 1/5 of the cost and power consumption.

System	Price	Power
1 Odroid	$69.00/572kr	5W
Cluster (4 Odroids)	$276/2288kr	20W
4-core Intel Xeon E3-1246	$410/3399kr	80W
8-core Intel Xeon E5-2650	$1125.99/9334kr	95W
*(1US=8.29kr) from hardkernel.com, amazon.com and newegg.com		

Table 7: Price and power comparison of validation systems

7 Future Work

In this section the possible future improvements of the prototype developed during this thesis will be discussed. The prototype can validate real-time electrical consumption stream data from SMs, by discarding incorrect values and interpolate missing ones. Scalability is provided through adding SBCs, with increased throughput and retained latency. This validation is changeable through reprogrammable rules. Functionality could be added by further research into other validation needed in the AMIs, such as editing of historical data. This could be done through interaction with an energy provider or supplying company. The occurrence of different data errors could also be explored, so that the error percentages used in the evaluation could be confirmed. Additional flexibility would be given, if the prototypes need for a wired network connection could be removed. Wireless connectivity can be achieved by purchasing a WiFi module from the manufacturer of Odroid-U3, Hardkernel. The connection then needs to be evaluated, as wireless connection gets worse with added distance. Additional separation could result in worse throughput rates and added latency for the validation. If the module provided by Hardkernel does not meet the requirements it is possible to use a WiFi module of industrial grade, by utilizing the available ports or connectors on the Odroid board. Another development possibility is to test the validation system on just the Odroids, without a dedicated server. Since, a server would not be available in a distributed deployment. This could be evaluated by running every entity of Storm and Kafka on the Odroids, to see how many are needed to keep the same throughput and latency. Of interest could also be to test a prototype outdoors, to develop a system that could withstand temperature changes and other weather-related stresses.

8 Conclusions

This thesis show that it is possible to validate electrical consumption data with a prototype built of four SBCs, with a throughput of 1000 to 10000 tuples per second and maintain low latency. If the SMs produce consumption data hourly the maximum hourly capacity of the system would be 36 million validations/hour as discussed in Section 6.5, which could serve quite a large city. These numbers are in the same range as those presented by Gulisano et al. [13], but the prototype presented in this thesis can achieve even better throughput by adding more SBCs to the system. Each added SBC has shown to scale the system with at least 500(t/s), which in an hourly schedule would provide 1.8 million additional validations. If the price is also taken in consideration the system used by Gulisano et al. is much more expensive, about $850/7046kr. Their system is pricier even without the inclusion of a motherboard or memory. It also consumes more power, 75W, and is harder to place in an AMI due to size. In conclusion this thesis has reached the objectives set in Section 1. Concerning the aspects of ethics and sustainability, the basic concept of this thesis is to improve the quality of data delivered to the utility head-end and thereby improve the whole system. The produced prototype could increase the sustainability of an AMI, by lowering the cost and power usage for validation.

9 List of Abbreviations

The abbreviations used in the report

AMI	-	Advanced Metering Infrastructure
DAG	-	Directed Acyclic Graph
DB	-	Database
MCU	-	Meter Concentrator Unit
SBC	-	Single-Board Computer
SBD	-	Single-Board Device
SM	-	Smart Meter
SPE	-	Stream Processing Engine
VEE	-	Validation, Estimation and Editing

References

[1] IBM InfoSphere Streams: Programming Model and Language Reference, Version 1.2.1. Technical report, IBM Corp., 2010.

[2] Hunt Allcott. Rethinking real-time electricity pricing. *Resource and Energy Economics*, 33(4):820–842, November 2011.

[3] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 577–588, New York, NY, USA, 2013. ACM.

[4] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491, Toronto, Canada, 2004. VLDB Endowment.

[5] A. Bartoli, J. Hernández-Serrano, M. Soriano, M. Dohler, A. Kountouris, and D. Barthel. Secure Lossless Aggregation for Smart Grid M2m Networks. In *2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 333–338, October 2010.

[6] Chakib Bekara, Thomas Luckenbach, and Kheira Bekara. A privacy preserving and secure authentication protocol for the advanced metering infrastructure with non-repudiation service. In *ENERGY 2012, The Second International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 60–68, 2012.

[7] R. Berthier and W.H. Sanders. Specification-based intrusion detection for advanced metering infrastructures. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 184–193, December 2011.

[8] Mar Callau-Zori, Ricardo Jiménez-Peris, Vincenzo Gulisano, Marina Papatriantafilou, Zhang Fu, and Marta Patiño-Martínez. STONE: A stream-based DDoS defense framework. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 807–812, New York, NY, USA, 2013. ACM.

[9] M. Costache, V. Tudor, M. Almgren, M. Papatriantafilou, and C. Saunders. Remote control of smart meters: Friend or foe? In *2011 Seventh European Conference on Computer Network Defense (EC2ND)*, pages 49–56, September 2011.

[10] A. Dieb Martins and E.C. Gurjao. Processing of smart meters data based on random projections. In *Innovative Smart Grid Technologies Latin America (ISGT LA), 2013 IEEE PES Conference On*, pages 1–4, April 2013.

[11] V. Gulisano, R. Jimenez-Peris, M. Patiño-Martinez, C. Soriente, and P. Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, December 2012.

[12] Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. METIS: A two-tier intrusion detection system for advanced metering infrastructures. In *Proceedings of the 5th International Conference on Future Energy Systems*, e-Energy '14, pages 211–212, New York, NY, USA, 2014. ACM.

[13] Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. Online and scalable data validation in advanced metering infrastructures. In *The 5th IEEE PES Innovative Smart Grid Technologies (ISGT) European 2014 Conference*, 2014.

[14] Yuanxiong Guo, Miao Pan, Yuguang Fang, and P.P. Khargonekar. Decentralized coordination of energy utilization for residential households in the smart grid. *IEEE Transactions on Smart Grid*, 4(3):1341–1350, September 2013.

[15] Apache Kafka. http://kafka.apache.org/.

[16] B. Lohrmann and Odej Kao. Processing smart meter data streams in the cloud. In *2011 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies (ISGT Europe)*, pages 1–8, December 2011.

[17] Björn Lohrmann, Daniel Warneke, and Odej Kao. Massively-parallel stream processing under QoS constraints with nephele. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 271–282, New York, NY, USA, 2012. ACM.

[18] T. Mikkola, E. Bunn, P. Hurri, G. Jacucci, M. Lehtonen, M. Fitta, and S. Biza. Near real time energy monitoring for end users: Requirements and sample applications. In *2011 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 451–456, October 2011.

[19] Nasim Beigi Mohammadi, Jelena Mišić, Vojislav B. Mišić, and Hamzeh Khazaei. A framework for intrusion detection system in advanced metering infrastructure. *Security and Communication Networks*, 7, 2014.

[20] ntp.org: Home of the Network Time Protocol. http://www.ntp.org/.

[21] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009. CCGRID '09*, pages 124–131, May 2009.

[22] Ian Richardson, Murray Thomson, David Infield, and Conor Clifford. Domestic electricity use: A high-resolution energy demand model. *Energy and Buildings*, 42(10):1878–1887, October 2010.

[23] F. Scarano and M. L. Riethmuller. Iterative multigrid approach in PIV image processing with discrete window offset. *Experiments in Fluids*, 26(6):513–523, May 1999.

[24] Yogesh Simmhan, Baohua Cao, Michail Giakkoupis, and Viktor K. Prasanna. Adaptive rate stream processing for smart grid applications on clouds. In *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ScienceCloud '11, pages 33–38, New York, NY, USA, 2011. ACM.

[25] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005.

[26] Apache Storm. http://storm.apache.org/.

[27] Valentin Tudor, Magnus Almgren, and Marina Papatriantafilou. Analysis of the impact of data granularity on privacy for the smart grid. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 61–70, New York, NY, USA, 2013. ACM.

[28] D. Warneke and Odej Kao. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):985–997, June 2011.

[29] Cheng Xu, Daniel Wedlund, Martin Helgoson, and Tore Risch. Model-based Validation of Streaming Data: (Industry Article). In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 107–114, New York, NY, USA, 2013. ACM.

[30] Apache ZooKeeper. http://zookeeper.apache.org.

10 Appendix

10.1 Other Stream Operators

Below is the description of the common stream operators that are not used in this thesis.

Union
Union is a stateless operator that merges several input tuples from different streams into one output stream. The schema for the operator is union()(I_1,I_2, ...)(O), tuples are assumed to have the same tuple field schema. For instance the operator can be used to produce tuples of consumption data from all SMs in one output stream.
$union()(I_1, I_2, ...)(O)$

Join
Join is a stateful operator that matches different input streams based on the value of tuple fields. Tupels are collected separately for each stream and uses sliding windows of typle or time type. join(WindowType, SizeOfWindow, Matching)(I_1,I_2)(O), the output is a combination of both incoming tuples if they match.
$join(time, 1h, 1.id = 2.id \wedge 1.e.cons = 2.e.cons * 2)(I_1, I_2)(O)$, join tuples if they are from the same SM and the consumption of the former is two times of the latter.

Sort
Sort is a stateful operator that collects a set of tuples over a sliding window. Every time the set is full a the tuple with minimum value is emitted as output. sort(WindowType,SizeofWindow,SlideAmount,in.field,Group(in.field))(I)(O), the a set of tuples are collected over a tuple-based or time-based sliding window. With every sliding the tuple with the minimum field is produced as output. Optional is to group by a certain field and thereby create a sliding window for each instance. $sort(time, 1h, 10min, e.cons, Group(id))(I)(O)$, produces the minimum consumption collected the last hour every ten minutes for each SM.