Thesis for The Degree of Licentiate of Engineering

Data stream processing meets the Advanced Metering Infrastructure: possibilities, challenges and applications

Joris van Rooij

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2020

Data stream processing meets the Advanced Metering Infrastructure: possibilities, challenges and applications

Joris van Rooij

ISSN 1652-876X
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

Abstract

Distribution of electricity is changing. Energy production is increasingly distributed, weather dependent and located in the distribution network, close to consumers. Energy consumption is increasing throughout society and the electrification of transportation is driving distribution networks closer to the limits.

Operating the networks closer to their limits also increases the risk for faults. Continuous monitoring of the distribution network closest to the customers is needed in order to mitigate this risk. The Advanced Metering Infrastructure introduced smart meters throughout the distribution network. Data stream processing is a computing paradigm that offers low latency results from analysis on large volumes of the data. This thesis investigates the possibilities and challenges for continuous monitoring that are created when the Advanced Metering Infrastructure and data stream processing meet.

The challenges that are addressed in the thesis are efficient processing of unordered (also called out-of-order) data and efficient usage of the computational resources present in the Advanced Metering Infrastructure.

Contributions towards more efficient processing of out-of-order data are made with eChIDNA and TinTiN. Both are systems that utilize knowledge about smart meter data to directly produce results where possible and storing only data that is relevant for late data in order to produce updated results when such late data arrives. eChIDNA is integrated in the streaming query itself, while TinTiN is a streaming middleware that can be applied to streaming queries in order to make them resilient against out-of-order data. Eventual determinism is defined in order to formally investigate the deterministic properties of output produced by such systems.

Contributions towards efficient usage of the computational resources of the Advanced Metering Infrastructure are made with the application LoCoVolt. LoCoVolt implements a monitoring algorithm that can run on equipment that is localized in the communication infrastructure of the Advanced Metering Infrastructure and can take advantage of the overlap between the communication and distribution networks.

All contributions are evaluated on hardware that is available in current AMI systems, using large scale data obtained from a real production AMI.

Keywords

Data stream processing, Advanced Metering Infrastructure

Acknowledgment

This work would not have been possible without the helpful support and guidance of my supervisors Marina Papatriantafilou and Vincenzo Gulisano at Chalmers, as well as Peter Berggren and earlier Katja Almberg at Göteborg Energi.

I am also very grateful for my employer, Göteborg Energi, for giving me the time to do research. A special thank you is due to my colleagues who have been running the AMI systems when I was busy doing research. Valuable experience and insights were shared by colleagues in the reference group for this project at Göteborg Energi, for which I am especially grateful.

Also the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, deserves my gratitude. Both for introducing me to a large number of students on a similar mission as my own, but also for making this work possible financially, together with the collaboration framework of Göteborg Energi and Chalmers Energy Area of Advance project STAMINA.

And finally I want to thank my wife and children: Thank you Frida, Julia and Fabian for your love and support when I need it the most.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] J. van Rooij, J. Swetzén, V. Gulisano, M. Almgren, M. Papatriantafilou "eChIDNA: Continuous Data Validation in Advanced Metering Infrastructures"
IEEE International Energy Conference (ENERGYCON), 2018.

[B] J. van Rooij, V. Gulisano, M. Papatriantafilou "TinTiN: Travelling in Time (if Necessary) to deal with out-of-order data in streaming aggregation"
Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS), 2020.

[C] J. van Rooij, V. Gulisano, M. Papatriantafilou "LoCoVolt: Distributed Detection of Broken Meters in Smart Grids through Stream Processing"
Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS), 2018.

Other publications

The following publication was published during my PhD studies. However, it is not appended to this thesis, due to contents not related to the thesis.

[a] V. Botev, M. Almgren, V. Gulisano, O. Landsiedel, M. Papatriantafilou, J. van Rooij "Detecting non-technical energy losses through structural periodic patterns in AMI data"
IEEE International Conference on Big Data (Big Data), 2016

Contents

Abstract **iii**

Acknowledgement **v**

List of Publications **vii**

1 Overview **1**
- 1.1 Introduction 1
- 1.2 Preliminaries 3
  - 1.2.1 Advanced Metering Infrastructures 3
  - 1.2.2 Data stream processing 6
- 1.3 Research Challenges 10
  - 1.3.1 Out-of-order data 11
  - 1.3.2 Deployment strategy 12
- 1.4 Contributions 13
  - 1.4.1 Intra-query out-of-order processing 13
  - 1.4.2 Middleware-based out-of-order processing 13
  - 1.4.3 Distributed voltage monitoring 14
  - 1.4.4 Industrial Relevance 15
- 1.5 Conclusions and Future Work 15
- Bibliography 16

2 eChIDNA: Continuous Data Validation in Advanced Metering Infrastructures **21**
- 2.1 Introduction 22
  - 2.1.1 Challenges 22
  - 2.1.2 Related work 23
  - 2.1.3 Contributions 23
- 2.2 System model 24
  - 2.2.1 Advanced Metering Infrastructure 24
  - 2.2.2 Data streaming 24
- 2.3 Architecture 25
  - 2.3.1 Data ingestion 26
  - 2.3.2 Validation modules and rule design 27
    - 2.3.2.1 Single event detection sub-module 27
    - 2.3.2.2 Composite event detection sub-module 28
- 2.4 Evaluation 28
  - 2.4.1 Evalution setup 29

2.4.2 Validation accuracy 29
2.4.2.1 Single event detection 29
2.4.2.2 Composite event detection 30
2.4.3 Efficiency 31
2.5 Conclusion 31
Bibliography 33

3 TinTiN: Travelling in Time (if Necessary) to deal with out-of-order data in streaming aggregation 35
3.1 Introduction 36
3.2 Preliminaries 38
3.2.1 Stream processing 38
3.2.2 Strict determinism 40
3.3 System Model 41
3.4 *D*-bounded eventual determinism 42
3.5 TinTiN's overview 44
3.6 TinTiN's core functionality 46
3.6.1 Answering Q1: Which results can be improved and forwarded to the end user 46
3.6.2 Answering Q2: Sufficient input to replay 48
3.6.3 Answering Q3: Replaying efficiently 49
3.6.4 Answering Q4: Replaying safely 49
3.6.4.1 Shifting timestamps of input to UDA$_R$ 50
3.6.4.2 Restoring timestamps of UDA$_R$ results 52
3.6.5 Synthesis: TinTiN's algorithmic design 52
3.7 Use Case and Evaluation 54
3.8 Related Work 58
3.9 Conclusion and Future Work 59
Bibliography 61

4 LoCoVolt: Distributed Detection of Broken Meters in Smart Grids through Stream Processing 65
4.1 Introduction 66
4.2 Preliminaries 66
4.2.1 Data streaming processing applications 67
4.2.2 AMIs and voltage monitoring 67
4.2.3 Streams correlation 68
4.3 Problem description 69
4.4 LoCoVolt 70
4.4.1 Input 72
4.4.2 Statistics 74
4.4.3 Accusations 74
4.4.4 Weighted accusations 75
4.4.5 Expectations 75
4.4.6 Output 75
4.5 Evaluation 75
4.6 Related Work 87
4.7 Conclusions and future work 88
Bibliography 89

Chapter 1

Overview

1.1 Introduction

The power distribution network takes electricity from local producers as well as the transmission network and distributes this to the energy consumers on the network. To minimize losses on the network, incoming electricity from the transmission network is distributed at a medium voltage (1kV-50kV) to substations where the voltage is transformed to low voltage (100V-240V) to which most customers are connected [1].

Measuring is an integral part of power distribution. The medium voltage distribution network can be connected to a *Supervisory Control And Data Acquisition* (SCADA) system [2]. Such systems contain sensors and remote controlled devices that operators use to monitor and control the distribution network. The low voltage distribution network is typically not connected to a SCADA system. Measurements in the low voltage distribution network were historically mainly for billing.

Smart Meters were introduced in the electricity grid in order to automate meter readings. The smart meters are part of the *Advanced Metering Infrastructure* (AMI) together with a communication network and servers [3]. This infrastructure can read the meters at a regular interval without human intervention.

Before the introduction of the AMI, meters were read manually by utility personnel, e.g. once a year. The AMI made it possible to increase the reading frequency and legislators have used this opportunity. Utilities in Sweden are required to be able to read smart meters every 15 minutes, starting 2025 [4].

The introduction of the AMI has not been the only change in the energy distribution network. Roof based photo-voltaic systems are installed in the distribution network, introducing local power generation. Electricity consumption patterns are also changing with for example the increasing penetration of electric cars.

Changes like these put increasing amounts of stress on the low voltage distribution network. For example reversing the flow of power when the sun shines, or power consumption close to the maximum ratings when customers in neighbourhoods come home from work and start charging their cars.

More stress on the low voltage distribution network increases the risk for faults. Faults are inconvenient, can be dangerous and are expensive to repair. Continuous and near real time monitoring can mitigate this risk by allowing the utility to take appropriate action before the fault occurs. The AMI with its smart meters offer the possibility for monitoring, however this means that the measurements for hundreds of thousands of meters must be processed continuously and with low latency in order to provide information while it is still relevant.

It would also be advantageous to be able to process parts of the data collected by smart meters closer to the source. Energy consumption data needs to be collected and stored for billing, but there is, for example, hardly any need to store voltage data from every smart meter in the grid. Voltage data can instead be processed and analyzed close to the source, forwarding only relevant results to the central server. This can reduce the load on the communication network as well as the cost of data transport.

Possibilities

The data stream processing paradigm allows processing of massive, unbounded data sets while producing results in near real time [5]. The massive amount of smart meters in the AMI continuously produce data and monitoring applications benefit from results with low latency. Therefore, the data stream processing paradigm could be a good match for such monitoring applications. In this paradigm, a continuous query processes input without the need to first store this input on disk. The need to store data is reduced, since data can be processed immediately and only the relevant results need to be stored. The store-then-process paradigm on the other hand first stores all input data before periodically running the query on a batch of input data.

Monitoring the low voltage distribution network ensures its reliability. More general analysis of the data produced by smart meters can result in valuable information, for example detection of fraud or non-technical losses [6]. Such information has economic value for the utility, but is also important for the safety of the distribution network and its users. Matching of demand and supply [7, 8] is another example of an application that can benefit from the data gathered through continuous monitoring.

Challenges

However, the data stream processing paradigm, applied to the AMI, comes with new challenges. The following challenges are in the scope of this thesis:

Out-of-order data Readings from the smart meters do not necessarily arrive to the utilities' servers in timestamp order. Data that does not arrive in timestamp order is throughout this thesis referred to as *out-of-order* data. A main reason for this is latency in the communication network. Data that arrives out-of-order is problematic if the analysis depends on multiple readings, which is often the case. The analysis could for example consist of identifying occurrences of a pattern in the energy consumption of an individual smart meter. Analyzing the data out-of-order might miss occurrences of patterns or even find occurrences where there are none if data is analyzed in order. A straightforward solution to data arriving out-of-order would be to wait until all data has arrived, sort the data and then process it. However such a solution also means that results are delayed by the amount of time that is spent waiting. Smart meter data can be days or even weeks late. A monitoring application that monitors the state of the low voltage network several weeks ago is not very useful. A more sophisticated solution is required for monitoring applications.

Deployment strategy Related to the out-of-order challenge is the question where to run the analysis of the data. Every device in the AMI, from the smart meters to the servers, comes with some amount of computational resources. This enables the possibility to analyze data at different places in the infrastructure. Running analysis at the smart meter means that all data will be in order, but the computing resources are limited. Computing resources are more readily available at the central servers, but the amount of out-of-order data will be larger, since every transportation step towards the server can cause delays.

1.2 Preliminaries

This section presents the Advanced Metering Infrastructure as well as the data stream processing paradigm in more detail.

1.2.1 Advanced Metering Infrastructures

Every customer of a utility pays for the energy that they consume. A smart meter measures the consumed energy and ensures that the customer pays for the right amount of energy. The meter is installed close to where the energy is consumed. Most smart meters are installed in the low-voltage (400 V line to line) distribution network since that is where most customers are connected.

History in Sweden

AMI rollouts are taking place all over the world and have reached various states of completion [9]. Sweden was the first European country to complete the large scale introduction of the Advanced Metering Infrastructure (AMI) in 2009 [10]. The main driver for the introduction was the requirement to read meters every month starting July 2009. Prior to July 2009, meters were manually read, e.g. once per year.

Automated Meter Reading was the main use case for the AMI systems introduced in Sweden in 2009. The smart meters in the systems read the active energy consumption and send this to the utility on a monthly basis. Since 2009, the legal requirements on the smart meters have increased. All customers in Sweden can opt to have their meter read every hour since 2012. All installed smart meters must be able to read every 15 minutes starting January 2025. From that date meters must also be able to read both production and consumption of active and reactive energy as well as voltage, current and power [4].

Smart Meters

Smart meters are metering devices that perform measurements, send these measurements to the utility and send events or alerts. Smart meters can be equipped with a breaker that can be used to remotely connect or disconnect a customer from the grid [11].

Smart electricity meters measure the voltage, current and the phase angle between these in order to calculate power and electricity consumption. Consumed active electric energy has been of most interest for utilities since that is what customers are billed for. Meters can measure the cumulative energy consumption, i.e. the total amount of energy consumed since the meter was installed, the amount of energy consumed during predefined intervals, or both. All readings are timestamped using a local clock in the smart meter.

An important property of smart meter readings is that they are *periodic*. The utility sets the periodicity, for example daily, hourly or every 15 minutes. For example, a smart meter with hourly periodicity creates a reading every time a new hour starts for the internal clock in the meter. This also implies that the timestamp of every reading is the start of an hour. For example, a reading that is taken at midnight on new year's day 2020 will have timestamp 2020-01-01 00:00:00.

Modern smart meters can measure and report production and consumption of active and reactive power, voltage, current and phase angle. Some meters can even measure the harmonic distortion on the voltage and current, a measure that indicates the amount of high frequency noise on the grid.

Aside from reporting measurements, smart meters can also generate and send events. Such events can for example be generated when there is a power outage or when the measured voltage is outside of a predefined interval.

Communication Networks

An integral part of the AMI is the communication network, which allows the smart meters to communicate with the servers at the utility. Many different kinds of communication networks are used over the world [12]. The most prevalent techniques are presented in the following list.

Powerline communication utilizes the grid for communication [13]. Smart meters send their data to a *concentrator unit* located in the transformer that the electricity cable connects to. The electricity cables that deliver power to the smart meter are the same as the cables that are used for communication. For this reason, there is perfect overlap between the communication network and the low voltage grid.

Cellular communication is point to point between smart meters and the central server. A cellular technology such as GPRS, 3G, 4G/LTE is used. More modern technologies include LTE-M and NB-IoT [14].

Radio mesh allows smart meters to connect to concentrator units with radio technology, e.g. ZigBee [15]. Smart meters serve as repeater nodes in order to increase the range of the network. Radio mesh networks can be dynamic, in which case smart meters can connect to another concentrator unit if the connection quality is insufficient [16]. Like powerline communication, radio mesh networks provide overlap between the communication network and the low voltage grid. The overlap is not perfect however, since the optimal radio links do not necessarily follow the power lines.

Common for all networks is that there can be temporary failures that can cause data to arrive late or not at all. Packet routing is another common cause for out-of-order data [17].

Communication networks consist of different components where some are common to all networks. Starting at the smart meter, all networks require a communication module present in, or connected to, the smart meter. The module is the link between the smart meter and the rest of the communication network. Radio mesh networks or powerline communication have intermediary nodes that collect data from multiple smart meters. Different manufacturers have different names for such nodes, throughout the thesis they are referred to as *Concentrator Units* (CU). Finally, there is one or more central servers located at the utility with the possibility to scale out to the private or public cloud [18].

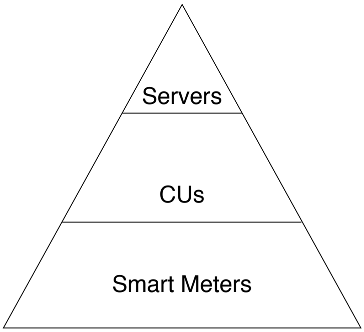


Figure 1.1: *Hierarchy of components in the AMI. A large number of smart meters make up the bottom layer. The middle layer containing CUs is optional. Smart meters either connect to a CU or directly to the server layer at the top.*

The components of the communication network can be structured in a hierarchy as illustrated in Figure 1.1. Smart meters make up the bottom layer in the hierarchy. If CUs are present in the network they make up the middle layer and the central servers the top layer. Figure 1.2 shows a schematic overview of a part of a low voltage distribution network where smart meters are connected to CUs with a radio mesh network.

Analysis of data could take place on all layers in the communication network. Every layer offers their own advantages and disadvantages.

The *edge-computing paradigm* [19] utilizes the nodes on the edge, in this case the smart meters, for analysis. The computational resources of a smart meter are limited. A smart meter doesn't have access to more data than what it can measure itself. For example, analysis that compares the measured voltage with the voltage measured by another meter, connected to the same cable, is not possible. An advantage of analysis at the smart meter is that data has not been transported and is therefore in order.

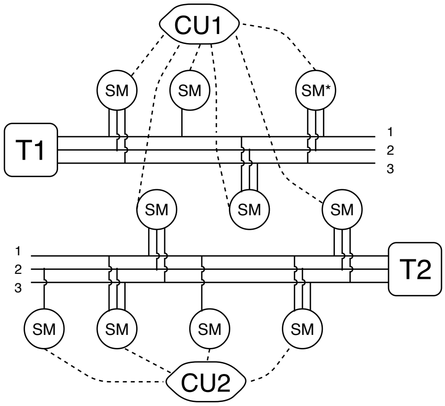


Figure 1.2: *Smart meters (SM) are connected to a transformer (T1/T2) in the distribution network. A radio link to a concentrator unit (CU1/CU2) connects the smart meters to the communication network. Note that there is overlap between the two networks.*

Fog computing [20] moves the analysis of data closer to the source, but not directly at the source like the edge computing paradigm. In the smart meter case, fog computing could take place on the middle layer of the communication infrastructure, containing the CUs. This layer offers more, but still limited, computational resources, compared with the bottom layer. Concentrators can connect to multiple smart meters and therefore the example where the voltage of two smart meters is compared is possible on this layer. Since data needs to be transported from the smart meter to the concentrator, it is possible that data is out-of-order.

The central server at the top layer has virtually unlimited access to computational resources by having the possibility to scale out to the cloud. All data that is collected by the AMI is available at the server, enabling advanced analysis. It is even possible to connect to other servers at the utility to obtain more data, for example the topology of the grid. A disadvantage of analysis on the servers is that data received from either smart meters or concentrators can be out-of-order.

1.2.2 Data stream processing

In the data stream processing paradigm, data is continuously processed by a *streaming query* [21]. The query produces results while data is passing through it, enabling data stream processing to deliver results with very little latency. The paradigm can be implemented with custom programming, but is also implemented in more general *Stream Processing Engine*s (SPE).

Aurora [22] was one of the first SPEs to be introduced. The goal of the system was to enable real-time monitoring applications with unbounded data. The Borealis SPE [23] introduced scalability to multiple machines, and StreamCloud [24] added elasticity. These developments made stream processing engines an alternative to traditional databases.

One of the main differences between running queries on a databases and

Table 1.1: *Example of smart meter data received by the central server.*

Arrival time	Meter ID	Reading (kWh)	Timestamp
2020-01-01 12:03	123456	1000	2020-01-01 12:00
2020-01-01 13:04	123456	1002	2020-01-01 13:00
2020-01-01 15:58	123456	1005	2020-01-01 15:00
⋮	⋮	⋮	⋮
2020-01-08 21:38	123456	2876	2020-01-01 14:00

a streaming query, is that the query on the database is executed on demand while the streaming query is executed continuously with incoming data. The difference between the approaches as well as the general concepts in data stream processing will be illustrated with the following example from the AMI domain.

Illustrative example The amount of energy used by a customer during one hour is limited by a fuse. A fuse will break if a current larger than its rating flows through it. This provides an upper limit on the energy consumption during an hour and any consumption exceeding this limit must be due to a faulty meter reading. Faulty meter readings are problematic for many reasons, including billing and any further analysis based upon the faulty reading. A utility might for example be interested in the sum of all energy consumption and production in a neighbourhood to know the direction of power flow. A faulty reading could make the difference between power flowing towards the neighbourhood or from it. Therefore a utility will want to validate all readings and identify faulty readings in order to solve the underlying problem as fast as possible.

Often, smart meters only measure the cumulative energy consumption, which means that the hourly consumption needs to be calculated by subtracting the reading of hour $t - 1$ from the reading of hour t.

An example of smart meter data is provided in Table 1.1. The reading for timestamp 2020-01-01 14:00 arrives at the central server one week after it was created by the meter. The hourly consumption between 13:00 and 14:00 cannot be calculated before this reading has arrived, i.e. not before 2020-01-08 21:38.

If a query, run on a database, is used for this validation, first all incoming data is stored in the database. Periodically, the query is executed and selects all new readings to combine those with the readings for the hour before. If the difference between these consecutive readings exceeds the fuse limit, the readings can be updated and written to the database again. Note that data might arrive out-of-order as mentioned in section 1.2.1, so a reading for the hour before might not exist in the database yet. Data might need to be read multiple times before it is either possible to verify whether the hourly consumption does not exceed the limit set by the fuse, or that it will never be possible to verify this because the earlier reading will never arrive.

The next section introduces the concepts in data stream processing that are required to understand the data stream processing approach to the validation example.

Operators and continuous queries

A *Stream Processing Engine* (SPE) continuously reads new input data and feeds this data to the *continuous query* which consists of *operators* arranged in a *directed graph.* Operators are located at the nodes in the graph and the edges that connect them represent streams. Figure 1.3 shows the query for the data validation example.

Data travels through the graph in the form of *tuples*, the streaming equivalent of a row in a database table. Tuples in a specific stream in the graph all share the same *schema*, analogous to all rows in a database table sharing the same columns. A schema consists of a number of *attributes.* The smart meter readings from our example consist of three attributes: an ID for the smart meter, the timestamp when the reading was created and the reading itself. The schemas for all streams in the example can be found in angle brackets in Figure 1.3. The SPE transports the input tuples to the first operator in the graph.

Operators can either be *stateless* or *stateful.* Stateful operators accumulate state from previous input in order to calculate the output. The amount of state that is accumulated is controlled by the *window* of the operator. For our example, an operator with a window of size two, performing keyed analysis by meter ID, could be used to find the difference between two readings. The output from this operator would have a different schema, for example smart meter ID, timestamp, reading, difference with previous reading. Stateful operators can perform their operation on the incoming tuples either based upon a *key* or not. It is for example possible to get the daily energy consumption per smart meter, or for all meters combined.

Basic stateful operators are:

Aggregate operators have a single input stream and use multiple tuples from the stream for an operation. The operation can either be computed incrementally, for example a sum, or can be computed on buffered data if incremental computation is not possible. It outputs a stream with the result of the operation.

Join operators buffer tuples from two input streams and join these based on a predicate. A join operator could for example join a tuple a from stream A with a tuple b from stream B if the value of one of the attributes of a exceeds a threshold. The join operator outputs a stream with tuples that are joined, i.e. the schema of the output stream contains attributes from both input streams.

Stateless operators perform an operation on a tuple without requiring more information than what is carried by that tuple. Basic stateless operators are:

Filter operators allow tuples to pass if a condition based on the value of one or more attributes is fulfilled. For example, only tuples where the value of an attribute exceeds a bound are allowed to pass.

Map operators apply a change to the schema of the input stream to create a new schema for the output stream. A map operator could for example remove one of the attributes of the incoming tuples.

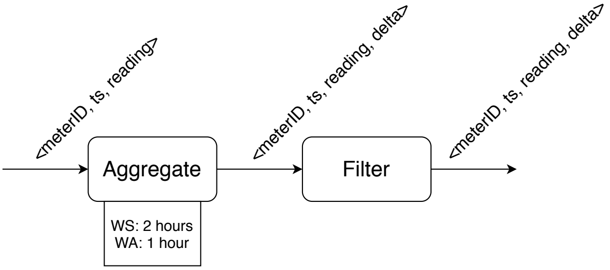


Figure 1.3: *The continuous query for the validation example. An aggregate operator processes the input data. The window size (WS) of the operator is two hours and window advance (WA) is one hour. The operator calculates the difference between the readings in the window and outputs tuples where this delta is included. The filter operator allows only tuples to pass where the delta exceeds a threshold. The schemas of the streams are noted in angle brackets.*

Time

Stream processing engines can use time to track stream progress, group tuples or to decide when an operator should execute its operation. However there are two different notions of time in data stream processing. Time can either be *processing time* or *event time*.

Processing time is the wall clock time of the machine that is running the analysis. For example, an operator with a window with size two hours and advance one hour (all in processing time) will contain all input that the operator receives during two hours. The window will shift every hour.

Event time on the other hand uses the timestamps in the input data as clock. The same example operator where the window size and advance is instead measured in event time, will contain all input data that has timestamps in the interval spanned by the window. The window will shift based upon the progressing of time by the timestamps in the input stream.

Windows

As mentioned in section 1.2.2, stateful operators work on a window of tuples. The window can either be a *sliding window* which moves over the input stream to keep a specific portion of the stream, or it can be a *landmark window*. Landmark windows are separated by so called landmarks in the stream, e.g. tuples where the value of an attribute exceeds a threshold [25]. One could also use the start of the execution as a landmark, for example to keep track of the largest value ever observed for a certain attribute.

The work described in this thesis does not use any landmark windows, all windows from here on refer to sliding windows.

All windows have a *size* and an *advance*, controlling how the window moves over the stream. A window covering a specific portion of the stream, is referred to as a *window instance*. For example, a window with size five and advance four, has a window instance that starts at every multiple of four. *Tumbling windows* are a special case of sliding windows where the advance of the window equals the size. The units for the size and advance are either *time* based or *count* based. The example query contains an aggregate operator with a (event) time-based window. The window size is two hours, the advance one hour.

The aggregate operator in the example query calculates the hourly electricity consumption. It does this by taking the difference between two readings from a smart meter that are one hour apart in time. It is necessary to use a time based window since the readings might arrive out-of-order. If readings are periodic and always arrive in order, the window could instead be count based. In that case the window size would be two and the advance one.

Note that the example is simple for illustrative purposes, a more advanced query can be found in chapter 4.

Metrics

The performance of a streaming query can be assessed with the following metrics:

Throughput measures the amount of tuples that a streaming query can process per unit of time. This metric is dependent on the SPE used and the computational resources of the used hardware. This must be taken into account when comparing the throughput of different streaming queries.

Processing latency measures the wall-clock time between (i) the production of a result by the streaming query and (ii), the ingestion of the last input required for this result to be produced. Like throughput, this metric depends on the SPE and hardware used.

Logical latency is similar to processing latency, but is instead measured with event time. The logical latency of a result is the difference between the timestamp of the result and the highest timestamp observed by the SPE when this result is produced. For example, if a result for 08:00 is produced when the SPE is processing data with timestamp 18:00, the logical latency of the result with timestamp 08:00 is ten hours. This metric is independent of the hardware used, but instead depends on the out-of-orderness of the input stream and the strategy used to deal with the out-of-order tuples.

Determinism is not a quantitative metric, but instead qualitative. A streaming query is deterministic if the results produced are independent of the order of the input data. This property is especially important for streaming queries that e.g. identify occurrences of patterns. For example, consider the data validation query and the data in Table 1.1. The reading for 2020-01-01 14:00 arrives one week late. If the query does not wait for this reading to arrive, it can not determine that the reading is invalid. A query that doesn't wait for late arrivals would be able to determine the reading invalid only if it would have arrived in order.

1.3 Research Challenges

Data stream processing offers new possibilities when combined with data from AMI systems. Continuous validation of data from all smart meters enables monitoring and analysis of events in the low voltage distribution grid. However applying data stream processing to smart meter data introduces new challenges.

1.3.1 Out-of-order data

As mentioned in section 1.2.1, some AMI data is prone to arrive out-of-order. Common causes are network latencies and temporary problems in the communication network. Out-of-order data in data stream processing is not something that is exclusive to AMIs. Different approaches to deal with out-of-order data exist [26]. All approaches try to find a balance between *logical latency* and *determinism*, since it is not possible to produce deterministic results when not all the relevant data is present for the computation.

The best approach for an application depends on the requirements. Some applications require deterministic results, while others benefit from fast yet possibly incomplete results [27, 28].

A straightforward approach to data that arrives out-of-order would be to wait until all the late data has arrived before sorting and processing it. This sacrifices logical latency but enables determinism. Such an approach is often impractical though, because there might not be a good answer to the question: How long to wait before one can be certain that all data has arrived. Even if the question has an answer, it might not be feasible to store and sort incoming data before processing. The amount of data to store could for example be prohibitively large. The logical latency of results also increases with longer waiting times, reducing the timeliness of the results.

Methods that utilize the concept of *slack* [22] try to balance logical latency and determinism. Methods based on slack will wait some amount of time for late arrivals before processing data. This results in deterministic results if all input data arrives within this slack time. Data that arrives later is discarded and cannot contribute to the results that the query produces.

Modern slack-based methods such as *EQ-K-slack* [29] use dynamic slack, where the system analyzes the lateness distribution of late arrivals in order to achieve a chosen result accuracy while minimizing the amount of slack.

An alternative approach to deal with out-of-order data is incorporated in the *dataflow model* [5] which is the model behind major SPEs as Apache Flink [30], Apache Beam [31] and Google Cloud Dataflow [32]. An *allowed lateness* parameter in the model controls which late data is allowed in a window instance and which data is discarded. The model allows multiple evaluations of a single window instance, e.g. once when the majority of the input data has arrived and once more when late data may have arrived in the window instance. This approach can produce multiple results for any single timestamp and key in the output but gives no guarantees on determinism.

A concept that controls the logical latency in systems based on the dataflow model is *watermarks*. Watermarks track the progress of event time throughout the streaming query. A watermark with timestamp *ts* tells operators that no tuples with timestamps smaller than *ts* will arrive. Operators can use watermarks to determine which window instances should be evaluated. Watermarks can either flow through the graph as *punctuation* [33] or be tracked by a central server as in Google Cloud Dataflow [5].

All of these approaches lead to results that either have a unusable large logical latency, i.e. results can not be delivered timely, or are not deterministic, i.e. the results are not calculated with all the relevant data. Besides, none of the approaches can take advantage of the fact that AMI data is periodic, which leads to the following research question:

Research question 1

Given that readings from smart meters are periodic, is it possible to improve regarding logical latency and to balance the trade-off between determinism and logical latency?

1.3.2 Deployment strategy

The AMI infrastructure can be viewed as a hierarchy as described in section 1.2.1 and in Figure 1.1. This infrastructure could be utilized for analysis since computing power exists on all layers in the hierarchy. Every layer comes with its own advantages and disadvantages, influencing the deployment strategy of a streaming query.

The bottom layer of the hierarchy consists of the smart meters themselves. One of the main advantages of running analysis on the smart meter is that data access is fast and free. There is no need to transfer data over the network, instead only the results from the analysis need to be transferred, minimizing costs associated with data transport. Minimizing the amount of data that is transported over the network is also preferable from a privacy point of view [34]. Another advantage that stems from processing data at its source is that the data is guaranteed to be ordered.

While there advantages for analysis on the bottom layer, there are significant disadvantages as well. Smart meters have limited computational resources and have only access to local data, i.e. data from the meter itself. Data that needs to be transported to the central server might undergo changes along the way. Such changes could be intentional, e.g. some calculation or correction, or unintentional due to hardware problems or software bugs. There is little point in analyzing data that might be changed or corrected at the central server, providing another disadvantage of running analysis at the smart meter.

A more promising layer for analysis is the middle layer. Any Concentrator Units (CUs) in the infrastructure are located in the middle layer of the hierarchy. Computational resources are still limited on this level, but useful amounts are available, comparable with single-board devices like RaspberryPi [35] or Odroid [36]. CUs connect to multiple smart meters, allowing analysis that depends on data from multiple meters. The number of meters that are connected to a single CU can range from tens to thousands or even tens of thousands, depending on the communication technology and implementation choices [37]. Which specific meters connect to a CU depends on the communication technology as well as the topology of the distribution network and the physical topology, e.g. hills or buildings in the area. Wireless mesh networks can be dynamic, allowing meters to switch between CUs [16]. There will nonetheless be overlap between the topology of the communication network and the distribution network. Smart meter data can arrive to the CU out-of-order, datapoints could even be missing due to faults or meters switching to another CU.

The central servers in the top of the hierarchy have best access to computational resources. Even scaling out to cloud-infrastructure could be an option at this level. Another advantage of this level is that data from all smart meters is available, as well as data from other systems at the utility. Disadvantages of running analysis in the top level include that data can be out-of-order and the cost to transport all data through all components of the communication infrastructure.

There is an abundance of work that utilizes central servers or cloud infrastructure for streaming applications related to AMIs and the possibilities for analysis on the smart meters themselves are limited. This leaves the following research question:

Research question 2

Given that the middle layer in the communication infrastructure has overlap with the distribution network, is it possible to design and run efficient monitoring applications on the concentrator units that leverage this?

1.4 Contributions

The contributions of this thesis to the research questions are presented in this section.

1.4.1 Intra-query out-of-order processing

Chapter 2 presents eChiDNA: a streaming validation system for energy consumption data from the AMI. eChIDNA calculates and validates the hourly consumption and identifies patterns of invalid hourly consumptions. Alerts for invalid readings are triggered as a step towards automatic correction of errors.

Since smart meter data is not perfectly ordered, there must be a trade off between determinism and logical latency. The system capitalizes on the fact that smart meter readings are periodic to optimize the trade off. The aggregation operators in eChIDNAs continuous query produce results on the available input and store only input that will be relevant for late arrivals. Late arrivals are processed as soon as they arrive. This approach produces all results that a deterministic system operating on sorted data would produce at significant lower logical latency.

eChIDNA is implemented in Apache Storm [38] and evaluated with real world data collected from 270 000 smart meters from a production AMI. The validation results are compared with the ground truth, as provided by system experts, and show that identification of patterns of invalid readings can reduce the time required from human operators. The performance results show that thousands of tuples per second can be validated with low processing latency.

1.4.2 Middleware-based out-of-order processing

Chapter 3 contributes to the research question in subsection 1.3.1 by formalizing the concept of *D-bounded eventual determinism* and proposing TinTiN, a streaming middleware that can deliver such determinism for streaming aggregation applications.

D-bounded eventual determinism allows for multiple results for a given key and timestamp, but guarantees that the final result, produced for this key and timestamp, is equal to the deterministic result which would have been obtained if the input data was ordered.

TinTiN supplies eventual determinism for applications by storing input data that is required to process late arrivals, similar to eChIDNA. Yet where eChIDNA stores data in all of its aggregation operators, TinTiN stores relevant data directly from the input stream.

TinTiN minimizes the logical latency for results that can be obtained with the data that arrives in order by forwarding such data directly to the streaming application it is providing eventual determinism for. Upon arrival of late data, the relevant input is forwarded so that updated results can be produced. TinTiN is implemented in Apache Flink [30] and evaluated with real world data from 50 000 smart meters.

Not only does TinTiN drastically reduce the logical latency, also the processing performance is improved with an order of magnitude compared with a state-of-the-art solution that waits for time proportional to D before processing the data and generating deterministic results.

1.4.3 Distributed voltage monitoring

An alternative approach to handling out-of-order data is to run the analysis closer to the source, where a smaller amount of data will be out-of-order, if any. This approach is chosen in chapter 4, where analysis of smart meter voltage data takes place at the CU level in the communication infrastructure.

Apart from providing an alternative answer to research question 1, chapter 4 also addresses research question 2, by proposing LoCoVolt (Local Comparison of Voltages).

LoCoVolt's purpose is to identify smart meters that stop accurately measuring the voltage. Inaccurate voltage measurements influence the energy consumption readings, leading to incorrect bills. Being able to detect such faulty meters also increases safety, since undetected high voltages can be dangerous for both humans and installed equipment.

Detection of faulty meters is accomplished by monitoring the actual and historical voltage difference for every pair of smart meters that are connected to the same CU and hence are physically close. Smart meters can report other meters as suspicious based upon the voltage differences and the correlation in the voltage measurements.

LoCoVolt contributes to research question 2 by proposing and implementing the algorithm described above, detecting malfunctioning smart meters at the middle layer of the AMI's communication infrastructure. The system is implemented in Apache Flink and evaluated with real world voltage data collected from 939 smart meters at 26 CUs, and runs on resource-constrained hardware with computational resources that are comparable to a CU. LoCoVolt shows that streaming processing on the CUs on the middle layer can deliver efficient and effective monitoring tools that can take advantage of the overlap between the communication infrastructure and the low voltage distribution network.

1.4.4 Industrial Relevance

This thesis does not only provide contributions targeting the research questions in section 1.3, it also shows that the contributions could be implemented directly in existing AMI systems. All proposed systems are evaluated on hardware that is currently available in existing AMI systems. The data used for evaluation is large scale and gathered directly from an AMI system that is running production data in Gothenburg, Sweden. A validation system based upon the work in this thesis has been implemented in a prototype system and is currently being tested.

1.5 Conclusions and Future Work

Continuous monitoring of the low voltage distribution network is increasingly required when the stress on the network is increased by local electricity production and increased consumption.

This thesis shows that one way to achieve this monitoring lies in the meeting between data stream processing and the Advanced Metering Infrastructure.

Out-of-order processing of AMI data and efficient usage of the computational resources present in the AMI are challenges that are studied in this thesis. Out-of-order processing is addressed in this thesis by (i) modified aggregation operators that are capable to deal with data that arrives out-of-order, (ii) a middleware solution that can take care of the out-of-order data as well as (iii) optimal placement of the analysis in the infrastructure. The thesis also shows that placement of analysis at the middle layer of the communication infrastructure is especially efficient for monitoring applications that can leverage the overlap between the communication network and the low voltage distribution network.

All contributions are evaluated on hardware that is available in current AMI systems, using large scale data obtained from a real production AMI. The proposed systems can be deployed at utilities with minimal effort, prototype solutions have already been implemented at Göteborg Energi and are currently under evaluation.

A future line of research that could improve out-of-order handling even more is to investigate if the Stream Processing Engine itself can become aware of missing input data and how this affects the results that it produces.

This thesis focuses on continuous monitoring of the low voltage distribution network. Continuous monitoring is one of the two major parts of a low voltage SCADA system, with the second part being control. The introduction of connected inverters for solar power as well as demand side flexibility with for example smart chargers for electric cars, increase the possibility to actively control the state of the low voltage distribution network. More research is needed to investigate the role of AMI data and data stream processing in this control of the low voltage distribution network in order to complete the low voltage SCADA system.

Bibliography

[1] T. Gonen, *Electric power distribution engineering.* CRC press, 2015.

[2] D. J. Gaushell and H. T. Darlington, "Supervisory control and data acquisition," *Proceedings of the IEEE*, vol. 75, no. 12, pp. 1645–1658, 1987.

[3] S. Karnouskos, O. Terzidis, and P. Karnouskos, "An advanced metering infrastructure for future energy networks," in *New Technologies, Mobility and Security.* Springer, 2007, pp. 597–606.

[4] Energimarknadsinspektionen, "Energimarknadsinspektionens föreskrifter om funktionskrav för mätsystem och mätutrustning (eifs 2019:5)," https://ei.se/Documents/Publikationer/foreskrifter/El/EIFS_2019_5.pdf, 2019, last accessed: October 26, 2020.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," in *Proceedings of the VLDB Endowment*, 2015, vol. 8, no. 12, pp. 1792–1803.

[6] V. Botev, M. Almgren, V. Gulisano, O. Landsiedel, M. Papatriantafilou, and J. van Rooij, "Detecting non-technical energy losses through structural periodic patterns in ami data," in *Big Data (Big Data), 2016 IEEE International Conference on.* Washington DC, USA: IEEE, 2016, pp. 3121–3130.

[7] G. Georgiadis, I. Salem, and M. Papatriantafilou, "Tailor your curves after your costume: supply-following demand in smart grids through the adwords problem," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 2127–2134.

[8] K. Ma, G. Hu, and C. J. Spanos, "Distributed energy consumption control via real-time pricing feedback in smart grid," *IEEE Transactions on Control Systems Technology*, vol. 22, no. 5, pp. 1907–1914, 2014.

[9] L. Östling, "Ami penetration and communication protocols," *Smart Energy International*, vol. 5, pp. 118–119, 2019.

[10] S. Zhou and M. A. Brown, "Smart meter deployment in europe: A comparative case study on the impacts of national policy schemes," *Journal of Cleaner Production*, vol. 144, pp. 22–32, 2017.

[11] Y. Arafat, L. B. Tjernberg, and P.-A. Gustafsson, "Experience from real tests on multiple smart meter switching," in *IEEE PES Innovative Smart Grid Technologies, Europe.* IEEE, 2014, pp. 1–6.

[12] "Which communications technologies for ami projects," https://www.bearingpoint.com/fr-fr/blogs/energie/which-communications-technologies-for-ami-projects/, 2017, last accessed: October 26, 2020.

[13] T. Sheppard, "Mains communication-a practical metering system," in *Seventh International Conference on Metering Apparatus and Tariffs for Electricity Supply 1992.* IET, 1992, pp. 223–227.

[14] M. Lauridsen, I. Z. Kovács, P. Mogensen, M. Sorensen, and S. Holst, "Coverage and capacity analysis of lte-m and nb-iot in a rural area," in *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall).* IEEE, 2016, pp. 1–5.

[15] P. Kinney *et al.*, "Zigbee technology: Wireless control that simply works," in *Communications design conference*, vol. 2, 2003, pp. 1–7.

[16] Z. Fu, O. Landsiedel, M. Almgren, and M. Papatriantafilou, "Managing your trees: Insights from a metropolitan-scale low-power wireless network," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS).* IEEE, 2014, pp. 664–669.

[17] V. Paxson, "End-to-end internet packet dynamics," *IEEE/ACM transactions on networking*, vol. 7, no. 3, pp. 277–292, 1999.

[18] R. R. Mohassel, A. Fung, F. Mohammadi, and K. Raahemifar, "A survey on advanced metering infrastructure," *International Journal of Electrical Power & Energy Systems*, vol. 63, pp. 473–484, 2014.

[19] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

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

[21] M. Garofalakis, J. Gehrke, and R. Rastogi, *Data stream management: processing high-speed data streams.* Springer, 2016.

[22] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[23] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, and others, "The design of the borealis stream processing engine." in *CIDR*, vol. 5, 2005, pp. 277–289.

[24] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez, “Streamcloud: A large scale data streaming system,” in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, 2010, pp. 126–137.

[25] J. Gehrke, F. Korn, and D. Srivastava, “On computing correlated aggregates over continual data streams,” *ACM SIGMOD Record*, vol. 30, no. 2, pp. 13–24, 2001.

[26] V. Gulisano, D. Palyvos-Giannas, B. Havers, and M. Papatriantafilou, “The role of event-time order in data streaming analysis,” in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 214–217.

[27] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, “Fault-tolerance in the Borealis distributed stream processing system,” *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 1, 2008.

[28] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas, “Maximizing determinism in stream processing under latency constraints,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems.* ACM, 2017, pp. 112–123.

[29] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer, “Quality-driven continuous query execution over out-of-order data streams,” in *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* ACM, 2015, pp. 889–894.

[30] “Apache Flink,” https://flink.apache.org/, 2014, last accessed: October 26, 2020.

[31] “Apache Beam,” https://beam.apache.org/, 2016, last accessed: October 26, 2020.

[32] “Google Cloud Dataflow,” https://cloud.google.com/dataflow, 2015, last accessed: October 26, 2020.

[33] J. R. Rao, “Eventual determinism: Using probabilistic means to achieve deterministic ends,” *J. of Parallel, Emergent and Distributed Systems*, vol. 8, no. 1, pp. 3–19, 1996.

[34] V. Tudor, V. Gulisano, M. Almgren, and M. Papatriantafilou, “Bes: Differentially private event aggregation for large-scale iot-based systems,” *Future Generation Computer Systems*, vol. 108, pp. 1241–1257, 2020.

[35] Raspberry Pi Foundation, “Raspberry pi,” https://www.raspberrypi.org/, 2012, last accessed: October 26, 2020.

[36] Hardkernel co., Ltd, “Hardkernel odroid xu4,” https://magazine.odroid.com/odroid-xu4/, 2015, last accessed: October 26, 2020.

[37] M. Martin, “Smart grid: Meter deployment strategies,” https://vividcomm.com/2014/12/18/smart-grid-meter-deployment-strategies/, 2014, last accessed: October 26, 2020.

[38] “Apache Storm.” [Online]. Available: http://storm.apache.org/

[39] J. Kreps, N. Narkhede, J. Rao, and others, “Kafka: A distributed messaging system for log processing,” in *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB)*, 2011.

Chapter 2

eChIDNA: Continuous Data Validation in Advanced Metering Infrastructures

J. van Rooij, J. Swetzén, V. Gulisano, M. Almgren, M. Papatriantafilou

IEEE International Energy Conference (ENERGYCON), 2018.

Abstract

New laws and regulations increase the demands for a more data-intense metering infrastructure towards more adaptive electricity networks (aka smart grids). The automatic measuring, often involving wireless communication, introduces errors both in software and during data transmission.

These demands, as well as the large data volumes that need to be validated, present new challenges to utilities. First, measurement errors cannot be allowed to propagate to the data stored by utilities. Second, manual fixing of errors after storing is not a feasible option with increasing data volumes and decreasing lead times for new services and analysis. Third, validation is not only to be applied to current readings but also to past readings when new types of errors are discovered.

This paper addresses these issues by proposing a hybrid system, eChIDNA, utilizing both the store-then-process and the data streaming processing paradigms, enabling for high throughput, low latency distributed and parallel analysis. Validation rules are built upon this paradigm and then implemented on the state of the art Apache Storm Stream Processing Engine to assess performance. Furthermore, patterns of common errors are matched, triggering alerts as a first step towards automatic correction of errors.

The system is evaluated with production data from hundreds of thousands of smart meters. The results show a performance in the thousands messages per second realm, showing that stream processing can be used to validate large volumes of meter data online with low processing latency, identifying common errors as they appear. The results from the pattern matching are cross-validated with system experts and show that pattern matching is a viable way to minimize time required from human operators.

2.1 Introduction

The development of an enhanced electricity grid has moved forward both technologically and politically in recent years. One driving force is an increased measurement rate, as a number of countries introduce mandatory hourly billing. When consumers are given the option to pay for their electricity based on the demand at that particular time of day, they gain a higher degree of control over their energy bills by facilitating energy efficiency. Shorter measurement intervals also open up new possibilities in load forecasting with the potential of tailoring electricity production to the consumption and increasing the amount of renewable energy.

These new requirements have lead to the construction of Advanced Metering Infrastructures (AMIs): networks of smart meters and data concentrator units that measure consumption and report it to the utility. These devices offer reduced computational power and can be remotely controlled.

Together with AMIs, the utility data analysis infrastructure also needs to be adjusted to accept the roughly 700 times increase in data that hourly readings give compared to monthly readings. Shorter sampling periods, such as 15 minutes, have been evaluated in a research setting and can improve load forecasting accuracy compared to hourly readings [1]. It is therefore reasonable to expect further increases in data rates.

This increasing amount of data collected by AMIs is known to be error prone, far from the clean and complete measurement series required for load forecasting or billing. Measurements can arrive out-of-order, duplicated or not at all, sometimes owing to communication losses and re-transmissions particularly common in wireless infrastructures. Erroneous measurements can be caused by meters that are broken, software issues or even by malicious users looking to decrease their electricity costs by unlawful means. These errors threaten the increasing quality demanded from the meter values by new applications such as demand-response and real time pricing. Therefore, all data collected must be swiftly validated and corrected if necessary.

2.1.1 Challenges

A system aiming to solve this validation problem faces several challenges. The first challenge is to process and validate this large volume of data with low latency before storing, billing or other analysis is run on the data. Fluctuating data rates and out-of-order readings contribute to this challenge as well as erroneous measurements.

The second challenge is continuously updating the validation rules and leveraging system expert knowledge. This is needed because new errors may arise at any moment and are usually identified by a human operator at some point in time. Closing the control loop creates an iterative learning process where the human controller finds new problems and subsequently uses the validation engine to find other instances of these problems (possibly re-running the validation for previous data to find earlier occurrences of a certain error).

Finally the third challenge is to run the analysis in a distributed and parallel fashion leveraging the cumulative computational power of AMIs' smart meters and data concentrators. Data concentrators could for example validate and compare voltage levels in the network for all meters in a specific area, forwarding only validated data to the utility.

2.1.2 Related work

Since the early 2000s, the data streaming paradigm has emerged as an alternative to traditional databases when it comes to processing large amounts of data in real-time. Among the many applications are financial trading and market analysis, as well as network intrusion detection and monitoring of denial of service attacks. Traditional databases are not designed to support this kind of streaming data and in response to these new challenges, Aurora was developed as one of the first SPEs [2]. Aurora was designed to run on a single machine without the possibility of distributing the work. Building upon the functionality of Aurora, among other works, Borealis, an SPE that can scale to several machines, was developed [23]. More recently, systems like StreamCloud [4] have been presented for improved scalability. Others have been released as open source software, like Apache Flink [30], Apache Spark [6], Apache Storm [38] and S4 [8].

Stream processing in the context of AMIs has been explored since 2010. Several different applications have been found for SPEs in this area, including Intrusion Detection Systems (IDSs) [9], real-time pricing [10] and adaptive measurement rates [11]. Data validation with a simplified system model was investigated in [12]. Here we designed and implemented a system that is directly deployable at a real world utility.

Out-of-orderness is a problem where two concepts have become prevalent: punctuation [13] and variations thereof (heartbeats [14], Window ID [15], out-of-order-processing [16]) and K-slack [17]. Common for these methods is that processing is halted until missing data has arrived unless too much time has passed after which special action is needed. Here we instead implemented a custom sorting mechanism that allows for processing all arrived data even if some values are missing.

2.1.3 Contributions

In this paper we introduce eChIDNA, a *Kappa* architecture [18] that addresses these challenges relying both on the traditional store-then-process (database) paradigm and the data streaming processing paradigm. The streaming paradigm has been proposed as an alternative to the traditional store-then-process (database) paradigm by applications demanding high processing capacity and low processing latency.

A prototype of eChIDNA has been implemented, extending the system of the utility providing the real use-cases we studied. The streaming analysis is performed relying on the state of the art Apache Storm [38] Stream Processing Engine (SPE). We evaluate the system both in terms of efficiency as well as performance and the results are compared with the ground truth provided by system experts.

The rest of the paper is organized as follows: the system model is introduced in Section 2.2. In Section 2.3, we discuss the architecture of eChIDNA. The

Table 2.1: *Format of tuples containing meter readings.*

Attribute	**Description**
t_n	timestamp
met	meter ID
loc	location ID
cc	cumulative consumption
cd	consumption delta

evaluation is presented in Section 2.4. Finally the conclusions are presented in Section 2.5.

2.2 System model

Metering data at a utility typically goes from the smart meters in the AMI through concentrator units and the head-end before being stored and processed in a Meter Data Management system (MDM).

2.2.1 Advanced Metering Infrastructure

AMIs are composed of a multitude of different devices; smart meters that measure the energy or water consumption, concentrators that collect the readings from the meters and forward these to the central server. The meters and concentrators in the field have limited memory and processing capabilities and can be organized in different topologies (e.g., point-to-point, hierarchical or mesh ones).

The smart meters can save the amount of consumed energy in different ways. The *cumulative consumption* since startup can be stored together with a timestamp at regular time intervals, a *consumption delta* - the amount of energy used between two timestamps - can be stored, or both. Readings are sent to the central server regularly after which they are stored in the Meter Data Management (MDM) system. The bulk of the data will have arrived within 24 hours but the server will continue waiting for data to arrive. If it has not arrived with a certain amount of days D_{max}, it is considered lost. D_{max} is typically in the order of days or weeks, yet regulations and future applications like real time pricing demand fast processing of data. Data must therefore be processed as soon as it is available even if some intermediate readings in the stream may not have arrived yet.

2.2.2 Data streaming

In the streaming data paradigm, a Stream Processing Engine (SPE) is used to process streams of data in a distributed and parallel fashion. Data is represented in tuples with a common schema, composed by a timestamp t_n and a set of attributes $\langle A_1, A_2, \ldots, A_n \rangle$. A schema for tuples with meter readings is presented in table 2.1. The operation of an SPE relies on viewing data as a flow of such tuples from beginning to end. The incoming data therefore consists of an unbounded stream of tuples. To extract meaningful information from

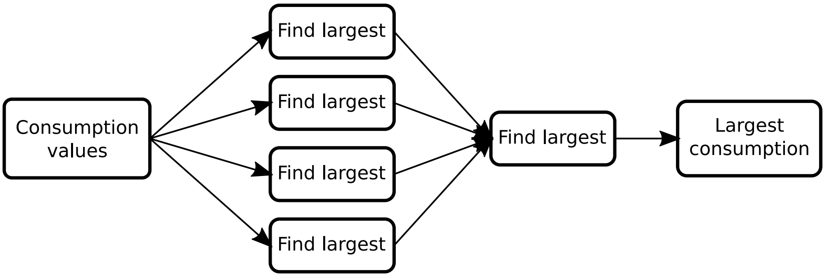


Figure 2.1: *A topology that finds the highest consumption value, the highest number of kWh drawn for specified time period, in a stream of consumption values. The first group of "Find largest" nodes are running in parallel, possibly on different machines. The last "Find largest" needs to be a single process to summarize the results.*

these tuples, continuous queries in the form of Directed Acyclic Graphs, DAGs, are formed. Edges in the DAG correspond to the flow of data between the operations, represented by vertices. Operations can be for example filters that drop tuples that do not fulfill certain criteria, mappers that transform tuples or union operations that merge several streams into one. All these are examples of stateless operations, they consider only a single tuple when producing a result. Another type of operators are the stateful ones, for example aggregate which performs a computation over several tuples in a single stream and join which combines tuples from several streams. A common way of performing aggregate operations is by using a (sliding) window. To handle the unbounded stream of tuples, only the latest tuples are considered. For example, a time-based window can contain all tuples from the last five hours and a tuple-based window can contain the 10 latest tuples.

The results of a continuous query can be alarms that are triggered or a modified data stream saved to a dedicated database. An example of a continuous query that finds the highest measured power consumption in a distributed manner can be found in figure 2.1.

2.3 Architecture

Here we describe the architecture of a system that can deal with the problems and challenges mentioned in Section 2.1. eChIDNA consists of two modules: data ingestion and validation. The latter in turn is composed of two submodules, single- and composite event detection, as shown in figure 2.2. The data ingestion module takes care of the incoming real time data from the different AMI systems as well as the historical data from the MDM system. The module parses the incoming data into tuples with a uniform format in order to maximize the systems interoperability. The tuples outputted from the data ingestion module are read into the validation module. Errors identified by the single event detection are sent to the composite event detection submodule, which finds common composite errors made up by sequences of these single events.

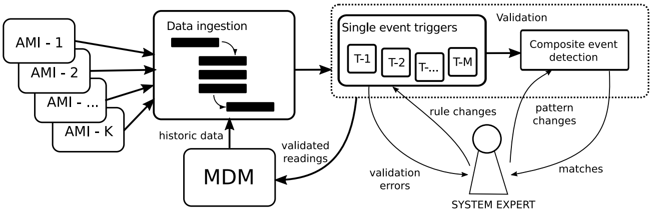


Figure 2.2: *An overview of the eChIDNA architecture. Data from one or more AMI systems and/or historical data from the MDM are read by the data ingestion module. The validation module is composed by two submodules: The single event triggers validate individual tuples and the composite event detection that looks for patterns in the single events. Validated data is stored in the MDM, while validation errors are sent to a system expert. New validation rules and patterns can be submitted by the system expert.*

2.3.1 Data ingestion

The data that is read into eChIDNA can either come from one or more AMI systems or, when data needs to be reprocessed, from the MDM. The data from different AMI systems, which in turn could have multiple meter brands and models, can have different formats. For example meters could send the cumulative consumption, a consumption delta for a fixed time period (eg. day, hour, 15 minutes), or both. Therefore the data ingestion module parses the data into a uniform tuple format presented in table 2.1. Because many validation rules require a consumption delta, these are calculated by eChIDNA, if not already present, by taking two consecutive tuples and computing the difference in the cumulated consumption. Data that has been processed by the entire system can be moved away or simply removed since reprocessing can always be done from the persistent storage in the MDM.

The calculation of the consumption delta is straightforward when all tuples arrive in order, but becomes more challenging when tuples are missing or out of order. There is extensive research dealing with tuples arriving out of order, as described in Section 2.1.2. Common for these methods is that the processing for a meter is halted until missing tuples have arrived. This makes these methods not compliant with the system requirements:

[a] Data must be processed as soon as possible.

[b] Data may arrive with a maximum delay D_{max}

Therefore we developed a tailored sorting mechanism described below.

A tuple $\tau.t_n$ with timestamp t_n is used to calculate the consumption delta between t_{n-1} and t_n as well as the delta between t_n and t_{n+1}. This implies that $\tau.t_n$ cannot be discarded until both $\tau.t_{n-1}$ and $\tau.t_{n+1}$ have arrived.

The sorting mechanism must thus keep track of which tuples have been fully processed and does this by using intervals. An interval spans between two timestamps and denotes that all tuples inside of the interval have been processed. An interval $[n..m]$ for example indicates that tuples with a timestamp between n and m have been processed. A tuple $\tau.t_n$ which can not attach to a existing interval will start a new interval $[n..n]$, spanning between n and n. At the

Table 2.2: *Single event triggers and their identifiers.*

Representation	**Validation error**
`N`	Negative consumption
`Z`	Zero consumption
`H`	Above fuse level

arrival of $\tau.t_{n+1}$, this interval is expanded to $[n..n+1]$ while neither of the tuples is discarded yet. As $\tau.t_{n+2}$ arrives the interval is expanded to $[n..n+2]$ and $\tau.t_{n+1}$ is discarded after the consumption deltas have been calculated. This approach allows us to store only the tuples that are still waiting to be processed. In case of out-of-order tuples, multiple intervals will be present to account for the tuples that are processed. Two intervals merge when the missing tuples between them arrive and are processed. Intervals and tuples with timestamps that exceed the maximum allowed delay D_{max} are deleted.

This mechanism accumulates state during system operation, the tuples that have not been processed completely as well as the intervals themselves. Persisting the state at a regular time interval and saving the input tuples processed during this interval ensures a robust system that can cope with crashes.

2.3.2 Validation modules and rule design

A basis for further analysis, the validation rules lay the groundwork on which the system is built. eChIDNA's architecture allows for rapid implementation and testing of new validation rules, both for single- and composite events.

2.3.2.1 Single event detection sub-module

Single event triggers contain a check for some condition and will output an identifier if this condition is met. The identifiers are stored for every meter in a sliding window of size D_{max}. Triggers can indicate an error of some kind in the meter reading, but can also be used to find other events. These events could then be used as a building block for composite events. Some examples for both types of triggers are described below.

1a) Negative values: As described in Section 2.2.1, the system model does not allow negative consumption deltas. Such deltas therefore always indicate an error of some kind.

1b) Zero consumption: Zero consumption is not an error in itself, but can be part of a bigger problem. This trigger but can be a useful building block for finding composite errors. For example, a longer period without energy consumption may be important to look at more closely.

1c) Above fuse level: As a maximum value for electricity consumption, the fuse provides a hard limit. For every individual meter, information about the installed fuses is available and has been used to calculate this limit. Although a fuse may temporarily allow values above its rating, it cannot sustain it over time and therefore consumption above this limit triggers an alert. This alert can be caused by a technical error, but it can also be caused by a customer who has installed a fuse with higher rating. Whether intentional or not, this needs to be corrected since the network tariffs for the consumer depends on the installed fuse.

2.3.2.2 Composite event detection sub-module

The errors described above can be part of a larger problem. Issues with hardware or software can give rise to so-called composite errors that can be identified by considering the ordering and kind of errors that have been reported for a smart meter. Automatic identification of these errors is a first step towards automated correction, which would save substantial amounts of time for human operators. In order to identify such errors automatically, a formal pattern matching framework is needed.

One common system for pattern matching is *regular expressions*, a search pattern defined by a character string. In order to apply regular expressions to the errors caught by the validation rules, they first need to be transformed into a string that can be matched by a regular expression. There is additional information that needs to be captured apart from the string of errors, for example finding if an error occurred at a specific time of day. If a match is found that also passes through these kinds of validation, it can be considered an instance of a composite error.

The sliding window where the errors found by the validation rules are saved contains timestamp ordered errors. Since there is a maximum delay for meter values, the size of the window is bounded by D_{max}. Matching is performed every time an error is found. The patterns for composite errors have a maximum length which is used to only look for matches within a partition of the stored errors. Matching an error with length l at timestamp t_n will consider values with a timestamp between $t_{n-(l-1)}$ and $t_{n+(l-1)}$ to ensure that every possible interval of length l containing the timestamp t_n is processed.

In order to match the errors with a regular expression, representations of the errors within the matching interval are concatenated into an error string. The expression 'HN' for example, using the identifiers specified in table 2.2 will match every time a consumption delta exceeding the fuse value is followed by a negative consumption value. Matches to the regular expression are complemented with the start and end times for the match, so that additional properties, like the timestamp of a specific event, may be tested for.

Due to the fact that partitions of the stored errors that are matched by regular expressions can overlap, there is a risk of finding some composite errors several times. This is illustrated by the following example: Consider the string with three high values followed by a negative value: 'HHHN'. The expression 'H.0-2N', a high consumption value followed by a negative consumption and at most 2 values in between, will match three times for the substrings 'HHHN', 'HHN', and 'HN'. For this reason, the start and end of any matched composite error is stored. In case a match is completely covered by a previously matched error, that match is not considered.

2.4 Evaluation

eChIDNA's accuracy is evaluated and compared with results from a system expert. The efficiency is evaluated as well. The throughput measured in

Table 2.3: *Number of matches for each validation rule in thousands. The total number of values analyzed is approximately 200 million.*

Validation rule	**Matches**
Zero consumption	4000
Negative consumption	1.5
Above fuse	5.5

processed tuples per second while the latency is defined as the time between system output and the most recent tuple that caused the output. The evaluation ran with 30 days of data for the validation rules, of which 12 days were used to assess the composite event detection capabilities.

2.4.1 Evalution setup

The data that is used for eChIDNA comes from an AMI with approx. 270 000 meters, 7 000 concentrator units and a central server. The meters register the cumulative consumption every hour. Produced energy is either not measured or recorded in a different register, guaranteeing that consumption values are monotonically increasing. The meters are queried for their readings by the concentrators twice a day. The concentrators send the data to the central server for processing. The maximum delay D_{max} for this AMI is 40 days. eChIDNA taps into the data stream between the central server of the AMI and the MDM, where the data is batched and persistent on disk. The tuples contain a meterID, locationID, timestamp and cumulative consumption for every reading.

The efficiency of the system is evaluated on existing hardware available at the utility. It runs on a virtual server with two AMD Opteron processing cores at 2.6 GHz and 4GB of RAM. This server runs Apache Kafka [39] as input and message queue in the data ingestion module while the consumption deltas are calculated by the SPE Apache Storm [38]. The validation module also runs on Apache Storm. Pyleus, a framework enabling the use of Python topologies on Storm, was used in consultation with system experts for usability reasons. Storm (and Pyleus) use topologies to represent the Directed Acyclic Graphs. The vertices in the DAG are called *bolts*. Reading the persistent file data into Apache Kafka is also performed on the server.

2.4.2 Validation accuracy

The single event detection was evaluated using the data produced by smart meters during one month, out of which twelve days were used to compare the results from the composite event detection with the results from a system expert.

2.4.2.1 Single event detection

All single event triggers in Section 2.3.2.1 were enabled: zero and negative consumption, and above fuse. The number of errors found for each validation rule can be seen in table 2.3. Approximately 2% of all processed values trigger a validation rule, the absolute majority of these values have zero consumption.

Table 2.4: *Example of consumption values from a meter with a rushing and reversing pattern.*

timestamp	18	19	20	21	22	23	24
consumption	2.1	*134.6*	*78.9*	4.7	3.8	2.7	***-204.2***

2.4.2.2 Composite event detection

To evaluate this in eChIDNA, an expression was defined in collaboration with a system expert, to identify meters suffering from a specific hardware failure common in the data from the used AMI. This failure expresses itself in the consumption values with a specific pattern: *rushing and reversing*. This pattern contains one or more extremely high consumption deltas during the day and a large negative consumption delta, that compensates the earlier large ones, at midnight. An example can be seen in table 2.4. The regular expression for this pattern was inputted in the system and was matched 640 times in one month of data. During a twelve day testing period, a system expert was asked to note all meters diagnosed with the hardware failure. 190 meters were identified. eChIDNA diagnosed 116 meters as experiencing this problem during the testing period, so a true positive rate of 61% was obtained for the hardware failures using the rushing and reversing pattern. The 39% of false negatives were investigated with help from the system expert and decomposed in the six cases following below. See also figure 2.3.

Missing data is the main cause for the false negative meters, the hardware failure not only causes errors in the consumption values but also affects the communication between the meter and the concentrator unit. Three cases where the root cause is missing data were identified: *1: only negative*, *2: only high* and *3: reversed*. The only negative case accounts for 23% of the false negative meters. For these meters no extremely high consumption deltas were found, only the negative delta at midnight was identified. The only high case, accounting for 22%, is the opposite: No negative delta was found, only the extremely high consumption deltas. Reversed, at 14%, is a combination of the previous two cases were only the negative delta was found during one day, and only extremely high values were found the day after, ie. a reversed version of the original pattern. Another large group of false negatives, *4: end time* at 38%, did have both extremely large consumption deltas and negative deltas, but the negative deltas did not occur at midnight. Instead these deltas occured at some other hour during the day. A case with 1% of the false negatives, *5: two days*, contains meters where the correcting negative consumption delta did not occur the same day as the extremely high values but instead the day after. Finally, *6: unseen data* with the last 2% is made up of meters where the system expert used data that arrived after the twelve day test period to diagnose the meters. This data was never analyzed by eChIDNA and therefore the pattern was not matched.

This decomposition shows that the original expression was not broad enough to identify all meters experiencing the hardware failure. The knowledge gained by the system expert during the decomposition can now be used to improve the expression and the kappa architecture can be leveraged to re-validate the data when new expressions are in place. This shows the importance of system

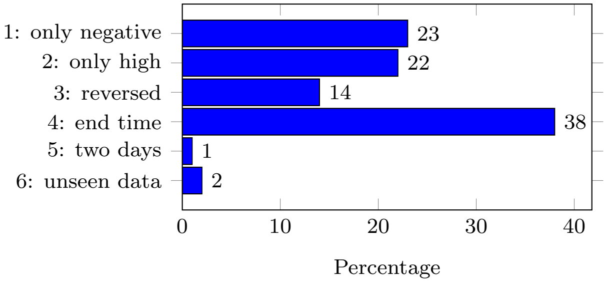


Figure 2.3: *The six different cases for errors caused by a specific hardware failure not being matched by the original pattern.*

expert feedback to the system, enabling swift improvements and maximizing the validation accuracy, closing the control loop.

2.4.3 Efficiency

The *throughput* of the system is defined as the number of tuples processed per second and the maximum throughput was found by running the topology with live data and increasing the input rate step by step until the CPU usage was at 100%.

The *latency* of the system is defined as follows: The time between output from the system and the arrival of the latest tuple that triggered the output. With this definition, the highest latency observed while processing data from 270 000 smart meters, was 4 seconds.

The throughput measured was 1 500 tuples per second on hardware with the processing power of contemporary deployed or immediately-deployable equipment. This means that the system has the potential to validate 5.4 million consumption values every hour. Only 5% of this capacity is required for the validation of the 270 000 hourly readings in the system. The remaining capacity can be used to deal with possible arrival bursts or for validation of 19 days of historic data every day.

2.5 Conclusion

The introduction of smart meters and legislation has given rise to a greatly increasing number of processed meter values at utilities. At the same time the quality demanded of these values has been increasing as well due to new applications, driving the need for a fast and accurate validation. Validation of live production data is complicated by out of order data and fluctuating data rates as well as the appearance of new problems. These problems can be cause by updates, hardware failure or other, often hard to foresee, sources. For this reason a live architecture, with system experts in the loop, is required until full automation can take over. In this paper, we have discussed how the data streaming processing paradigm in a kappa architecture can be used to provide scalable and adaptable validation for both real time and historical data. An implementation built in Kafka and Storm, eChiDNA, shows that data for millions of meters reporting hourly values can be validated on commodity hardware, with a configurable validation rule set. These results encourage the continued efforts in this direction, with more advanced types of validation detecting deviations from known distributions, rapid changes, even patterns that may indicate malfunctioning measurements.

Bibliography

[1] M. Ghofrani, M. Hassanzadeh, M. Etezadi-Amoli, and M. Fadali, "Smart meter based short-term load forecasting for residential customers," in *North American Power Symposium (NAPS)*, 2011, pp. 1–5.

[2] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, and others, "Monitoring streams: a new class of data management applications," in *Proceedings of the 28th international conference on Very Large Data Bases*, 2002, pp. 215–226.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, and others, "The design of the borealis stream processing engine." in *CIDR*, vol. 5, 2005, pp. 277–289.

[4] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 12, pp. 2351–2365, 2012.

[5] "Apache Flink," https://flink.apache.org/, 2014, last accessed: October 26, 2020.

[6] "Apache Spark." [Online]. Available: http://spark.apache.org

[7] "Apache Storm." [Online]. Available: http://storm.apache.org/

[8] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on.* IEEE, 2010, pp. 170–177.

[9] V. Gulisano, M. Almgren, and M. Papatriantafilou, "METIS: a two-tier intrusion detection system for advanced metering infrastructures," in *Proceedings of the 5th international conference on Future energy systems.* ACM, 2014, pp. 211–212.

[10] B. Lohrmann and O. 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)*, 2011, pp. 1–8.

[11] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, "Adaptive rate stream processing for smart grid applications on clouds," in *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ser. ScienceCloud '11, 2011, pp. 33–38.

[12] V. Gulisano, M. Almgren, and M. Papatriantafilou, “Online and scalable data validation in advanced metering infrastructures,” in *Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2014 IEEE PES*, 2014, pp. 1–6.

[13] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, “Exploiting punctuation semantics in continuous data streams,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 3, pp. 555–568, 2003.

[14] U. Srivastava and J. Widom, “Flexible time management in data stream systems,” in *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2004, pp. 263–274.

[15] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, “Semantics and evaluation techniques for window aggregates in data streams,” in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 311–322.

[16] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, “Out-of-order processing: a new architecture for high-performance stream systems,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008.

[17] S. Babu, U. Srivastava, and J. Widom, “Exploiting k-constraints to reduce memory overhead in continuous queries over data streams,” *ACM Transactions on Database Systems (TODS)*, vol. 29, no. 3, pp. 545–580, 2004.

[18] “Kappa architecture,” 2014. [Online]. Available: http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html

Chapter 3

TinTiN: Travelling in Time (if Necessary) to deal with out-of-order data in streaming aggregation

J. van Rooij, V. Gulisano, M. Papatriantafilou

Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS), 2020.

Abstract

Cyber-Physical Systems (CPS) rely on data stream processing for high-throughput, low-latency analysis with correctness and accuracy guarantees (building on deterministic execution) for monitoring, safety or security applications. The trade-offs in processing performance and results' accuracy are nonetheless application-dependent. While some applications need strict deterministic execution, others can value fast (but possibly approximated) answers. Despite the existing literature on how to relax and trade strict determinism for efficiency or deadlines, we lack a formal characterization of levels of determinism, needed by industries to assess whether or not such trade-offs are acceptable. To bridge the gap, we introduce the notion of D-bounded eventual determinism, where D is the maximum out-of-order delay of the input data. We design and implement TinTiN, a streaming middleware that can be used in combination with user-defined streaming applications, to provably enforce D-bounded eventual determinism. We evaluate TinTiN with a real-world streaming application for Advanced Metering Infrastructure (AMI) monitoring, showing it provides an order of magnitude improvement in processing performance, while minimizing delays in output generation, compared to a state-of-the-art strictly deterministic solution that waits for time proportional to D, for each input tuple, before generating output that depends on it.

3.1 Introduction

Data stream processing [1] is widely adopted for analysis of continuous streams of data produced in Cyber-Physical Systems (CPSs), for extraction of information useful for the operation, protection and dependability of the systems (e.g., smart meters data validation [2, 3] or vehicular data analysis [4–6]). Moreover, it is compliant with the needs for decentralized processing and furthermore, the research community is investing significant efforts in encompassing parallelism for stream processing for a large spectrum of devices, from embedded edge units to high-end servers.

Streams consist of sequences of tuples and are unbounded by definition. Therefore one-pass analysis is commonly performed on *windows* of data, whose boundaries change following the time carried by tuples' *timestamps*. A key challenge in processing data from distributed sources resides in its processing order, since the latter can influence the results. Simply put, the results for a certain window of tuples are accurate and can be produced as *deterministic* outcomes, depending on the condition that there are no still-to-be-processed tuples (because of late arrivals) contributing to such window. In this sense, totally ordered streams with no late arrivals simplify the generation of accurate, deterministic results.

Tools such as Viper [4] make sure that results from processing parallel streams are deterministic, by building on sorting techniques. *Relaxed determinism* guarantees are nonetheless desirable and preferable for some applications for which fast (but possibly not accurate) results are more valuable than accurate but late ones [27, 28]. Notice that, sorting of all input data and delaying processing due to few late arrivals, can unnecessarily penalize parts of the analysis that do not depend on late arrivals. An example application is data validation in an Advanced Metering Infrastructure (AMI) system of an electricity grid, to distinguish out-of-range values or value-patterns by Smart Meters (SM) that malfunction [2].

Why existing approaches fall short?

Available approaches for relaxed determinism fall short for at least three reasons. First, there is lack of a formal characterization of the possible results produced by a streaming application with relaxed determinism guarantees. Such a characterization is needed by data analysts in order to understand and estimate whether the effects of relaxed determinism are adequate or not for sensitive applications, when the latter's outcomes influence the dependability of a system.

Second, existing approaches that deal with out-of-order tuples are either integrated within a specific Stream Processing Engine (SPE) or require ad-hoc coding to maintain fine-grained control. The Apache Flink Streaming API [30] and Apache Beam Streaming pipelines [31] are examples of SPE-specific solutions. Both allow for processing of out-of-order tuples by introducing watermarks and multiple evaluations of windows, as discussed in the Dataflow model [5]. However, both require careful considerations about how duplicate or updated results are handled within the query. Enhanced stateful operators [12] or storing and restoring state for late arrivals [13] are other example approaches that, by requiring additional functionality of the SPE, can result in limited usability. Data analysts might not have the option to choose which SPEs should be used and could also lack the advanced programming skills needed to integrate an approach in a given SPE. Avoiding enhanced operators allows the analysts to use any SPE that supports basic aggregation operators.

Third, existing solutions can have prohibitive memory overheads when keeping all data in memory for a given lateness interval, as detailed later in the paper. Hence, their usage in large CPSs, composed of computationally-constrained devices, can also be limited.

Contributions

Motivated by these observations, we formalize the concept of *D-bounded eventual determinism* (D being a known bound on the timestamp-based out-of-order delay of late input tuples). We also propose TinTiN; *a streaming middleware*, that can top-up the guarantees of aggregation applications that originally ensure determinism for sorted input sequences, to enforce D-bounded eventual determinism for out-of-order input sequences, without requiring modification of the application or the SPE, if the application conforms to a set of assumptions (essentially providing some information about its semantics and the data fed to it; cf. section 3.3).

TinTiN does not delay results that can be accurately generated when no data is missing, while it "replays" portions of input data, when there are late arrivals. The "replayed" data is fed to an *application's replica*; to prevent the arbitrary time order (and possible overlap) of the relayed data, TinTiN manipulates their timestamps, (hence, their "time travelling") safely and according to the application's semantics. Since data can be "replayed" by TinTiN, some results are not guaranteed to be delivered exactly once. Such a behavior has been proposed in pioneer SPEs such as Borealis [27] (with the introduction of special UNDO or TENTATIVE tuples) and more recently in the Dataflow model [5]. Existing approaches, nonetheless, have large memory requirements since they maintain large windows and also demand that all operators can handle updated results.

In summary, we show that TinTiN enables, without changes on an SPE's operators, (i) timely processing of data, i.e. as soon as it is available, allowing the user to act on preliminary results immediately, (ii) the generation of final results, identical to the ideal case of no late arrivals, as soon as the relevant data has arrived, and (iii) small memory and time overheads, compared with state-of-the-art solutions (e.g. Apache Flink), which can guarantee determinism by processing data when the D bound expires (i.e., when late arrivals can no longer be seen), with output latency proportional to D and large memory overheads. These properties can make the difference between an approach being impossible and possible to consider in deployments, as we show in the example massive-data industrial use-case on data validation in our evaluation; in particular this use-case has been the key motivation for working on the problem.

The rest of the paper is organized as follows: the preliminaries are covered in section 3.2 after which we describe our system model in section 3.3, followed by the formal definition, goals and evaluation metrics of D-bounded eventual determinism in section 3.4. TinTiN's overview and core functionality, including algorithmic design are in section 3.5 and section 3.6, while section 3.7 evaluates TinTiN with our real-world application. Other related work and concluding remarks are discussed in section 3.8 and section 3.9.

3.2 Preliminaries

3.2.1 Stream processing

In data streaming *applications*, data is processed by *queries*, i.e., Directed Acyclic Graphs of *streams* and *operators*, deployed and run by SPEs. In the remainder, we use the terms query and application interchangeably. Each stream carries *tuples* sharing a schema $\langle ts, A_1, \ldots, A_n\rangle$, where ts is the tuple's *event* timestamp (which carries the notion of time for the query's operators [5]), and $A_1, \ldots, A_n$ are application-specific attributes.

We focus on applications composed by a sequence of one or more *stateful* aggregate operators, as well as by an arbitrary number of *stateless* operators. We use the term *user-defined application* (UDA) to refer to one such application. *Stateful aggregate operators* (also refered to as *Aggregates*) produce results that depend on multiple input tuples (e.g., to compute an average value). *Stateless operators* on the other hand process tuples without maintaining state that depends on the processed tuples (e.g., by filtering tuples based on their values or mapping their input schema to a different output schema). Stateless operators do not change timestamps, as in e.g. Apache Flink [30].

Commonly each Aggregate maintains a *sliding window*, a portion of the recent input tuples, that are processed to deliver results as output tuples. More specifically, we consider that the stateful aggregation of each Aggregate of a UDA is defined over a time-based sliding window W, characterized by its size WS and advance WA, and a set of functions $\{f_1, f_2, \ldots\}$.For example, an Aggregate could maintain a sliding window with WS 2 hours and WA 1 hour to maintain consecutive readings for a smart meter (SM) in order to calculate the hourly consumption by taking the difference between the readings. Notice that different Aggregates of the same UDA can have different size and advance parameters for their windows.

Each Aggregate defines an optional *key-by paramete*r (a subset of the input tuples' schema). If such a parameter is set, the aggregate maintains dedicated windows for each distinct set of key-by values observed in the stream. For the AMI measurements validation example, the input schema is $\langle ts, smID, cc\rangle$, where $smID$ is the ID for the SM and cc is the cumulative consumption that the meter has registered at timestamp ts. For ease of notation, we assume such a key attribute, denoted by k, is defined for all tuples. This does not affect generality, since all input tuples can share the same k value if they are to be aggregated in the same window.

In the remainder, we use the term *window* to refer to the object that is maintained by an Aggregate for each key-by value and evolves according to the tuples being processed, while we use the term *window instance* to refer to the *window covering a specific time interval.* As an example, for an Aggregate with WS and WA set to 1 hour and 30 minutes, respectively, a window instance could refer to the window covering the interval [08:00,09:00), while the subsequent instance is the one covering the interval [08:30,09:30).

Being W the window an Aggregate maintains for a key-by value, each window instance W^i covers the information of all tuples $t_i | t_i.ts \in [W^i_L, W^i_R)$, where W^i_L is the *left boundary* of W^i, and $W^i_R = W^i_L + WS$ is the *right boundary* of W^i. Initially, W^0 covers the first WS-long interval at event time 0. Then, the evolution of window W depends on three methods: `add`, `fire` and `remove`. These methods are invoked by the SPE maintaining W as specified in the following:

S1 Method `add` is invoked for each input tuple $t|t.ts \in [W_L^i, W_R^i)$ and (optionally) used to update the state of functions $\{f_1, f_2, \ldots\}$ (if the latter can be updated incrementally).

S2 Method `fire` is invoked as soon as an input tuple $t|t.ts \geq W_R$ is received. Then, the outcome of functions $\{f_1, f_2, \ldots\}$ is retrieved and forwarded as an output tuple. The timestamp of such output tuple is set to W_L. Let that tuple be called the *result* of that window instance, denoted by result(W_L).

S3 Method `remove` is invoked immediately after the `fire` method is invoked. All tuples $t|t.ts \in [W_L^i, W_L^i + \mathit{WA})$ are removed from W and the state of functions $\{f_1, f_2, \ldots\}$ is updated accordingly (if they define one). Then, W is shifted forward by WA (i.e., W_L^i and W_R^i are updated to $W_L^i + \mathit{WA}$ and $W_R^i + \mathit{WA}$, respectively), thus moving to window instance W^{i+1}.

Methods `fire` and `remove` are repeatedly invoked, one after the other, until the input tuple t' triggering the invocation of the `fire` method falls within W's left and right boundaries. Continuing the previous example, if the current window instance covers [08:00,09:00) and the next input tuple has timestamp 10:15, methods `fire` and `remove` would be invoked 3 times each, for W to eventually cover [09:30,10:30), to which the input tuple falls in.

We assume method `fire` is only invoked for window instances containing at least one tuple. Hence, no results are initially produced for window instance $W^0, \ldots, W^i$, where W^i is the earliest window instance to which the first tuple of a given key falls in.

We assume that all the windows maintained for the different keys observed in the input stream are aligned. That is, if a window for a certain key shifts to a certain $[W_L^i, W_R^i)$ period, so do all the windows of other keys maintained by the application. This is enforced by invoking methods `fire` and `remove` on all existing windows when an input tuple $t'|t'.ts \geq W_R^i$ is processed. Also, we assume that a window for a new key value is created when an input tuple carrying such value is processed and deleted if, after invoking the method `remove` for it, no tuple is left in the window.

Running Example We introduce an example query, based on a real world use case to illustrate the concepts in the paper. Smart Meters (SM) can break down gradually, resulting in unreliable readings. In the example the start of the breakdown can be detected by a *pattern*: one or more large hourly consumption values followed by a negative hourly consumption within four hours. As soon as the pattern is detected for a specific SM, a technician is deployed to replace the SM. The pattern is identified by a query consisting of two Aggregates. Aggregate 1 calculates the hourly consumption by taking the difference between every two consecutive readings. It therefore has a WS 2 hours and WA 1 hour. Aggregate 2 has WS 4, WA 1 hour, and produces an alert if the pattern is matched. The value of the alert is the number of large consumption values in the match. Figure 3.1 shows a "cross-section" of an execution of the query and the results it produces for an input stream consisting of 10 tuples for a single SM.

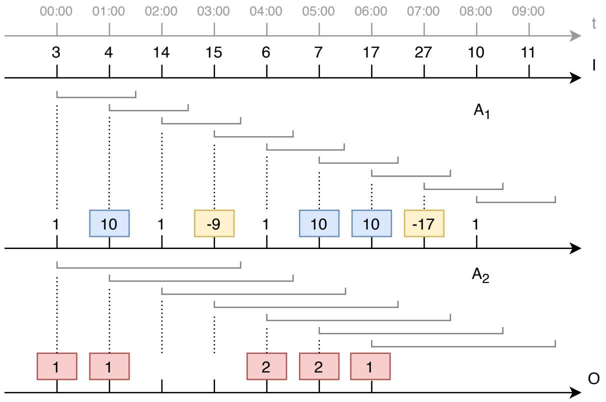


Figure 3.1: *Example "cross-section" of a query execution, processing data for one Smart Meter. The input stream I shows the readings, while their timestamps are given on the time axis t. The query has two aggregate operators. A_1 calculates the hourly consumption by taking the difference between two consecutive readings. A_2 outputs alerts if a pattern is found in its window. The pattern is one or more large consumption values (marked in blue) followed by a negative value (marked in yellow) within four hours. The value of the produced alerts (in red) indicates the number of large values in the match. A_1 has WS 2 and WA 1, while A_2 has WS 4 and WA 1. The timestamp of results is equal to the left (inclusive) boundary of the window instance that produced it. The right boundary of a window is exclusive.*

3.2.2 Strict determinism

The execution of a stateful operator is deterministic if the operator's semantics are correctly enforced, independently of (i) the operator implementation and deployment and (ii) the input data ordering. For instance, when running an aggregate operator counting tuples on a per-key basis over a window with *WS* and *WA* set to one hour, if 5 tuples referring to key k and having timestamps $\in [08{:}00, 09{:}00)$ are delivered in the input stream, the Aggregate's execution is deterministic if the operator produces the correct count for key k and window $[08{:}00, 09{:}00)$ independently of whether (i) the operator is run sequentially by one thread or in parallel by several threads (e.g., assigning each thread a subset of keys) and (ii) the input tuples are delivered in timestamp order or not to the thread(s) running the Aggregate's analysis.

For a *single-threaded aggregate operator* whose sliding window's execution evolves over time upon the invocations of methods `add`, `fire` and `remove` as described in subsection 3.2.1, it is known from the literature that deterministic execution is enforced if:

[a] Functions' $\{f_1, f_2, \ldots\}$ analysis uses no randomness and depends exclusively on input tuples' attributes, and

[b] Input tuples are fed in timestamp order.

If these conditions hold, the method `fire` is invoked for each window W^i only after the method `add` is invoked for all the tuples contributing to W^i, and each output tuple depends exclusively on the values carried by the tuples contributing to it (including the timestamp, which determines the order in which tuples are added to the window). For a *multi-threaded aggregate operator*, in which distinct threads carry out the analysis of different keys, the above set of sufficient conditions to imply determinism, are commonly complemented with the following one:

[c] Exactly one of the threads running the analysis in parallel is responsible for the analysis of a given key.

This is a sufficient condition to prevent inconsistencies of state updates in the analysis. It can be possible to prevent the latter with other methods, however this is a common one practice.

Regarding guaranteeing *determinism for a UDA*, a *sufficient condition* is to ensure (i) determinism on the operator level and (ii) time-stamp-ordering in the data flows to the operators, including the internal, inter-operator ones [4,14,15]. In the following, we say that a streaming aggregation application enforces *strict determinism* if such a condition is met. We use the term strict to differentiate the determinism from the relaxed one we propose here.

3.3 System Model

Here we specify some more detail about the type of the User-Defined Applications (UDAs, serial aggregation applications) targeted. We assume the UDA is fed with one input stream and it can run in parallel the analysis of different *keys*. We assume that strict determinism is guaranteed for the UDA by guaranteeing determinism for all the operators composing the UDA and the inter-operator flows, as explained in subsection 3.2.2. We wish to note that each *result* t_o of the UDA, given that it is a tuple that bears the timestamp of the last aggregate in the UDA, can be uniquely indexed by that timestamp (this is due to the fact that output tuples of operators are timestamped using the left boundary of the window instance they correspond to, as mentioned in S2 in section 3.2).

We also require the following to hold (we refer the reader to section 3.6 for further discussions and the justification of these assumptions):

A1 By observing the last two tuples t_A, t_B received for a certain key k s.t. $t_B.ts > t_A.ts$, it is possible to know whether a *hole* exists, i.e. there exists a tuple with timestamp $\in (t_A.ts, t_B.ts)$ that can either arrive late or not at all. This is the case, for instance, when the input data sampling period is known or when an enumerator attribute is defined for the input data.

A2 A known maximum out-of-order delay D allows to distinguish late arrivals that can still be received, from those that will not (i.e., that can be ignored). More concretely, given any arbitrary point in any arbitrary execution, being τ the highest timestamp received by the application, late arrivals with timestamps $\in [\tau - D, \tau]$ can still be received, while those with timestamps smaller than $\tau - D$ cannot (i.e., they can be ignored).

A3 Analysis of data might be related to its seasonality, i.e. results could differ depending on e.g. the hour of the day, or the day of the week.

We define the **periodicity** P of a UDA as the maximum period that is relevant for the seasonality of the data; e.g., P is 24 hours if tuples are treated different depending on the time of day of their timestamp, while it is one week if treatment also depends on the day of the week. If the analysis of the UDA is not related to the seasonality of the data, we consider $P = 1$, else, we assume that its P is known.

A4 A sorted sequence of tuples, denoted RC_{flush} that triggers at least one output tuple is made available to TinTiN.

A5 The sequence of stateless and aggregate operators composing the UDA is known, as well as the finite window sizes and advances of all the aggregate operators in the UDA.

Note that the UDA developer who wants to use TinTiN to deal with out-of-order input data will have all the information required.

3.4 D-bounded eventual determinism

As mentioned in section 3.1, outputting information in a timely fashion is useful or critical in certain applications. We propose *D-bounded eventual determinism* to formalize guarantees that enable timeliness of output that depends on timely available input, while loosening only part of the requirements, compared to strict determinism.

Definition 1. *Given a stream I that is not timestamp-sorted, we say that I is* **within lateness bound** *D if, for any t in I, for all subsequent tuples t' in I for which $t'.ts < t.ts$, the condition $t.ts - t'.ts \leq D$ holds.*

Definition 2. *Given:*

- *A, a streaming application that supports deterministic execution for timestamp-sorted streams,*
- *I, a timestamp-sorted input stream,*
- *O, the output stream produced by A when all input tuples from I are fed to A, and*
- *E, the set of all possible executions in which A is fed with a permutation of I that is within lateness bound D (Definition 1);*

we say that A is extended to a D-**bounded eventually deterministic** *streaming application A' if, for A, A' and any $e \in E$, being:*

- *$o_{ts,k} \in O$ the tuple output by A for timestamp ts and key k,*
- *$o'^{1}_{ts,k}, \ldots, o'^{n}_{ts,k}$ the ordered sequence of output tuples produced by A' for timestamp ts and key k (with $n \geq 1$),*

then $o_{ts,k} = o'^{n}_{ts,k}$.

Definition 2 says that an eventually deterministic streaming application A' (derived from A that is strictly deterministic when it processes timestamp-ordered tuples), processing a stream within lateness bound D, produces all tuples that A produces when it processes the same input but sorted. A' might produce more tuples than A, but for every tuple produced by A, the latest tuple with identical timestamp and key produced by A' will be equal. Note that, if A can produce multiple output tuples for a specific timestamp and key, then these should be distinguishable, by e.g. another attribute.

Evaluation metrics

When comparing the results observed for a streaming application A' with relaxed deterministic guarantees (when fed with an input stream I that might be unsorted), with those of a UDA A that enforces strict determinism for a sorted version of I, we say an output tuple produced by A' is (i) **exact** if an output tuple carrying the same attribute values (for the same timestamp and key k) is produced by A, (ii) **duplicate** if another, exact tuple (for the same timestamp and key k) has already been produced by A', or (iii) **different** if an output tuple with different attribute values for the same timestamp and key k (i.e. not exact) is produced by A. (iv) It is also possible to have tuples **omitted** by A', i.e. tuples produced by A but not by A'.

Note that if A' enables D-bounded eventual-determinism for an input stream I that is within lateness bound D, no omitted output tuples exist and, for each different output tuple (if any), one or more exact output tuples are also later produced for a given timestamp and key k. If, on the contrary, A' does not enable D-bounded eventual determinism, both omitted and different output tuples not followed by at least an exact output tuple can be observed.

Example continued: Recall the running example where occurrences of a specific pattern indicating Smart Meter hardware failure are identified. Figure 3.1 shows the query processing data for a single SM where the pattern occurs twice. Figure 3.2 shows the same example, but with two missing input tuples. The first occurrence of the pattern is not identified when this input is missing, the alerts from this occurrence are *omitted*. The second occurrence is only partially affected by the missing input and is identified. The output with timestamps 04:00 and 05:00 is *different*. The output for timestamp 06:00 does not depend on the missing data and is *exact*. If exact output is produced once more, for example by processing the relevant data at a later time when the missing tuples have arrived, it would instead be *duplicate*. The figure also illustrates one of TinTiN's advantages: by being able to continue the processing even if data is missing, TinTiN identifies the second occurrence of the pattern without waiting for the missing data to arrive. A technician can be dispatched to the affected SM immediately after identification, minimizing the amount of unreliable data sent by the SM.

Let us also define the **logical latency** of an output tuple t, the difference between $t.ts$ and the highest timestamp observed in the stream when the exact result for t is produced.

Note that for each output tuple t, if strict determinism is enforced by simply postponing the processing of each input tuple by D time units from its timestamp (e.g., as done by Apache Flink's Complex Event Processor) its logical latency is D, while it can be made (significantly) smaller than D, by leveraging finer-grained techniques for managing out of order data, as we show with TinTiN, when enforcing D-bounded eventual determinism.

Similarly to logical latency, **responsiveness** is defined as the difference between the highest timestamp observed in the stream when a late tuple t_l arrives and the highest timestamp in the stream when the final exact result for t_l is produced. In the case where strict determinism is enforced by postponing processing as described above, the responsiveness will be D minus the lateness of t_l, while it can be made smaller by TinTiN just as the logical latency.

3.5 TinTiN's overview

Here we give an overview of TinTiN, while in the subsequent section we describe its core design and its algorithmic implementation.

TinTiN processes data even though some tuples are late. Our pattern matching example with Smart Meter data shows that this allows some matches to be identified despite missing data. Figure 3.2 illustrates this, the alerts with timestamps 04:00, 05:00 and 06:00 are produced. The alerts from 04:00 and 05:00 have a different value compared with the alerts produced when no data is missing. The implications of such differences are application specific. In this particular example a technician will be deployed regardless of the value of the alert, so there is no direct implication. In order to eventually deliver the exact output, TinTiN *replays* portions of data when late data arrives. Such portions are processed by a copy of the UDA which produces updated results. This is illustrated in Figure 3.4 where a portion of data is replayed. The extra alerts due to replaying might cause another technician to be dispatched; i.e. the dispatcher has to take extra care when such results are produced.

Considering the example, let us proceed with the description of the middleware: TinTiN does not delay results that can be accurately generated in the cases of no missing data. When intervals of data with missing tuples have been processed, as also mentioned in the example, it later *replays* sufficient portions of the input, when late data arrives. Moreover, it aims at achieving the aforementioned behaviour efficiently, both from the point of view of computational and memory overheads, as well as from the perspective of limiting the amount of "unnecessary", partial, results. Furthermore, it works as a wrapper of the UDA in any SPE, without requiring to modify the internals of the latter. The following list of steps and Figure 3.3, outline at a high level the aforementioned procedures.

---

[a] TinTiN forwards to the UDA the input tuples that arrive in increasing timestamp order.

[b] TinTiN also temporarily maintains a sufficiently large portion of the input stream that initially contained some *hole(s)* (caused by tuple(s) being late), named **relevant context** of the holes(s).

[c] Later, if late tuples arrive within the bound D, TinTiN "replays" the relevant context of the respective hole(s), to get refined results. To avoid

interference with the processing of the in-order data, the replayed data is fed to a *replica of the UDA* (UDA_R; c.f. Fig 3.3).

[d] To prevent that the arbitrary time-order and the potential overlap of relevant context of late tuples to affect consistency of results, when replayed at UDA_R, TinTiN shifts forward by a given offset all the timestamps of each relevant context (hence, "time travelling"), safely and according to the application's semantics and shifts back the final results' timestamps to the original ones when forwarding those results to the user.

Why can TinTiN guarantee eventual determinism? The tuples that arrive in timestamp order and without holes, generate the same result as the strictly deterministic case when fed to the UDA. If a portion of input forwarded to the UDA contains a hole though, such a portion is also sent to the UDA$_R$ (at least once) when holes are later filled in (with tuples arriving with maximum lateness D). This implies that multiple and possibly different, improved, versions of the same result could be delivered to the user. However each result will be based on a relevant context of hole(s) that is gradually filled in by late tuples. Following this procedure, TinTiN eventually delivers the output that would be generated in the ideal case (in which there would have been no late arrivals) for the respective window instances, i.e. it satisfies D-bounded eventual determinism when fed with a stream that is within bound D.

How is TinTiN's processing safe and consistent? The aforementioned 4 steps need to be carried out in order to make sure that safety in processing is preserved, i.e. that state created by the processing of different portions of data (overlapping intervals, intervals that are forwarded out of order) is not inconsistently mixed up when data is being replayed. This is achieved by replication, in a two-folded fashion: (i) for separating the processing of replayed tuples from the processing of in-order tuples, the replaying of the relevant context of holes are carried out by UDA$_R$ and not by UDA; (ii) for avoiding UDA$_R$ to mix up replaying of overlapping relevant contexts of holes, the latter are replayed with modified time, which is modified back to the original, when the result is produced.

For TinTiN to efficiently carry out the above, the following questions need to be answered, as explained in the next section.

Q1 Which results can be improved and forwarded to the end-user? In case of late data, it is straightforward to identify such results for a query composed by a single aggregate operator, but there is more to think about for arbitrary UDAs.

Q2 What relevant context to replay? For each result that can be improved, how to identify which source data is sufficient to maintain in memory, in order to replay if/when late tuples arrive?

Q3 How to replay efficiently? If, due to one or more late tuples, TinTiN needs to re-produce multiple results, can it do it efficiently without replaying many times overlapping relevant contexts?

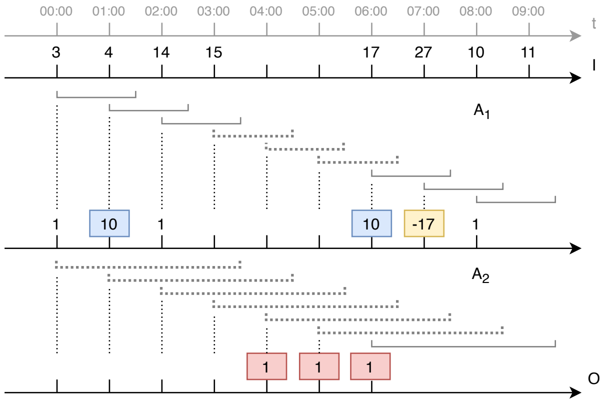


Figure 3.2: *The same "cross-section" as seen in Figure 3.1, but with readings from 04:00 and 05:00 missing at the time of processing. All window instances that are affected by the missing input are dashed.*

Q4 How to replay safely? How to ensure that the "time travelling" is correct, i.e. that the processing state of the UDA_R does not get mixed up with that of other replayed data?

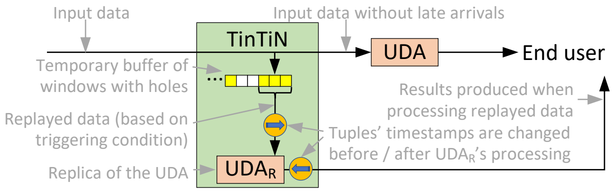

Input data
Input data without late arrivals
TinTiN
UDA
End user
Temporary buffer of windows with holes
...
Results produced when processing replayed data
Replayed data (based on triggering condition)
Tuples' timestamps are changed before / after UDAR's processing
Replica of the UDA
UDAR

Figure 3.3: *Overview of TinTiN's architecture.*

3.6 TinTiN's core functionality

This section covers TinTiN's design and its "*time travel*" mechanism, answering the questions of the previous section. Then, it presents TinTiN's algorithmic implementation and the main argument for satisfying D-bounded eventual determinism.

3.6.1 Answering Q1: Which results can be improved and forwarded to the end user

A hole in the input stream of a UDA can affect multiple results, as shown in Figure 3.2. The quality of such results can be improved by reprocessing the relevant context of the holes once the late tuples have arrived. Let us first determine the results that are *affected* (i.e., that can be improved) by late values for a UDA with a *single Aggregate*:

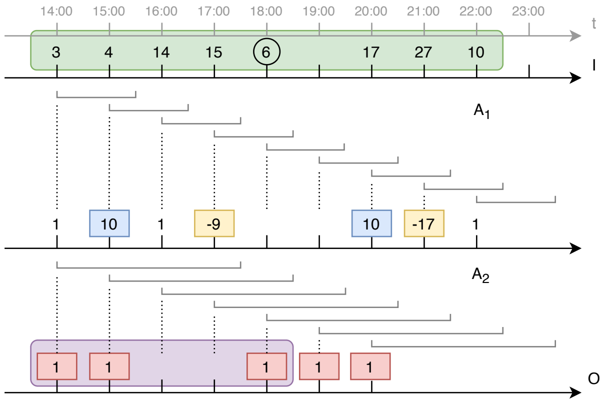


Figure 3.4: *Example "cross-section" of an execution at the UDA$_R$ showing the relevant context of a hole (subsection 3.6.2, Figure 3.2), in green, as it will be replayed, when the hole is filled by the encircled late arrival. The timestamps of the input data have been shifted by TinTiN (subsection 3.6.4), e.g. timestamp 00:00 in Figure 3.2 is shifted to 14:00. The output affected by the late arrival, i.e. that can be improved by the late arrival (subsection 3.6.1), is shown in purple. For example, the alert with timestamp 14:00 could not be produced without the late arrival, cf. Figure 3.2.*

Lemma 1. *Given an Aggregate A with window size WS, the timestamps of A's results that are potentially affected by a missing input tuple t with timestamp ts, are in $(ts - WS, ts]$.*

Proof. The timestamp of any affected result, potentially improvable by a late input t, equals the left boundary of any window instance that contains t (cf. S2 in section 3.2). The window with the earliest left boundary that contains t starts no earlier than $ts - WS$. The window with the latest left boundary containing t cannot start after ts, since the left boundary of a window is inclusive and windows with a left boundary larger than ts do not include t. □

Consider again the *example* in Figure 3.2 with two Aggregates. A hole in the input stream affects results from A_1, which in turn affects more results from A_2. An interval that contains the timestamps for all of the affected results is given by the following lemma.

Lemma 2. *Consider a UDA with n Aggregates in series with window sizes $WS_i; i \in [1, n]$, and a missing input tuple t with timestamp ts. The timestamps of the UDA results that are potentially affected due to the missing input are in $(ts - \sum_i WS_i, ts]$.*

Proof. The timestamps of the affected results of the first Aggregate A_1 are contained in the interval $(ts - WS_1, ts]$ (Lemma 1). The same argumentation, applied for the second Aggregate A_2 and for $ts - WS_1$ and ts, allows to find the bounding interval for timestamps of A_2's affected results, implying the interval $(ts - (WS_1 + WS_2), ts]$. The same reasoning applied recursively for any subsequent Aggregate, results in the interval in the lemma statement. □

It should be noted that Lemma 2 implies:

Observation 1. *The results to forward to the end user when data is replayed due to a late input tuple with timestamp ts, are the ones with timestamps in* $(ts - \sum_i WS_i, ts]$.

This is illustrated in Figure 3.4 where the relevant context for the late arrival is *replayed.* The updated results to forward are marked, while results outside the interval in Observation 1 (i.e. not to be forwarded) are removed from the output stream.

3.6.2 Answering Q2: Sufficient input to replay

To determine the exact content of the sufficient relevant context of a hole, we need to find all input tuples that are relevant to the potentially affected results, so that those tuples are stored and replayed together with late tuples if/when the latter arrive.

Lemma 3. *Consider a* result tuple *t with timestamp ts, produced by an Aggregate A with window size WS. The timestamps of all input tuples to A that are relevant for (i.e. potentially affect) t are in* $[ts, ts + WS)$.

Proof. The lemma follows directly from the fact that a result with timestamp *ts* is produced by a window instance whose inclusive left boundary is *ts* and exclusive right boundary is *ts* + *WS*. □

Figure 3.4 shows the *running example*, marking all relevant input tuples for a set of results from the example-UDA. The bounding interval for these tuples follows from the following lemma.

Lemma 4. *Consider a UDA with n Aggregates in series with window sizes* $WS_i; i \in [1, n]$*, and a* result tuple *t with timestamp ts. The timestamps of all input tuples to the UDA that are relevant for t are in* $[ts, ts + \sum_i WS_i)$.

Proof. Lemma 3 implies that the timestamps of the relevant input to the first Aggregate are in $[ts, ts + WS_1)$. The same argumentation can be applied for the second Aggregate for *ts* and $ts + WS_1$, to find the bounding interval for timestamps of affected results from the second Aggregate, implying the interval $[ts, ts + (WS_1 + WS_2))$. The same reasoning can be applied recursively for any subsequent Aggregate, resulting in the interval in the lemma statement. □

Combining the lemmas from subsection 3.6.1 with lemmas 3 and 4, we get:

Lemma 5. *Consider a UDA with n aggregate operators in series with window sizes* $WS_i; i \in [1, n]$*, and a* hole *in the input stream with timestamp ts. The relevant context of the hole is contained in the interval* $(ts - \sum_i WS_i, ts + \sum_i WS_i)$.

This is illustrated in Figure 3.4 for the *running example* UDA and the relevant context of the encircled late arrival.

3.6.3 Answering Q3: Replaying efficiently

The logic with which the relevant context*s* of late tuples, temporarily stored at TinTiN, are replayed, depends on a user-defined *triggering condition TC*. TCs can imply a trade-off between different properties, e.g. between efficient processing and how fast a result for a late arrival is produced, as explained in the following.

An *eager TC* could trigger the replay of a relevant context for a hole as soon as the late arrival filling it is received. Such a condition, reacting immediately to each late arrival, minimizes the time between receiving the late arrival and producing potential results to which it contributes. This could be beneficial for applications that need up-to-date (possibly different) results as soon as possible.

Alternatively, a *lazy TC* could instead trigger the replay of a relevant context of hole(s) for a certain key k when multiple holes have been filled. For use cases where late arrivals arrive in batches, this can be achieved by delaying the firing of the trigger until an on-time tuple is observed for k. Such TC trades increased logical latency for better processing throughput. Note that it is possible to construct a TC that favors efficient processing even more by waiting even longer before triggering. More efficient processing is achieved by combining the relevant context for multiple late arrivals where it overlaps. This is possible due to the associative property of the relevant context, shown here:

Lemma 6. *Consider a UDA with n aggregate operators and two holes with timestamps ts and $ts + x$, for any $x > 0$ s.t. there is overlap in their respective relevant contexts. The set of affected results produced by replaying each relevant context separately is equal to the set of affected results produced by replaying the union of the relevant contexts once.*

Proof. The affected results produced by replaying the relevant context for the late arrival with timestamp ts have timestamps in the interval $(ts - \sum_i WS_i, ts]$ according to Lemma 2. Results for the late arrival with timestamp $ts + x$, have timestamps in interval $(ts+x-\sum_i WS_i, ts+x]$. The size of the relevant context for the late arrival with timestamp ts is $(ts - \sum_i WS_i, ts + \sum_i WS_i)$, according to Lemma 4. Since the relevant context*s* of the two late arrivals overlap, $x < \sum_i WS_i$. Therefore $ts + x - \sum_i WS_i < ts$, implying that intervals for the affected results overlap and are contained in the interval $(ts - \sum_i WS_i, ts + x]$. This interval is equal to the interval obtained for the affected results by the union of the relevant context for ts and $ts + x$. □

Special consideration is due for holes that are filled after a *short amount of time* (ie. before their entire relevant context has arrived), while to obtain eventual determinism, it is required to replay *all* tuples that contribute to a result that can be improved by a late arrival. One way to ensure this, is to delay replaying until a tuple with timestamp larger than the largest timestamp in that relevant context arrives.

3.6.4 Answering Q4: Replaying safely

Replaying the relevant context of a late arrival directly to the UDA would cause the input stream of the UDA to become out of order and interfere with data currently being processed. For this reason, TinTiN replays data to a replica of the UDA, UDA$_R$. Note that UDA$_R$, being a replica of UDA, guarantees deterministic processing only for timestamp sorted input. However, replaying the relevant context of two different holes can be problematic for two reasons:

[a] It causes the input stream to UDA$_R$ to become out of order if the second relevant context starts with a timestamp lower than the highest timestamp of the first relevant context.

[b] It can cause erroneous results if the data for the different late arrivals ends up in common windows.

Intuitively, a straightforward solution to prevent this from happening, is to deploy a "fresh" UDA$_R$ (i.e., that has not yet processed any tuple) before replaying any past portion of input tuples. Deploying a fresh UDA$_R$ incurs large overhead though. Relying on a method that resets the UDA$_R$'s state is also not an option, since, for the sake of generality, we do not assume that the SPE or the UDA's programmer provide it.

TinTiN's novel approach is to (i) shift the timestamps of the tuples in the relevant context with an offset, so that UDA$_R$ is fed with an input stream with strictly monotonically increasing timestamps, and tuples from one replay cannot interfere with other replays; and (ii) shift back the timestamp of the result by the same offset.

3.6.4.1 Shifting timestamps of input to UDA$_R$

When processing data in the UDA, an input tuple will belong to a specific number of window instances for the first Aggregate. The same is true for a tuple produced by the first Aggregate, it will belong to a specific number of window instances for the second Aggregate and so on and so forth. The number of window instances a tuple belongs to depends on its timestamp as well as the window advance and size of the Aggregate. An example of this can be seen in Figure 3.1, where the output of the first Aggregate belongs to either one or two window instances of the second Aggregate.

In order to obtain correct results when reprocessing data, it is required that every tuple contributes to the same number of window instances as for the on-time processing. The way the tuples timestamps can be changed without affecting the number of window instances they contribute to is based on the following observations:

Observation 2. *Consider a UDA with* n *aggregate operators in series with window advances* $WA_i; i \in [1, n]$. *The starting times of the windows for all aggregate operators are aligned at time* 0, *since all windows start in* 0. *Such alignment also occurs at timestamps that are a multiple of* $LCM(WA_i)$ *(the least common multiple of all* WA_i*).*

Observation 3. *Consider a UDA and an input tuple* t *with timestamp* ts. *Let* d *denote the difference between* ts *and the nearest alignment of windows in the UDA smaller than* ts. *Now consider another tuple* t' *with timestamp* ts', *with difference equal to* d *between* ts' *and its nearest alignment of windows smaller than* ts'. *The number of window instances that* t *contributes to is equal to the number of window instances that* t' *contributes to.*

Consider for example a query with two Aggregates, the first one having *WS* 3, *WA* 2 and the second one *WS* 3, *WA* 1. Since subsequent windows for the first Aggregate overlap one hour, input tuples can contribute to either one or two window instances. Window alignment occurs every multiple of 2 ($LCM(1,2)$). The tuples with timestamps 5 and 7 both have distance 1 to their nearest window alignment and both contribute to two window instances.

We conclude from the observations that when changing timestamps, any multiple of $LCM(WA_i)$ can be added to the original timestamps to ensure that all tuples contribute to the same number of window instances in UDA$_R$ and the on-time UDA.

If the UDA has an internal periodicity P as described in assumption A3, cf. section 3.3, then this should be taken into consideration as well when shifting timestamps of data to replay. Tuples should not only contribute to the same number of window instances, but the periodicity should be preserved as well. For example if a UDA has P of one week, a tuple with timestamp 12:00 on a Monday should also have a timestamp 12:00 on a Monday after the timestamp is changed. This is accomplished based on the following observations:

Observation 4. *The periodicity P of a UDA is conceptually the same as a window with WS and WA equal to P. In other words: a new period starts at every multiple of P and has a duration of P.*

Observation 5. *Consider a UDA with n aggregate operators in series with window advances $WA_i; i \in [1, n]$ and periodicity P. The windows for all aggregate operators, as well as the start of period, are aligned at time 0, since all windows start in 0, as does the periodicity. Alignment also occurs at timestamps that are a multiple of $LCM(WA_i, P)$ (the least common multiple of all WA_i and P).*

The final consideration when shifting timestamps is that one sequence of replayed tuples should not interfere with another sequence of replayed tuples. This is achieved when the sets of results are produced by the sequences are disjoint. This can be accomplished by separating the timestamps from both sequences with a *safety distance*, *SD*. The size of SD is given by the following lemma, which follows directly from Lemma 2.

Lemma 7. *Given a UDA with n aggregate operators with window sizes WS_i and two sequences of tuples S_1 and S_2 where all timestamps in S_2 are larger than any timestamp in S_1, no results are affected by tuples from both S_1 and S_2 if the smallest timestamp in S_2 is separated from the greatest timestamp in S_1 by at least* $\mathrm{SD} = \sum_i WS_i$.

In conclusion, the following lemma justifies how timestamps can be shifted in a way to guarantee safety in the processing.

Lemma 8. *Consider a UDA with periodicity P and n aggregate operators in series with window advances $WA_i; i \in [1, n]$, processing two tuple-sequences S_1 and S_2. Adding $z \cdot LCM(WA_i, P)$ to the timestamps of the tuples in S_2, where $LCM(WA_i, P)$ is the least common multiple of all WA_i and P, and $z \in \mathbb{Z}$ so that $z \cdot LCM(WA_i, P) > SD$, guarantees that (1) all tuples in S_2 still contribute to the same number of window instances as they would have, had their timestamps not been changed, and (2) no results are produced that are affected by tuples from both S_1 and S_2.*

Proof. Property 1 follows directly from Observation 5. Property 2 follows directly from Lemma 7. □

3.6.4.2 Restoring timestamps of UDA$_R$ results

Results produced by the UDA$_R$ cannot be forwarded to the end user without restoring the timestamps, for obvious reasons. Therefore TinTiN should store a *mapping* of the changed and original timestamps in order to restore the timestamps for produced results. There is no guarantee that replaying a relevant context to UDA$_R$ will produce any results. Mappings that were stored by TinTiN can therefore become *stale*. Whether a mapping is stale or not cannot be inferred by setting a timeout for the result of a relevant context, since the UDA processing latency is outside TinTiN's control.

Notice that, since UDA$_R$ supports deterministic execution and is fed a timestamp-sorted stream (based on TinTiN's manipulation of timestamps), it results in a timestamp-sorted output stream (cf. subsection 3.2.2). Hence, if a replayed sequence S results in output, stale mappings of changed and original timestamps for sequences replayed earlier can be safely discarded by TinTiN, since results for them will not be produced after S's.

To prevent the size of changed and original timestamp mappings to grow beyond a maximum size, TinTiN can replay the sample sequence of tuples that is known to trigger an output (assumption A4 in section 3.3) to flush stale mappings (in this case, without forwarding the result to the UDA user).

3.6.5 Synthesis: TinTiN's algorithmic design

Table 3.1: *Abbreviations and parameters used in Algorithm 1*

Parameter	Definition
UDA, UDA$_R$	User-Defined Application and TinTiN's replica of it
WA[]	All WA in the UDA
D	Max delay on UDA's input tuples
TC	Triggering condition
SD	Safety distance between successive replays by UDA$_R$ as defined in subsection 3.6.4
P	UDA's periodicity as defined in section 3.3
β[]	TinTiN's internal array of sorted tuples (one array per key k, used to store the most recent tuples)
B[]	TinTiN's internal array of sorted tuples (one array per key k, used to store tuples contributing to the relevant contexts of holes)
T[]	TinTiN's internal array of timestamps and corresponding manipulated timestamps
λ[]	TinTiN's internal array for late arrivals
max_ts	highest timestamp seen so far
max_ts_{man}	highest manipulated timestamp replayed to UDA$_R$
RC_t	The relevant context for tuple t
RC_{flush}	Sample data that triggers an output from the UDA

Algorithm 1: TinTiN's algorithm, upon receiving tuple t

```
1  $max\_ts = \mathtt{max}(max\_ts, t.ts)$;
2  if $t.ts \geq max\_ts$ then
3      forward $t$ to UDA and add $t$ to $\beta[t.k]$;
4      if $\beta[t.k]$ contains holes then
5          add $\beta[t.k]$ to $B[t.k]$ (excluding duplicates);
6  else if $t.ts > max\_ts - D$ then
7      add $t$ to $B[t.k]$;
8      add $t$ to $\lambda[t.k]$
9  for all $t^i$ in $\lambda[t.k]$ for which $TC$ holds do
10     replay $(RC_{t^i})$;
11 if $\exists t^1$ in $\lambda[t.k]$ so that $t^i.ts < t.ts - D$ then
12     replay $(RC_{t^i})$;
13 if $\exists t^i$ in $B[t.k]$ so that $t^i.ts < t.ts - (D + size(RC_t))$ then
14     remove $t^i$ from $B[t.k]$;
15 if $size(T[]) > threshold$ then
16     replay $(RC_{flush})$
17 replay($RC_t$)
18     $ts_{min} = min(ts \in RC_t)$;
19     $ts_{max} = max(ts \in RC_t)$;
20     $M = LCM(WA[], P)$;
21     find min $(z \in \mathbb{Z}) : ts_{min} + z \cdot M > max\_ts_{man} + SD$;
22     shift $RC_t$ tuples' timestamps with $z \cdot M$;
23     for results affected by $t$ do
24         store $t.ts, t.ts_{man}$ pairs in $T[]$;
25     $max\_ts_{man} = ts_{max} + n \cdot M$;
26 send $RC_t$ tuples to UDA$_R$ in timestamp-order; get results $t_0[]$;
27 flush old state from $T[]$;
28 if $RC_t \neq RC_{flush}$ then
29     shift $t_0[]$'s timestamps back;
30     forward $t_0[]$ to output ;
```

Here we focus on the algorithmic description of TinTiN, also shown in Algorithm 1 (based on the abbreviations and parameters listed in Table 3.1). For each key k, each input tuple t with a timestamp greater than or equal to that of the previous input tuple observed by TinTiN is forwarded to the UDA. When t is observed, TinTiN could identify t as part of the relevant context of a previously observed hole, if such hole is within time-distance $\sum_i WS_i$ from t (Lemma 5). Even if no such hole has been observed, t could still turn out to be part of the relevant context for a hole later observed within time-distance $\sum_i WS_i$ from t (Lemma 5). To be efficient, TinTiN aims at maintaining t only if t is part of at least one relevant context. To do this, TinTiN initially adds each new incoming tuple that is not a late arrival into a key-dedicated map $\beta[t.k]$ of size $\sum_i WS_i$. If a hole is observed while t is in $\beta[t.k]$, then t is part of a relevant context that could be replayed in the future. Only in this case, t is moved to a larger key-dedicated map $B[t.k]$, in which relevant contexts are kept as long as a late arrival within bound D could still be received (L5). If t is a late tuple, but it is no more than D time units late compared to the highest timestamp observed so far, t is added to both $B[t.k]$ (L7) and $\lambda[t.k]$, a map that stores late arrivals. If t is more than D time units late, it is discarded.

Subsequently, the triggering condition is checked for all late arrivals in $\lambda[t.k]$. For each late arrival for which the triggering condition TC holds (cf. subsection 3.6.3), method `replay` (L21-30) is invoked. In this case, RC_t, i.e. the tuples in the relevant context for the late arrivals from $B[t.k]$ are forwarded to UDA$_R$ once their timestamp is changed, while respecting the periodicity P of the UDA as well as the window advances of the UDA, as described in subsection 3.6.4.

Manipulated timestamps for which an affected result can be produced by the UDA are paired with the original timestamps and stored in $T[]$, to accommodate shifting back the timestamps. If the size of $T[]$ exceeds a predefined threshold, RC_{flush} is replayed to remove stale mappings from the array (L16), as described in subsection 3.6.4. If results are produced by UDA$_R$, from any RC_t that is not RC_{flush}, the timestamps of the results are then moved back and the results are forwarded to the end user. All tuples in $B[]$ contributing to a relevant context for which late arrivals will not be received (based on D) are replayed to UDA$_R$ and then removed from $B[]$ (L14).

Lemma 9. *Given a UDA that supports deterministic execution for timestamp-ordered input streams, Algorithm 1 guarantees D-bounded eventual determinism when the input is within lateness bound D.*

Proof. Based on the questions in section 3.6, we need to ensure that (i) all relevant contexts for all D-bounded late arrivals are stored; (ii) after all late arrivals have arrived, the relevant contexts are forwarded in timestamp-order at least once to UDA$_R$ (which will run method `fire` for all relevant window instances); (iii) when forwarding relevant context, all tuples in it neither precede other UDA$_R$-maintained tuples nor contribute to any of the windows maintained by UDA$_R$ (once the timestamps of the tuples in the relevant context are changed). Algorithm 1 implies this is achieved since (i) all late arrivals are stored together with other tuples in the relevant context they contribute to (L1-5 and L7) as described in Lemma 4, (ii) tuples are removed from $B[t.k]$ only when no more late arrivals will be received for the relevant context they belong to (L14) and (iii) method `replay` is run at least once (after all possible late arrivals have been added to it in timestamp order) once its timestamps have been changed according to Lemma 8 (L14). □

3.7 Use Case and Evaluation

TinTiN is implemented in Apache Flink [30] and evaluated using a UDA based on a real-world validation application for Smart Meter (SM) readings in an Advanced Metering Infrastructure (AMI). 55 days of hourly data from 50.000 SMs are validated in the use case. We evaluate TinTiN's *output* (cf. section 3.4), *throughput*, *processing latency*, *logical latency*, *responsiveness* and *memory requirements.*

Use-case and experiment set-up

The SMs periodically send the cumulative energy consumption to the utility's central servers. The readings, used for billing (among other things), are validated by calculating SMs' hourly consumption (by taking the difference between two consecutive readings) and verifying that the latter is positive and bounded by the installed fuses. Invalid readings are marked and processed to identify patterns indicating hardware failure (when readings exceeding the bounds are followed by a negative one within 24 hours). The data validation application outputs alerts for matched patterns as well as excessive or negative consumption values. Hourly readings can reach the utility up to 40 days late. Hence, parameter D is set to 40 days. There are two Aggregates in the query, one with WS 2 hours for calculating the hourly consumption and one with WS 24 hours, the pattern's maximum length. The size of $\beta[]$ is therefore set to 26 hours (the sum of the window sizes, cf. subsection 3.6.5). Both aggregates have WA 1 hour. TinTiN and the UDA are evaluated for (sub)sets of increasing size of the 50K SMs (statistics are given in Table 3.2), and run on a virtual server with 4 dedicated 2.6 GHz cores and 16 GB RAM. Throughput and processing latency results are averaged over 10 runs.

Parameters for TinTiN and baselines for comparison

We evaluate TinTiN with the triggering conditions (TCs) from section 3.6:

TC-eager (TinTiN-TCE) reprocesses the relevant context for holes as soon as late data fills them. This approach minimizes logical latency but its output (cf. section 3.4) can contain multiple different and duplicate results before the final exact result is produced.

TC-lazy (TinTiN-TCL) reprocesses relevant context as soon as the next in-order reading (i.e. $t.ts \geq \tau$, the largest timestamp seen so far by TinTiN) arrives. This TC reduces the amount of different and duplicate results at the cost of a higher logical latency.

TinTiN and these TCs are compared against the following baselines:

SortedNoWait (SNW): an ideal baseline fed timestamp-sorted input and thus strictly deterministic. Note SNW cannot be used in practice (data is not sorted in the real-world application), it is included to characterize TinTiN's and other baselines' output in terms of different, duplicate, omitted and exact tuples, and logical latency.

UnsortedWait (UW): the baseline where the allowed delay of the input data is based on D (i.e., 40 days). The UDA processes data after storing it and waiting for D time units, incurring D time units logical latency penalty and large memory requirements but enforcing strict determinism. UW is essentially the option for system experts that are not stream processing experts to get accurate results.

UnsortedDiscard (UD): a baseline that discards all late arrivals, thus not validating all data and omitting final results when there are late arrivals. This can be the fastest but least accurate, hence not really usable in systems where accuracy and reliability are required.

Table 3.2: *Data statistics for the used datasets.*

Dataset size (keys)	10k	20k	30k	40k	50k
Number of tuples	11.4M	22.8M	34.2M	45.5M	56.9M
Number of late tuples	90.1k	179k	262k	350k	435k
Number of holes	406k	808k	1.22M	1.66M	2.03M
Number of relevant contexts with holes	899k	1.79M	2.66M	3.58M	4.41M

Evaluation of quality of output

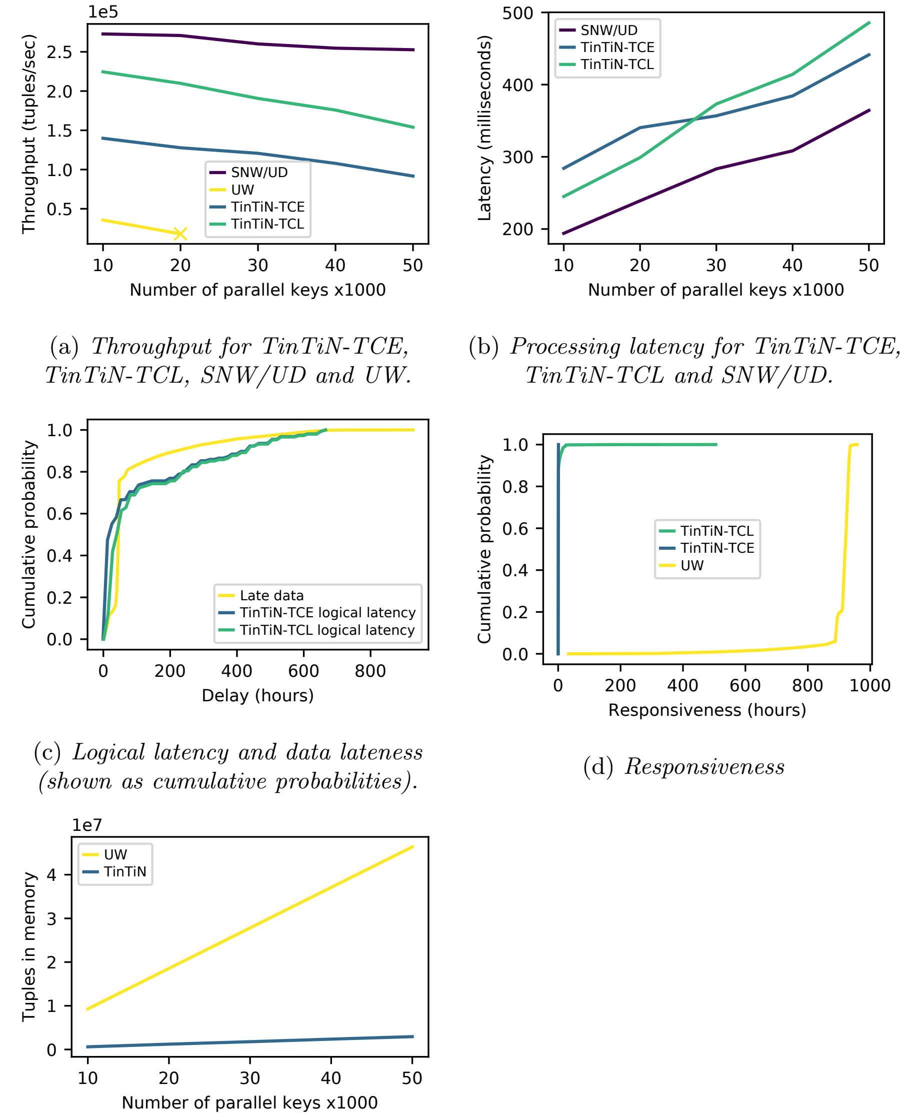
Table 3.3 compares the output of both TCs and UD. As expected, TinTiN does not omit any results; UD omits approximately 10 percent of the exact results. TinTiN-TCL also gives fewer duplicate results than TinTiN-TCE, since the latter prioritizes reprocessing as soon as possible. This causes a result to be produced multiple times if it is affected by more than one hole, which in turn can result in multiple duplicate results.

Table 3.3: *Output of TinTiN's TCs and UD compared with strictly deterministic output (such as from SNW or UW).*

Dataset	AlgID	Exact	Omitted	Different	Duplicate
10k	TinTiN-TCE	240	0	0	351
10k	TinTiN-TCL	240	0	0	0
10k	UD	213	27	0	0
20k	TinTiN-TCE	388	0	0	645
20k	TinTiN-TCL	388	0	0	5
20k	UD	350	38	0	0
30k	TinTiN-TCE	518	0	0	2261
30k	TinTiN-TCL	518	0	0	37
30k	UD	460	58	0	0
40k	TinTiN-TCE	729	0	0	2536
40k	TinTiN-TCL	729	0	0	37
40k	UD	659	70	0	0
50k	TinTiN-TCE	850	0	0	2780
50k	TinTiN-TCL	850	0	0	39
50k	UD	765	85	0	0

Processing throughput

Figure 3.5a shows the *processing throughput*, i.e. the number of processed tuples per second, for increasing number of parallel keys for TinTiN's TCs, SNW, UW and UD. Due to its processing overhead, TinTiN's throughput is lower than that of SNW's or UD's (notice though the latter baselines cannot be used in production). Nonetheless, it largely improves UW, which cannot sustain more than 20K keys (since it runs out of memory for larger number of keys). As expected, TinTiN-TCE's throughput is lower than TinTiN-TCL's since the former prioritizes low logical latency over the number of times data is potentially reprocessed.



(a) *Throughput for TinTiN-TCE, TinTiN-TCL, SNW/UD and UW.*

(b) *Processing latency for TinTiN-TCE, TinTiN-TCL and SNW/UD.*

(c) *Logical latency and data lateness (shown as cumulative probabilities).*

(d) *Responsiveness*

(e) *Memory footprint for TinTiN and UW measured in number of tuples.*

Figure 3.5: *Evaluation graphs*

Processing latency

Figure 3.5b shows the *processing latency*, i.e. the difference in wall clock time between the creation time of an output tuple and the ingestion time of the input tuple that triggers such output. TinTiN adds approximately 100 ms to the latency when compared with UD and SNW. Also for this metric, it nonetheless performs significantly better than UW (not plotted since it is orders of magnitude larger, 98 and 191 seconds for 10K and 20K keys, respectively).

As discussed in section 3.6, deploying a fresh UDA$_R$ before replaying a window is not a viable solution. In our experiments, the time taken to deploy a fresh UDA$_R$ instance is 1 order of magnitude larger than TinTiN's processing latency (between 3 and 4 seconds).

Logical latency

Figure 3.5c shows the logical latency (cf. section 3.4); it naturally depends on the input data's lateness, which is drawn on the plot for convenience. As shown, TinTiN-TCE's logical latency is some hours smaller than TinTiN-TCL's and, for both TCs, it is substantially better than UW's (40 days). Since SM late data often arrives in batches, the logical latency penalty for TinTiN-TCL is relatively small compared with the throughput gain over TinTiN-TCE.

Responsiveness

Figure 3.5d shows one of TinTiN's key strengths: its responsiveness compared to UW (i.e. the time between the arrival of a late tuple and the processing of the window this tuple belongs to, cf. section 3.4). Both TinTiN's TCs enable faster processing of late data. While TinTiN-TCE prioritizes swift reprocessing over performance, TinTiN-TCL offers a compromise between fast reprocessing of late data and performance. TinTiN reprocesses 90% of late data within 2 hours and 99.8% within 24 hours. For UW, 95% of the late data is processed more than 37 days after its arrival.

Memory

Figure 3.5e shows the extra memory (in number of tuples maintained temporarily in order to process all data) required by TinTiN and UW. SNW and UD are not shown since they do not temporarily maintain tuples. The amount of memory required by TinTiN is two orders of magnitude smaller than UW's. This is expected, since UW needs to keep 1920 tuples in memory for every key (24 hours·40 days·2 aggregates), while TinTiN keeps 26 tuples per key in addition to the tuples that belong to a relevant context.

Our results show that TinTiN processes on-time data without delays, providing timely results for late data, based on its triggering condition, thus minimizing utilities' response time for actions.

3.8 Related Work

One of the eight requirements for real time stream processing as defined in [1] is resiliency against missing and out-of-order data. We propose a way for resiliency against missing and out-of-order data, one of the key requirements for streaming processing [1]. Earlier methods to handle such stream imperfections are *slack* [22] and *punctuation* [17], both methods introduce waiting in order to deal with out-of-order data which we aim to minimize. Recent work [28] utilizes a Slack-ScaleGate data structure in order to process out-of-order input strictly deterministic as long as a logical latency constraint can be fulfilled, but without guarantees otherwise. Slack can be combined with speculative processing and buffering for event processing [13], but this method requires the event processor to be able to export its internal state in order to be consistent. Our approach does not require any changes to the application, that is wrapped in order to be able to process out-of-order events.

The term eventual determinism has earlier been used also in a different context, i.e. algorithms with a probabilistic and a deterministic mode, for problems where randomization is needed to break symmetries; processes eventually enter, and stay in, the deterministic mode [18]. Differently, here, the term is to characterize the output of processing whose input can be influenced by non-deterministic reorderings due to e.g. varying network delays.

An alternative approach to handle out-of-order data is to enhance the stateful operators in the streaming queries; [12] is early work in this direction which allows all stateful operators to store their state when data is late and to process late data with this stored state. Unlike ours, this method requires changes in the SPE or the original streaming application, and does not guarantee determinism.

The dataflow model [5], adopted by SPEs as Apache Flink [30] and Google Cloud Dataflow [32], allows for multiple evaluations of window instances, if the late data arrives no later than specified by an *allowed lateness* parameter. However the dataflow model cannot identify holes in the input stream and therefore cannot determine which window instances can receive late arrivals. For this reason all window instances need to be stored until the allowed lateness has expired, leading to excessive memory demands.

Orthogonal work, studying efficient merge-sorting of interleaving streams for strictly deterministic analysis, is presented in [4].

Data stream processing is a good match for smart grid challenges, as shown in [2, 3] where both applications disregard late data. Yet since occurrence of late data is common for smart energy meters, both are examples of applications that could leverage TinTiN.

3.9 Conclusion and Future Work

We introduce the concept of *D-bounded eventual determinism* to control streaming applications' trade-offs, in result correctness and quality versus timeliness, in CPS contexts where data fed to such applications comes out of timestamp order. We also present TinTiN, a middleware that enforces D-bounded eventual determinism, and evaluate it for a real-world Smart Grid use case. As shown, TinTiN induces minimal overhead in logical latency and enables processing of larger streams of data compared to other state-of-the-art methods. It enables out-of-order stream processing for 50K keys in parallel, where the strictly deterministic baseline is bound to 20K.

Future work includes the extension and refinement for different granularity of eventual determinism, including *weak* and *strong* variations (specifying whether multiple – possibly different – results can be produced for the same window of tuples) and variations of use-cases [6, 21, 22]. Since the processing order also impacts the cost of parallelization techniques for stream processing [27–29], it is also worth investigating (i) how TinTiN's semantics can be encapsulated in basic streaming operators, in order to leverage SPEs' distribution and parallelization techniques, and (ii) how TinTiN's methodology can benefit distribution and parallelization of queries, to provide guarantees about their degree of determinism.

Acknowledgment

Work partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, the collaboration framework of Göteborg Energi and Chalmers Energy Area of Advance project STAMINA, the Swedish Research Council proj. "HARE" grant nr. 2016-03800, the Swedish Foundation for Strategic Research proj. FiC, grant nr. GMT14-0032, the Chalmers Energy Area of Advance proj. INDEED, and the EU Horizon 2020 Framework Programme, grant nr. 773717.

Bibliography

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

[2] J. van Rooij, J. Swetzén, V. Gulisano, M. Almgren, and M. Papatriantafilou, “echidna: Continuous data validation in advanced metering infrastructures,” in *2018 IEEE International Energy Conference.* IEEE, 2018, pp. 1–6.

[3] J. van Rooij, V. Gulisano, and M. Papatriantafilou, “Locovolt: Distributed detection of broken meters in smart grids through stream processing,” in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, 2018, pp. 171–182.

[4] I. Walulya, D. Palyvos-Giannas, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas, “Viper: A module for communication-layer determinism and scaling in low-latency stream processing,” *Future Generation Computer Systems*, vol. 88, pp. 297 – 308, 2018.

[5] S. Costache, V. Gulisano, and M. Papatriantafilou, “Understanding the data-processing challenges in intelligent vehicular systems,” in *2016 IEEE Intelligent Vehicles Symposium (IV).* IEEE, 2016, pp. 611–618.

[6] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, A. C. Koppisetty, and M. Papatriantafilou, “Driven: a framework for efficient data retrieval and clustering in vehicular networks,” in *2019 IEEE 35th International Conference on Data Engineering (ICDE).* IEEE, 2019, pp. 1850–1861.

[7] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, “Fault-tolerance in the Borealis distributed stream processing system,” *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 1, 2008.

[8] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas, “Maximizing determinism in stream processing under latency constraints,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems.* ACM, 2017, pp. 112–123.

[9] “Apache Flink,” https://flink.apache.org/, 2014, last accessed: October 26, 2020.

[10] "Apache Beam," https://beam.apache.org/, 2016, last accessed: October 26, 2020.

[11] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," in *Proceedings of the VLDB Endowment*, 2015, vol. 8, no. 12, pp. 1792–1803.

[12] I. E. Kuralenok, N. Marshalkin, A. Trofimov, and B. Novikov, "An optimistic approach to handle out-of-order events within analytical stream processing," in *CEUR Workshop Proceedings*, vol. 2135. RWTH Aahen University, 2018, pp. 22–29.

[13] C. Mutschler and M. Philippsen, "Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares," in *DEBS 2013 - Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, 2013, pp. 147–158.

[14] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *Proceedings of the 17th SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2012, pp. 181–192.

[15] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[16] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[17] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008.

[18] J. R. Rao, "Eventual determinism: Using probabilistic means to achieve deterministic ends," *J. of Parallel, Emergent and Distributed Systems*, vol. 8, no. 1, pp. 3–19, 1996.

[19] "Google Cloud Dataflow," https://cloud.google.com/dataflow, 2015, last accessed: October 26, 2020.

[20] V. Botev, M. Almgren, V. Gulisano, O. Landsiedel, M. Papatriantafilou, and J. van Rooij, "Detecting non-technical energy losses through structural periodic patterns in ami data," in *Big Data (Big Data), 2016 IEEE International Conference on*. Washington DC, USA: IEEE, 2016, pp. 3121–3130.

[21] Z. Fu, M. Almgren, O. Landsiedel, and M. Papatriantafilou, "Online temporal-spatial analysis for detection of critical events in cyber-physical systems," in *Big Data (Big Data), 2014 IEEE International Conference on.* Washington DC, USA: IEEE, 2014, pp. 129–134.

[22] V. Gulisano, Y. Nikolakopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas, "Deterministic real-time analytics of geospatial data streams through scalegate objects," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, 2015, pp. 316–317.

[23] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer, "Quality-driven continuous query execution over out-of-order data streams," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* ACM, 2015, pp. 889–894.

Chapter 4

LoCoVolt: Distributed Detection of Broken Meters in Smart Grids through Stream Processing

J. van Rooij, V. Gulisano, M. Papatriantafilou

Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS), 2018.

Abstract

Smart Grids and Advanced Metering Infrastructures are rapidly replacing traditional energy grids. The cumulative computational power of their IT devices, which can be leveraged to continuously monitor the state of the grid, is nonetheless vastly underused.

This paper provides evidence of the potential of streaming analysis run at smart grid devices. We propose a structural component, which we name LoCoVolt (Local Comparison of Voltages), that is able to detect in a distributed fashion malfunctioning smart meters, which report erroneous information about the power quality. This is achieved by comparing the voltage readings of meters that, because of their proximity in the network, are expected to report readings following similar trends. Having this information can allow utilities to react promptly and thus increase timeliness, quality and safety of their services to society and, implicitly, their business value. As we show, based on our implementation on Apache Flink and the evaluation conducted with resource-constrained hardware (i.e., with capacity similar to that of hardware in smart grids) and data from a real-world network, the streaming paradigm can deliver efficient and effective monitoring tools and thus achieve the desired goals with almost no additional computational cost.

4.1 Introduction

Smart Grids, in which communication-enabled IT devices can share information with energy utilities, are replacing traditional energy grids. At the lower-voltage distribution tiers (i.e., at the level where energy is distributed to private customers and businesses), this transformation has been enabled by Advanced Metering Infrastructures (AMIs).

These consist of Smart Meters (SMs) and a communication infrastructure to communicate with the energy utilities' data center. A common communication infrastructure consists of Concentrator Units (CUs) that use either a wireless radio network or power line communication to communicate with the SMs [1].

In any large system continuous monitoring is needed to detect faulty components. This is especially important in an AMI because of safety-related, economic and administrative implications. To our advantage, AMIs make available the cumulative distributed computational power of their devices, which can be used to efficiently monitor the state of an AMI in a continuous and distributed fashion. In this context, data streaming fits well thanks to its inherently distributed, parallel and low-latency analysis properties.

In this paper, we show this for the detection of broken SMs that report incorrect voltage readings. Incorrect voltage readings can indicate that the SM is broken and will influence the amount of energy billed to the customers. Furthermore, early detection of broken SMs boosts safety, since undetected high voltages can cause damage to electric equipment or injury to persons. We present LoCoVolt (Local Comparison of Voltages). With LoCoVolt the differences of voltage readings between any pair of close-in-space SMs are continuously monitored by each CU. Based on the observed differences (instantaneous and average) and the correlations between voltage readings from each SM connected to the same CU, an SM is reported as broken when its number of suspicious readings exceeds a given threshold (as we explain in detail in § 4.4) . LoCoVolt can function in a manner that is agnostic of the precise topology of the underlying electricity network and this property makes its approach applicable in a variety of deployments.

We implemented LoCoVolt on top of Apache Flink [30] and tested it with real-world electricity network data from a deployed network. Our evaluation, conducted on resource-constrained hardware whose capacity resembles AMI hardware capacity, shows that the streaming paradigm delivers efficient and effective monitoring tools.

The rest of the paper is organized as follows. § 4.2 overviews preliminary concepts. § 4.3 describes the problem in more detail, explaining also the bigger challenges. LoCoVolt's technique to detect broken SMs is presented in § 4.4 and evaluated in § 4.5. § 4.6 discusses related work while § 4.7 concludes the paper.

4.2 Preliminaries

In this section we overview introductory and background information about data streaming applications, AMIs and voltage monitoring, as well as correlation measures between data streams.

4.2.1 Data streaming processing applications

Data streaming processing applications are designed as graphs composed by *streams* of data and *operators*.

Each *stream* carries tuples sharing the schema $\langle ts, A_1, \dots, A_n \rangle$, where ts is the tuple's creation timestamp and $A_1, \dots, A_n$ are application-related attributes. In a DAG, streams specify how tuples flow from the data sources through the operators and, eventually, to the data sinks (delivering results to analysts or other applications).

Operators are provided by the Stream Processing Engine (SPE) being used to run the application. Despite the fact that each SPE provides its own definition (and implementation) of basic streaming operators, a common subset of the operators provided by different SPEs includes *Aggregate*, *Join*, *Stateless* and *Merge* operators [3].

Aggregate operators apply aggregation functions over *sliding windows* of tuples. Windows are defined by their size, their advance and, optionally, by a group-by parameter referring to one or more of the input tuples' attributes when the aggregation function is applied independently to each group of tuples sharing such attributes. The Join operator matches tuples from two streams (keeping a sliding window for each stream) and forwards the pairs for which a given predicate holds. Stateless operators, as the name suggests, do not maintain a state evolving with the tuples being processed, and can produce zero, one or more output tuples for each input tuple, applying a user-defined function that specifies the input tuples' attributes to be copied to the output tuples and the functions to be applied to them. Finally, Merge operators allow to merge multiple streams into a single one. As we discuss in Section 4.4, these basic operators can be composed to implement LoCoVolt's analysis.

4.2.2 AMIs and voltage monitoring

Electricity Network: Private customers and businesses, along with their SMs, are connected to the grid via *transformers*. Each SM connects to exactly one transformer while a single transformer can host multiple SMs. Each SM is connected to a transformer by one, two or three lines as well as a neutral.
Data Network: At the same time, each *Smart Meter* SM is also connected to the utilities' servers, often via a *Concentrator Unit* (CU) that aggregates data from multiple SMs. Each SM connects to exactly one CU (but can change it over time) while a single CU is connected to multiple SMs. SMs that are physically close (e.g., deployed in the same building) have high chances of being connected to the same transformer and CU. In the case of wireless communication between SMs and CU there is nonetheless no guarantee about the overlap between the SMs and transformers topology and the SMs and CUs topology (e.g., two physically close SMs could for instance be connected to different transformers but the same CU). This is illustrated in the schematic overview in Figure 4.1.

The voltage measured by the SMs depends on the input voltage at the transformer, the length of the connecting cable connecting and local loads in the distribution network. SMs that are not broken and are connected to the same transformer are thus expected to display a high correlation between their voltages time series [4]. Figure 4.2 illustrates this by showing the voltages

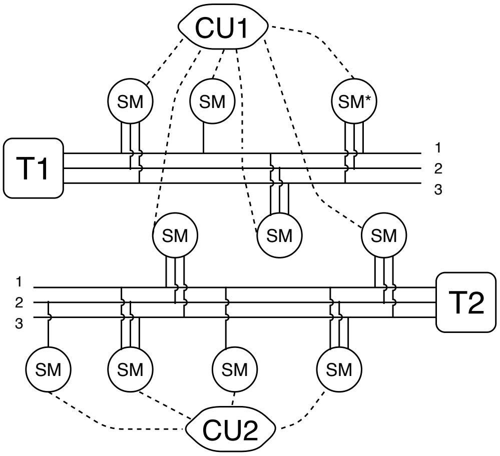


Figure 4.1: *A schematic overview of the SMs connected to two transformers and two CU. Note that the data network overlaps only partially with the electricity network. SMs are connected to the transformer with at least one and at most three lines. The line order at the SMs can differ, e.g. here the line order of SM* is reversed compared to the other SMs. Besides, the set of SMs connected to each CU may vary with time, e.g. depending on conditions that affect the signal strength.*

measured by two SMs that are physically close and connected to the same transformer.

Among other things, SMs report the voltage readings for each of their lines with tuples composed by attributes $\langle ts, CU, SM, L\rangle$, where ts denotes the timestamp for the reading, CU and SM indicate the corresponding device identities and L is a 3-entry vector, containing the root-mean-square voltage readings for the respective lines, i.e. the equivalent steady (DC) value which gives the same effect as the sinusoid signal [5].

4.2.3 Streams correlation

In LoCoVolt, we make use of correlation of time series. Given two time series a and b with n elements, the Pearson correlation coefficient, r, is suitable to be calculated incrementally [6] as:

$$r_{a,b} = \frac{P - \frac{AB}{n}}{\sqrt{A_2 - \frac{A^2}{n}}\sqrt{B_2 - \frac{B^2}{n}}},$$

where $A = \sum_i a_i$, $B = \sum_i b_i$, $A_2 = \sum_i a_i^2$, $B_2 = \sum_i b_i^2$ and $P = \sum_i a_i \cdot b_i$. For incremental calculation only the sums and the number of values need to be stored. The range of r is $[-1, 1]$, where -1 indicates maximum negative correlation and 1 maximum positive correlation. A correlation coefficient of 0 indicates that there is no correlation between the time series.

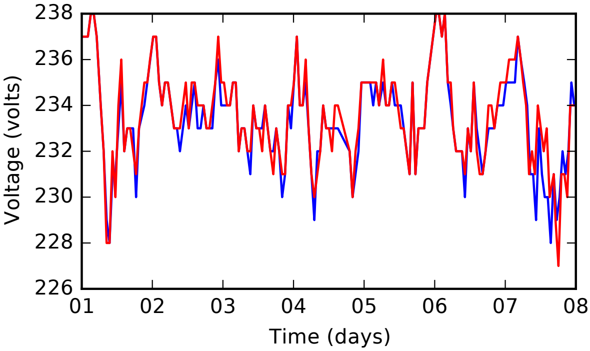


Figure 4.2: *The voltage for two SMs that are physically close and connected to the same transformer during one week. The correlation coefficient for the time series is 0.94.*

4.3 Problem description

In AMIs, SMs are expected to report consumption and quality measurements periodically. Over time, SMs can break and stop reporting or start reporting measurements that are not correct. An SM is defined to be broken when its measured voltage differs from the actual voltage. Detection of broken SMs is challenging since variations in consumption readings cannot be easily distinguished between customer-dependent variations and variations dependent on broken SMs. Because of the physical properties of electricity, a broken SM can be potentially identified by comparing the voltage readings of its *lines* with the ones of the lines of a working SM connected to the same transformer (as explained in Section 4.2 they are expected to have readings that follow the same temporal curves) [4].

LoCoVolt aims at detecting broken SMs based on this observation. It should be noticed that several practical aspects of AMI deployments can make this a challenging task:

[a] *Line ordering*: SMs' lines are not connected to transformers in a fixed order. That is, the same physical line connecting two SMs to the same transformer is not necessarily plugged as line 1 (or 2 or 3) in both SMs.

[b] *Asynchronous measurements*: The primary task of SMs is to measure the consumed energy. For this reason SMs don't necessarily take voltage readings in a synchronized fashion, voltage readings can even be measured by the CUs sequentially with a coarse-grained periodicity (e.g. 15 minutes, 1 hour). This implies that we might not have simultaneous measurements at all SMs and hence the differences might be inaccurate. Moreover, due to intermittent connectivity and noise, not all readings made in nearby time-intervals reach the CU in time (i.e. the continuous analysis must tolerate missing values).

[c] *Symmetric differences*: A suspicious difference between the voltage readings of the lines of two SMs indicates that one of the two is potentially

broken, but does not give any hint about which one of the two is the broken one.

[d] *Electricity network and communication network topology*: The topology of the electricity network (i.e. which SMs are on the same transformer) might not be known, due to distinctions between administrative domains of the utility or for security reasons. Moreover, wireless communication may imply that the set of SMs connected to the same CU is dynamic. (cf. Figure 4.1). I.e. we are looking for a method that is agnostic to the transformer - SM network, as mentioned in the introduction.

From an implementation perspective, an additional challenge is to employ methods that are intuitive to communicate with the utility system experts, as unnecessary complexity introduces risks of errors due to misunderstandings. For that reason, we propose to analyze the problem so as to use standard SPE operators, usable by engineers with common programming skills.

4.4 LoCoVolt

LoCoVolt addresses the challenges described in the previous section, through continuous monitoring of voltage readings of SMs at the CU level. Following LoCoVolt's rules, SMs can "accuse" each other if their voltage readings are suspiciously far. On a CU containing some broken SMs, this will result in many-to-few accusations (from the working SMs to the broken ones) and few-to-many accusations (from the broken SMs to the working ones). Accusations made by an SM M_i are weighed by the number of accusations it receives, introducing a reliability measure. The weighted accusations for M_i are then summed and compared with the number of SMs expected to accuse M_i is the latter is broken. In the following, we discuss LoCoVolt's semantics. We present the stream operators (Figure 4.3) that implement them and an example in the following subsections (subsections' names correspond to the boxes in the figure, the stream operators are also referenced in the text).

When do SMs accuse each other? The difference measured by two SMs fluctuates over time due to the reasons mentioned in Section 4.2. Hence, rather than having SMs accuse each other when their readings are arbitrarily distant, we base accusations on the difference between their instantaneous (ΔV) and average ($\overline{\Delta V}$) difference. Furthermore, we weigh such accusations by the correlation (c) observed between them, so that accusations from SMs observing similar differences over time count more than those from SMs observing fluctuating differences. Two SMs accuse each other when:

$$|\Delta V - \overline{\Delta V}| \cdot c_{a,b} > \theta$$

where (i) θ is a parameter related to the measurement resolution, the accuracy and local load changes occurring between the readings of a pair of SMs; and (ii) $c_{a,b} = (r_{a,b} + 1)/2$, i.e. a mapping of the correlation of formula 4.2.3. This mapping of the correlation preserves the monotonic relation between the correlation and $\overline{\Delta V}$. The mapping also allows uncorrelated and inversely correlated SMs to accuse each other if $|\Delta V - \overline{\Delta V}|$ grows large enough. Systems experts maintaining the smart grid used in our evaluation identified a suitable value for θ from a set of working SMs. Since SMs can have up to three lines, $\Delta V, \overline{\Delta V}$ as well as c are 3x3 matrices. Therefore, the number of accusations between two SMs, M_i and M_j, is an integer in the range [0,9]. Notice that accusations are symmetric, i.e. if M_i accuses M_j, then M_j accuses M_i too.

How to prevent accusations by SMs from different transformers from affecting broken SMs detection? When M_i and M_j are not connected to the same transformer, their voltage reading difference can fluctuate more than if they share the same transformer (independently of whether any of the two is broken). When two big-enough groups of SMs are connected to the same CU but different transformers, this can result in many-to-many accusations (because of the legit different voltage readings across groups). To mitigate this, we normalize accusations weighting them by:

$$w_i = \frac{1}{\sum_{S_j} acc(j,i)}$$

where w_i is the weight for M_i, S_j is the set of SMs accusing M_i, and $acc(j,i)$ is the number of accusations between M_j and M_i. The value of w_i is small for SMs that receive a large number of accusations, indicating that it is less trustworthy. The weighted accusation received by M_i from M_j is then defined as:

$$wacc(i,j) = w_j \cdot acc(i,j).$$

Each $wacc()$ is a real number in the range [0,1]. M_i weighted accusation from M_j is 1 when M_i is the only SM accusing M_j. Note that while the accusations are symmetric between M_i and M_j, this does not necessarily hold for the weighted accusations. The total amount of weighted accusations can now be calculated for all SMs by:

$$J_i = \sum_S wacc(i,j)$$

where S is the set of SMs whose readings are compared with M_i.

If an SM breaks, how many other SMs at its CU will accuse it? Value J_i depends not only on how much M_i's readings deviate from the other SMs, but also on the number of SMs that the reading is compared with, as well as how correlated these SMs are with M_i (i.e., how likely they will trigger an accusation). When M_i breaks and starts reporting inaccurate values, the probability of receiving an accusation from M_j will be related to the correlation between M_i and M_j, with an increasing probability for increasing correlations.

We estimate E_i, the expected number of SMs accusing M_i, by the likelihood of receiving at least one accusation from another SM, which we approximate through the correlation matrix entries $c_{i,j}$. More concretely, E_i is defined as:

$$E_i = \sum_S max(c_{i,j})$$

Notice that each SMs in S contributes to the sum with value 1 if its correlation with M_i is 1. A weight θ' is introduced to specify which portion of E_i is

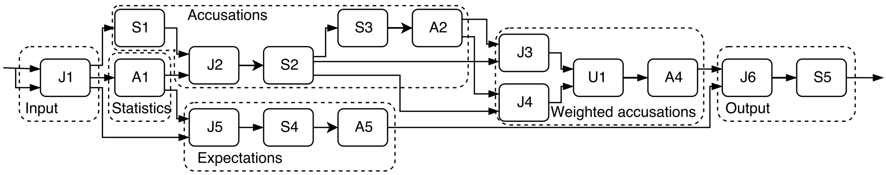


Figure 4.3: *The graph for LoCoVolt. The letter in the operator name refers to the type of operator: J for Join, A for Aggregate, S for Stateless and U for Union.* The operators are arranged in dashed logical blocks. The function and details of the logical blocks are described under the equally named headings in Section 4.4. Details for the individual operators can be found in Table 4.1.

sufficient for an accused SM to be reported as broken. The value of the weight influences the number of simultaneous broken SMs that can be detected since a larger number of broken SMs will cause $wacc$ to decrease. The weight is set by a system expert to match the expected maximum number of simultaneous broken SMs. An alert is eventually triggered for M_i if $J_i > \theta' E_i$.

The resulting graph of streams and operators implementing the method is presented in detail in Figure 4.3 as well as in the following sections. The names of the sections correspond to the boxes in the figure and the stream operators in the figure are referenced in the text with (OperatorID). A running example in the text is used to illustrate the query. The example consists of three SMs, A, B and C on CU X, where C is accused by A and B. For simplicity, all SMs have a single line.

4.4.1 Input

As described in Section 4.2.2, the highest correlation between voltage time series will be between SMs that are close in the distribution network. Therefore LoCoVolt compares SM readings that are within a certain time-window and connect to the same CU. This can be implemented by joining the stream of measurements onto itself in a window. The size of the window (w_s), and advance (w_a), is a trade off between the number of pairs that can be compared and the correlation between the readings. Here it is set to 10 minutes with the help of a system expert and the window is evaluated for every incoming tuple.

Since an SM can report values for 1, 2 or 3 lines, the readings are stored in an array of size 3. If there are fewer than three values the corresponding place in the array will be set to a predefined null value.

The example starts with the following three readings:

$$< ts_A, X, A, 230 >, < ts_B, X, B, 231 >, < ts_C, X, C, 219 >,$$

where $ts_A < ts_B < ts_C$. When all timestamps are within the window for the Join operator, the following tuples will be produced:

$$< ts_B, X, A, B, 230, 231 >, < ts_C, X, A, C, 230, 219 >,$$
$$< ts_C, X, B, C, 231, 219 > .$$

ID	Description	Tuple schema
J_1	Match any pair of tuples from two different SMs that share the same CU over a sliding window of 10 minutes, outputting the SMs' IDs and their voltage readings per line (3X1 matrices L_1 and L_2).	$\langle ts, CU, m_1, m_2, L_1[3], L_2[3]\rangle$
S_1	Produce a 3X3 matrix (ΔV) with the difference between each pair of lines between two SMs joined by J_1.	$\langle ts, CU, M_1, M_2, \Delta V[3X3]\rangle$
A_1	Produce a 3X3 matrix (c) with the scaled correlation (Pearson coefficient) as well as a 3X3 matrix ($\overline{\Delta V}$) with the average difference observed for each pair of lines between two SMs joined by J_1 over a sliding window of 28 days with an advance of 14 days.	$\langle ts, CU, M_1, M_2, c[3X3], ...$ $...\overline{\Delta V}[3X3]\rangle$
J_2	Match tuples referring to the same pair of SMs and produce a 3X3 matrix (D) carrying the value $c \circ \lvert\Delta V - \overline{\Delta V}\rvert$, where $\circ$ denotes the entrywise product.	$\langle ts, CU, M_1, M_2, D[3X3]\rangle$
S_2	Applies a threshold to the elements in D, setting the element to 1 if the value exceeds the threshold and 0 otherwise.	$\langle ts, CU, m_1, m_2, D[3X3]\rangle$
S_3	Produce the sum of the elements in D that equal 1 (accusations, *acc*) and output this in two tuples, one for each SM in the input tuple.	$\langle ts, CU, M, acc\rangle$
A_2	Produce the sum of *acc* for all SMs on all CUs in a 1 hour tumbling window.	$\langle ts, CU, M, \sum acc\rangle$
J_3	Match tuples where M from A_2 equals M_1 from S_2. Produces a tuple for M_2 containing weighted accusations *wacc*.	$\langle ts, CU, M_2, wacc\rangle$
J_4	Similar to J_3 but matches tuples where M equals M_2. Produces a tuple for M_1.	$< ts, CU, M_1, wacc >$
U_1	Union of the outputs from J_3 and J_4.	$\langle ts, CU, M, wacc\rangle$
A_4	Produce the sum of *wacc*, (J) for all SMs in a 1 hour tumbling window.	$\langle ts, CU, M, J\rangle$
J_5	Matches tuples referring to the same pair of SMs in a sliding window of 14 days, removes $\overline{\Delta V}$ from the tuple..	$\langle ts, CU, M_1, M_2, c[3X3]\rangle$
S_4	Splits the input tuple into two, one for each SM and keeps containing the maximum element in c.	$\langle ts, CU, M, max(c)\rangle$
A_5	Produces the expected number of accusations (E), in a 1 hour tumbling window.	$\langle ts, CU, M, E\rangle$
J_6	Join A_4 and A_5 for each SM.	$\langle ts, CU, M, J, E\rangle$
S_5	Report suspicious SMs for which the number of J exceeds $\theta' E$.	$\langle ts, CU, M\rangle$

Table 4.1: *Detailed description of the Join, Aggregate, Stateless and Union operators used in LoCoVolt's query.*

4.4.2 Statistics

The statistics required for LoCoVolt are the correlation (c) and the average difference ($\overline{\Delta V}$) between all line pairs of all SMs. This can be implemented with an aggregation operator that incrementally calculates the average for the difference as well as the correlation in a window. The window size should be large so that $\overline{\Delta V}$ and c are stable values. The window size for LoCoVolt is set to 28 days with a window advance of 14 days.

Since the readings are arrays with length three, the average difference, as well as the correlation between all possible line pairs are stored in two 3x3 matrices. (A1)

For the example we assume that the correlation between SMs A and B is 0.91 while the average difference is 1.2. Values for the other pairs can be found below.

$$< ts, X, A, B, 0.91, 1.2 >, < ts, X, A, C, 0.87, 2.3 >,$$
$$< ts, X, B, C, 0.89, 2.8 > .$$

4.4.3 Accusations

The calculation of the accusations is accomplished by an operator that calculates ΔV for all line pairs (S1). This stream is then joined with the statistics stream with a window size and advance that equals the window advance for the statistics operator. The join function can now calculate the value of $|\Delta V - \overline{\Delta V}| \cdot c$. (J2)

$$< ts_B, X, A, B, 0.182 >, < ts_C, X, A, C, 7.569 >,$$
$$< ts_C, X, B, C, 8.188 > .$$

The threshold θ can now be applied by a stateless operator. (S2) θ is set to 3, which for the example results in the following tuples:

$$< ts_B, X, A, B, 0 >, < ts_C, X, A, C, 1 >, < ts_C, X, B, C, 1 > .$$

The number of accusations between any pair of SMs is symmetric and can be processed in a single tuple containing the IDs for both SMs as well as the number of accusations. This is a number between 0 and 9, depending on the number of lines for the SMs and the result from the previous equation. In order to get the sum of the accusations per SM, two tuples with the number of accusations are created, one for each SM (S3).

$$< ts_B, X, A, 0 >, < ts_B, X, B, 0 >,$$
$$< ts_C, X, A, 1 >, < ts_C, X, C, 1 >,$$
$$< ts_C, X, B, 1 >, < ts_C, X, C, 1 > .$$

These tuples are aggregated in a tumbling window and counted (A2). The window size is chosen to match the frequency of the readings which in our case are hourly.

$$< ts_C, X, A, 1 >, < ts_C, X, B, 1 >, < ts_C, X, C, 2 > .$$

4.4.4 Weighted accusations

The weighted accusations are obtained by combining the accusations between the SMs with the sum of the accusations per SM. Two join operators are required to accomplish this since the accusations between the SMs is stored in a single tuple for every pair of SMs, while the total number of accusations received has one tuple per SM. (J3,J4) The window size and advance of the join operators match the frequency of the readings. The union of these streams results in a single stream with all *wacc* values. (U1)

The tuples at this point in the query contain *wacc* as shown in the example:

$$< ts_B, X, A, 0/1 >, < ts_B, X, B, 0/1 >,$$
$$< ts_C, X, A, 1/2 >, < ts_C, X, C, 1/1 >,$$
$$< ts_C, X, B, 1/2 >, < ts_C, X, C, 1/1 > .$$

The tuples now need to be aggregated in a window and counted in order to obtain J (A4). The window size and advance match the values chosen for the join operators in this block.

$$< ts_C, X, A, 1/2 >, < ts_C, X, B, 1/2 >, < ts_C, X, C, 2 > .$$

4.4.5 Expectations

The expected amount of accusations (E) for a broken SM is estimated in the query by joining the tuples containing the readings that are being compared with the statistics. (J5). The window size and advance equal the window advance for the statistics aggregator. A stateless operator selects the maximum correlation value for every pair of SMs and outputs a tuple containing this value for each SM in the pair (S4). These tuples can then be aggregated in order to obtain the sum per SM, ie. the value of E (A5).

The expected number of accusations for the SMs in our example is given by:

$$< ts_C, X, A, 1.78 >, < ts_C, X, B, 1.80 >, < ts_C, X, C, 1.76 > .$$

4.4.6 Output

The final step in the query is then to combine the expected number of accusations with the total number of weighted accusations (J6) and output an alert if the number of weighted accusations exceeds the expected number weighted by θ'. (S5) The window size and advance for the join operator match the frequency of the readings.

θ' is set to 0.5 (majority voting) which renders the following final output in the example:

$$< ts_C, X, C >$$

4.5 Evaluation

In this section, we present LoCoVolt's evaluation. We first introduce the evaluation criteria followed by the evaluation setup. Subsequently, we discuss how we simulated broken SMs based on real cases observed at the energy company. Finally, we evaluate LoCoVolt effectiveness in detecting such broken SMs and LoCoVolt's performance.

Evaluation criteria

In order to evaluate LoCoVolt's detection capabilities, the metrics we take into account are (1) the *detected percentage of manipulated readings*; (2) the *number of true positive (TP) and false positive (FP) alarms*; (3) the *detection time* for the different voltage manipulation rates. TP alarms are alarms that are generated for the manipulated SM after the manipulation has started, while FP alarms are alarms for SMs that have not been manipulated. We also evaluate(4) the *precision*, *recall* and *accuracy* which are common measures in classification problems [7]. These metrics are defined with TP and FP as well as true negative (TN) and false negative (FN) alarms. FN is the number of manipulated readings that did not trigger an alarm, while TN are the readings that neither were manipulated nor triggered an alarm. Precision is now defined as the quotient of TP by TP+FP, recall as the quotient of TP by TP+FN and accuracy is defined as the quotient of TP+TN by TP+TN+FP+FN.

In the case where there are multiple broken SMs simultaneously, we investigate (5) the percentage of detected broken SMs.

Finally (6) the performance in terms of *processing throughput* and *latency* is evaluated, in order to assess the possibility to run LoCoVolt on the hardware available in the AMI.

The detection time is especially important in order to minimize the duration and impact of the problems described in Section 4.3. The number of FP alarms should be as low as possible to minimize the manpower needed to investigate the alarms, while a high number of TP alarms generated by a large percentage detected readings helps to ascertain that a TP alarm truly is TP.

The results are compared with a baseline consisting of the current situation at the utility where SMs are investigated manually by system experts when the SM readings are outside of a predefined interval.

Evaluation setup

We evaluated LoCoVolt with data collected from 939 SMs during a period of 9 months. Each SM reports the voltage observed for each of its lines every hour, for a total of 4 million readings (about 1.5 million readings are missing). SMs connect to 26 Concentrator Units (CUs). The average number of SMs per CU is 36 while the CU with the largest set of SMs connects 152 and the CU with the smallest set connects 7 SMs. The SMs are connected to the grid by 30 different transformers. 26 CUs contain SMs that are connected to a single transformer. Two CUs contain SMs connected to two transformers while a single CU has SMs that are connected to three different transformers. Three kinds of SMs exist in the dataset, types 1 and 2 are three line SMs while type 3 is a single line SM. 366 SMs are of type 1 with a voltage resolution of 1 volt, 361 of type 2 with a resolution of 0.1 volts and finally 212 SMs of type 3 with a 0.1 volts resolution. The data is sanitized by removing values that would have been ignored based on existing validation rules.

We implemented LoCoVolt on top of Apache Flink [30] version 1.4.0. In order to test LoCoVolt's performance when potentially deployed at CUs, we run the performance evaluation experiments using a single-board device called Odroid-XU4 [36] (or simply Odroid in the remainder), equipped with a Samsung Exynos5422 Cortex-A15 2Ghz and Cortex-A7 Octa core CPUs and with 2 GB of memory. All other experiments were run on a standard off-the-shelf laptop computer.

Simulation of a broken smart meter

To the best of our knowledge, the SMs selected to conduct the evaluation are not broken during the period of time covered by the data. In order to check LoCoVolt's effectiveness in detecting broken SMs, we simulate the latter by manipulating their data, in ways that comply with the experts' description of experienced failures that are studied in a *post-mortem* fashion, i.e. after an SM is known to be broken through e.g. the damage caused by it. More concretely, we simulate two ways for which broken SMs have been observed in the AMI. In case I (*all-line*), we pick a random SM and, starting from a certain reading, we decrease its reported voltage every day with a constant rate. The rates we simulate are 0.005, 0.01, 0.015, 0.02 and 0.025. In case II (*single-line*), once a random SM is picked, we only alter one of its lines decreasing it every day with a constant rate. Also in this case, we simulate different decrease rates: 0.005, 0.01, 0.015, 0.02.

For both cases, we run 1000 experiments for each voltage manipulation rate, picking a random SM and a random starting date each time. The experiments start 28 days before the starting date of the manipulation, to ensure that the window for the statistics operator is filled, and ends after 14 days of manipulation.

Simulation of multiple bad SMs per CU

We also evaluate LoCoVolt when more than one SMs break simultaneously for two different cases. In case III (*multi-SM*) we simulate multiple SMs breaking independently from each other by selecting the correct number of SMs randomly. In case IV (*multi-correlated-SM*) we instead simulate the case where multiple SMs connected to the same transformer break, for instance due to lightning. For this case one SM is selected randomly while the other broken SMs are selected randomly from the set of SMs connected to the same transformer as - and physically close to - the first SM. We try different numbers of broken SMs: 2, 4, 6, 8 and 10 for both case III and IV.. 100 experiments are run for every number of broken SMs on a subset of the data described earlier. For these experiments we use the data from a single CU with 64 SMs that connect to two transformers (45 SMs to first transformer and 19 to the second one). For every experiment we decrease the readings for all lines on all selected SMs with a rate of 0.001 for every following day.

LoCoVolt detection capabilities

The detection capabilities of LoCoVolt are evaluated with the criteria described earlier in this section.

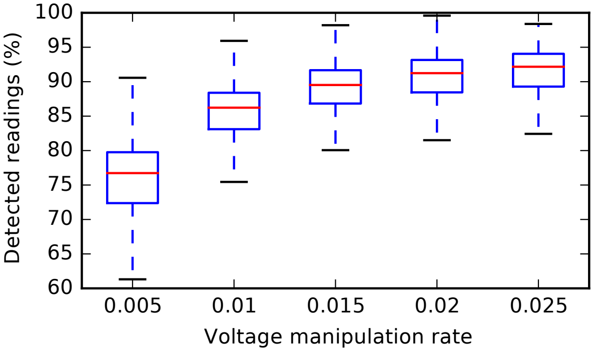


Figure 4.4: *The detected percentage of manipulated readings for different voltage manipulation rates when all lines are manipulated. The largest outliers are typically between 20 and 30%.*

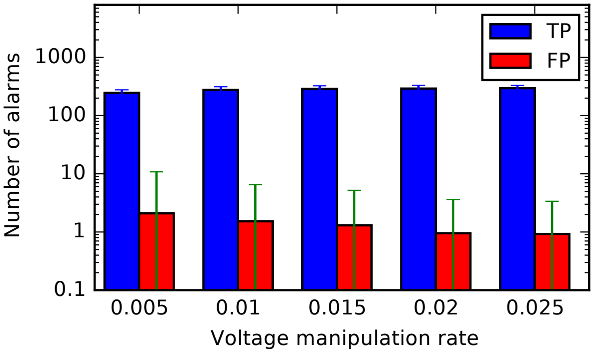


Figure 4.5: *The mean and standard deviation of the number of TP and FP alarms respectively for different voltage manipulation rates when all lines are manipulated. The number of FP alarms is 2 orders of magnitude smaller than the number of TP alarms.*

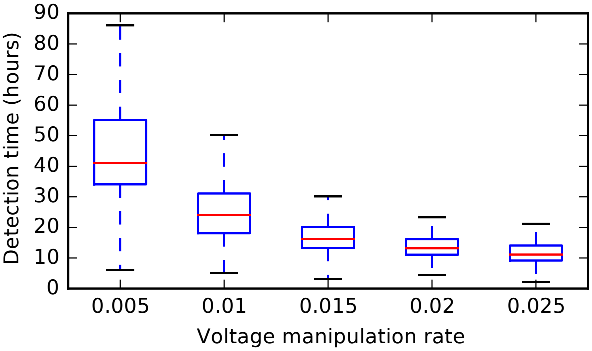


Figure 4.6: *The detection time for different voltage manipulation rates when all lines are manipulated. The largest outliers can be found between 250 and 320 hours for all manipulation rates.*

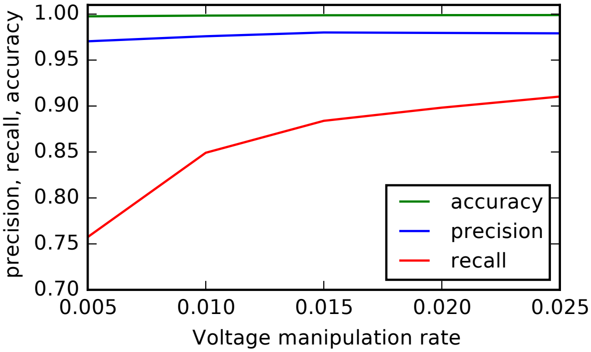


Figure 4.7: *The precision, recall and average for different voltage manipulation rates when all lines are manipulated.*

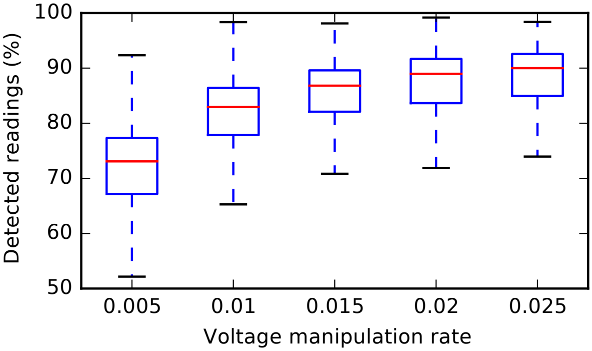


Figure 4.8: *The detected percentage of manipulated readings for different voltage manipulation rates when a single line is manipulated. The largest outliers are typically between 20 and 30%.*

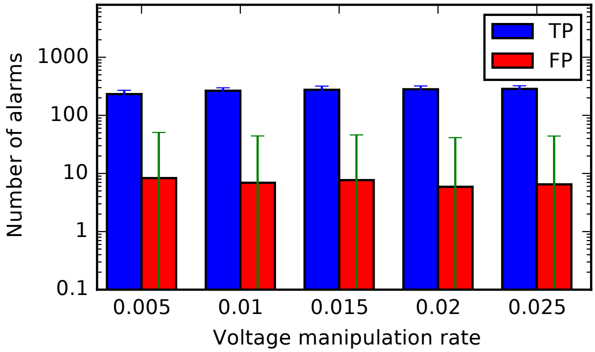


Figure 4.9: *The mean and standard deviation for the number of TP and FP alarms respectively for different voltage manipulation rates when a single line is manipulated.*

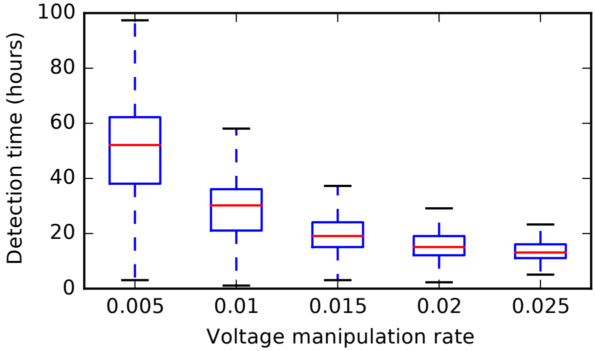


Figure 4.10: *The detection time for different voltage manipulation rates when a single line is manipulated. The largest outliers can be found between 250 and 320 hours for all manipulation rates.*

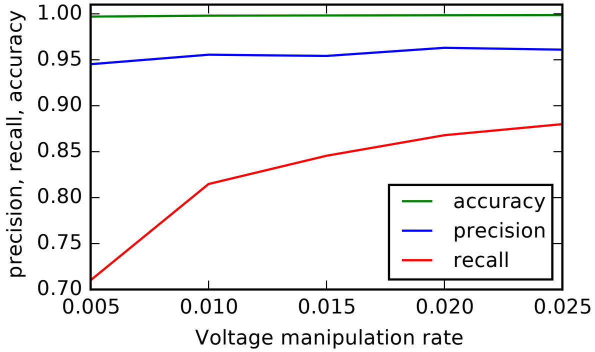


Figure 4.11: *The precision, recall and average for different voltage manipulation rates when a single line is manipulated.*

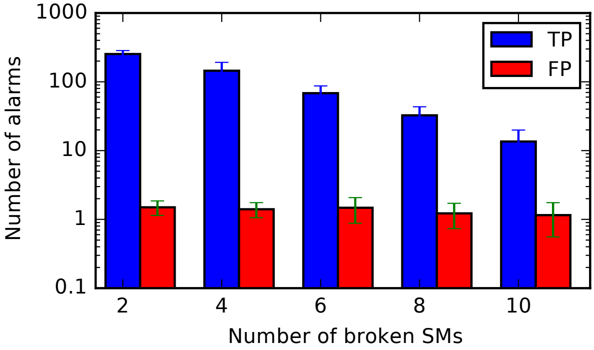


Figure 4.12: *The mean and standard deviation for the number of TP alarms per manipulated SM and FP alarms per SM reporting such alarms respectively for different numbers of manipulated SMs.*

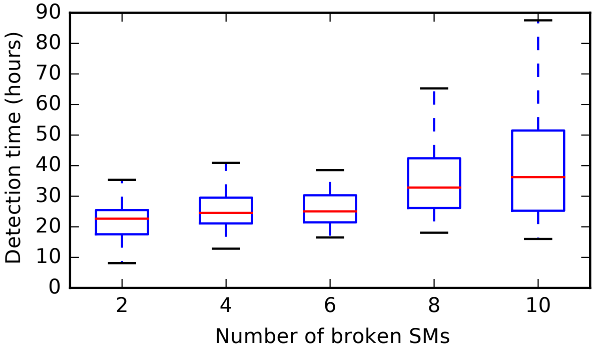


Figure 4.13: *The detection time for the detected manipulated SMs for different numbers of manipulated SMs.*

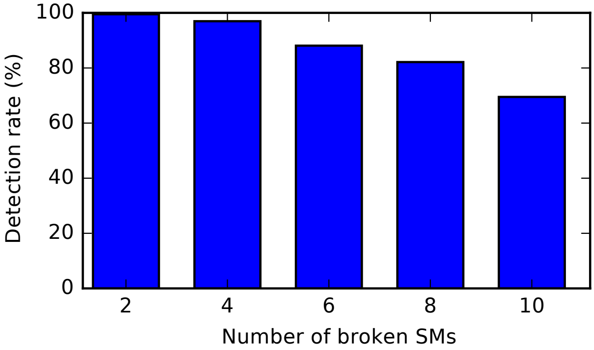


Figure 4.14: *LoCoVolt's detection rate for different numbers of simultaneous manipulated SMs.*

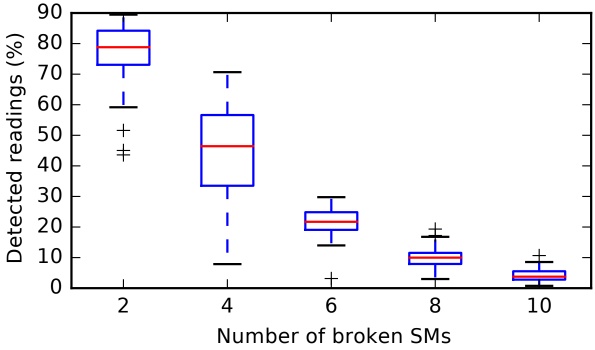


Figure 4.15: *The detected percentage of manipulated readings for different numbers of simultaneous manipulated SMs.*

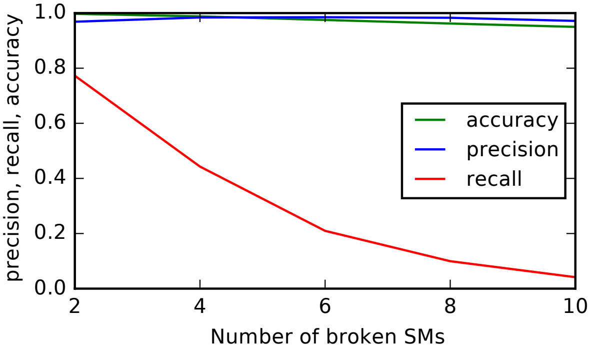


Figure 4.16: *The precision, recall and average for different numbers of simultaneous manipulated SMs.*

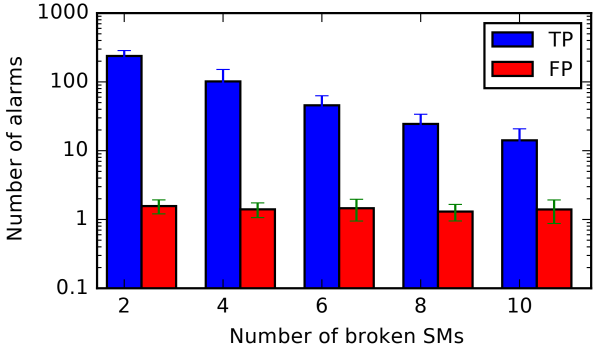


Figure 4.17: *The mean and standard deviation for the number of TP alarms per manipulated SM and FP alarms per SM reporting such alarms respectively for different numbers of manipulated correlated SMs.*

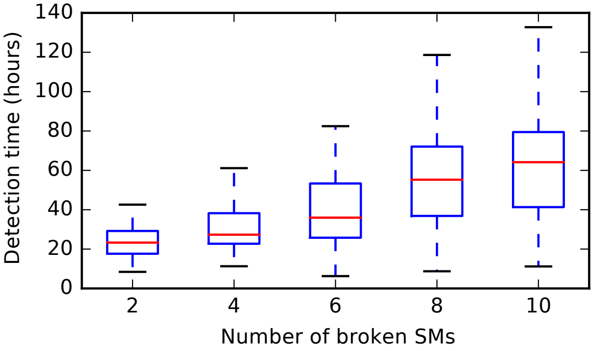


Figure 4.18: *The detection time for the detected manipulated SMs for different numbers of manipulated correlated SMs.*

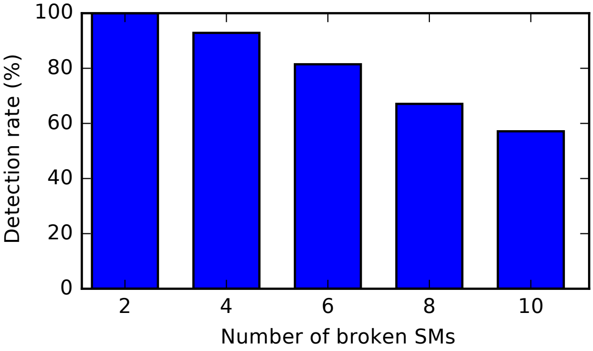


Figure 4.19: *LoCoVolt's detection rate for different numbers of simultaneous manipulated correlated SMs.*

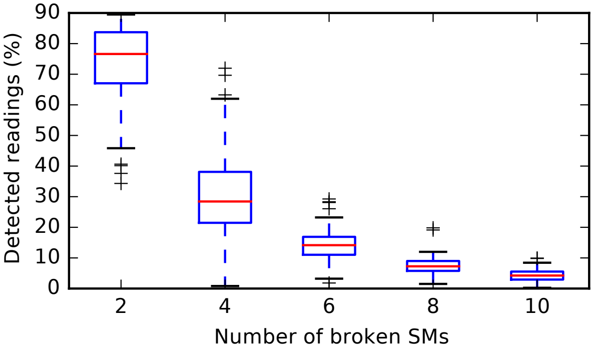


Figure 4.20: *The detected percentage of manipulated readings for different numbers of simultaneous manipulated correlated SMs.*

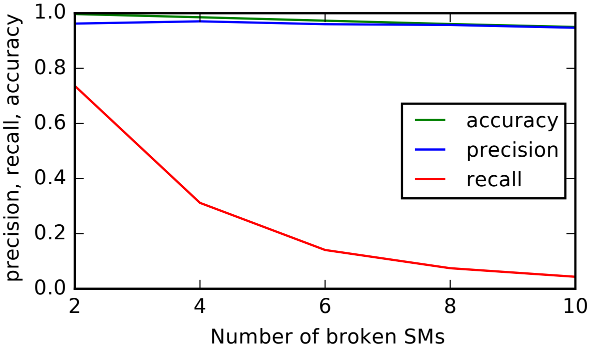


Figure 4.21: *The precision, recall and average for different numbers of simultaneous manipulated correlated SMs.*

Figures 4.4, 4.5 and 4.6 present the results for the broken SMs in the all-line case. As presented in figure 4.4, the median percentage of detected manipulated readings grows from approximately 75% (voltage manipulation rate 0.005) to 90% (manipulation rate 0.025). This trend is reflected in Figure 4.5 showing the mean total number of TP and FP alarms that are detected as well as the standard deviation. This can also be observed in Figure 4.7, which shows that the accuracy in this case is very close to one independent of the manipulation rate. Both the recall and the precision increase for larger manipulation rates, consistent with the larger percentage of detected manipulated readings for larger voltage manipulation rates. The mean number of SMs reporting FP alarms is approximately 2.4 with a standard deviation of 5. The average number of FP alarms per SM reporting such alarms is around 3, but the actual number has quite some variation with a standard deviation of 15. The number of TP alarms is about 250 (voltage manipulation rate 0.005) and increases to 290 (manipulation rate 0.025). This shows that the number of TP alarms detected greatly outnumber the number of FP alarms for every manipulation rate. The detection time of the manipulation depends on the time it takes for the manipulation to become greater than the threshold θ. For a voltage manipulation rate of 0.005 and a normal voltage of 230, this will take approximately 62 hours. Figure 4.6 shows that the mean detection time at this manipulation rate is some hours faster, and similarly for the larger rates.

Figures 4.8, 4.9, 4.10 and 4.11 present the result for the single-line case. The figures show similar trends with the percentage of detected manipulated readings, as well as the number of alarms increasing with the voltage manipulation rate as can be seen in Figures 4.8 and 4.9. The absolute amount of alarms is however about 7% lower, with matching results for the percentage of detected readings. This is reflected by the detection time in Figure 4.10 which is increased by a similar amount. The amount of SMs reporting FP alarms is similar as in the all-line case, but a difference is that the amount of alarms per SM is increased to 7 on average and the standard deviation is increased to 40 as can be seen in Figure 4.9. The number of TP alarms still greatly outnumber the FP alarms. The recall and precision values follow this trend as can be observed in Figure 4.11. The accuracy however remains very close to one.

Alarms are triggered both by manipulated SMs as well as SMs that have not been manipulated. However the number of alarms triggered by manipulated SMs is at least one order of magnitude larger than the number triggered by non manipulated SMs, as shown in Figures 4.5 and 4.9. This is also the case when multiple SMs are manipulated simultaneously, as shown in Figure 4.12. As currently done by system experts, all reported SMs need to be inspected. System experts usually start inspecting the SMs with the higher amount of alarms and stops when a false positive is encountered. If we define the detection rate as the percentage of broken meters that are indeed inspected, we can see in Figure 4.14 that the range of broken meters that are correctly detected goes from 100% (for 2 simultaneously broken meters) to approximately 65% for 10 simultaneously broken meters.. LoCoVolt's detection capability declines with increasing numbers of broken SMs with a rate for two bad SMs above 99% but then starts to decrease to reach 70% for ten bad SMs. Figure 4.12 shows that the number of TP alarms per SM decreases rapidly from a mean of 250 for two broken SMs to just 13 when the number of broken SMs is increased to 10. The average number of FP alarms however is independent of the number of bad SMs and smaller than two in all cases. The number of detected manipulated readings shows a similar trend as observed in Figure 4.15. The mean detection times for the broken SMs increases from 20 to 40 hours as can be seen in Figure 4.13. These trends are reflected in Figure 4.16 showing the precision, recall and accuracy. Both the precision and accuracy are close to one independent of the number of manipulated SMs, while the recall drops from 0.75 for two manipulated SMs to 0.05 when the number of manipulated SMs is increased 10. Even though the recall is only 0.05, the average number of alarms for a manipulated SM is still an order of magnitude larger than for a non-manipulated SM.

The results for the multi-correlated-SMs show similar trends. Figure 4.17 shows that the number alarms reported per TP or FP SM hardly changes, however the number of manipulated SMs that is detected declines faster as seen in Figure 4.19. This is also reflected in the number of manipulated readings that are identified which can be seen in Figure 4.20 as well as in Figure 4.21 which shows that the recall decreases faster in this case. The detection time increases faster when the broken SMs are highly correlated, as shown in Figure 4.18, with the mean detection time when for 10 broken SMs growing slightly over 60 hours.

LoCoVolt performance

In order to study LoCoVolt performance, we also evaluate its throughput and latency when running on the Odroid. The throughput, measured in tuples/second, represents the rate with which an CU can process input tuples. The latency, measured in ms, represents the average time elapsed between the production of an output tuple and the receiving of the latest input tuple contributing to it. Note however that since LoCoVolt produces a reduced number of output tuples (only for alerts about broken SMs), the latency is measured at the input of operator S_5 in figure 4.3.

Figure 4.22 and 4.23 show the mean and standard deviation of the measured

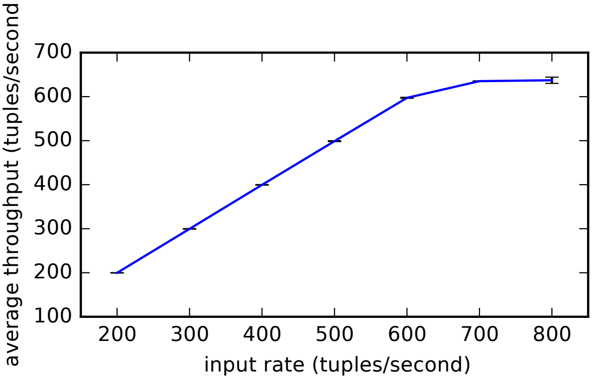


Figure 4.22: *The mean and standard deviation of the* throughput on the Odroid for different input rates.

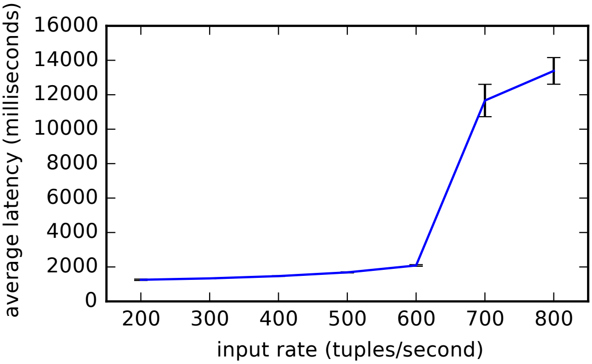


Figure 4.23: *The mean and standard deviation of the* latency on the Odroid for different input rates.

throughput and latency over three runs for every tested input rate. As shown, the Odroid can maintain a stable input rate of 600 tuples per second which is enough to sustain a reading interval of 1 second with a big margin. This throughput is more than sufficient for future applications, considering that the current reading interval is 1 hour. The latency is approximately 2 seconds at an input rate of 600 tuples/second, which is very small compared to LoCoVolt's detection time which is in the range of hours.

Evaluation summary

Broken SMs are currently usually detected when their readings exceed the allowed voltage range which in the case of this data set is $230V \pm 10\%$. A broken SM with a voltage manipulation rate of 0.005 would be detected after 20 days at the earliest. We show that the detection time using LoCoVolt is significantly smaller with a mean of 48 hours. The number of FP alarms generated by LoCoVolt is very small and greatly outnumbered by the number of TP alarms, enabling utilities to act on alarms swiftly since the possibility that the work required for an investigation is wasted due to a false alarm is very small. LoCoVolt can also detect situations in which there multiple broken SMs simultaneously, regardless of the correlation between these SMs. The detection rate decreases with increasing numbers of broken SMs, which is expected since a larger number of broken SMs will decrease the value of the weighted accusations. The detection rate is largest and the detection time smallest for the case where all broken SMs are picked randomly, but LoCoVolt outperforms the current detection method with a large margin even when the broken SMs are highly correlated.

We also show that it is possible to run the analysis on the next generation CUs. Carrying out processing in the deployed infrastructure would minimize the amount of data that needs to be uploaded to the utilities' central servers.

4.6 Related Work

There is rising interest in the benefits from processing data in digitalized systems and especially so in the context of improving sustainable development in cities, where electricity networks is a key component in the infrastructure (c.f. e.g., [9,10] and references therein). Especially in the latter it is highlighted that as data flows continuously in the systems, it is useful to process in a streaming fashion, before the data (or summaries of it [11]) is stored in big data-bases where it becomes infeasible to extract useful information in timely fashion. The latter is also one of the focal points of this paper.

The reliability of data and the robustness of the digitalized systems themselves are key issues. As examples, it is possible to mention that in [12] the authors have shown how to deal with data validation through continuous data-stream processing of the electricity consumption measurements so as to have trustworthy data for billing and for further processing to e.g. use in planning operations. In [13,14] the authors study the problem of detecting potential intrusions in AMI, since these are highly motivated.

Besides protecting the robustness and the reliability of the infrastructures, it has been shown that general data processing and stream processing can generate valuable information, for e.g. detecting fraud, non-technical losses, power outages [6,15,21], thus protecting both safety/societal and economic aspects.

Our work combines stream processing of AMI voltage data with correlations of time series in order to detect broken SMs.

Correlation of consumption value time series has been used in a streaming fashion in order to clusters of similar customers [6].

Voltage data has seen increasing use in Smart Grids recently. E.g. voltage time series correlations have been used in order to verify the documented grid topology, either by correlating smart meter data with transformers [4,18], or by correlating only smart meter data [19]. To the best of our knowledge, voltage data has never been used to identify broken smart meters nor have voltage correlations been done in a streaming and distributed fashion. Other uses of voltage data include power quality estimation in the grid [20].

Identification of bad individuals by peer to peer accusations has also been explored in wireless [21] and vehicular networks [22], where the notion of a group has a physical interpretation. Here we need to induce this information through the temporal dimension of the measurements of SMs, among the dynamic set reporting to the CU that locally processes the data.

4.7 Conclusions and future work

The digitalization of electrical grids and in particular AMIs can provide the means to not only take and report measurements, but also to process the data in the deployed IT infrastructure and generate valuable information at the edge of the network, without relying on big cloud infrastructures and data centers. We strengthen this statement by addressing the problem of continuous distributed monitoring of voltage measurement streams, for detecting broken smart meters. Having this information is important for reliable billing, for prompt reaction for safety reasons, and, consequently, for the business value of the utility. We show that it is possible to have high accuracy and timely detection, even when the processing is done through resource-constrained devices such as the ones that are common in AMIs. The latter implies that this is achievable at a negligible cost for the utility.

Continuous stream-based monitoring can be beneficial for a series of other purposes, including facilitating planning operations, use of renewables and identifying other types of anomalies and unwanted situations.

Acknowledgment

This work was partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, by the Swedish Research Council (Vetenskapsrådet) proj. “HARE: Self-deploying and Adaptive Data Streaming Analytics in Fog Architectures” grant nr. 2016-03800, the Swedish Foundation for Strategic Research, project FiC, grant nr. GMT14-0032, by the EU Horizon 2020 Framework Programme under grant agreement 773717 and by the collaboration framework of Göteborg Energi and Chalmers Energy Area of Advance project STAMINA and project INDEED.

Bibliography

[1] Navigant Research, "Smart grid networking and communications," 2012.

[2] "Apache Flink," https://flink.apache.org/, 2014, last accessed: October 26, 2020.

[3] V. Gulisano, "Streamcloud: an elastic parallel-distributed stream processing engine," Ph.D. dissertation, Universidad Politécnica de Madrid, 2012.

[4] R. Mitra, R. Kota, S. Bandyopadhyay, V. Arya, B. Sullivan, R. Mueller, H. Storey, and G. Labut, "Voltage correlations in smart meter data," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* Sydney, Australia: ACM, 2015, pp. 1999–2008.

[5] J. Bird, *Higher engineering mathematics.* London: Routledge Publishers, 2007.

[6] P. P. Rodrigues and J. Gama, "A system for analysis and prediction of electricity-load streams," *Intelligent Data Analysis*, vol. 13, no. 3, pp. 477–496, 2009.

[7] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.

[8] Hardkernel co., Ltd, "Hardkernel odroid xu4," https://magazine.odroid.com/odroid-xu4/, 2015, last accessed: October 26, 2020.

[9] D. Alahakoon and X. Yu, "Smart electricity meter data intelligence for future energy systems: A survey," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 425–436, 2016.

[10] V. Gulisano, M. Almgren, and M. Papatriantafilou, "When smart cities meet big data," *Smart Cities*, vol. July 2014, no. 98, p. 40, 2014.

[11] V. Gulisano, V. Tudor, M. Almgren, and M. Papatriantafilou, "Bes: Differentially private and distributed event aggregation in advanced metering infrastructures," in *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security.* Xi'an, China: ACM, 2016, pp. 59–69.

[12] V. Gulisano, M. Almgren, and M. Papatriantafilou, “Online and scalable data validation in advanced metering infrastructures,” in *Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2014 IEEE PES*. Istanbul, Turkey: IEEE, 2014, pp. 1–6.

[13] A. A. Cárdenas, R. Berthier, R. B. Bobba, J. H. Huh, J. G. Jetcheva, D. Grochocki, and W. H. Sanders, “A framework for evaluating intrusion detection architectures in advanced metering infrastructures,” *IEEE Transactions on Smart Grid*, vol. 5, no. 2, pp. 906–915, 2014.

[14] V. Gulisano, M. Almgren, and M. Papatriantafilou, “Metis: a two-tier intrusion detection system for advanced metering infrastructures,” in *International Conference on Security and Privacy in Communication Systems*. Beijing, China: Springer, 2014, pp. 51–68.

[15] C. Barreto and A. A. Cárdenas, “Detecting fraud in demand response programs,” in *Decision and Control (CDC), 2015 IEEE 54th Annual Conference on*. Osaka, Japan: IEEE, 2015, pp. 5209–5214.

[16] V. Botev, M. Almgren, V. Gulisano, O. Landsiedel, M. Papatriantafilou, and J. van Rooij, “Detecting non-technical energy losses through structural periodic patterns in ami data,” in *Big Data (Big Data), 2016 IEEE International Conference on*. Washington DC, USA: IEEE, 2016, pp. 3121–3130.

[17] Z. Fu, M. Almgren, O. Landsiedel, and M. Papatriantafilou, “Online temporal-spatial analysis for detection of critical events in cyber-physical systems,” in *Big Data (Big Data), 2014 IEEE International Conference on*. Washington DC, USA: IEEE, 2014, pp. 129–134.

[18] H. Pezeshki and P. Wolfs, “Correlation based method for phase identification in a three phase lv distribution network,” in *Power Engineering Conference (AUPEC), 2012 22nd Australasian Universities*. Bali, Indonesia: IEEE, 2012, pp. 1–7.

[19] W. Luan, J. Peng, M. Maras, J. Lo, and B. Harapnuk, “Smart meter data analytics for distribution network connectivity verification,” *IEEE Transactions on Smart Grid*, vol. 6, no. 4, pp. 1964–1971, 2015.

[20] C. Almeida and N. Kagan, “Using genetic algorithms and fuzzy programming to monitor voltage sags and swells,” *IEEE Intelligent Systems*, vol. 26, no. 2, pp. 46–53, 2011.

[21] S. Y. Lim and Y.-H. Choi, “Malicious node detection using a dual threshold in wireless sensor networks,” *Journal of Sensor and Actuator Networks*, vol. 2, no. 1, pp. 70–84, 2013.

[22] M. Raya, P. Papadimitratos, I. Aad, D. Jungels, and J.-P. Hubaux, “Eviction of misbehaving and faulty nodes in vehicular networks,” *IEEE Journal on Selected Areas in Communications*, vol. 25, no. 8, pp. 1557–1568, 2007.