A Distributed Stream Processing based Architecture for IoT Smart Grids Monitoring

Otávio Carvalho
Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
omcarvalho@inf.ufrgs.br

Eduardo Roloff
Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
eroloff@inf.ufrgs.br

Philippe O. A. Navaux
Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
navaux@inf.ufrgs.br

ABSTRACT

Sensor networks have become ubiquitous – being present from personal smartphones to smart cities deployments – and are producing large volumes of data at increasing rates. Distributed event stream processing systems, in its turn, are a specific kind of systems that help us to parallelize event processing. Therefore, they provide us capabilities to produce quick insights and decisions, in near real-time, on top of multiple data streams.

However, current systems for large scale processing do not focus on Internet of Things and Sensor Network workloads, which makes the performance decrease quickly as the workload size increases. In order to process large scale events with acceptable latency percentiles and high throughput, special systems are needed, such as distributed event stream processing systems.

In this work, we propose an architecture for Internet of Things data workloads, in a combination of sensor networks data sources and distributed event stream processing systems, focused on smart grid data profiles. In our evaluations, the system was able to process up to 45K messages per second using 8 processing nodes, while providing stable latencies for micro-batches above 30 seconds.

KEYWORDS

Smart Grids, Internet of Things, Sensor Networks, Stream Processing, Distributed Processing, Cloud Computing

1 INTRODUCTION

The Internet has made a significant impact in our economy and society by offering a remarkable networking infrastructure for communication. In global information and media sharing, the Internet has been a major driver. It is now turning each time more ubiquitous, mainly due to the arrival of wireless broadband connectivity at each time lower costs [20].

Advancements in technologies related to data collection, such as embedded devices and Radio-Frequency Identification (RFID) technology, has led to an increase into the number of devices connected to networks producing data, leading to the advent of Wireless Sensor Networks (WSNs). The continuation of this trend is called Internet of Things (IoT), where the web provides the medium to the objects interact between themselves [3].

Although the proliferation in connectivity and pervasivity of data produced by sensors provides large benefits for everyone, it also produces large challenges related to data processing. The datasets produced by IoT sensors represent a challenge in the data velocity aspect of big data, which we need to overcome in order to guarantee that data will be processed and we will be able produce insights for organizations in the expected time spans [17].

Smart grids will allow consumers to receive near real-time feedback about their energy consumption and price, enabling them to make their own informed decisions about consumption and spending. On the producer point-of-view, we can leverage home consumption data to produce energy forecasts, enabling near real-time reaction and a better scheduling of energy generation and distribution [4]. In this way, smart grids will save billions of dollars on both sides in the long run, for consumers and the generators, according to recent forecasts [16].

Since millions of end-users will be taking part into processes and information flows of smart grids, high scalability of these methods turns into an important issue. To solve these issues, cloud computing services present themselves as a viable solution, by providing reliable, distributed and redundant capabilities at global scale [5].

In this way, the main challenge of this work is to provide an easily scalable platform for smart grids that is able to process load measurements and measurement predictions over sensor networks data flows.

2 RELATED WORK

This work contributions are placed between the evolving architectures for Smart Grids and the novel distributed architectures for event processing. In this section, we analyse the state-of-the-art under these two aspects and how they relate to our work.

2.1 Load Forecasting Using Event Processing

There are many research groups working recently on the field of smart grids. A subset of them has been focusing on applying Event Processing systems to solve smart grids scalability and data processing issues. Zhou et al [23] and Ziekow et al [24] provide a middleware solution for smart grids, and the ladder extends the former to address load demand balancing issues. Dunning et al [8] provides a highly scalable quantile estimator called t-digest, which was designed for parallel online operation.

The Grand Challenge [25] held in the 8th ACM International Conference on Distributed Event-Based Systems (DEBS) was one

UCC'17 Companion, , December 5–8, 2017, Austin, TX, USA.

ACM ISBN 978-1-4503-5195-9/17/12…$15.00
https://doi.org/10.1145/3147234.3148105

of the motivations for this work. The solutions presented to the challenge represent some interesting techniques for load prediction and outlier detection for smart grid systems. In Perera et al [15] is proposed a technique with a histogram of values to predict the median and a min-max heap approach. In Martin et al [12] they propose an approach for data completion based on work measures, generating missing load measurements based on work measurements data. In Sunderrajan et al [18] it is provided an approach that is similar to ours, but using Apache Storm as a baseline, providing better overall latencies but worse average throughputs.

Our work relates to Aprelkin [2], that also works with Short Term Load Forecast (STLF), but focus on prediction algorithms and do not explore distributed processing. Waalinder et al [19] also works with Advanced Metering Infrastructure (AMI), but with focus on analytics and social networks, providing metrics for the end-user to take smart decisions about energy consumption. Finally, Kumar et al [10] provides an architecture for processing sensor network data, for a Water Distribution Network using Apache Storm. He manages to provide analytics for detection of anomalies and explores the scalability of the system.

2.2 Distributed Architectures for Internet Scale

Applications that require real-time or near real-time processing functionalities are changing the way that traditional data processing systems infrastructures operate. They are pushing the limits of current architectures forcing them to adapt in order to provide better throughputs with the lowest possible latencies.

Hadoop [21] has proved that the development of large scale distributed processing systems on the cloud is achievable. However, its bottlenecks were soon exposed, and it is now widely understood that its architecture is more suitable to workloads that are batch oriented [14]. Their architecture inspired other approaches to develop large scale distributed systems for Internet scale, that started to focus each time more into near real-time processing [7]. The most prominent ones are, respectively: Lambda Architecture and Kappa Architecture.

2.2.1 Lambda Architecture. In its architectural design, the data input is sent both to an offline and to an online processing systems. Both systems execute the same processing logic and output results to a service layer. Queries from back-end systems are executed based on the data on service layer, joining the results produced by offline and online processing systems [13].

The use of this pattern allows organizations to adapt their current infrastructures to support near real-time applications, but it is costly to develop and maintain, as you need to keep the same processing logic updated for both the batch and stream layers.

2.2.2 Kappa Architecture. In this architecture, a single near real-time system, e.g. an event stream processing platform, processes the input data. To re-process data, a new job starts in parallel to an existing one. It re-processes the data from scratch and outputs the results to a service layer. After the job has finished, back-end systems read the data loaded by the new job from the service layer. This architecture provides low latency and high throughput, but it has a higher storage footprint. [9].

While the Lambda Architecture provides a complete solution for both high velocity data and batch processing, Kappa Architecture focuses only on high velocity data, which is desirable for an IoT architecture, but does not provides a solution for queries on the output data.

In order to process output data in a Kappa Architecture approach, data needs to iterate over the stack, leading to unnecessary reprocessing. Our approach, on the other hand, is implemented as a speed layer only solution, inspired by Kappa Architecture, but does not rely on such strict rules and single processing path, which could lead to higher storage footprints and a single immutable processing path.

3 DESIGN AND IMPLEMENTATION

The designed architecture was built based on the state-of-the-art research on distributed processing for event stream processing, maintaining characteristics such as a high throughput and low latency, while keeping large scalability and availability in mind. In order to obtain the desired characteristics, we have studied the patterns for successful implementation of large scale processing, mainly those focused on the velocity aspect of big data [17].

In this section, we present the reasoning behind our design and how our initial design goals were achieved through our implementation.

3.1 Cyclic Architecture

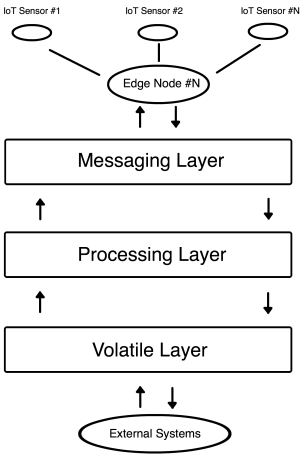

Figure 1: An overview of the proposed Cyclic Architecture

The proposed architecture is called Cyclic architecture, and it is represented by a three layered structure, based on a Messaging Layer, a Processing Layer and a Volatile Layer, as it can be seen on Figure 1.

Together, these layers should provide a reliable architecture upon which we can build highly scalable systems, that should be able to process large amounts of data with high throughput and low latency.

In comparison with other systems, like the Lambda Architecture, we might consider that all of these three layers are inside of a Speed

Layer, as the idea of our design is to process flows of events and not large batches of data.

3.1.1 Messaging Layer. This layer is responsible for communicating with the underlining systems that will provide input data to be processed by the next layer. It should only store the amount of data that is needed by the Processing Layer at a given point in time. It is also responsibility of this layer to provide ways for the processing layer to provide outputs to the underlying systems to receive updates about the current status of the data on the Processing Layer. Thus, for example, IoT Edge Nodes can provide raw data and receive processed updates that, in this turn, can be used by IoT systems to take action over the messages received.

3.1.2 Processing Layer. On this layer the events are processed as they arrive, which can be implemented on event windows or actions that need to be taken per event. After these events have been processed, we might output results to the underlining systems, as queues or other methods of communication through the Messaging Layer, as well as provide updates to the Volatile Layer, that will provide information to External Systems interested on these outputs.

3.1.3 Volatile Layer. The Volatile Layer, in our design, is represented as a way to cache our results from Processing Layer to external systems interested on being updated by what is the current result of what is being processed by the Processing Layer. The main idea is to provide eventually consistent data to external systems, without transactional guarantees or compromise of storing complete historical data.

4 IMPLEMENTATION

The architectural implementation of the design discussed on Section 3 is presented on this section. The implementation consists of a series of layers, each one represented by the framework and tools which were used for its implementation, as it can be found on Figure 2. Next, we are going to explain the decisions made during the implementation phase and its motivations.

4.1 Architectural Implementation

In the first layer, we have used a distributed message framework, called Apache Kafka, as our *Messaging Layer*. Several options were considered for this layer, such as: Message Queues (MQs) like ZeroMQ[1] and Apache Qpid[2]; Higher level messaging protocols like XMPP[3]; Low memory footprint protocols like CoAP[4] and MQTT[5]. However, we ended up choosing Apache Kafka[6] mainly due to its proved capability of supporting large scale and high throughput systems, with strong fault tolerance and rebalancing algorithms.

The next layer is the *Processing Layer*, represented here by the chosen stream processing framework Spark Streaming [22]. As we have discussed in the previous section, it is important because of its ability to provide high processing throughputs and exactly-once processing semantics through a simple programming API.

Once the measurements are received, they are processed by Spark Streaming transformations and the results sent to the *Volatile Layer* to be queried by external systems. Depending on the algorithm needs, the messaging layer can also be used to trigger other computations or as a way to communicate with the sensors that are generating the raw data to the Processing Layer.

We have also considered using Apache Storm [7] and Apache Flink [8] as a medium to implement the Processing Layer. However, we have found that despite the fact that Apache Storm could provide better latencies, it was difficult to achieve comparable throughputs in comparison to Spark Streaming. Also, we have found that Flink Streaming windowing system was still under development at the time we started our development, lacking some important features that were needed by our design. Finally, the *Volatile Layer* is implemented using a Redis [9] key-value in-memory data store. Using Redis, we can do simple and fast distributed I/O and, since we do not rely on complex queries and do not need to store data permanently, the model fits well the architectural design needs.

4.2 Processing Flow

The processing starts with the data being read from disk and put into Kafka topics. Once the data is in Kafka topics, the system makes data partitions and starts to send them to Spark Streaming nodes. When the data arrives at Spark Streaming nodes, the system does the processing and, in the end of a time window, stores load prediction data into Redis.

The most complex part of processing is done using Spark Streaming, which can be described as the steps of the processing flow inside of the Spark Streaming block on Figure 2.

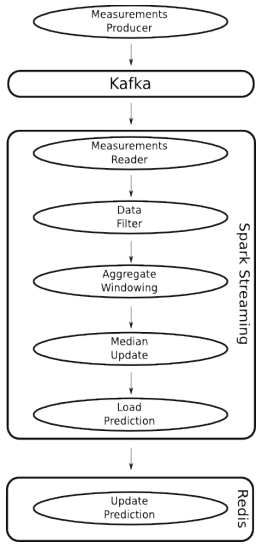

Figure 2: An overview of the data processing flow

[1] http://http://zeromq.org/
[2] http://qpid.apache.org/
[3] http://xmpp.org/
[4] http://coap.technology/
[5] http://mqtt.org/
[6] http://kafka.apache.org
[7] http://storm.apache.org/
[8] http://flink.apache.org/
[9] http://redis.io

The processing starts with the *Measurements Producer*, which is a Scala application which reads from the input file and sends data to the Kafka destination topic. Then the *Measurements Reader* gets the input from the Kafka topic using the Kafka's High Level API. The *Data Filter* cleans the dataset, which in our case means get rid of work measurements in the dataset, as well as invalid measurements, and keep only the valid power measurements.

Aggregate Windowing is the step where we group similar load measurements for the prediction algorithm processing, which in our case means grouping measurements within the same house and the same plugs. The next step is the *Median Update*, which gets the average for the current time slice and updates the set of averages for the current time slice, in order to keep data updated for the system to calculate the median in the next iteration (as it was previously described, there are a certain number of time slices that are circularly updated based on calculations based on their timestamp).

Finally, the set of measurements within the time window, together with the previous median for the current time slice are used to calculate the *Load Prediction* step.

The final result is stored on Redis, which represents our *Volatile Layer*, and can be used for external queries to the AMI. The processed events can also be redirected to another Kafka topics and the same architecture could be used to reprocess this data in another queries, as well as feed another systems that could want to keep further processing this data.

5 EVALUATION

In this section, we explain the approach we have used to evaluate our platform and what results were found on these evaluations. Section 5.1 describes the platform used as the basis for running our tests. Section 5.4 explores the system regarding to the achievable latency, measuring the end-to-end time taken by events to traverse the system. Section 5.5 does a series of tests to evaluate the throughput of the platform, helping us to better understand how the system behaves when the number of nodes increases.

5.1 Platform

In order to evaluate the system, we have decide to use an IaaS cloud platform provided to us by Microsoft Azure. The platform built upon Microsoft Azure to host our application was configured using the settings described in Table 1.

The system was then configured to operate with one node acting exclusively as the Spark master, with up to 8 nodes as Spark slave processing nodes. We also separate one node exclusively for Kafka, to receive writes from input readers and receive reads from processing nodes. Kafka was configured to use a single broker, providing an average write throughput of 45K events per second in all of the following tests.

5.2 Data

The dataset used to evaluate the platform is originated from the 8th DEBS Grand Challenge. This conference provides competitions with problems which are relevant for the industry and, in 2014, the conference challenge focus was on the ability of CEP systems to apply on real-time predictions over a large amount of sensor data.

Table 1: Platform evaluation: Virtual machines and toolset description

Parameter	Description
Instance Type	Standard_A3 (4 cores, 7 GB of RAM)
Nodes	10
Operating System	Ubuntu 14.04 LTS
Location	West Europe
Kafka Version	0.8.2.1
Scala Version	2.10
Java Version	1.7.0_80
Zookeeper Version	3.4.6
Spark Version	1.5.1
Redis Version	3.0.4

For this purpose, household energy consumption measurements where generated, based on simulations driven by real-world energy consumption profiles, originating from smart plugs deployed in households. [25]

For the purpose of the challenge a number of smart plugs have been deployed in households with data being collected roughly every second for each sensor in each smart plug. It has to be noted that the dataset is collected in an uncontrolled, real-world environment, which implies the possibility of malformed data as well as missing measurements.

In Table 2 we describe how is the layout of each one of the measurements into the dataset. It is uses a hierarchical structure to represent the relation of the smart plugs, households, and houses. A house is identified by a unique house id. A house is the topmost entity. Every house contains one or more households, identified by a unique household id. Each household id is unique only within a given house. Every household contains one or more smart plugs, each identified by a unique plug id. Similar to household id, the plug id is unique only within a given household. Every smart plug contains exactly two sensors: 1) a load sensor measuring current load with Watt as unit and 2) a work sensor measuring total accumulated work since the start (or reset) of the sensor with kWh as unit.

For our tests, we have used a subset of the original file, in order to decrease the total processing time and being able to execute a greater number of simulation tests. The subset of the file contains 100 Million measurements, the same amount of plugs and houses, for a total amount of 3.6 GB. The subset file covers a period of two days, which is an important characteristic to test our prediction algorithm using historical data (data from the previous day).

5.3 Forecasting Method

Smart grid deployments carry the promise of allowing better control and balance of energy supply and demand through near real-time, continuous visibility into detailed energy generation and consumption patterns. Methods to extract knowledge from near real-time and accumulated observations are hence critical to the extraction of value from the infrastructure investment.

In this context, STLF refers to the prediction of power consumption levels in the next hour, next day, or up to a week ahead. Methods

Table 2: Dataset: Schema and overview

Name	Description	Unit
id	Unique identifier	Number
timestamp	Measurement timestamp	Number of seconds (since January 1, 1970, 00:00:00 GMT)
value	Measurement value	kWh or Watt
property	Type of measurement	0 or 1
plug_id	Identifier of the smart plug	Number
household_id	Identifier of where the plug is located	Number
house_id	Identifier of the house where the household with the plug is located	Number

for STLF consider variables such as date (e.g., day of week and hour of the day), temperature (including weather forecasts), humidity, temperature-humidity index, wind-chill index and most importantly, historical load. Residential versus commercial or industrial uses are rarely specified.

Time series modeling for STLF has been widely used over the last 30 years and a myriad of approaches have been developed. These methods [11] can be summarized as follows:

- Regression models that represent electricity load as a linear combination of variables related to weather factors, day type, and customer class.
- Linear time series-based methods including the Autoregressive Integrated Moving Average (ARIMA) model, auto regressive moving average with external inputs model, generalized auto-regressive conditional heteroscedastic model and State-Space Models (SSMs).
- SSMs typically relying on a filtering-based (e.g., Kalman) technique and a characterization of dynamical systems.
- Nonlinear time series modeling through machine learning methods such as nonlinear regression.

Shawkat Ali [1] argues that the three most accurate models for load prediction are, respectively, Multilayer Perceptron (MLP), Support Vector Machine and Least Mean Squares. Due to the model fit in relation to the distributed architecture, we decide to pursue the approach suggested by the conference committee [25], that is schematically described in Equation (1). This approach could be interpreted as a mixed approach between MLP and ARIMA. It brings together characteristics from both Linear time series-based methods and SSMs [6].

More specifically, the set of queries provide a forecast of the load for: 1) each house, i.e., house-based and 2) for each individual plug, i.e., plug-based. The forecast for each house and plug is made based on the current load of the connected plugs and a plug specific prediction model. The aim of these queries is not at the over the better prediction model, but at stressing the interplay between modules for model learning that operate on long-term (historic) data with components that apply the model on top of live, high velocity data.

$$L(s_{i+2}) = \frac{avgL(s_i) + median(avgL(s_j))}{2} \quad (1)$$

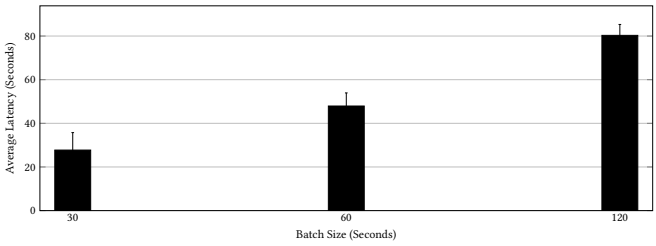

Figure 3: Best case scenario - Large batches with 8 processing nodes.

In the Equation (1), $avgL(s_i)$ represents the current average load for the slice s_i. The value of $avgL(s_i)$, in case of plug-based prediction, is calculated as the average of all load values reported by the given plug with timestamps $\in s_i$. In case of a house-based prediction the $avgL(s_i)$ is calculated as a sum of average values for each plug within the house. $avgL(s_j)$ is a set of average load value for all slices s_j such that:

$$s_j = s_{i+2-n*k} \quad (2)$$

In the Equation (2), k is the number of slices in a 24 hour period and n is a natural number with values between 1 and $floor(\frac{i+2}{k})$. The value of $avgL(s_j)$ is calculated analogously to $avgL(s_i)$ in case of plug-based and house-based (sum of averages) variants.

5.4 Latency

The latency analysis consists on the measurement of the time taken from an event since it was generated or read to its arrival at the end of the processing pipeline. In our case, due to the predominant time taken by the processing step, the time taken from an event to be generated, or read from the dataset, and received into the data pipeline is not considered. Due to this fact, the focus of our analysis was on the processing step, which was done through measurement of the time taken by events to be processed from the beginning to the end of the Spark Streaming pipeline.

As it can seen in Figure 3, when batch sizes are large the system has enough time to schedule and process events through application pipeline, not incurring into schedule delays or processing pressure due to time constraints. In this way, the system is able to maintain itself below the time limit, which means below the value of the batch size.

5.5 Throughput

The throughput is an important metric in an AMI, since it bounds the number of possible clients that can be reach by the AMI smart grid systems, given the number of messages per second a single meter will provide. To measure the throughput in our platform, we analyse the system behavior when the number of nodes increases and also how it behaves with different batch sizes.

As we can see in Figure 4, the system scales linearly up to 8 nodes, doubling the input processing rate when the number of machines doubles. That can be also perceived that, up to the boundary of the input rate, the system is able to handle the data input of a

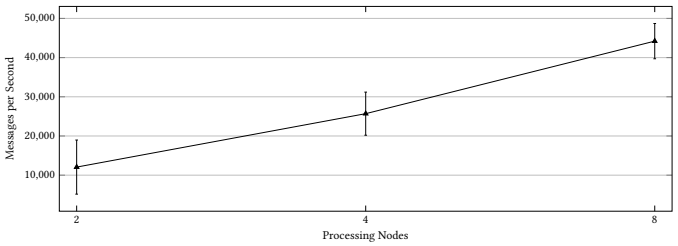

Figure 4: Average message throughput, by number of nodes, with 30 seconds batch

single *Measurements Producer*. In order to add more nodes to the processing system, we would need to add more *Measurements Producer* nodes, and distribute them among different Kafka brokers, in order to parallelize not only the reads from Kafka, but also the parallel writes to the system queue.

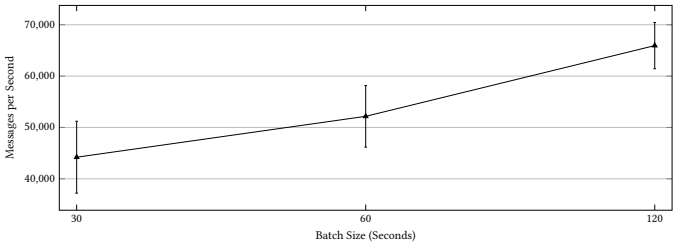

Figure 5: Average message throughput, by batch sizes, with 8 processing nodes

The second step of the testing process was the testing of some other important parameters of the system. We have decided to test the effect of batch sizes into the overall throughput of the system, due to the impact that we have seen on latency. We also decided to use the maximum number of available nodes (8 nodes) and batch sizes that presented a stable performance on the previous tests — batches from 30 seconds to 120 seconds —.

It is possible to analyse from these results that batch sizes affect latency more than throughput as it is shown on Figure 5. However, we can see that the throughput do not exponentially grows, as it happens with latency when the batch sizes increase.

6 CONCLUSION AND FUTURE WORK

The main goal of this work was to provide a high performance scalable architectural solution for distributed event stream processing, focusing on smart grids data profiles. The main goal was achieved, the system was able to handle the pressure with high throughput and was able to scale linearly up to 8 processing nodes.

Future works will include a deeper research on prediction forecasting and results on forecast accuracy, as well as other techniques to increment prediction quality, such as recovery of lost load energy measurements to use weather predictions into energy load forecast prediction algorithms.

Furthermore, we would like to explore opportunities to improve load balancing by moving computation into IoT edge nodes, in order to improve both throughput and latency performance by moving computation to the network edges.

ACKNOWLEDGMENTS

This research received partial funding from CYTED for the RICAP Project. It has also received partial funding from the EU H2020 Programme and from MCTI/RNPBrazil under the HPC4E project, grant agreement no. 689772.

Additional funding was provided by FAPERGS in the context of the GreenCloud Project.

REFERENCES

[1] A. B. M. Shawkat Ali. 2013. *Smart Grids: Opportunities, Developments, and Trends.* Springer Science & Business Media.
[2] Alexander Aprelkin. 2014. Short Term Household Electricity Load Forecasting Using a Distributed In-Memory Event Stream Processing System. (2014).
[3] Kevin Ashton. 2009. That Internet of Things Thing. *RFID Journal* 22, 7 (2009), 97–114.
[4] Richard E Brown. 2008. Impact of Smart Grid on Distribution System Design. In *Power and Energy Society General Meeting*. IEEE, 1–4.
[5] Rajkumar Buyya et al. 2009. Cloud Computing and Emerging IT Platforms. *Future Generation Computer Systems* 25, 6 (2009), 599–616.
[6] Tom Bylander et al. 1997. A Perceptron-like Online Algorithm for Tracking the Median. In *Neural Networks, International Conference on*, Vol. 4. IEEE, 2219–2224.
[7] Otávio Carvalho, Eduardo Roloff, et al. 2013. Beyond Hadoop: An Analysis of The Evolution of New Technologies for Cloud Computing. In *XXV Symposium in Computational Systems, WSCAD-WIC*.
[8] Ted Dunning et al. 2014. *Practical Machine Learning: A New Look at Anomaly Detection.* O'Reilly Media, Inc.
[9] Jay Kreps. 2014. Questioning the Lambda Architecture. (August 2014). http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html
[10] Simpal Kumar. 2014. Real Time Data Analysis for Water Distribution Network using Storm. (2014).
[11] Elias Kyriakides et al. 2007. Short Term Electric Load Forecasting: A Tutorial. In *Trends in Neural Computation*. Springer, 391–418.
[12] André Martin et al. 2014. Predicting energy consumption with StreamMine3G. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 270–275.
[13] Nathan Marz et al. 2015. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems.* Manning Publications Co.
[14] Andrew Pavlo et al. 2009. A Comparison of Approaches to Large-Scale Data Analysis. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. ACM, 165–178.
[15] Srinath Perera et al. 2014. Solving the Grand Challenge Using an Open Source CEP engine. In *Proceedings of the 8th International Conference on Distributed Event-Based Systems*. ACM, 288–293.
[16] Reuters. 2011. U.S. Smart Grid to Cost Billions, Save Trillions. (2011). http://www.reuters.com/article/2011/05/24/us-utilities-smartgrid-epri-idUSTRE74N7O420110524
[17] Seref Sagiroglu et al. 2013. Big Data: A Review. In *Collaboration Technologies and Systems (CTS), International Conference on*. IEEE, 42–47.
[18] Abhinav Sunderrajan et al. 2014. Real Time Load Prediction and Outliers Detection Using Storm. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 294–297.
[19] Carl Walinder et al. 2015. BCStream - A Data Streaming Based System for Processing Energy Consumption Data and Integrating with Social Media. (2015).
[20] Mark Weiser et al. 1999. The Origins of Ubiquitous Computing Research at PARC in the Late 1980s. *IBM systems journal* 38, 4 (1999), 693–696.
[21] Tom White. 2012. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc.
[22] Matei Zaharia et al. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, Vol. 10. 10.
[23] Qunzhi Zhou et al. 2013. On Using Complex Event Processing for Dynamic Demand Response Optimization in Microgrid. In *Proceedings of IEEE Green Energy and Systems Conference*. IEEE.
[24] Holger Ziekow et al. 2013. Forecasting Household Electricity Demand with Complex Event Processing: Insights from a Prototypical Solution. In *Proceedings of the 13th ACM International Middleware Conference*. ACM, 2.
[25] Holger Ziekow and Zbigniew Jerzak. 2014. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-based Systems, DEBS*, Vol. 14.