UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

OTÁVIO MORAES DE CARVALHO

Distributed Near Real-Time Processing of Sensor Network Data Flows for Smart Grids

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Philippe O. A. Navaux
Coadvisor: Prof. M.Sc. Eduardo Roloff

Porto Alegre
December 2015

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do Curso de Ciência de Computação: Prof. Carlos Arthur Lang Lisbôa
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"The one who has conquered himself is a far greater hero than he who has defeated a thousand times a thousand men."

— Shakyamuni Buddha

ACKNOWLEDGMENTS

First of all, I would like to give a special thanks to my parents, Jorge and Nadia, for all the love and support. You are my life lessons on the importance of kindness and wholeheartedness. I would also like to wholeheartedly thank my uncle Marcelo and my aunt Ana — without whom I would never being here — not only for their extensive support throughout my life, but also for sharing their family values with me and for making me part of their family.

Second, I would like to thank everyone in the Informatics Institute at Federal University of Rio Grande do Sul (UFRGS), by the opportunities and education of excellence that was provided to me and my colleagues. Thanks also to my advisor and co-advisor, Prof. Dr. Philippe Navaux and Prof. MSc. Eduardo Roloff, respectively, for all of the support, openness to new ideas and confidence into my work. Furthermore, I would also like to thank everyone at Parallel and Distributed Research Group (GPPD), without whom my education certainly would not have been as complete and my path would not have been as enjoyable as it was in the last few years.

Finally, I would like to thank my friends, colleagues and co-workers for the support, insights and great moments. You are a considerable part of what made possible this accomplishment.

ABSTRACT

Sensor networks have become ubiquitous, ranging from personal mobile phones to smart grids, and are producing each time higher amounts of data, in ever shorter time intervals. Distributed event stream processing systems, in its turn, are systems that help us to parallelize not only the processing, but also the input of multiple data streams into a single processing engine, providing us capabilities to produce near real-time insights based on multiple data streams, as well as make decisions more quickly.
Joining together these ideas, in this work, we propose an architecture based on open source tools that represent the state-of-the-art in distributed event stream processing systems. In this manner, we aim to provide a platform for processing large scale sensor network data, focused on data profiles of smart grids.
To evaluate the feasibility of a system of this kind, we use a dataset based on a sensor network for smart energy consumption meters, in order to generate load forecasts based on this dataset. In the end, we evaluate the proposed architecture regarding to processing scale and latency issues. Achieving the conclusions that it is possible to build a distributed processing platform, for processing of sensor network data flows coming from smart grids, as it was designed on this work. The platform is able to process up to approximately 45K messages per second using 8 processing nodes, while providing stable latencies for micro-batches above 30 seconds.

Keywords: Event Stream Processing. Distributed Processing. Sensor Networks. Smart Grids. Internet of Things. Cloud Computing.

Processamento Distribuído em Quase Tempo Real de Fluxos de Dados de Redes de Sensores para Redes Inteligentes de Energia

RESUMO

Redes de sensores tornaram-se ubíquas, indo desde telefones móveis pessoais até redes inteligentes de energia, e estão produzindo cada vez maiores quantidades de dados, em intervalos de tempo cada vez menores. Sistemas para o processamento distribuído de fluxo de eventos, por sua vez, são sistemas que ajudam-nos a paralelizar não somente o processamento, mas também a inserção de múltiplos fluxos de dados em um único mecanismo de processamento, proporcionando-nos capacidades para produzir análises em tempo real baseadas em múltiplos fluxos da dados, assim como tomar decisões mais rapidamente.
Unindo essas idéias, neste trabalho, propomos uma arquitetura baseada em ferramentas de código aberto que representam o estado-da-arte em processamento distribuído de fluxo de eventos. Desta maneira, nosso objetivo é oferecer uma plataforma para o processamento de dados em redes de sensores em grande escala, focada em perfis de dados de redes inteligentes de energia. Para avaliar a viabilidade de um sistema desse tipo, nós utilizamos um conjunto de dados baseado em uma rede de sensores para medidores de consumo de energia inteligentes, a fim de gerar previsões de carga baseadas nesse conjunto de dados. No final, nós avaliamos a arquitetura proposta com relação à escala de processamento e problemas de latência. Alcançando as conclusões de que é possível contruir uma plataforma de processamento distribuída, para o processamento de fluxos de dados de redes de sensores provenientes de redes inteligentes de energia, como foi projetado nesse trabalho. A plataforma é capaz de processar até 45 mil mensagens por segundo utilizando 8 nós de processamento, enquanto provê latências estáveis para micro-lotes acima de 30 segundos.

Palavras-chave: Processamento de Fluxo de Eventos, Processamento Distribuído, Redes de Sensores, Redes Inteligentes de Energia, Internet das Coisas, Computação em Nuvem.

LIST OF FIGURES

Figure 2.1 The internet of things paradigm as the convergence of different visions 24
Figure 2.2 Comparison between a simple and a windowed DStream..................................... 32
Figure 2.3 Cloud computing service models stack and their relationships............................. 34

Figure 3.1 Lambda Architecture 44
Figure 3.2 Kappa Architecture... 45
Figure 3.3 Liquid Architecture... 46
Figure 3.4 An overview of the proposed Cyclic Architecture 47
Figure 3.5 An overview of the stack used to implement the Cyclic Architecture 53
Figure 3.6 An overview of the data processing flow.. 54

Figure 4.1 Best case scenario - Large batches with 8 processing nodes................................. 57
Figure 4.2 Sample Execution - 8 processing nodes, 30 seconds batch and stable overall processing 57
Figure 4.3 Worst case scenario - Small batches with 1 processing node................................ 58
Figure 4.4 Sample Execution - 1 processing node, 5 seconds batch and increasing input queueing... 58
Figure 4.5 Average message throughput, by number of nodes, with 30 seconds batch........... 59
Figure 4.6 Average message throughput, by batch sizes, with 8 processing nodes 60

LIST OF TABLES

Table 3.1 Dataset: Schema and overview 49

Table 4.1 Platform evaluation: Virtual machines and toolset description................................ 55

LIST OF ABBREVIATIONS AND ACRONYMS

AMI	Advanced Metering Infrastructure
API	Application Programming Interface
ARIMA	Autoregressive Integrated Moving Average
CEP	Complex Event Processing
CLI	Command Line Interface
DAG	Directed Acyclic Graph
DBMS	Database Management System
DEBS	Distributed Event-Based Systems
DSM	Demand Side Management
DSMS	Data Stream Management System
GFS	Google File System
HDFS	Hadoop Distributed File System
IaaS	Internet as a Service
IoT	Internet of Things
JVM	Java Virtual Machine
MLP	Multilayer Perceptron
NIST	National Institute of Standards and Technology
OS	Operating System
PaaS	Platform as a Service
RDD	Resilient Distributed Dataset
RFID	Radio-frequency Identification
SSM	State-Space Model
STLF	Short Term Load Forecasting
SVM	State Vector Machine

SaaS	Software as a Service
VM	Virtual Machine
WSN	Wireless Sensor Networks

CONTENTS

1 INTRODUCTION **19**
1.1 Motivation **19**
1.2 Objective **20**
1.3 Outline **20**
2 BACKGROUND **23**
2.1 Internet of Things **23**
2.2 Distributed Event Stream Processing Systems **25**
2.2.1 Apache Flink 27
2.2.1.1 Nephele Streaming 28
2.2.2 Apache Spark 29
2.2.2.1 D-Streams 30
2.2.3 Apache Storm 32
2.2.3.1 Spouts 33
2.2.3.2 Bolts 33
2.2.3.3 Topologies 33
2.3 Cloud Computing **33**
2.4 Big Data **37**
2.5 Smart Grid **38**
2.5.1 Advanced Metering Infrastructure 39
2.5.2 Demand Side Management 40
2.5.3 Consumption Forecasting 40
2.6 Chapter Remarks **41**
3 DESIGN AND IMPLEMENTATION **43**
3.1 Architecture **43**
3.1.1 Lambda Architecture 43
3.1.2 Kappa Architecture 44
3.1.3 Liquid Architecture 45
3.1.4 Cyclic Architecture 46
3.2 Data **48**
3.3 Forecasting Method **49**
3.4 Distributed Data Input **50**
3.5 Implementation **52**
3.5.1 Architectural Implementation 52
3.5.2 Processing Flow 53
4 EVALUATION **55**
4.1 Platform **55**
4.2 Latency **56**
4.3 Throughput **58**
5 RELATED WORK **61**
5.1 Energy Consumption and Smart Grids Infrastructure **61**
5.2 Load Forecasting Using Event Processing **62**
6 CONCLUSION AND FUTURE WORK **63**
REFERENCES **65**

1 INTRODUCTION

In Section 1.1 we introduce the motivation behind this work. In Section 1.2 we describe our objectives when we started this work. Finally, in Section 1.3, we describe an outline of the following chapters.

1.1 Motivation

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 (WEISER et al., 1999).

Advancements in technologies related to data collection, such as embedded devices and Radio-Frequency Identification (RFID) technology, had let to an increase in the number of devices connected to networks producing data, leading to the advent of Wireless Sensor Networks (WSNs). The continuation of this trend is Internet of Things (IoT), where the web provides the medium to the objects interact between themselves (ASHTON, 2009).

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 (SAGIROGLU; SINANC, 2013a).

Furthermore, finding ways to achieve a more sustainable lifestyle plays a large hole on the path of our society growth, and poses some challenges to a society that depends each time more on a increasing number of electrical devices. To achieve this, the smart grid incentives pretend to improve the legacy systems for energy production and consumption, based on research, to advance technologies in the energy field.

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 (BROWN, 2008). 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 (REUTERS, 2011).

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 (BUYYA et al., 2009).

Given the volume of data produced and the number of users in smart grid systems, there are large IoT challenges with the growth of these technologies, not only by their huge data flows, but also by the difficulties in scaling those systems. In this way, the main challenge of this work is to build a platform that provides load measures and predictions over sensor networks data while maintaining the capacity of the network to growth over time, and without compromising the quality of their measures and predictions.

1.2 Objective

The objective of this work is to provide an adaptable architecture for processing IoT data flows. We aim to provide a stack based on open source tools that represent the state-of-the-art in distributed event stream processing, providing a reliable and cheap data platform.

In order to show the applicability of the platform, we will build a series of load prediction algorithms for smart grids, test them using a series of realistic datasets, and further discuss what is achievable using this platform, focusing on the aspects of throughput and latency.

Furthermore, we will discuss how a platform of its type could be able to scale without compromising its ability to properly produce measures and predictions, and what is needed to achieve it from the point of view of data ingestion, data processing and data output.

1.3 Outline

Chapter 2 introduces the background needed to better understand our design and implementation, explaining from the trends of the IoT and big data that are part of what motivates this work, to part of the tools that we have used to build it: Distributed event stream processing frameworks and the cloud computing infrastructure.

In the Chapter 3 we present the design of our architecture, citing several architectural patterns and explaining why we believe that our proposed architecture fulfills some needs not

addressed by previous works. Here we also explain where the data used in the evaluation comes from, how the algorithm works inside of the architecture, and how we manage to implement it in a real software platform.

In Chapter 4 we explore the capabilities and limitations of the architecture, in order to better understand what we can expect when using it. After the platform was built and evaluated, in Chapter 5 we discuss works related to the platform we built, what points of the platform were not explored and what we think that still could be improved. Finally, in Chapter 6, we review what we achieved with this work and cite some plans for the improvement of the presented ideas.

2 BACKGROUND

In this chapter, we introduce the necessary background in order to further understand the basic concepts that will be used as a basis along this work. Section 2.1 introduces what is IoT, what are WSNs and why sensors have been becoming ubiquitous in everyday life. Section 2.2 explain what are distributed event stream processing systems, why they are important in fields that require processing for unbounded amounts of data and the relationship of those systems with the IoT field. Section 2.3 we explain what is cloud computing and what are the advantages of using it as the basis to design large scale online systems. Section 2.4 discuss what is big data and what are the challenges to address the need to process this kind of data. Finally, in Section 2.5 we discuss what are smart grids, their importance on the control of global energy consumption, as well as the importance of the field in global environmental protection.

2.1 Internet of Things

IoT is a novel paradigm that is rapidly gaining ground in the scenario of modern wireless telecommunications. The IoT builds on the pervasive presence around us of a variety of things or objects – such as RFID tags, sensors, actuators, mobile phones, etc. – which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbors to reach common goals (ATZORI et al., 2010).

However, for the IoT vision to successfully emerge, the computing criterion will need to go beyond traditional mobile computing scenarios that use smartphones and portables, and evolve into connecting everyday existing objects and embedding intelligence into our environment. For technology to disappear from the consciousness of the user, the IoT demands: (1) a shared understanding of the situation of its users and their appliances, (2) software architectures and pervasive communication networks to process and convey the contextual information to where it is relevant, and (3) the computational artifacts in the IoT that aim for autonomous and smart behavior. With these three fundamental grounds in place, smart connectivity and context-aware computation via anything, anywhere, and anytime can be accomplished (YAN et al., 2008).

Gartner, Inc. forecasts that the IoT will reach 26 billion units by 2020, up from 0.9 billion in 2009, and will impact the information available to supply chain partners and how the supply chain operates. From production line and warehousing to retail delivery and store shelving, the IoT is transforming business processes by providing more accurate and real-time visibility into

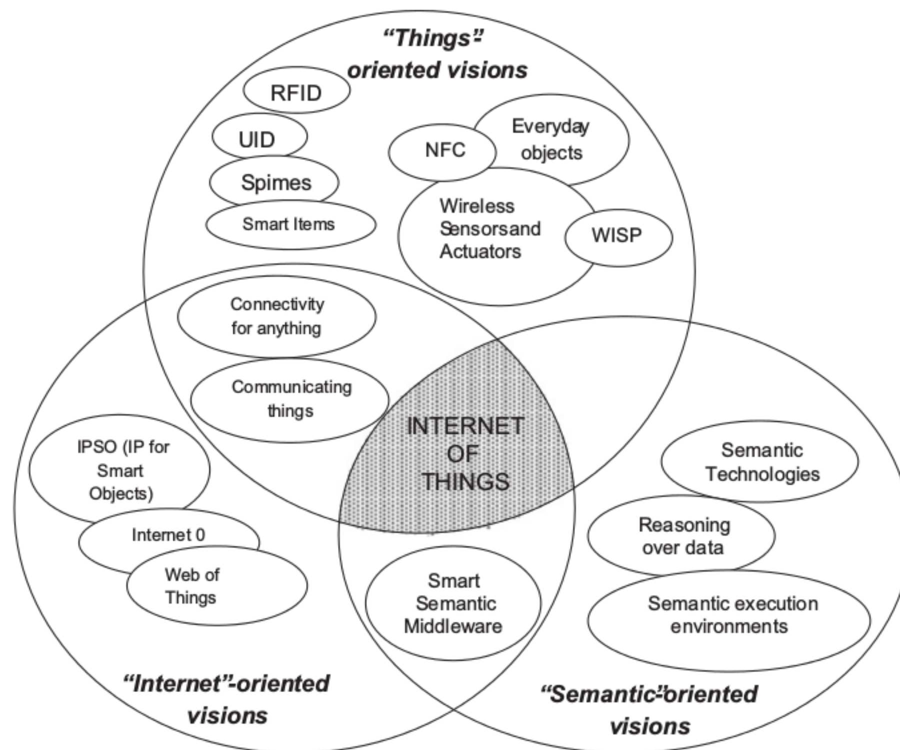

Figure 2.1 – The internet of things paradigm as the convergence of different visions (ATZORI et al., 2010)

the flow of materials and products. Firms will invest in the IoT to redesign factory workflows, improve tracking of materials, and optimize distribution costs. For example, large enterprises such as John Deere and UPS are already using IoT-enabled fleet tracking technologies to cut costs and improve supply efficiency (LEE et al., 2015).

In Figure 2.1, the main concepts, technologies and standards are highlighted and classified with reference to the IoT visions they contribute to characterize best. From such an illustration, it clearly appears that the paradigm shall be the result of the convergence of three main visions. The "Internet oriented" together with the "Things oriented" perspective imply the huge number of interconnected devices connected through unique addressing protocols. The "Semantic oriented" perspective, on the other hand, represents the challenges in data representation for storage and information exchange (ATZORI et al., 2010).

2.2 Distributed Event Stream Processing Systems

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 processing systems forcing them to provide better throughputs with the lowest possible latencies.

The main problems to be solved nowadays are not primarily focused on raw data, but rather in the high-level intelligence that can be extracted from it. As a response, systems were developed to filter, aggregate and correlate data, and notify interested parties about its results, abnormalities, or interesting facts.

However, the distributed processing ecosystem today is mostly focused on Hadoop (WHITE, 2012), which itself is the result of the Google's Inc. research effort into large scale processing, that ended up producing several tools — such as MapReduce (DEAN; GHEMAWAT, 2004) and Google File System (GFS) (GHEMAWAT; GOBIOFF; LEUNG, 2003) — that were rewritten for the open source community through the Apache Foundation.

Hadoop has proved that the development of large scale distributed processing systems on the cloud is achievable. After understanding that it was possible to develop such systems, better approaches were proposed using Hadoop's infrastructure, but focusing on improving the performance of these kinds of systems. The aim was not to limit them only to batch processing, but to evolve them into systems of near real-time processing (CARVALHO; ROLOFF et al., 2013).

During the development of applications that aim to achieve better throughputs using Hadoop, its bottlenecks were exposed, proving that it is not the best platform for certain kinds of intensive data processing systems, being better for workloads that are more batch processing oriented. Aiming improvements in the fields in which Hadoop failed, new approaches to distributed processing were proposed, focusing each time on more reliable processing systems that are not heavily bounded by intensive processing workloads (PAVLO et al., 2009).

These efforts generated a convergence between event processing systems and distributed processing systems, in direction for a merge between those fields. Nowadays event processing systems have been developed focusing on ways for distribute data processing. On the other hand, the most recent distributed processing systems also include complex built-in tools and specific Advanced Programming Interface (API) for easier the process of analysing data (CARVALHO; ROLOFF; NAVAUX, 2013).

We can trace the development of the event stream processing area back to Data Stream

Management Systems (DSMSs), such as TelegraphCQ (CHANDRASEKARAN et al., 2003) and Aurora/Borealis (ABADI et al., 2003) (ABADI et al., 2005), which are similar to Database Management Systems (DBMSs), but focused on managing continuous data streams. In contrast to DBMSs, they execute a continuous query that is not only performed once, but is permanently executed until it is explicitly stopped. The development of the area could also be traced back to the origins of Complex Event Processing (CEP) systems, which are event processing systems that combine data from multiple sources to infer events or patterns that suggest more complicated situations. These systems are represented broadly by traditional content-based publish-subscribe systems like Rapide (LUCKHAM et al., 1995) and TESLA/T-Rex (CUGOLA; MARGARA, 2010) (CUGOLA; MARGARA, 2012).

In the evolution of both kinds of systems, a process of convergence between DSMSs systems and CEP systems had generated intersections between those fields, complicating more the characterization of them in distinct and clear groups.

Aiming to solve this naming problems, efforts were done to group all those kinds of systems into a common terminology. The term Information Flow Processing (MARGARA; CUGOLA, 2011) was created to refer to an application domain in which users need to collect information produced by multiple sources, to process it in a timely way, in order to extract new knowledge as soon as the relevant information is collected.

As well as the information processing systems had aggregated characteristics of distributed processing systems, many distributed processing system are aggregating information flow processing capabilities into their platforms. These changes are making it harder to explain the differences between them, because they are merging into tools that offer characteristics of both of them (CARVALHO; ROLOFF; NAVAUX, 2013).

These new systems have their designs strongly driven by the trend towards cloud computing, which requires the data stream processing engines to be highly scalable and robust towards faults. We can see this trend through well-known system of this generation, that will be discussed in the following subsections, such as Flink 2.2.1, Spark Streaming 2.2.2 and Storm 2.2.3.

More specifically, this generation of streaming systems present a pattern towards a set of common requirements: 1) the scenarios typically involve input streams with high up to very high data rates (> 10000 event/s); 2) they have relaxed latency constraints (up to a few seconds); 3) the use cases require the correlation among historical and live data; 4) they require systems to elastically scale and to support diverse workloads and; 5) they need low overhead fault tolerance, supporting out-of-order events and exactly once semantic (HEINZE et al., 2014).

2.2.1 Apache Flink

Apache Flink Streaming is a distributed stream analytics system that is part of the Apache Flink Stack (former Stratosphere). Apache Flink is architectured around a generic runtime engine uniformly processing both batch and streaming jobs composed of stateful interconnected tasks. Analytics jobs in Flink are compiled into directed graphs of tasks. Data elements are fetched from external sources and routed through the task graph in a pipelined fashion. Tasks are continuously manipulating their internal state based on the received inputs and are generating new outputs (CARBONE et al., 2015).

The Stratosphere software stack consists of three layers, termed the Sopremo, Parallelization Contract (PACT), and Nephele layers. Each layer is defined by its own programming model (the API that is used to program directly the layer or used by upper layers to interact with it) and a set of components that have certain responsibilities in the query processing pipeline.

Sopremo is the topmost layer of the Stratosphere stack. A Sopremo program consists of a set of logical operators connected in a Directed Acyclic Graph (DAG), akin to a logical query plan in relational DBMSs. Programs for the Sopremo layer can be written in Meteor, an operator-oriented query language that uses a JSON-like data model to support the analysis of unstructured and semi-structured data. Meteor shares similar objectives as higher-level languages of other big data stacks, such as Pig (OLSTON et al., 2008) and Hive (THUSOO et al., 2009) in the Hadoop ecosystem, but is highlighted by extensibility and the semantically rich operator model Sopremo, which also lends its name to the layer.

The output of the Sopremo layer and, at the same time, input to the PACT layer of the Stratosphere system is a PACT program. PACT programs are based on the PACT programming model, an extension to the MapReduce programming model. Similar to MapReduce, the PACT programming model builds upon the idea of second-order functions, called PACTs. Each PACT provides a certain set of guarantees on what subsets of the input data will be processed together, and the first-order function is invoked at runtime for each of these subsets. That way, the first-order functions can be written (or generated from a Sopremo operator plan) independently of the concrete degree of parallelism or strategies for data shipping and reorganization. Apart from the Map and Reduce contracts, the PACT programming model also features additional contracts to support the efficient implementation of binary operators. Moreover, PACTs can be assembled to form arbitrarily complex DAGs, not just fixed pipelines of jobs as in MapReduce.

The first-order (user-defined) functions in PACT programs can be written in Java by the user, and their semantics are hidden from the system. This is more expressive than writing

programs in the Sopremo programming model, as the language is not restricted to a specific set of operators. However, PACT programs still exhibit a certain level of declarativity as they do not define how the specific guarantees of the used second-order functions will be enforced at runtime. In particular, PACT programs do not contain information on data repartitioning, data shipping, or grouping. In fact, for several PACT input contracts, there exist different strategies to fulfill the provided guarantees with different implications on the required effort for data reorganization. Choosing the cheapest of those data reorganization strategies is the responsibility of a special cost-based optimizer, contained in the PACT layer. Similar to classic database optimizers, it computes alternative execution plans and eventually chooses the most preferable one. To this end, the optimizer can rely on various information sources, such as samples of the input data, code annotations (possibly generated by the Sopremo layer), information from the cluster's resource manager, or runtime statistics from previous job executions.

The output of the PACT compiler is a parallel data flow program for Nephele, Stratosphere's parallel execution engine, and the third layer of the Stratosphere stack. Similar to PACT programs, Nephele data flow programs, also called Job Graphs, are also specified as DAGs with the vertices representing the individual tasks and the edges modeling the data flows between those. However, in contrast to PACT programs, Nephele Job Graphs contain a concrete execution strategy, chosen specifically for the given data sources and cluster environment. In particular, this execution strategy includes a suggested degree of parallelism for each vertex of the Job Graph, concrete instructions on data partitioning as well as hints on the co-location of vertices at runtime.

Nephele itself executes the received Job Graph on a set of worker nodes. It is responsible for allocating the required hardware resources to run the job from a resource manager, scheduling the job's individual tasks among them, monitoring their execution, managing the data flows between the tasks, and recovering tasks in the event of execution failures. Moreover, Nephele provides a set of memory and I/O services that can be accessed by the user tasks submitted. At the moment, these services are primarily used by the PACT data preparation code mentioned above (ALEXANDROV et al., 2014).

2.2.1.1 Nephele Streaming

Nephele Streaming is the research prototype that has driven the creation of the Flink Streaming project. It uses Nephele engine as an underline to develop a massively parallel event stream processing engine with latency constraints.

Nephele Streaming shares some goals with Apache Storm: It is a software framework

for massively parallel real-time computation on large clusters or clouds. Nephele Streaming jobs are continuous data flows where tasks communicate via sending records along predefined channels.

The main feature of Nephele Streaming is that users can annotate their applications (jobs) with latency constraints. A latency constraint as declaration of a non-functional application requirement. It specifies a desired upper latency bound in milliseconds for a portion of the job's dataflow graph. At runtime the engine attempts to enforce the constraints by three techniques: 1) Adaptive Output Batching: Emitted records are by default sent immediately to the receiver for low latency. Adaptive output batching flushes those buffers in a time driven fashion to enforce the constraint. The lower the constraint, the more often buffers are flushed (and vice versa); 2) Dynamic Task Chaining: The mapping of pipeline parallel tasks on the same worker process (task manager) is changed ad-hoc at runtime by the framework, depending on CPU utilization. This eliminates queues between pipeline parallel tasks; 3) Elastic Scaling: Queues are a major source of latency in event stream processing. In order to fulfill latency constraints despite variations in stream rates and computational load, Nephele Streaming autoscales data parallel tasks. This technique uses a predictive latency model based on queueing theory.

Its main differences regarding to the original Nephele engine are related to the removal of specific batch processing capabilities, such as Hadoop Distributed File System (HDFS) support and higher-level programming models (e.g. PACTs). The original Nephele Streaming currently differs from the Flink Streaming, but the main ideas behind the system are still related to Nephele Streaming (LOHRMANN; WARNEKE; KAO, 2014).

2.2.2 Apache Spark

Apache Spark relies on a main abstraction that is called Resilient Distributed Dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark is implemented in Scala programming language, a statically typed high-level pro-

gramming language for the Java Virtual Machine (JVM), and exposes a functional programming interface similar to DryadLINQ (YU et al., 2008). In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster (ZAHARIA et al., 2010).

In order to address the nowadays needs for processing data arriving in real-time, the Spark research team proposed a new programming model, Discretized Streams (D-Streams), that offers a high-level functional programming API, strong consistency, and efficient fault recovery. D-Streams support a new recovery mechanism that improves efficiency over the traditional replication and upstream backup solutions in streaming databases: parallel recovery of lost state across the cluster. They have prototyped D-Streams in an extension to the Spark cluster computing framework called Spark Streaming, which lets users seamlessly intermix streaming, batch and interactive queries.

2.2.2.1 D-Streams

The key idea behind D-Streams is to treat a streaming computation as a series of deterministic batch computations on small time intervals. Two immediate advantages of the D-Stream model are that consistency is well-defined (each record is processed atomically with the interval in which it arrives), and that the processing model is easy to unify with batch systems.

There are two key challenges in realizing the D-Stream model. The first is making the latency (interval granularity) low. Traditional batch systems like Hadoop and Dryad (ISARD et al., 2007) fall short here because they keep state on disk between jobs and take tens of seconds to run each job. Instead, to meet a target latency of several seconds, we keep intermediate state in memory. However, simply putting the state into a general-purpose in-memory storage system, such as a key-value store, would be expensive due to the cost of data replication. Instead, they build on RDDs, a storage abstraction that can rebuild lost data without replication by tracking the operations needed to recompute it.

Along with a fast execution engine (Spark) that supports tasks as small as 100 ms, they can achieve latencies as low as a second. They argue that this is sufficient for many real-world big data applications, where the timescale of events monitored (e.g., trends in a social network) is much higher. The second challenge is recovering quickly from failures. In order to do this, they use the deterministic nature of the batch operations in each interval to provide a new recovery mechanism that has not been present in previous streaming systems: parallel

recovery of a lost node's state. Each node in the cluster works to recompute part of the lost node's RDDs, resulting in faster recovery than upstream backup without the cost of replication. Parallel recovery was hard to implement in record-at-a-time systems due to the complex state maintenance protocols needed even for basic replication, but is simple with the deterministic model of D-Streams (ZAHARIA et al., 2012).

D-Stream Operators provide two types of operators to let users build streaming programs:

- Transformation operators, which produce a new D-Stream from one or more parent streams. These can be either stateless (i.e., act independently on each interval) or stateful (share data across intervals).
- Output operators, which let the program write data to external systems (e.g., save each RDD to HDFS).

D-Streams support the same stateless transformations available in typical batch frameworks, including map, reduce, groupBy, and join. They reuse all of the operators already present in Spark.

In addition, D-Streams introduce new stateful operators that work over multiple intervals. These include:

- Windowing: The window operator groups all the records from a range of past time intervals into a single RDD. As we can see in Figure 2.2, a window operator can, for example, combine RDDs at each three time units into a single windowed RDD. Window is the most general stateful operator, but it is also often inefficient, as it repeats work.
- Incremental aggregation: For the common use case of computing an aggregate value, such as a count or sum, over a sliding window, D-Streams have several variants of a reduceByWindow operator. The simplest one only takes an associative "merge" operation for combining values.
- Time-skewed joins: Users can join a stream against its own RDDs from some time in the past to compute trends.

Finally, the user calls output operators to transfer results out of D-Streams into external systems (e.g., for display on a dashboard). There are provided two such operators: 1) save, which writes each RDD in a D-Stream to a storage system, and; 2) foreach, which runs a user code snippet (any Spark code) on each RDD in a stream.

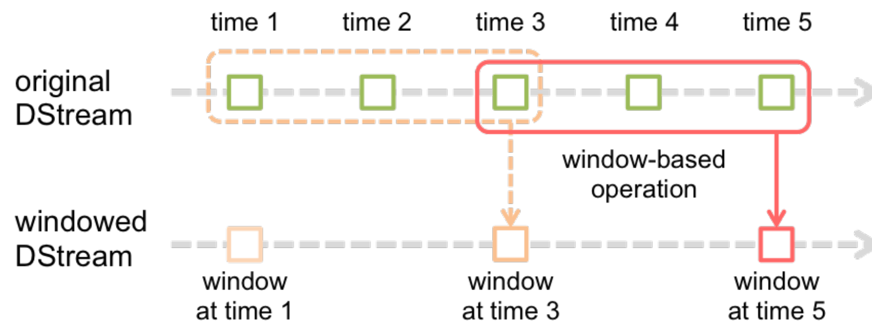

Figure 2.2 – Comparison between a simple and a windowed DStream

2.2.3 Apache Storm

Apache Storm is a free Java and Clojure programming languages based framework that supports distributed real-time computations. Clojure is a functional programming language, which is based on Lisp and can be executed on the JVM. Storm was first developed by Nathan Marz when working at BackType, which was later acquired by Twitter in 2011. In 2014, Storm graduated to an Apache top-level project and became an open source framework. Storm supports the development of applications, which are processing large amount of data in real-time. In contrast to the batch processing framework Apache Hadoop, Storm has become an important platform for real-time processing (JAIN; NALYA, 2014).

Storm provides four different levels of parallelism in its architecture. The following levels are used to run a topology in Storm (STORM, 2015):

- Supervisor (Slave)
- Worker (JVM)
- Executor (Thread)
- Task

First of all Storm supports the distribution of work among multiple slave nodes. Each one of these worker nodes runs a single Supervisor daemon. A Supervisor executes one or multiple Worker processes within a dedicated JVM instance. At the next level each Worker is able to use multiple Executor threads within its JVM process. And finally each Executor thread executes one or more Tasks serially. By default Storm runs one Task per each Executor thread but an Executer might also execute multiple Tasks serially. The feature of having multiple Tasks that are executed in a serial fashion within an Executor, allows to test the parallelism of the system.

2.2.3.1 Spouts

The terminology Spout defines the source of tuples in Storm. A Spout reads data from an external source and provides it to the Storm topology. For example, a Spout might listen to a Twitter stream and emits this data into a Storm stream. Storm supports reliable and unreliable Spouts. If a tuple fails during the execution within a topology, then a reliable Spout would replay it. In contrast, an unreliable Spout forgets a tuple as soon as it is emitted.

2.2.3.2 Bolts

In Storm the actual processing of a task is executed by Bolts. A Bolt takes the tuples of one or multiple input streams, processes and emits them to one or multiple output streams. A Bolt can transform, filter, aggregate, join or execute other functions on tuples.

2.2.3.3 Topologies

Storm uses its own terminology to describe the workflow. A topology in Storm defines a DAG, which represents the structure and logic of a Storm real-time application. Each node in this DAG processes and forwards tuples in parallel. A topology typically consists of the two major components Spouts and Bolts. The connection between Spouts and Bolts is called stream. A stream represents an infinite sequence of tuples, which can be processed in parallel. Storm supports different stream groupings, which specify how streams should be partitioned among different Bolts.

2.3 Cloud Computing

Cloud computing is a new paradigm of computing. It was developed with the combination and evolution of distributed computing and virtualization, with strong contributions from grid and parallel computing (BUYYA et al., 2009). There are many efforts to provide a definition of cloud computing, such as the work of Grid Computing and Distributed Systems Laboratory (BUYYA et al., 2009) and the initiative from Berkeley University (BUYYA et al., 2009). In 2011, NIST (National Institute of Standards and Technology) has published its definition that consolidates several studies and became widely adopted.

Cloud computing has two main actors that are defined as the user and the provider. The user is defined as the consumer and can be a single person or an entire organization. The

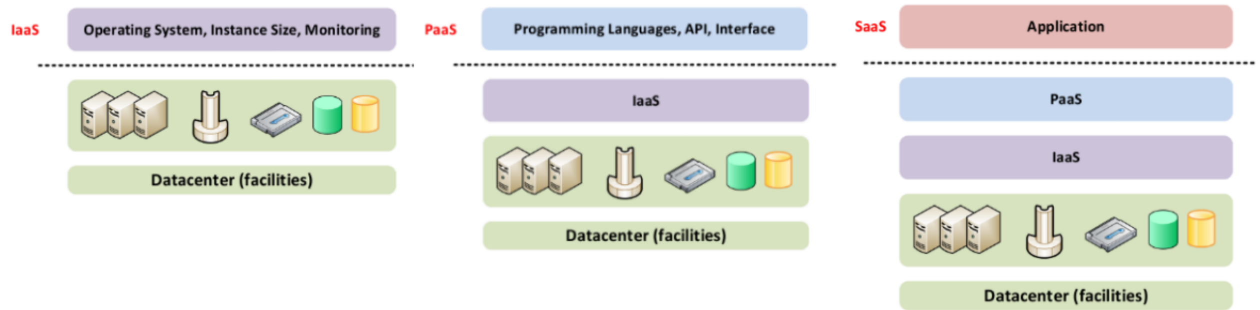

Figure 2.3 – Cloud computing service models stack and their relationships

provider is an organization that provides the services to the user. According to NIST definition (MELL; GRANCE, 2011), cloud computing is a model that conveniently provides on-demand network access to a shared pool of configurable computing resources that can be provisioned and released quickly without large management efforts and interaction with the service provider. This model definition is composed of five essential characteristics, three service models and four implementation models, which will be discussed in this section.

A cloud computing service needs to present the following characteristics to be considered adherent to the NIST definition: On-demand service; Broad network access; Resource Pooling; Rapid Elasticity and Measured Service.

As it is shown on Figure 2.3, the services provided by a cloud provider are categorized into three service models (BADGER et al.,): Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Some providers denominates other service models such as Database as a Service and Framework as a Service. Those models are commonly defined as Everything as a Service (XaaS), but it is possible to classify these models into one of the three defined by NIST.

The NIST cloud definition lists four implementation models: private, community, public and hybrid. Each one of these models has its particularities regarding to aspects such as network dependency, security, quantity of resources, among others. In the public model the user rents the resources from a provider, the private and community models can be used in two configurations, outsourced, where the user rents exclusive resources from a provider, and on-site, where the resources are owned by the user. The hybrid model is a combination between any of the other three models.

Generally we can say that a cloud service is controlled by a cloud platform, which is responsible for all procedures related to the service. A cloud platform is a very abstract term. In a practical approach it is made up of several components that are responsible for its operation.

The base of any cloud service is the hardware. By hardware we mean servers, storage and networking equipment. In a public cloud provider this hardware is maintained in a datacenter facility. The hardware is normally grouped into a container, that holds thousands of physical machines and storages interconnected with a high-speed network. This strategy is used by providers to optimize energy consumption. The containers are switched on and off according to user demand, each container has its own cooling system that is only used when the container is turned on.

In a cloud service normally the machines and storages offered to the users are a virtualization of real hardware. The component that performs this virtualization is the hypervisor. Basically the hypervisor controls the underlying hardware and provides VMs to the upper layers of the cloud platform. The hardware is a group of machines composed by different sizes and configurations and also a different type of processor architectures and operating systems. The purposes of the use of a hypervisor are to take care of all this heterogeneity and provide a standard interface to the cloud platform. Example of hypervisors that can be used are: Xen [1], KVM[2], Virtualbox[3], Hyper-V[4] and VMware[5]. The hypervisor layer delivers basically virtual machines to the other layers. The Virtual Machines (VMs) are abstractions of real hardware and can be used for general purposes.

The resource manager is responsible for providing the interface between the resources and the cloud platform. It controls the VM allocation and deallocation, also the VM migration between different servers is controlled in this layer. The security of resources is defined in this layer too. For example, the policies of VMs interconnection are defined by the resource manager. This is the main component regarding the energy consumption. Because this is the layer that decides when to power up a new machine, or an entire container, according to the demand. The resource manager has the responsibility to perform the VM consolidation to be possible turn off a server.

The main part of a cloud service is the cloud manager. This component performs the entire administrative tasks of the cloud platform. The user authentication is performed by the manager, that has a complete user record system controlling each user rights. The consumption of resources that each user makes of the system and the pricing mode are also controlled by this component. This control is used for billing purposes, in this way it can be stated that the pay-per-use charging model is implemented here. The instance sizes of VMs and the standard

---

[1]<http://www.openstack.org/>
[2]<http://www.linux-kvm.org/>
[3]<https://www.virtualbox.org/>
[4]<http://www.microsoft.com/hyper-v-server>
[5]<http://www.vmware.com/>

operating systems are also defined in the cloud manager. All the capabilities of customization of the images, size changing, multiple creation, user access security, among others are controlled in this layer too. The user reports are generated and provided here. Examples of cloud managers are: OpenStack [6], Eucalyptus [7], OpenNebula [8] and Nimbus [9]. Several providers implement their own proprietary cloud managers (ROLOFF, 2013).

The user interface is the front-end layer of the cloud platform. All the user-provider interaction is made through this layer. The user interface is normally a web page or a smart phone application, from the point of view of the user this layer is the entire cloud platform. Commonly the cloud managers provide a standard user interface, but each provider customizes it.

Microsoft started its initiative in cloud computing with the release of Windows Azure[10] in 2008, which initially was a PaaS to develop and run applications written in the programming languages supported by the .NET framework. At these days, the company owns products that covers all types of service models. Online Services[11] is a set of products that are provided as SaaS, while Windows Azure provides both PaaS and IaaS.

Windows Azure PaaS is a platform developed to provide to the user, the capability to develop and deploy a complete application into Microsoft's infrastructure. To have access to this services, the user needs to develop his application following the provided framework. The Azure framework has support to a wide range of programming languages, including all .NET languages, Python, Java and PHP. A generic framework is provided in which the user can develop in any programming language that is supported by Windows Operating System (OS).

Windows Azure IaaS is a service developed to provide to the user access to VMs running in Microsoft's infrastructure. The user has a set of base images of Windows and Linux OS, but other images can be created using Hyper-V. The user can also configure an image directly into the Azure and capture it to use locally or to deploy to another provider that supports Hyper-V.

In this work, we use Microsoft Azure IaaS extensively, as the basis for our deployment and evaluation. It completely fulfills our needs for a stable and flexible large scale platform to deploy VMs, as well as provides several tools to partially automate our deployments of VMs using the Azure Command Line Interface (CLI) [12].

---

[6]<http://www.openstack.org/>
[7]<http://www.eucalyptus.com/>
[8]<http://opennebula.org/>
[9]<http://www.nimbusproject.org/>
[10]<http://www.windowsazure.com/>
[11]<http://www.microsoftonline.com/>
[12]<https://azure.microsoft.com/pt-br/documentation/articles/xplat-cli/>

2.4 Big Data

Big data is an abstract concept. Apart from masses of data, it also has some other features, which determine the difference between itself and "massive data" or "very big data." At present, although the importance of big data has been generally recognized, people still have different opinions on its definition. In general, big data shall mean the datasets that could not be perceived, acquired, managed, and processed by traditional information technology and software/hardware tools within a tolerable time. Because of different concerns, scientific and technological enterprises, research scholars, data analysts, and technical practitioners have different definitions of big data. The following definitions may help us have a better understanding on the profound social, economic, and technological connotations of big data.

In 2010, Apache Hadoop defined big data as "datasets which could not be captured, managed, and processed by general computers within an acceptable scope." On the basis of this definition, in May 2011, McKinsey, a global consulting agency announced big data as the next frontier for innovation, competition, and productivity. Big data shall mean such datasets which could not be acquired, stored, and managed by classic database software. This definition includes two connotations: First, datasets volume that conform to the standard of big data are changing, and may grow over time or with technological advances; Second, datasets' volumes that conform to the standard of big data in different applications differ from each other. At present, big data generally ranges from several TB to several PB (MANYIKA et al., 2011).

From the definition by McKinsey, it can be seen that the volume of a dataset is not the only criterion for big data. The increasingly growing data scale and its management that could not be handled by traditional database technologies are the next two key features.

As a matter of fact, big data has been defined as early as 2001. Doug Laney, an analyst of META (presently Gartner) defined challenges and opportunities brought about by increased data with a "3Vs" model, i.e., the increase of Volume, Velocity, and Variety, in a research report (LANEY, 2001). Although such a model was not originally used to define big data, Gartner and many other enterprises, including IBM (ZIKOPOULOS; EATON et al., 2011) and some research departments of Microsoft (MEIJER, 2011) still used the "3Vs" model to describe big data within the following ten years (BEYER, 2011). In the "3Vs" model, Volume means, with the generation and collection of masses of data, data scale becomes increasingly big; Velocity means the timeliness of big data, specifically, data collection and analysis, etc. must be rapidly and timely conducted, so as to fully utilize the commercial value of big data; Variety indicates the various types of data, which include semi-structured and unstructured data such as audio,

video, webpage, and text, as well as traditional structured data. However, others have different opinions, including International Data Corporation, one of the most influential leaders in big data and its research fields. In 2011, an International Data Corporation report defined big data as "big data technologies describe a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling the high-velocity capture, discovery, and/or analysis." (GANTZ; REINSEL, 2011). With this definition, characteristics of big data may be summarized as four Vs, that are: Volume (great volume); Variety (various modalities); Velocity (rapid generation) and Value (huge value but very low density). Such 4Vs definition was widely recognized since it highlights the meaning and necessity of big data, i.e., exploring the huge hidden values. This definition indicates the most critical problem in big data, which is how to discover values from datasets with an enormous scale, various types, and rapid generation.

In addition, NIST defines big data as "Big data shall mean the data of which the data volume, acquisition speed, or data representation limits the capacity of using traditional relational methods to conduct effective analysis or the data which may be effectively processed with important horizontal zoom technologies", which focuses on the technological aspect of big data. It indicates that efficient methods or technologies need to be developed and used to analyze and process big data.

There have been considerable discussions from both industry and academia on the definition of big data (CHEN; MAO; LIU, 2014). In addition to developing a proper definition, the big data research should also focus on how to extract its value, how to use data, and how to transform "a bunch of data" into "big data." (SAGIROGLU; SINANC, 2013b).

2.5 Smart Grid

For 100 years, there has been no change in the basic structure of the electrical power grid. Experiences have shown that the hierarchical, centrally controlled grid of the 20th Century is ill-suited to the needs of the 21st Century. To address the challenges of the existing power grid, the new concept of smart grid has emerged. The smart grid can be considered as a modern electric power grid infrastructure for enhanced efficiency and reliability through automated control, high-power converters, modern communications infrastructure, sensing and metering technologies, and modern energy management techniques based on the optimization of demand, energy and network availability, and so on. While current power systems are based on a solid information and communication infrastructure, the new smart grid needs a different

and much more complex one, as its dimension is much larger (GÜNGÖR et al., 2011).

According to the U.S. Department of Energy report, the demand and consumption for electricity in the U.S. have increased by 2.5% annually over the last 20 years (GUNGOR; LU; HANCKE, 2010). Today's electric power distribution network is very complex and ill-suited to the needs of the 21st Century. Among the deficiencies are a lack of automated analysis, poor visibility, mechanical switches causing slow response times, lack of situational awareness, etc (ENERGY, 2015). These have contributed to the blackouts happening over the past 40 years. Some additional inhibiting factors are the growing population and demand for energy, the global climate change, equipment failures, energy storage problems, the capacity limitations of electricity generation, one-way communication, decrease in fossil fuels, and resilience problems (EROL-KANTARCI; MOUFTAH, 2011). Also, the greenhouse gas emissions on Earth have been a significant threat that is caused by the electricity and transportation industries (SABER; VENAYAGAMOORTHY, 2011). Consequently, a new grid infrastructure is urgently needed to address these challenges.

2.5.1 Advanced Metering Infrastructure

The Advanced Metering Infrastructure (AMI) is regarded as the most fundamental and crucial part of smart grid. It is designed to read, measure, and analyse the energy consumption data of consumers through smart meters in order to allow for dynamic and automatic electricity pricing.

AMI requires a two way communication and spans through all the network components of smart grid from the private networks and Field Area Networks to Wide Area Networks. AMI goes beyond automatic meter reading scenarios which according to IEC 61968-9 — a series of standards under development that will define standards for information exchanges between electrical distribution systems — only have to do with meter reading, meter events, grid events and alarms. AMI will include customer price signals, load management information, power support for prepaid services, Home Energy Management Systems and Demand Response. It can also be used to monitor power quality, electricity produced or stored by Distributed Energy Resources units as well as interconnect Intelligent Electronic Devices (ANCILLOTTI; BRUNO; CONTI, 2013).

In addition, AMI is also expected to support customer switch between suppliers and help in detection and reducing electricity theft. Electricity theft has plagued many utilities companies especially in developing countries. To address these issues, authors in (ANAS et

al., 2012) have reviewed electricity theft and reduction issues using security and efficient AMIs (TSADO; LUND; GAMAGE, 2015).

2.5.2 Demand Side Management

Demand Side Management (DSM) is the action that influences the quantity or pattern of energy consumption by end users. These actions may include targeting reduction of peak demand by end users during periods when energy supply systems are constrained. Energy peak management does not necessarily decrease the amount of total energy consumption, but it will reduce the need for investments on power generation sources or spinning reserves at peak periods (WANG; XU; KHANNA, 2011) (DAVITO; TAI; UHLANER, 2010). DSM includes the following:

- Demand Response enabling the utility operator to optimally balance power generation and consumption either by offering dynamic pricing programs or by implementing various load control programs.
- Load Management through dynamic pricing which helps to reduce energy consumption during peak hours by encouraging customers to limit energy usage or shifting demand to other periods. Existing dynamic pricing programs include: Time-of-use, Real-Time Pricing, Critical Peak timing and Peak time Rebates.
- Conservation of energy through load control program which involve performing remote load control programs where communicating networks are used to control usage of appliances remotely to use less energy across many hours (TSADO; LUND; GAMAGE, 2015).

2.5.3 Consumption Forecasting

The term forecasting is frequently confused with the terms prediction and predictive analytics. A prediction in the general sense involves an imagination of an oracle which can reason about the past based on some experience and which on this basis is able to look into future to predict a certain event.

Prediction and predictive analytics in the scientific sense means predicting a behavior of someone or a trend characterized with a probability and based on statistical data analysis and the current evolution. In contrast, forecasting refers to predicting an (aggregated) value, or an

occurrence of an event at certain time point, based on historical data analysis, the current state and sometimes on predictive analytics (ANALYTICSWORLD, 2015).

Electricity load forecasts provide a prediction of an amount of electricity consumed at a certain point of time. The purpose of electricity load forecasting is in most cases an efficient economic and quality planning. Good forecasts ensure economic profitability of the service and safety of the network.

Energy consumption forecasts can be performed on different levels of time interval resolution. The range of the forecasts generally depends on the available reliable data and the goal of the forecast. Usually, the following three terms for forecasting interval are used: short term, medium term and long term forecasts (ALFARES; NAZEERUDDIN, 2002).

- Short Term Load Forecasting (STLF) means to give forecasts for the next minutes up to one day on minutes or hourly basis. Such forecasts are required for the scheduling, capacity planning and control of power systems.
- Medium Term Load Forecasting (MTLF) are required for planning and operation of power systems. Such forecasts can be provided from one day to months ahead on hourly or days basis.
- Long Term Load Forecasting (LTLF), in contrast to short and medium term forecasting which support operational decisions, has the aim to support strategic decisions, more than a year ahead.

Metrics to measure the quality of load forecasting can be subdivided into two main categories: Measuring the forecast accuracy and measuring the processing delay (latency).

2.6 Chapter Remarks

Throughout this chapter, we discussed several topics that are important as the underlining background of this work. We started with IoT in Section 2.1, which is an important research topic, represented in this work by the sensors networks that collect data from smart grids. In Section 2.2 we presented the field of distributed event stream processing and discussed several frameworks that represent their state-of-the-art. From the available options, we decided to choose Apache Spark Streaming for our implementation, from reasons we discuss on the implementation section of Chapter 3. In Section 2.3 we discuss the importance of cloud computing and present Microsoft Azure, which will be our IaaS provider for the evaluation of our platform, for reasons we also discuss on the implementation section of the Chapter 3.

Finally, in Section 2.5 we present the field of smart grids and the subfields of AMI, DSM and consumption forecasting. Those areas are important for this work because it is proposed an architecture for consumption forecasting, based on AMI data and for DSM, which have its design and implementation shown along the Chapter 3.

3 DESIGN AND IMPLEMENTATION

In this chapter, we explain how we designed our architecture, describe important project decisions and finally show how it was implemented. Section 3.1 describes architectural patterns that we found in the state-of-the-art, how we planned our own architecture and what thoughts lead us to design this specific architecture. The final design of our architecture, which we named *Cyclic Architecture*, can be found in Subsection 3.1.4.

Section 3.2 explain the dataset used to design and test our application, and what are the implications and constraints by using it. Section 3.3 describe the forecast method used, why it was chosen and how it fits our proposed architecture. Section 3.4 describe the challenges needed to overcome the difficulties imposed by data input in parallel and how we overcome those limitations Section 3.5 focuses on the implementation specifics, tools and framework used to turn the proposed architecture into a real application.

3.1 Architecture

The proposed 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 (SAGIROGLU; SINANC, 2013b). We found that, after the large success of MapReduce architecture and the subsequent understanding of its faults — mainly due to the difficulties of adapting a batch-oriented architecture for solving online and near real-time problems — research goals went from adapting this architectural design for building new ones.

3.1.1 Lambda Architecture

In this architectural design, as we can see in Figure 3.1, input data is sent to both an offline and an online processing system. 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 in the service layer, reconciling the results produced by the offline and online processing systems (MARZ; WARREN, 2015).

The use of this pattern allows organizations to adapt their current infrastructures to support near real-time applications (FERNANDEZ et al., 2015). This comes at a cost, though: developers must write, debug, and maintain the same processing code for both the batch and stream layers, and the Lambda architecture increases the hardware footprint.

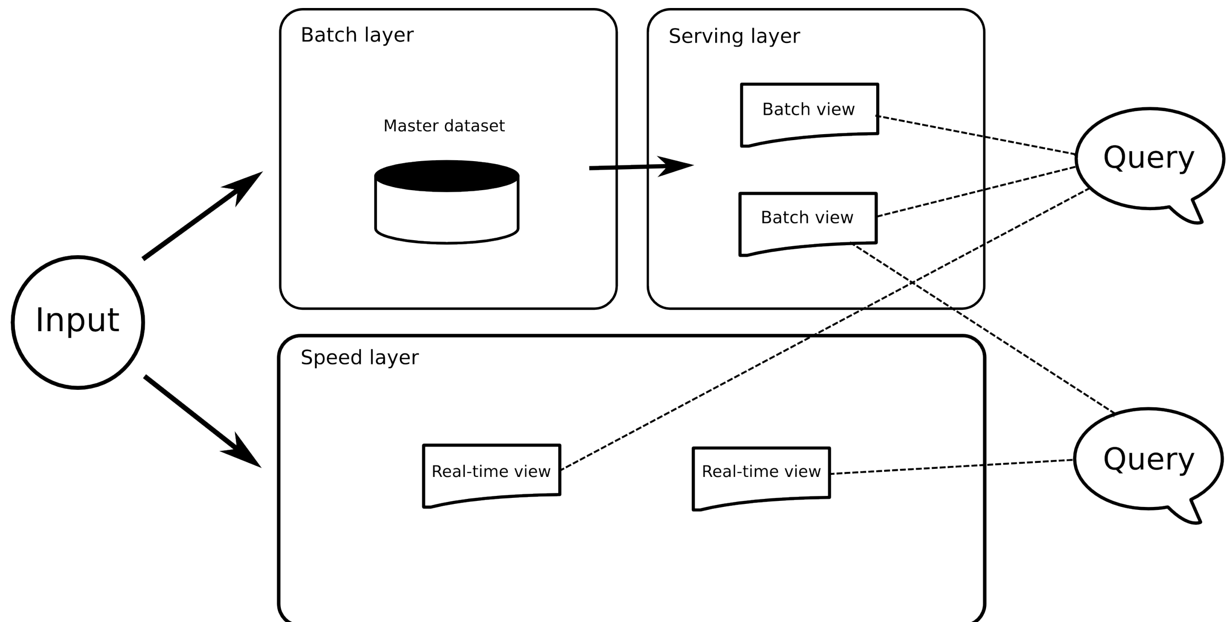

Figure 3.1 – Lambda Architecture

3.1.2 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 approach only requires a single processing path, as we can see Figure 3.2. However, its architecture has a higher storage footprint, and applications access stale data while the system is reprocessing data.

In summary, implementing the above architectural patterns in current MapReduce/GFS-based data integration stacks introduces a range of problems, including an increased hardware footprint, data and processing duplication and more complex management. To overcome these issues and provide more flexibility for the new requirements of near real-time applications, they describe a new data integration stack, that acts as a more efficient substrate for back-end systems by providing low-latency data access by default (KREPS, 2014).

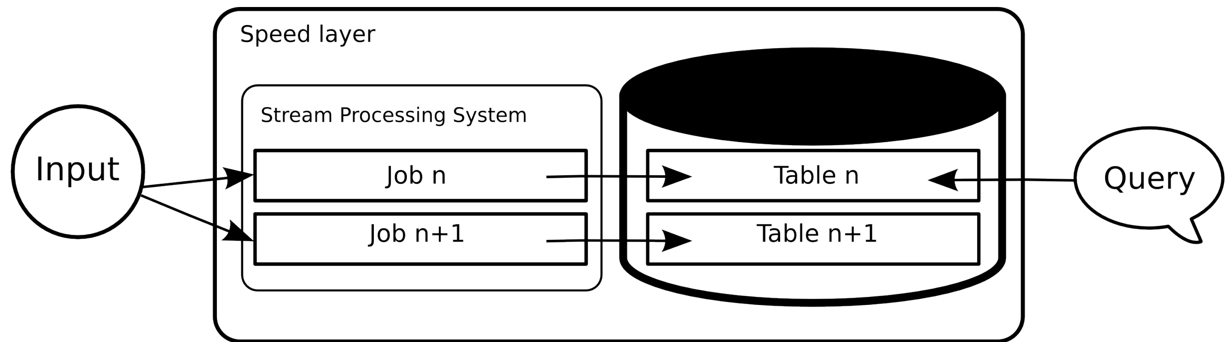

Figure 3.2 – Kappa Architecture

3.1.3 Liquid Architecture

Liquid is a data integration stack with two independent, yet cooperating layers, as it can be seen in the Figure 3.3. A processing layer 1) executes jobs similar to extract-transform-load process for different back-end systems according to a stateful event stream processing model (FERNANDEZ et al., 2013); 2) guarantees service levels through resource isolation; 3) provides low latency results; and 4) enables incremental data processing. A messaging layer supports the processing layer. It 1) stores high-volume data with high availability; and 2) offers rewindability, i.e. the ability to access data through metadata annotations.

The two layers communicate by writing and reading data to and from two types of feeds, stored in the messaging layer: source-of-truth feeds represent primary data, i.e. data that is not generated within the system; and derived data feeds contain results from processed source-of-truth feeds or other derived feeds. Derived feeds contain lineage information, i.e. annotations about how the data was computed, which are stored by the messaging layer. The processing layer must be able to access data according to different annotations, e.g. by timestamp. It also produces such annotations when writing data to the messaging layer.

Back-end systems read data from the input feeds, after Liquid has pre-processed them to meet application-specific requirements. These jobs are executed by the processing layer, which reads data from input feeds and outputs processed data to new output feeds. The division into two layers is an important design decision. By keeping both layers separated, producers and consumers can be decoupled completely, i.e. a job at the processing layer can consume from a feed more slowly than the rate at which another job published the data without affecting each other's performance. In addition, the separation improves the operational characteristics of the data integration stack in a large organization, particularly when it is developed and operated by

independent teams: separation of concerns allows for management flexibility, and each layer can evolve without affecting the other (FERNANDEZ et al., 2015).

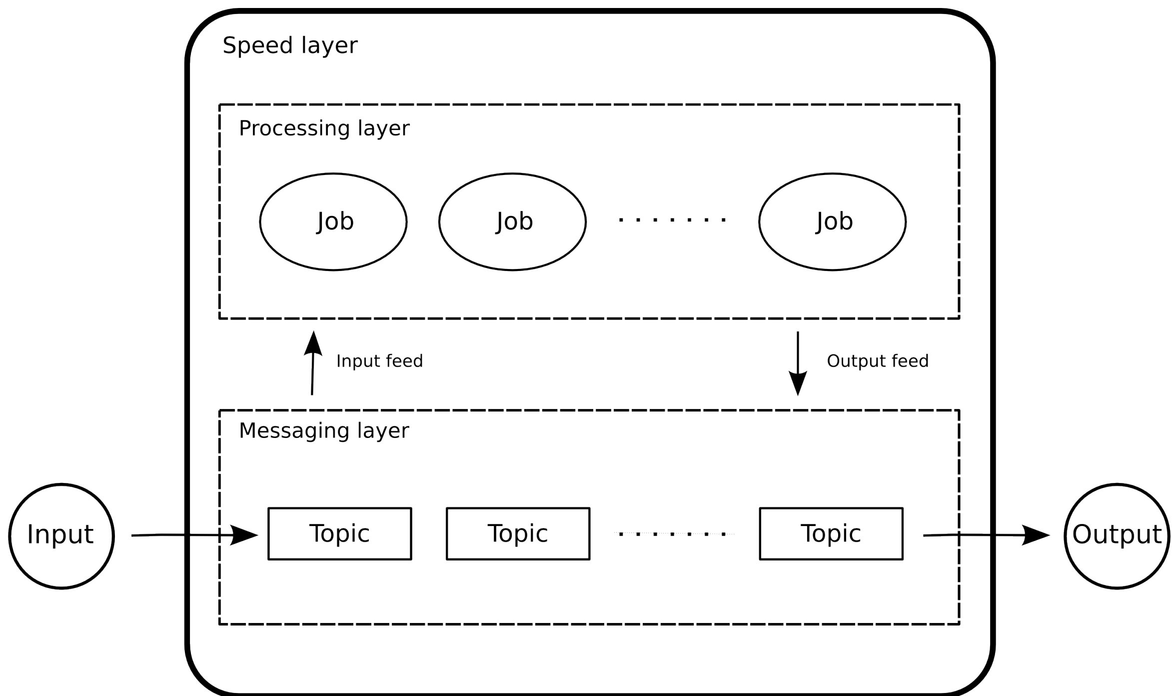

Figure 3.3 – Liquid Architecture

3.1.4 Cyclic Architecture

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. Liquid architecture, on the other side, provides a solution that dissociates processing from the flow of data completely. That dissociation from the flow of data can be required for some specific profiles of data, but it does not cover a broadly amount of cases where raw data has not value to the system at all.

We then propose the Cyclic architecture, which can be seen on Figure 3.4, that is a hybrid solution mixing architectural solutions from Kappa architecture and Liquid architecture. Our solution, like Kappa architecture, is based on the idea of continuous process over the data, but doesn't relies on such strict (and high) storage footprint and a single processing path. Likewise, our solution relies on some ideas behind liquid architecture, mainly on the idea that we can split

the design in two layers (messaging layer and processing layer). On the other hand, we do not separate the output of data from the processing path, relying on the processing layer itself to provide answers (which can also induce computations) to output systems.

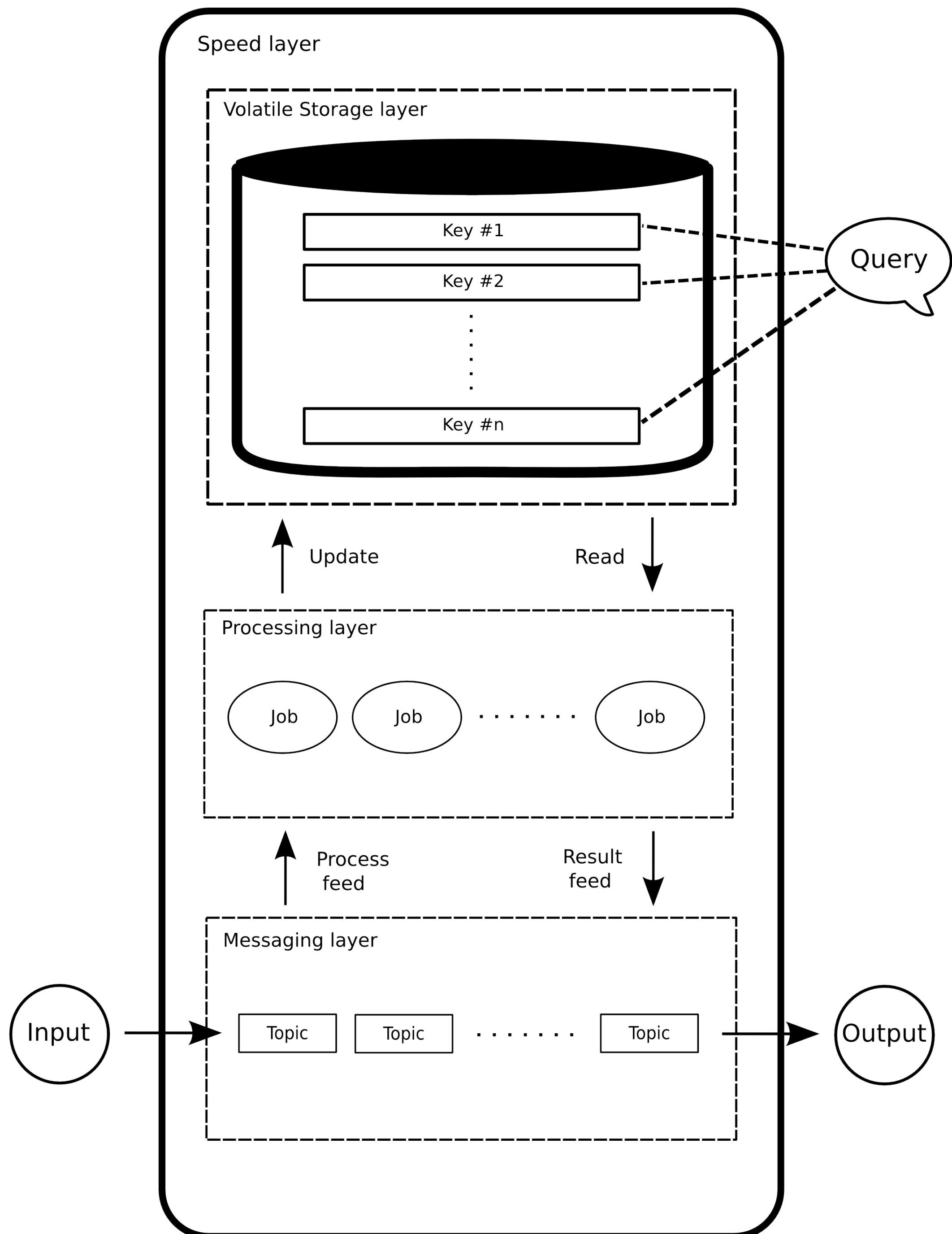

Figure 3.4 – An overview of the proposed Cyclic Architecture

3.2 Data

The dataset used to evaluate the platform is originated from the 8th ACM International Conference on Distributed Event-Based Systems (DEBS). 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. 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.

For the purpose of the challenge a number of smart plugs has 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 3.1 we describe how is the layout of each one of the measurements into the dataset. It is used 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.

The synthesized data file contains over 4055 Millions of measurements for 2125 plugs distributed across 40 houses, for a total amount of 136 GB. Generated measurements cover a period of one month with the first timestamp equal to 1377986401 (Sept. 1st, 2013, 00:00:00) and the last timestamp equal to 1380578399 (Sept. 30th, 2013, 23:59:59). All events in the data file are sorted by the timestamp value. Events with the same timestamp are ordered randomly with respect to each other. For our tests, we used a subset of this file, in order to decrease the total processing time and being able to make a greater numbers 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).

Table 3.1 – Dataset: Schema and overview

Name	Description	Type	Unit
id	Unique identifier	32 bit unsigned integer	Number
timestamp	Timestamp of measurement	32 bit unsigned integer	Number of seconds (since January 1, 1970, 00:00:00 GMT)
value	Measurement value, with unit type dependent of the property field value	32 bit floating point	kWh or Watt
property	Type of measurement	32 bit unsigned integer	0 or 1
plug_id	Identifier of the smart plug	32 bit unsigned integer	Number
household_id	Identifier of where the plug is located	32 bit unsigned integer	Number
house_id	Identifier of the house where the household with the plug is located	32 bit unsigned integer	Number

3.3 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 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 (KYRIAKIDES; POLYCARPOU, 2007) 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 (ALI, 2013) 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 (ZIEKOW; JERZAK, 2014), that is schematically described in Equation (3.1). 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 (BYLANDER; ROSEN, 1997).

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.

$$L(s_{i+2}) = \frac{avgL(s_i) + median(avgL(s_j))}{2} \tag{3.1}$$

In the Equation (3.1), $avgL(s_i)$ 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:

$$s_j = s_{i+2-n*k} \tag{3.2}$$

In the Equation (3.2), 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.

3.4 Distributed Data Input

We have decided to implement the core of our architecture using Spark Streaming. Our decision was based on evidence that we found that, while Apache Storm could provide better

latencies, it was difficult to achieve comparable throughputs in comparison to Spark Streaming (ZAHARIA et al., 2012) (TOSHNIWAL et al., 2014) (CARVALHO; NAVAUX, 2014). We also found that there are many issues related to the platform stability over time (KULKARNI et al., 2015). Similarly, when we started our project implementation using Flink Streaming (our first implementation approach) we have found some issues. We have found that their windowing system — which is important due to the importance of time frames in our architecture — was still unstable and under development phase, which have made us decide to shift our implementation to use Spark Streaming as the underlining event stream processing framework.

In order to being able to distribute the data input to all processing machines, we needed to research ways of parallelizing not only the processing, but also the data input, otherwise we could have been heavily bounded by the data throughput into the event stream processing system. Several options exist for this task, 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]. All of these options were considered, but we ended up using Apache Kafka[6], mainly due to its proved capacity of supporting large scale and high throughput systems, with strong fault tolerance and rebalancing algorithms.

We do agree that low memory footprint protocols should be used for data collection. However, we found that most protocols, such as MQTT queues, can be easily connected into Apache Kafka — and sometimes provide their own implementations of the Kafka API — which can then use these topics to distribute the data input to processing nodes.

Apache Kafka is a scalable publish-subscribe messaging system with its core architecture as a distributed commit log. It was originally built at LinkedIn as its centralized event pipelining platform for online data integration tasks. Over the past years developing and operating Kafka, LinkedIn extended its log-structured architecture as a replicated logging backbone for much wider application scopes in the distributed environment (WANG et al., 2015).

Log processing has become a critical component of the data pipeline for consumer internet companies, and is through Kafka that LinkedIn is able to collect and deliver high volumes of log data with low latency. The system incorporates ideas from existing log aggregators and messaging systems, and is suitable for both offline and online message consumption. Experimental results show that Kafka has superior performance when compared to two popular messaging

[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>

systems. LinkedIn have been using Kafka in production for some time and uses it to process hundreds of gigabytes of new data each day.

Kafka organizes messages as a partitioned write-ahead commit log on persistent storage and provides a pull-based messaging abstraction to allow both real-time subscribers such as online services and offline subscribers such as Hadoop and data warehouse to read these messages at arbitrary pace. Since Oct. 2012, Kafka has become a top-level Apache open source software and be widely adopted outside LinkedIn as well [7].

The Kafka functionality addresses not only data ingestion and distributed log processing, but also the renewed interest in using log-centric architectures to build distributed systems that provides efficient durability and availability (BALAKRISHNAN et al., 2013) (LIN et al., 2008) (OUSTERHOUT et al., 2011). In its approach, a collection of distributed servers can maintain a consistent system state via a replicated log that records state changes in sequential order. When some of the servers fail and come back, their states can be deterministically reconstructed by replaying this log upon recovery.

3.5 Implementation

3.5.1 Architectural Implementation

Once the architecture was designed, we worked to materialize it through the implementation flow described in the Figure 3.5. The implementation consists of a series of layers, each one represented by the framework or tool responsible for its implementation.

In the first layer, as our *Messaging Layer*, we have used Apache Kafka, the distributed message framework, as a way to provide large scale input and output with high throughput and fault tolerance capabilities.

The next layer is the *Processing Layer*, represented here by the chosen Streaming Processing framework Spark Streaming. As we discussed in the last chapter, it provides high processing throughputs, exactly-once processing semantics, and all of it through a simple programming API. We can do the processing of sets of tuples itself in this layer, and then store the results on the volatile store or put tuples back into another Kafka topics to be processed by other connected platforms.

Finally, we have the *Volatile Layer* implemented using a Redis key-value store in-memory database. Using Redis, we can do simple and fast distributed I/O and, since we do

[7] <http://kafka.apache.org>

not rely on complex queries and do not need to store data permanently, the model fits well to the architectural design needs.

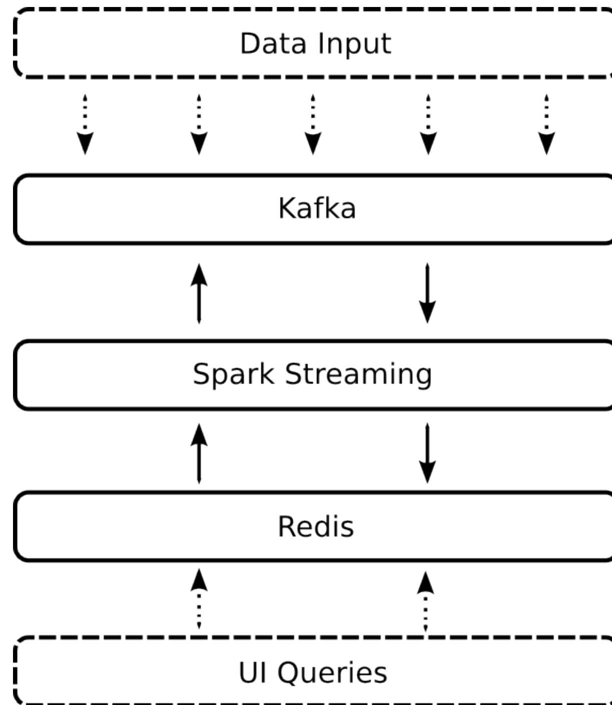

Figure 3.5 – An overview of the stack used to implement the Cyclic Architecture

3.5.2 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 the processing is done using Spark Streaming processing, which we can describe as the processing flow in the Figure 3.6.

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 (there are known approaches for recover load measurements from work measurements, which are not under the scope of this work and are discussed in Chapter 5). *Aggregate Windowing* is the step

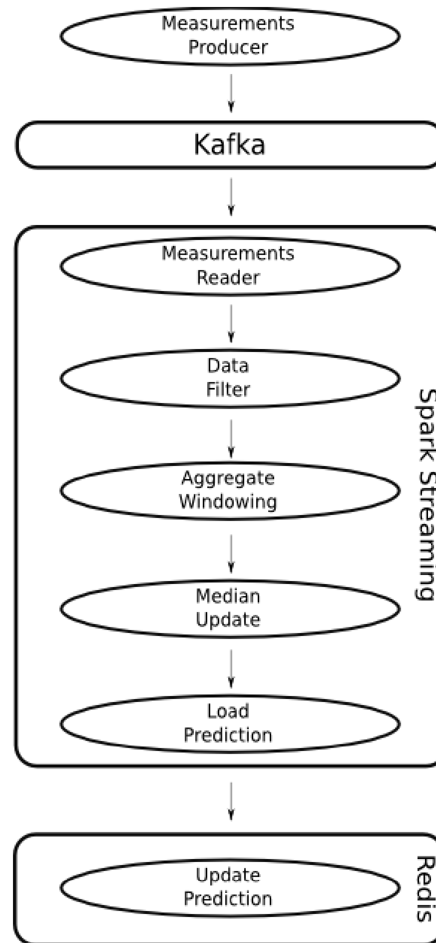

Figure 3.6 – An overview of the data processing flow

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.

4 EVALUATION

In this chapter, we explain the approach we have used to evaluate our platform and what results were found on these tests. Section 4.1 describes the platform used as the basis for running our tests. Section 4.2 explores the system regarding to the achievable latency, measuring the end-to-end time taken by events to traverse the system. Section 4.3 does a series of tests to evaluate the throughput of the platform, and better understand how the system behaves when the number of nodes increases.

4.1 Platform

In order to evaluate the system, we need a platform able to execute our tests. As we have previously discussed in the Chapter 2, cloud computing platforms presents features desirable to host these kinds of applications, and then we decided to use the IaaS platform provided by Microsoft Azure. The platform was chosen not only because it is one of the largest cloud computing platforms that provide IaaS capabilities, but also because it was available at our university free of utilization costs, through a research project partnership. The platform built using Microsoft Azure to host our application was configured using the settings described in Table 4.1.

Table 4.1 – 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

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.

4.2 Latency

It is desirable for a smart grid system to provide the smallest possible latencies, due to the discussed impact in monetary costs as well as for the environment. Nevertheless, when dealing with distributed systems, it is impossible to get rid completely of latencies, and often we need to decide between larger throughputs or smaller latencies. Because of these limitations, we tried to balance acceptable latencies (given the application needs) with the highest throughput possible, in order to accept the highest possible amount of clients.

The analysis of latency consists of the measurement of the time taken from an event arrival — in our case, an energy measurement — to its arrival into the end of the processing and delivering to the desired location. In our case, due to the predominant time taken by the processing step, time taken from event development to data pipeline, and from data pipeline to storage, is negligible. The focus of the measurement then goes to the processing step, where we measure the time taken by the processing of an event into the application pipeline of Spark Streaming.

As we can see in Figure 4.1, when batch sizes are large the system has enough time to schedule and process the DAG of the application pipeline, not incurring into schedule delays or processing pressure due to time constraints. This way, the system is able to maintain itself below the time limit, which means below the value of the batch size. In Figure 4.2, we can observe that even when there are scheduling spikes, the system is able to recover itself and compensate the processing time spikes in the long run.

However, when batches are too small, the system is not capable of providing the low latencies — in Spark Streaming, latencies are directly bound by the configured batch sizes — and latency spikes. To overcome that, it is needed to decrease the input rate (to an amount which the system is able to process) or to increase the batch size (which implies into greater latencies).

To analyse the system behavior when the batch sizes are small, we did a second set of tests using batch sizes from 5 to 15 seconds. In the Figure 4.3, we can see that to its batch sizes, the system is not able to handle the amount of data as it is received — an overall of 45K messages per second —, and the latencies are now greater then the batch sizes, which did not

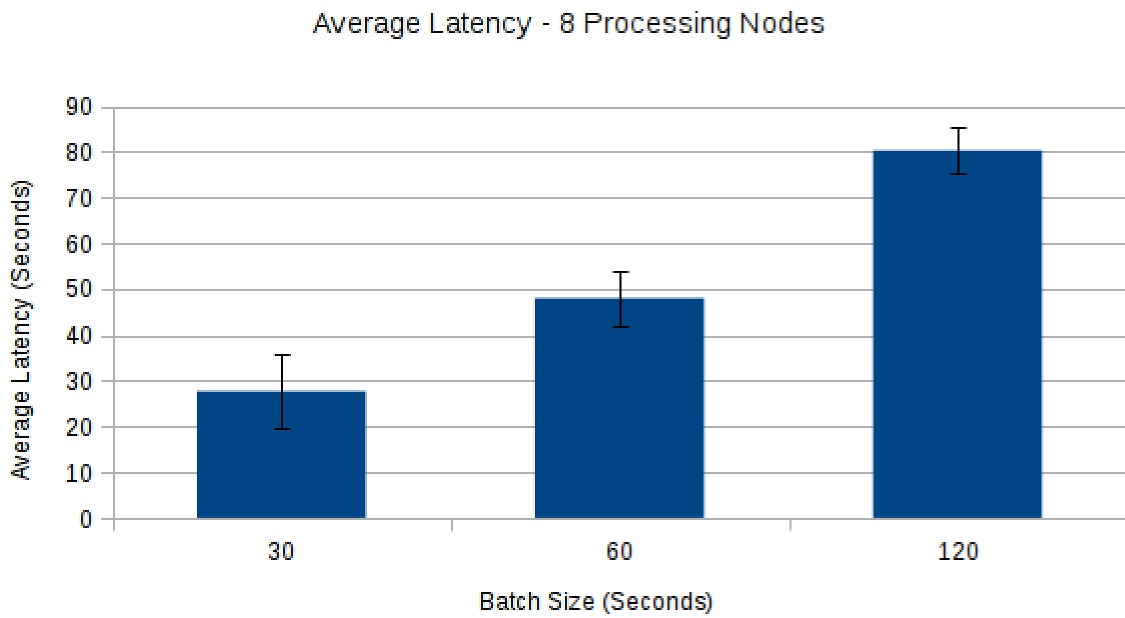

Figure 4.1 – Best case scenario - Large batches with 8 processing nodes

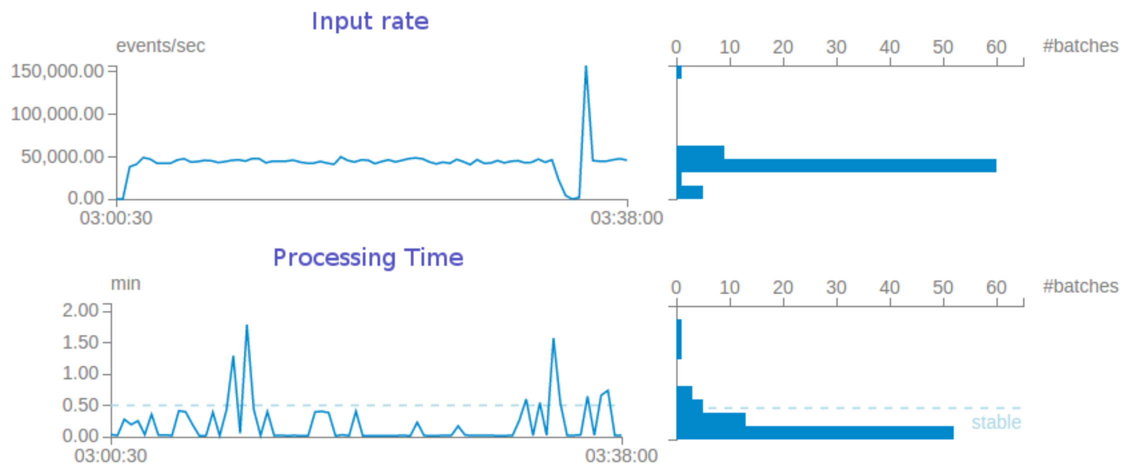

Figure 4.2 – Sample Execution - 8 processing nodes, 30 seconds batch and stable overall processing

happened for the batch sizes in the Figure 4.1. When latencies are larger than batch sizes, the system starts queueing those messages into memory buffers. The overall processing time keeps increasing, until there are not enough space anymore in the pre-configured queue buffer, as we can see in Figure 4.4.

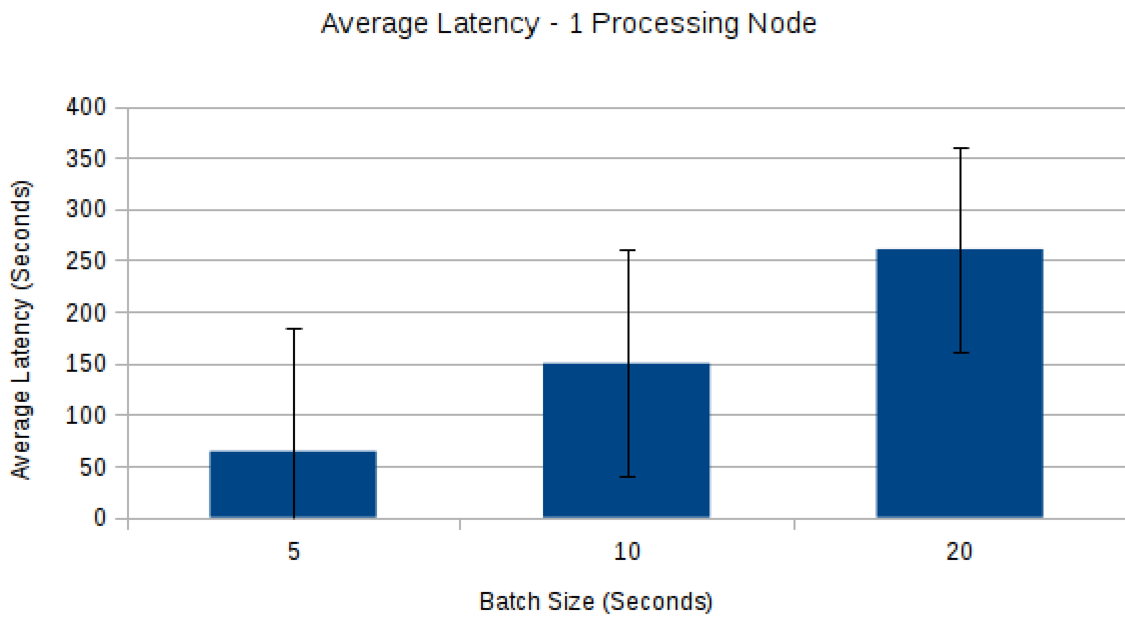

Figure 4.3 – Worst case scenario - Small batches with 1 processing node

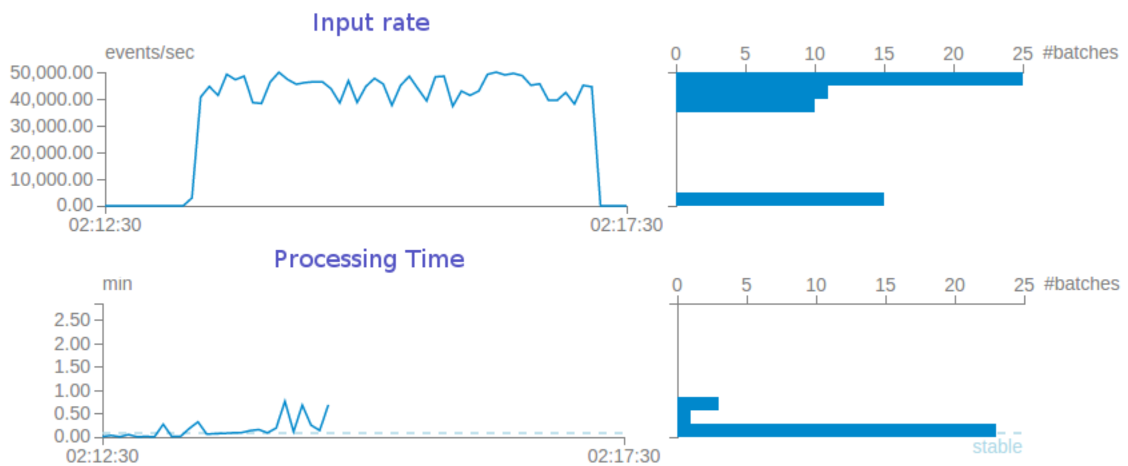

Figure 4.4 – Sample Execution - 1 processing node, 5 seconds batch and increasing input queueing

4.3 Throughput

The throughput is an important metric in an AMI, since it delimits 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 4.5, the system scales linearly up to 8 nodes, doubling the input processing rate when the number of machines doubles. By analyzing the Figure 4.5 we can

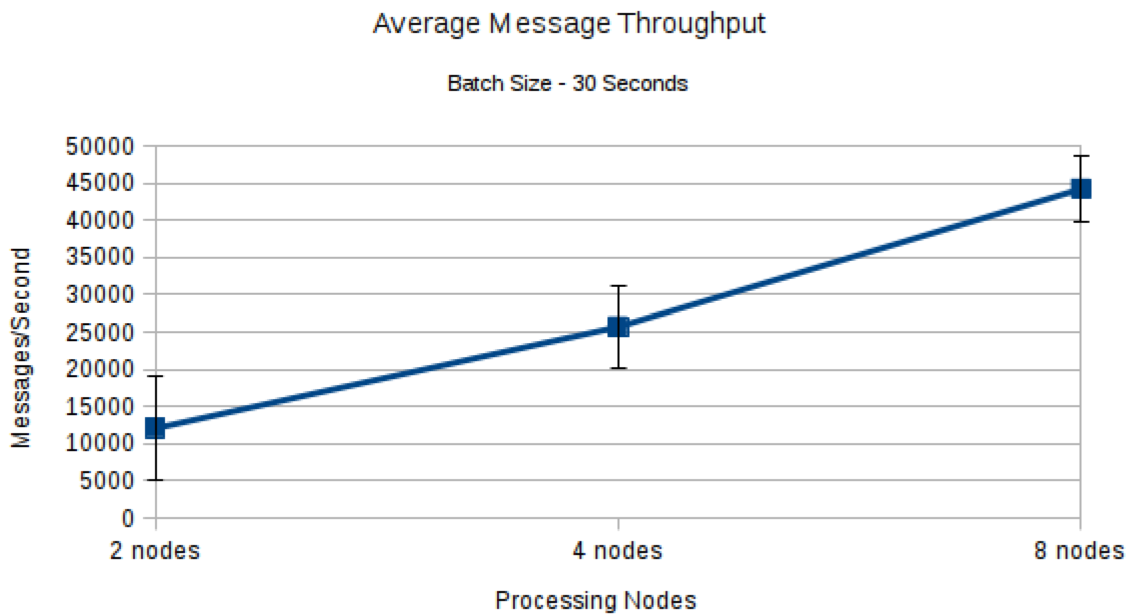

Figure 4.5 – Average message throughput, by number of nodes, with 30 seconds batch

perceive that, up to the boundary of the input rate, the system is able to handle the data input of a 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.

The system achieves a maximum median throughput of almost 45K events per second with 8 processing nodes. In this way, the system is able to handle all of the data being processed within their batch size. In this way, the system does not generate increasing queues on Kafka buffers, which happens with the same input rate but less than 8 processing nodes.

The second step of the testing process was the testing of some parameters of the system. We decided then to test the effect of batch sizes into the overall throughput of the system, due to the impact that we had seen into 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 —.

Contrary to our previous suspect, the batch sizes affect the latency more than the throughput — which is not exactly true to extremely low batch sizes, which can turn the system unstable and then have a great impact in the throughput — as we show in the Figure 4.6. Either way, the batch sizes still have some effects over the throughput of the system, but their impact is much lower than into the latency performance, generating a tiny constant increment into the overall throughput as the batch sizes grows up. However, we can see that the throughput do not

exponentially grows, as it happens with latency, when the batch sizes increase.

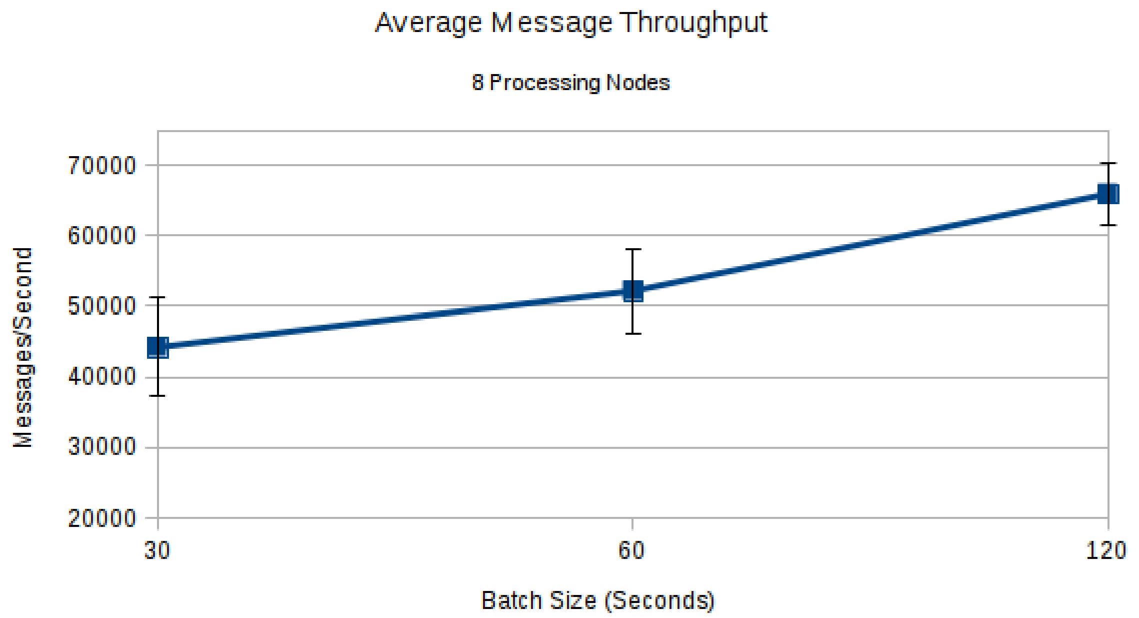

Figure 4.6 – Average message throughput, by batch sizes, with 8 processing nodes

5 RELATED WORK

In this chapter, we present works related to our proposal from different points-of-view. In Section 5.1 we present related works in the fields of energy consumption and smart grids infrastructure. In Section 5.2 we present more specific related works, in the area of load forecasting using event processing.

5.1 Energy Consumption and Smart Grids Infrastructure

The field of smart grids is highly active and there are ongoing research in topics such as energy forecasting, information security, energy consumption scheduling, etc. In this section we introduce studies related to energy consumption, smart grids and AMIs.

(METKE; EKL, 2010) discuss the importance of minding security issues when designing a smart grid solution, and provides key security technologies that must be adopted to assure minimal security for a smart grid environment, such as public key infrastructures and trusted computing. (FANG et al., 2012) describes in its survey the differences between the old electromechanical grids and the new digital grids, providing and important notion dividing the smart grids in 3 subsystems: Smart Infrastructure System, Smart Management System and Smart Protection System. The Smart Protection System is exclusively designed to take care of failure protection and security of the system.

(AUNG et al., 2012) provides an statistical model for load forecasting in smart grids, providing an method that can obtain approximately 98% of average accuracy on predictions and that is also computationally efficient, with potential for being used in large scale forecasting. (KALYVIANAKI et al., 2012) provide a technique to control average latency by dropping some incoming packages, in order to keep system stability over time.

The main advantage of AMIs is the ability to make the DSM. But it is not only limited to the ability of monitor and control the client devices. AMIs can also provide data to customers, which in turn make smart decisions for when it is preferable to spend or to save energy. (MOHSENIAN-RAD et al., 2010) proposes a technique based on game-theory to achieve a Nash equilibrium in the smart grid, based on the two-way communication provided by Smart Meters with the smart grid, the system balances itself to better schedule energy consumption for the energy companies and better energy costs for the end users.

5.2 Load Forecasting Using Event Processing

There are lots of research going related to smart grids, and a subset of these research uses Event Processing systems to address its problems. (ZHOU et al., 2013) and (ZIEKOW et al., 2013) provide a middleware solution for smart grids, and in the ladder extends the former to address load demand balancing. (DUNNING; FRIEDMAN, 2014) provides a highly scalable quantile estimator called t-digest, which was designed for parallel online operation.

The DEBS challenge (ZIEKOW; JERZAK, 2014) held in the DEBS conference of 2014 was one 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., 2014) is proposed a technique with a histogram of values to predict the median and a min-max heap approach. In (MARTIN et al., 2014) they propose an approach for data completion based on work measures, generating missing load measurements based on work measurements data. In (SUNDERRAJAN; AYDT; KNOLL, 2014) 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, 2014), that also works with STLF, but focus on prediction algorithms and do not explore distributed processing. (WÅLINDER; HOANG, 2015) also works with AMIs, but has its focus on analytics and social networks, providing metrics for the end-user to take smart decisions about energy consumption. Finally, (KUMAR, 2014) 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.

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 and the system was able to handle the pressure with high throughputs, while scaling linearly up to 8 processing nodes.

We found that for tiny batch sizes the system could turn unstable and have difficulties to process data. It could lead to increasing data queue until there are not space anymore and packets will start being dropped. It is possible and desirable to adjust batch sizes to fit the latency needs in a way that the system could be able to deliver the proper latencies without data loss.

It was also found that greater batch sizes improve throughput performance, leading to a greater number of events being processed per second, in expense of latencies, which start to increase proportionally.

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

We also plan to improve overall throughput by increasing the number of parallel data feeds, as well as platform settings such as data serialization and improvements into the algorithm itself.

The system availability could be improved by adding certain parameters and configurations to the platform. All of the most parts used to build the platform, Apache Kafka, Apache Spark and Apache Redis, present features for fault tolerance, but we still need more research about parameters and the interconnections between them to assure that.

A final important improvement would be to add an abstraction layer for machine deployment, such as Apache YARN [1] or Apache Mesos[2] with Docker[3] containers. It would provide advanced scheduling for our application, as well as to provide an abstraction layer from the underlining IaaS platform, turning it easier to replace the IaaS provider. We have tried a Docker deployment, but decided to stay with the standard Azure CLI due to several issues with Docker networking — that was still under development when this work was finished —.

[1] <https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>
[2] <http://mesos.apache.org>
[3] <http://www.docker.com>

REFERENCES

ABADI, D. J. et al. The Design of the Borealis Stream Processing Engine. In: **2nd Biennial Conference on Innovative Data Systems Research (CIDR)**. Asilomar, CA: [s.n.], 2005.

ABADI, D. J. et al. Aurora: A New Model and Architecture for Data Stream Management. **The VLDB Journal - The Int. Journal on Very Large Data Bases**, Springer-Verlag New York, Inc., v. 12, n. 2, p. 120–139, 2003.

ALEXANDROV, A. et al. The Stratosphere Platform for Big Data Analytics. **The VLDB Journal - The International Journal on Very Large Data Bases**, Springer-Verlag New York, Inc., v. 23, n. 6, p. 939–964, 2014.

ALFARES, H. K.; NAZEERUDDIN, M. Electric Load Forecasting: Literature Survey and Classification of Methods. **International Journal of Systems Science**, Taylor & Francis, v. 33, n. 1, p. 23–34, 2002.

ALI, A. B. M. S. **Smart Grids: Opportunities, Developments, and Trends**. [S.l.]: Springer Science & Business Media, 2013.

ANALYTICSWORLD, P. **How is Predictive Analytics Different from Forecasting?** 2015. Disponível em: <http://www.predictiveanalyticsworld.com/faq.php#q3-2>.

ANAS, M. et al. Minimizing Electricity Theft Using Smart Meters in AMI. In: IEEE. **P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), Seventh International Conference on**. [S.l.], 2012. p. 176–182.

ANCILLOTTI, E.; BRUNO, R.; CONTI, M. The Role of Communication Systems in Smart Grids: Architectures, Technical Solutions and Research Challenges. **Computer Communications**, Elsevier, v. 36, n. 17, p. 1665–1697, 2013.

APRELKIN, A. **Short Term Household Electricity Load Forecasting Using a Distributed In-Memory Event Stream Processing System**. Dissertação (Mestrado) — Technische Universität München, 2014.

ASHTON, K. That Internet of Things Thing. **RFID Journal**, v. 22, n. 7, p. 97–114, 2009.

ATZORI, L. et al. The Internet of Things: A Survey. **Computer networks**, Elsevier, v. 54, n. 15, p. 2787–2805, 2010.

AUNG, Z. et al. Towards Accurate Electricity Load Forecasting in Smart Grids. In: **The Fourth International Conference on Advances in Databases, Knowledge and Data Applications (DBKDA)**. [S.l.: s.n.], 2012.

BADGER, L. et al. Cloud Computing Synopsis and Recommendations. **NIST Special Publications**, v. 800.

BALAKRISHNAN, M. et al. CORFU: A Distributed Shared Log. **ACM Transactions on Computer Systems (TOCS)**, ACM, v. 31, n. 4, p. 10, 2013.

BEYER, M. Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data. **URL http://www.gartner.com/newsroom/id/1731916**, 2011.

BROWN, R. E. Impact of Smart Grid on Distribution System Design. In: IEEE. **Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE**. [S.l.], 2008. p. 1–4.

BUYYA, R. et al. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. **Future Generation computer systems**, Elsevier, v. 25, n. 6, p. 599–616, 2009.

BYLANDER, T.; ROSEN, B. A Perceptron-like Online Algorithm for Tracking the Median. In: IEEE. **Neural Networks, 1997., International Conference on**. [S.l.], 1997. v. 4, p. 2219–2224.

CARBONE, P. et al. Lightweight Asynchronous Snapshots for Distributed Dataflows. **Computing Research Repository (CoRR)**, abs/1506.08603, 2015.

CARVALHO, O.; NAVAUX, P. Um Estudo da Performance de Sistemas Distribuídos para o Processamento de Streams. In: **XXVI Salão de Iniciação Científica da UFRGS**. [S.l.: s.n.], 2014.

CARVALHO, O.; ROLOFF, E.; NAVAUX, P. A Survey of the State-of-the-art in Event Processing. In: **11th Workshop on Parallel and Distributed Processing (WSPPD)**. [S.l.: s.n.], 2013.

CARVALHO, O.; ROLOFF, E. et al. Beyond Hadoop: An Analysis of The Evolution of New Technologies for Cloud Computing. In: **Anais do, Workshop de Iniciacao Cientifica, XXV Simposio em Sistemas Computacionais, WSCAD-WIC**. [S.l.: s.n.], 2013.

CHANDRASEKARAN, S. et al. TelegraphCQ: Continuous Dataflow Processing. In: ACM. **Proc. of the 2003 ACM SIGMOD Int. Conference on Management of Data**. [S.l.], 2003. p. 668–668.

CHEN, M.; MAO, S.; LIU, Y. Big Data: A survey. **Mobile Networks and Applications**, Springer, v. 19, n. 2, p. 171–209, 2014.

CUGOLA, G.; MARGARA, A. TESLA: A Formally Defined Event Specification Language. In: ACM. **Proc. of the Fourth ACM Int. Conf. on Distributed Event-Based Systems**. [S.l.], 2010. p. 50–61.

CUGOLA, G.; MARGARA, A. Complex Event Processing with T-REX. **Journal of Systems and Software**, Elsevier, v. 85, n. 8, p. 1709–1728, 2012.

DAVITO, B.; TAI, H.; UHLANER, R. The Smart Grid and the Promise of Demand-side Management. **McKinsey on Smart Grid**, p. 38–44, 2010.

DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In: **Symposium on Operating System Design and Implementation (OSDI)**. [S.l.: s.n.], 2004. p. 137–150.

DUNNING, T.; FRIEDMAN, E. **Practical Machine Learning: A New Look at Anomaly Detection**. [S.l.]: O'Reilly Media, Inc., 2014.

ENERGY Department of. **U.S. Department of Energy**. 2015. Disponível em: <http://www.oe.energy.gov>.

EROL-KANTARCI, M.; MOUFTAH, H. T. Wireless Multimedia Sensor and Actor Networks for the Next Generation Power Grid. **Ad Hoc Networks**, Elsevier, v. 9, n. 4, p. 542–551, 2011.

FANG, X. et al. Smart Grid - The New and Improved Power Grid: A Survey. **Communications Surveys & Tutorials, IEEE**, IEEE, v. 14, n. 4, p. 944–980, 2012.

FERNANDEZ, R. C. et al. Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management. In: ACM. **Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data**. [S.l.], 2013. p. 725–736.

FERNANDEZ, R. C. et al. Liquid: Unifying Nearline and Offline Big Data Integration. In: **Biennial Conf. on Innovative Data Systems Research**. [S.l.: s.n.], 2015.

GANTZ, J.; REINSEL, D. Extracting Value from Chaos. **IDC iview**, n. 1142, p. 9–10, 2011.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The Google File System. In: ACM. **Proc. of ACM Int. Conf. SIGOPS Operating Systems Review**. [S.l.], 2003. v. 37, p. 29–43.

GUNGOR, V. C.; LU, B.; HANCKE, G. P. Opportunities and Challenges of Wireless Sensor Networks in Smart Grid. **Industrial Electronics, IEEE Transactions on**, IEEE, v. 57, n. 10, p. 3557–3564, 2010.

GÜNGÖR, V. C. et al. Smart Grid Technologies: Communication Technologies and Standards. **Industrial informatics, IEEE transactions on**, IEEE, v. 7, n. 4, p. 529–539, 2011.

HEINZE, T. et al. Tutorial: Cloud-based Data Stream Processing. 2014.

ISARD, M. et al. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: ACM. **ACM SIGOPS Operating Systems Review**. [S.l.], 2007. v. 41, n. 3, p. 59–72.

JAIN, A.; NALYA, A. **Learning Storm**. [S.l.]: Packt Publishing, 2014. ISBN 1783981326, 9781783981328.

KALYVIANAKI, E. et al. Overload Management in Data Stream Processing Systems with Latency Guarantees. In: **7th IEEE International Workshop on Feedback Computing**. [S.l.: s.n.], 2012.

KREPS, J. **Questioning the Lambda Architecture**. 2014. Disponível em: <http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html>.

KULKARNI, S. et al. Twitter Heron: Stream Processing at Scale. In: **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2015. (SIGMOD ’15), p. 239–250. ISBN 978-1-4503-2758-9. Disponível em: <http://doi.acm.org/10.1145/2723372.2742788>.

KUMAR, S. **Real Time Data Analysis for Water Distribution Network using Storm**. 2014.

KYRIAKIDES, E.; POLYCARPOU, M. Short Term Electric Load Forecasting: A Tutorial. In: **Trends in Neural Computation**. [S.l.]: Springer, 2007. p. 391–418.

LANEY, D. 3-D Data Management: Controlling Data Volume. **Velocity and Variety, META Group Original Research Note**, 2001.

LEE, I. et al. The Internet of Things (IoT): Applications, Investments and Challenges for Enterprises. **Business Horizons**, Elsevier, 2015.

LIN, W. et al. **PacificA: Replication in Log-based Distributed Storage Systems**. [S.l.], 2008.

LOHRMANN, B.; WARNEKE, D.; KAO, O. Nephele Streaming: Stream Processing Under QoS Constraints at Scale. **Cluster Computing**, Springer, v. 17, n. 1, p. 61–78, 2014.

LUCKHAM, D. C. et al. Specification and Analysis of System Architecture Using Rapide. **Software Engineering, IEEE Transactions on**, IEEE, v. 21, n. 4, p. 336–354, 1995.

MANYIKA, J. et al. Big Data: The Next Frontier for Innovation, Competition and Productivity. 2011.

MARGARA, A.; CUGOLA, G. Processing Flows of Information: From Data Stream to Complex Event Processing. In: ACM. **Proc. of the 5th ACM Int. Conf. on Distributed Event-based Systems**. [S.l.], 2011. p. 359–360.

MARTIN, A. et al. Predicting energy consumption with StreamMine3G. In: ACM. **Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems**. [S.l.], 2014. p. 270–275.

MARZ, N.; WARREN, J. **Big Data: Principles and Best Practices of Scalable Realtime Data Systems**. [S.l.]: Manning Publications Co., 2015.

MEIJER, E. The World According to LINQ. **Queue**, ACM, v. 9, n. 8, p. 60, 2011.

MELL, P.; GRANCE, T. The NIST definition of Cloud Computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

METKE, A. R.; EKL, R. L. Security Technology for Smart Grid Networks. **Smart Grid, IEEE Transactions on**, IEEE, v. 1, n. 1, p. 99–107, 2010.

MOHSENIAN-RAD, A.-H. et al. Autonomous Demand-Side Management Based On Game-Theoretic Energy Consumption Scheduling for the Future Smart Grid. **Smart Grid, IEEE Transactions on**, IEEE, v. 1, n. 3, p. 320–331, 2010.

OLSTON, C. et al. Pig Latin: A Not-so-foreign Language for Data Processing. In: ACM. **Proceedings of the 2008 ACM SIGMOD international conference on Management of data**. [S.l.], 2008. p. 1099–1110.

OUSTERHOUT, J. et al. The Case for RAMCloud. **Communications of the ACM**, ACM, v. 54, n. 7, p. 121–130, 2011.

PAVLO, A. et al. A Comparison of Approaches to Large-Scale Data Analysis. In: ACM. **Proc. ACM SIGMOD Int. Conf. on Management of Data**. [S.l.], 2009. p. 165–178.

PERERA, S. et al. Solving the Grand Challenge Using an Open Source CEP engine. In: ACM. **Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems**. [S.l.], 2014. p. 288–293.

REUTERS. **U.S. Smart Grid to Cost Billions, Save Trillions**. 2011. Disponível em: <http://www.reuters.com/article/2011/05/24/us-utilities-smartgrid-epri-idUSTRE74N7O420110524>.

ROLOFF, E. **Viability and Performance of High-Performance Computing in the Cloud**. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, 2013.

SABER, A. Y.; VENAYAGAMOORTHY, G. K. Plug-in Vehicles and Renewable Energy Sources for Cost and Emission Reductions. **Industrial Electronics, IEEE Transactions on**, IEEE, v. 58, n. 4, p. 1229–1238, 2011.

SAGIROGLU, S.; SINANC, D. Big Data: A Review. In: IEEE. **Collaboration Technologies and Systems (CTS), 2013 International Conference on**. [S.l.], 2013. p. 42–47.

SAGIROGLU, S.; SINANC, D. Big Data: A review. In: IEEE. **Collaboration Technologies and Systems (CTS), 2013 International Conference on**. [S.l.], 2013. p. 42–47.

STORM, A. **Understanding the Parallelism of a Storm Topology**. 2015. Disponível em: <http://storm.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology>.

SUNDERRAJAN, A.; AYDT, H.; KNOLL, A. Real Time Load Prediction and Outliers Detection Using Storm. In: ACM. **Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems**. [S.l.], 2014. p. 294–297.

THUSOO, A. et al. Hive: A Warehousing Solution Over a Map-reduce Framework. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 2, n. 2, p. 1626–1629, 2009.

TOSHNIWAL, A. et al. Storm @ Twitter. In: ACM. **Proceedings of the 2014 ACM SIGMOD international conference on Management of data**. [S.l.], 2014. p. 147–156.

TSADO, Y.; LUND, D.; GAMAGE, K. A. Resilient Communication for Smart Grid Ubiquitous Sensor Network: State of the Art and Prospects for Next Generation. **Computer Communications**, Elsevier, 2015.

WÅLINDER, C.; HOANG, B. BCStream - A Data Streaming Based System for Processing Energy Consumption Data and Integrating with Social Media. 2015.

WANG, G. et al. Building a Replicated Logging System with Apache Kafka. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 8, n. 12, p. 1654–1655, 2015.

WANG, W.; XU, Y.; KHANNA, M. A Survey on the Communication Architectures in Smart Grid. **Computer Networks**, Elsevier, v. 55, n. 15, p. 3604–3629, 2011.

WEISER, M. et al. The Origins of Ubiquitous Computing Research at PARC in the Late 1980s. **IBM systems journal**, v. 38, n. 4, p. 693–696, 1999.

WHITE, T. **Hadoop: The Definitive Guide**. [S.l.]: O'Reilly Media, Inc., 2012.

YAN, L. et al. **The Internet of Things: from RFID to the Next-generation Pervasive Networked Systems**. [S.l.]: CRC Press, 2008.

YU, Y. et al. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In: **OSDI**. [S.l.: s.n.], 2008. v. 8, p. 1–14.

ZAHARIA, M. et al. Spark: Cluster Computing with Working Sets. In: **Proceedings of the 2nd USENIX conference on Hot topics in cloud computing**. [S.l.: s.n.], 2010. v. 10, p. 10.

ZAHARIA, M. et al. Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In: USENIX ASSOCIATION. **Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing**. [S.l.], 2012. p. 10–10.

ZHOU, Q. et al. On Using Complex Event Processing for Dynamic Demand Response Optimization in Microgrid. In: IEEE. **Proceedings of IEEE Green Energy and Systems Conference**. [S.l.], 2013.

ZIEKOW, H. et al. Forecasting Household Electricity Demand with Complex Event Processing: Insights from a Prototypical Solution. In: ACM. **Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference**. [S.l.], 2013. p. 2.

ZIEKOW, H.; JERZAK, Z. The DEBS 2014 Grand Challenge. In: **Proceedings of the 8th ACM International Conference on Distributed Event-based Systems, DEBS**. [S.l.: s.n.], 2014. v. 14.

ZIKOPOULOS, P.; EATON, C. et al. **Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data**. [S.l.]: McGraw-Hill Osborne Media, 2011.