FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

**Christian Gulden**

# Towards Energy-Efficient Distributed Data-Stream Processing

Masterarbeit im Fach Informatik

2. Oktober 2017

Please cite as:
Christian Gulden, "Towards Energy-Efficient
Distributed Data-Stream Processing" Master's Thesis, University of Erlangen,
Dept. of Computer Science, October 2017.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

# Towards Energy-Efficient Distributed Data-Stream Processing

Masterarbeit im Fach Informatik

vorgelegt von

**Christian Gulden**

angefertigt am

**Lehrstuhl für Informatik 4**
**Verteilte Systeme und Betriebssysteme**

**Department Informatik**
**Friedrich-Alexander-Universität Erlangen-Nürnberg**

| | |
|---|---|
| Betreuer: | **Dipl.-Inf. Christopher Eibel**<br>**Dr.-Ing. Tobias Distler** |
| Betreuender Hochschullehrer: | **Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat** |
| Beginn der Arbeit: | **30. März 2017** |
| Abgabe der Arbeit: | **2. Oktober 2017** |

## Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

## Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Christian Gulden)
Erlangen, 2. Oktober 2017

# ABSTRACT

Distributed data-stream–processing systems have emerged in response to the growing amount of data generated by sensors, devices, and users in an increasingly interconnected world. As opposed to traditional batch-oriented processing, these systems are able to meet the demand for analyzing streams of data and extracting insights on-the-fly and in near realtime.

Simultaneously, awareness for the ecological impact of computing has risen and reducing wasteful energy usage in favor of sustainability and energy efficiency has become a shared goal of cloud-computing providers and software developers.

This work aims to reconcile both the emergence of stream-processing with the need for more energy-efficient computing by applying power capping to a distributed stream-processing cluster to decrease power usage while maintaining application performance. Furthermore, this work shows how leveraging unique characteristics of common stream-processing applications can lead to additional energy savings.

We provide an implementation and evaluation of the solution based on Heron, a distributed data-stream–processing platform used in production environments, to validate our approach. We show that power consumption can be reduced by up to 35 %, thus demonstrating the effectiveness of the implementation in conserving power while maintaining performance at an acceptable range.

# KURZFASSUNG

Verteilte Datenstromverarbeitungssysteme werden eingesetzt, um die wachsenden Datenmengen zu verarbeiten, die von Sensoren, Geräten und Nutzern in einer zunehmend vernetzten Welt erzeugt werden. Im Gegensatz zu herkömmlicher Stapelverarbeitung sind diese Systeme in der Lage, Datenströme ohne vorheriges Zwischenspeichern in Echtzeit zu analysieren und relevante Informationen aus den Daten zu extrahieren.

Gleichzeitig ist das Bewusstsein für die ökologischen Auswirkungen der Datenverarbeitung gestiegen und die Reduzierung des verschwenderischen Energieverbrauchs zugunsten von Nachhaltigkeit und Energieeffizienz zu einem gemeinsamen Ziel von Cloud-Computing-Anbietern und -Softwareentwicklern geworden.

Diese Arbeit zielt darauf ab, das zunehmende Aufkommen an Datenstromverarbeitungssystemen in Einklang mit dem Bewusstsein für energieeffiziente Informationstechnologie zu bringen. Dazu wird Power-Capping auf einen verteilten Datenstromverarbeitungs-Cluster angewendet, um den Stromverbrauch zu reduzieren und gleichzeitig die Anwendungsleistung beizubehalten. Darüber hinaus zeigt diese Arbeit, wie die Nutzung einiger Eigenschaften verbreiteter Datenstromverarbeitungsanwendungen zu zusätzlichen Energieeinsparungen führen kann.

Die in dieser Arbeit vorgestellte Lösung wird unter dem Einsatz von Heron, einer im Produktivbetrieb eingesetzten, verteilten Datenstromverarbeitungsplattform validiert. Wir zeigen, dass der Energieverbrauch um bis zu 35 % gesenkt werden kann. Dies belegt die Effektivität der Implementierung, Energie zu sparen und gleichzeitig die Leistung der Anwendungen in einem akzeptablen Bereich zu halten.

# CONTENTS

# 1 INTRODUCTION

With the rising number of connected people and devices, the need to process the flood of generated data has increased. Analyzing and extracting information from this data is an integral part of any business strategy: data analytics is driving profits and allows companies to gain insight into the behavior of their customers [Wal15; BC12; Sim13]. The shift towards more dynamic and user-generated content in the web leads to a stream of information that is only valuable for a short time and thus has to be processed immediately. Companies such as Netflix have already re-adjusted their way of processing data, for example, by monitoring user activity to optimize product or video recommendations for the current user [Net16]. Twitter performs continuous sentiment analysis to inform users on trending topics as they come up and Google has superseded batch processing for indexing the web to minimize the latency by which it reflects new and updated sites [PD10]. Traditional batch-processing systems cannot meet the demand for Internet-scale, high-throughput, realtime data analysis required by these use cases, so distributed data-stream processing has emerged as the computational paradigm of choice. Streams of input data are fed into a network of interconnected processing elements (topologies) and analyzed on-the-fly, avoiding the performance penalty associated with store-and-forward data management.

The need to process continuous streams of data has increased the prevalence of high-performance, high-dimensional, data-based applications, resulting in significant power-demands of the computing and storage infrastructure used. Reducing this energy consumption without sacrificing performance are worthwhile goals from both the perspective of data-center operators and from an ecological sustainability point of view. According to estimates by Amazon [Ham09], energy-related costs amount to 42 % of the total operational expenditure. On the other hand, data centers are responsible for 2 % of global greenhouse gas emissions – about the same as air travel [Coo+14].

With regards to energy efficiency, cloud computing has recently received considerable attention: providers are calling for the development of new techniques that deal with the unique opportunities and design challenges associated with implementing energy-efficient solutions for the cloud-computing era. Power capping [Pet+15] is a technique to control the peak power consumption of servers and has emerged as a reasonable way to increase energy efficiency in some distributed applications [Eib+18; Sch16; Lo+14]. However, to the best of our knowledge, research has not yet been published that investigates data-streaming engines specifically for their energy-savings potential or that introduces an automatic, distributed elastic power-capping system.

## 1.1 Goals

In this work, we aim to reconcile both the demand for continuous, realtime data analysis and for sustainable computing by locating and exploiting energy-saving opportunities in distributed

stream-processing applications. We leverage power capping as a way to reduce the total power consumption of a stream-processing cluster in a transparent manner. Our approach maintains the performance of the streaming application and elastically reacts to changes in the workload.

We concern ourselves with three major goals:

1. We show that power capping is an effective method to reduce the energy consumption of a distributed stream-processing application
2. We show how the unique properties of some streaming topologies can lead to further energy savings
3. We implement a system to transparently, automatically, and elastically determine the lowest possible power caps of the whole cluster without deteriorating the performance of the application

To achieve elastic, transparent, and automatic power capping, the implementation needs to be capable of determining the lowest possible power cap for each node in the streaming cluster without knowledge of the application being executed. Additionally, the solution needs to be able to satisfy SASO properties [LX08]. That is, the implementation ensures that for a fixed throughput, the power cap does not oscillate (stability), converges to the ideal cap that maximizes throughput (accuracy), reaches the minimal power cap quickly (short settling time) and avoids lowering the cap further than necessary (overshoot).

We address the challenge of showing the effectiveness of power capping in the context of data-stream processing by comparing the total cluster energy consumption before and after setting power caps. The approach is evaluated on multiple realistic benchmarking topologies at preset static workloads. We demonstrate that some streaming applications can lead to further energy gains by investigating stream-grouping algorithms and showing how even a minor modification can be an effective power and energy saver. Finally, we address the challenge of implementing an elastic, transparent, and automatic power-capping system as a master–slave design with a single master observing runtime metrics and each slave setting the local power cap as instructed. A *capping strategy* run on the master determines the ideal power caps and adheres to SASO properties by addressing overshoot as well as accuracy based on the runtime metrics. Both stability and settling time are functions of a set of configuration parameters for these strategies.

The techniques we introduce all have general applicability and can be implemented in any stream-processing system. To evaluate our work, we use Heron, a distributed stream-processing engine developed at Twitter [Twi17a].

## 1.2 Outline

The remainder of this thesis is structured as follows: Chapter 2 introduces the fundamentals of distributed data-stream processing and describes power management in computer systems. Chapter 3 illustrates the main concepts and design of our energy regulator. Chapter 4 shows how this design was practically implemented on top of Twitter Heron. In Chapter 5 we describe the applications we implemented and used for the evaluation. In Chapter 6 we evaluate our implementation using Heron with multiple realistic topologies. Chapter 7 briefly explores a potential energy-saving technique we discovered while pursuing our main idea that was not yet thoroughly evaluated, but might provide the basis for future work. Chapter 8 gives an overview of similar works and how they relate to this publication. Finally, Chapter 9 concludes this work with a brief summary of its content.

# 2 FUNDAMENTALS

In this chapter, we introduce concepts and terminology required to understand the fundamental ideas presented in this work. The first part focuses on distributed data-stream processing and describes defining principles present in this paradigm. It details the structure of data-streaming applications as well as concepts of the underlying data model. The second section deals with power management in servers and the mechanisms we eventually exploit in order to reduce energy consumption.

## 2.1 Data-Stream Processing

The paradigm of data-stream processing is not an inherently new concept, however the rise of big data and cloud computing in the past years lead to the emergence of various Distributed Stream-Processing Engines (DSPEs) [Kam+13]. While each system has its own unique properties and use cases, we focus on giving an abstract view on what defines them in general, their shared data model, and common rationale.

Traditional data-analytics systems initially store and only periodically process large volumes of static data. A well-known example for such a system is Apache Hadoop MapReduce [Apa17a]: it uses the two-phase MapReduce programming model, where intermediate data is persisted on the disk before being processed. These systems are designed for batch-oriented work loads without strict limits on the processing latency. Streaming-analytics engines instead process data as it becomes available [Win+16]. This approach keeps data in motion and is beneficial in most scenarios where new, dynamic data is generated continuously. This data is processed sequentially on a record-by-record basis or over sliding time windows, and used for a wide variety of analytics, such as correlations, aggregations, filtering, and sampling. Data can come from various sources: log files, e-commerce purchases, information from social networks, financial trading or telemetry metrics from connected devices or instrumentation in data centers. Streaming applications provide insights into many aspects of business and consumer activity and allow its users to respond to changes and emerging situations as they occur. In the context of scientific research, the demand for near real-time analysis of streaming data is rising rapidly [Cho+16]. Scientific experiments and simulations are creating streaming data at an increasing velocity and volume, making it a natural fit for data-stream processing to handle.

### 2.1.1 Stream-Processing Model

A stream is defined as an unbounded sequence of tuples generated over time. The elements of each tuple are called attributes or fields.

Applications running on stream processors are organized as a logical network of Processing Elements (PEs), forming a Directed Acyclic Graph (DAG), which is also referred to as a *topology*. An example can be seen in Figure 2.1. Each PE is a node in the graph, inputs and outputs are modeled as edges. Tuples of a stream flow through this graph from left to right and PEs consume them, emitting new tuples themselves based on application logic. If tuples flow from node A to node B, A is called the *upstream* node of B and B is called the *downstream* node of A [Hwa+05]. This relationship is transitive and does not require adjacent edges between any two PEs. Generally, a set of PEs running in parallel (e.g., nodes B and C in Figure 2.1) are not synchronized and can process data at any rate, depending on the available resources.

Communication between components can happen either push- or pull-based. In the former, components send tuples to their subsequent neighbor to be stored in its ingress queue until ready for processing. In pull-based implementations, components poll their antecedent neighbor for available tuples.

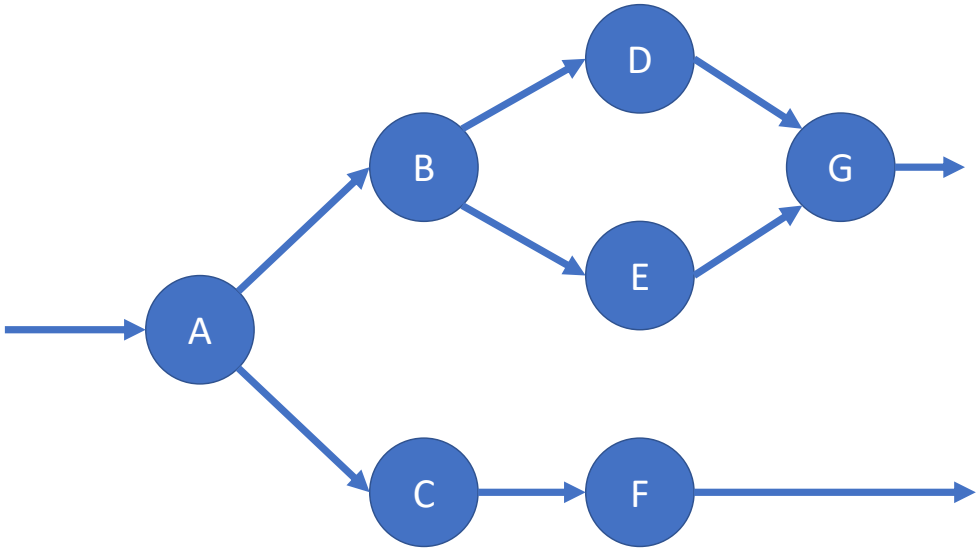



**Figure 2.1** – Logical view of a stream-processing topology. The PEs are labeled A to G and form the directed acyclic graph.

### Requirements of Realtime Stream Processing

Stonebraker et al. [SÇZ05] have defined eight requirements for realtime stream processing. We identify the three key requirements relevant for understanding this work:

1. **Data Mobility** is the term used to describe how data moves through the topology; it is the key to maintaining low latency as blocking operations and passive PEs can decrease data mobility.

2. **Data Partitioning** describes strategies used to partition and distribute large volumes of data across the nodes of the topology. The strategies affect how well data is processed in parallel and how the deployment scales.

3. **Stream Imperfections** emerge, for example, when tuples in a stream are delayed, arrive out of order, are duplicated, or lost. DSPEs must provide functionality to handle these imperfections, for example, using delayed processing or temporary storage.

#### Processing Element Operations

Processing elements in a DSPE can be categorized into three classes: producers, consumers, and operators, which act as data generators, receivers, and processors, respectively. Producers are also called *spouts*, consumers *sinks*, and operators *bolts*.

Tuples are injected into a topology from a spout. Data like user clicks, billing information or unstructured content such as images or Tweets are collected from various sources inside an organization and then moved to a streaming layer from which it is accessible to the stream processor. Spouts are pulling data from this layer – which can be an HTTP service or a queuing system like Kafka, Kinesis, or ZeroMQ.

Bolts are receiving or pulling tuples from the upstream PEs in order to transform them in some way. The functionality of each bolt is implemented at the discretion of the developer. DSPEs allow topologies to be written in either a domain-specific or general purpose programming language. Common operations performed by bolts are *filtering*, where tuples are discarded based on a predicate, *joining*, where tuples from multiple streams are combined based on one or more attributes, and *aggregating*, where tuples are stored over a time or quantity window, an aggregation function is performed, and one or more tuples are emitted.

To ensure data mobility and achieve low latency, PEs should process tuples in-stream without executing blocking operations that rely on external components or services. Performing these operations in the critical path leads to unnecessary latency when processing tuples. Blocking operations can be an invocation of a sleep system call, reading or writing from a file on disk, calling thread synchronization primitives, or executing database transactions. Besides avoiding blocking operations, DSPEs should ideally use an active processing model. In passive systems, the bolts are required to continuously poll for data, which results in overhead and additional latency, because half the polling interval is added to the processing delay on average. Active systems avoid this by leveraging event-driven processing capabilities.

### 2.1.2 Physical Distribution of Components

The preceding section has shown how the logical view of a streaming topology describes the relationship between PEs and the flow of data through them. However, when deploying a streaming application on a cluster of servers, an additional view of the topology is required: the *physical view*. It describes how logical elements are placed on the servers of the cluster and how edges are mapped to physical connections. In order to achieve high throughput, redundancy, and scale, each PE can be distributed in parallel among the computing nodes.

Figure 2.2 shows a possible distribution of the first three nodes from the example topology depicted in Figure 2.1: six instances of PE A are running in parallel on the same host as the two instances of B, the physical instances of C are running on a different host. Tuples sent from A to B do not leave the physical host boundary, while tuples sent from A to C do. The edge connecting A to C is mapped to a physical network connection and the edge connecting A to C physically manifests itself in the form of inter-host communication.

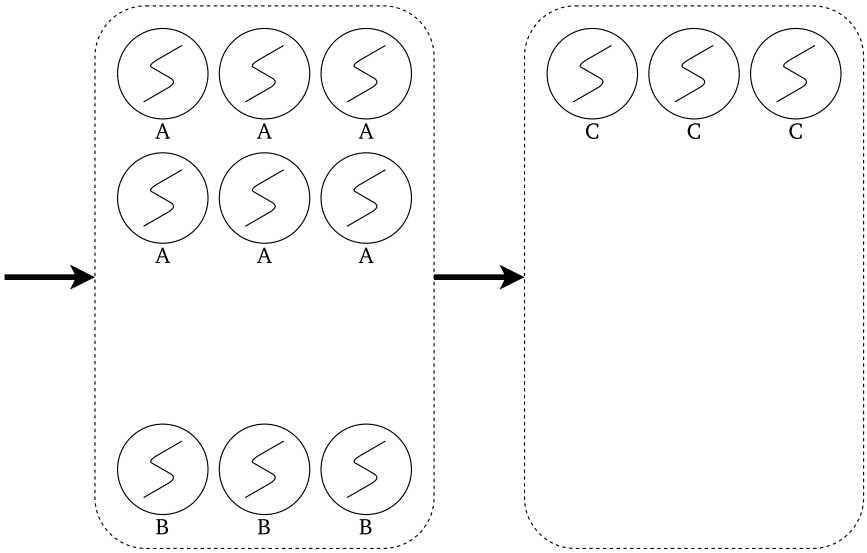



**Figure 2.2** – Physical distribution of logical PEs A, B and C across two hosts.

### 2.1.3 Data Partitioning

Tuples that are sent through a topology can take multiple paths, for example, in Figure 2.1, tuples emitted by A can be sent to C, B, or both simultaneously. When designing a streaming application, the developer can create paths through the topology and explicitly define under which circumstances tuples should take a certain path. These paths are called *streams* and logically group together tuples with a shared processing goal. For example, imagine Figure 2.1 is part of a larger machine-learning–streaming topology, tuples emitted by A are classified as either training or production data from the real world use case. PEs C and F are used for statistics about the real world data, while B, D, E, and G represent the actual machine learning implementation and require both training and production data. In this use case, the developer would connect node A to C using the production stream and A to B using both the training and production stream.

Besides defining streams for a topology, the developer also has to specify a stream grouping, which controls how tuples of a stream are partitioned among the physical instances.

The following is an incomplete list of possible stream-grouping schemes [Apa17b]:

- **Shuffle Grouping** distributes tuples in a uniform, random way across the instances, ensuring that each physical instance will process an equal number of tuples. It distributes the work load uniformly across the instances and is ideal when there is no need for any partitioning based on the attributes of each tuple.

- **Fields Grouping** partitions the stream by one or more attributes specified in the grouping. For example, if the stream is grouped by a “tweet id” field, tuples with the same “tweet id” will always be sent to the same instance. For example, this behavior is desirable in cases where a bolt implements a local cache.

- **All Grouping** sends a copy of each tuple to all instances of the receiving node. This grouping can be used for signaling instances, for example, when requesting a cache refresh.

- **Global Grouping** sends tuples generated by all instances of the source to a single target instance. This can be used to implement a reduce-phase, combining results from previous steps in a single instance.
- **Direct Grouping** allows the emitter to decide for each tuple individually which physical instance it should be sent to.

### 2.1.4 Back Pressure

In physics and engineering, back pressure describes pressure exerted by a liquid or gas against the forward flow in confined places such as a pipe. In information technology, it describes the buildup of data packets in buffers when the input rate is larger than the egress rate. In the context of stream processing, it occurs if one PE is struggling to keep up with the processing rate of the topology, requiring the streaming system as a whole to respond in a sensible way. Since it is generally not desirable for the component under stress to fail or to drop messages in an uncontrolled fashion, it should communicate the fact that it is under stress to upstream components and get them to reduce the load. This back pressure is an important feedback and flow control mechanism that allows systems to gracefully respond to load rather than collapse under it. Additionally, back pressure is an important indicator when it comes to tuning and dimensioning the number of instances and the degree of parallelism of the physical topology: the workload can be distributed between several instances of the same PE, avoiding any one instance to collapse under back pressure during normal topology operation.

## 2.2 Power Management

Improving energy efficiency has become an important goal when designing modern systems, not only for battery-powered mobile devices but also for datacenter-scale distributed applications. In the face of ever-increasing power consumption and associated economic and ecologic footprint, providing and using tools for managing the power intake of computer systems has gained in popularity.

Power management refers to the ability of an electric system to manage its energy usage by switching to a low-power state or turning off when inactive or the performance requirements allow for it. Each hardware component in a server employs some form of power management, however compared to other components, the CPU represents the largest share of total power consumption [BCH13]. Therefore, this work focuses on techniques to control the power consumption of the CPU specifically. Power management mechanisms for the CPU can force it to execute either in an *inactive idle* or an *active throttled* state. In the former, the CPU is running in an inactive state, where some components are switched off, significantly lowering the energy consumption. In a throttled state, the CPU frequency is lowered and instructions are executed at a reduced rate, leading to energy savings as well.

Even though peak load and thus peak power consumption is rarely, or ever, reached in servers, it is still used as the basis for dimensioning data centers and power infrastructure [BCH13]. This over-provisioning of resources during typical load conditions means a waste of limited Capital Expenditure (CAPEX) and Operating Expenditure (OPEX) resources. One method of handling this inefficiency is to explicitly set the power limit of the servers lower than the peak power consumption. This is called power capping and allows for granular control of the peak energy utilization of individual servers in a cluster.

A variety of techniques implementing power capping exist, seven of which have been described and evaluated in [Pet+15]. The most commonly used one is *Dynamic Voltage-Frequency Scaling*

*(DVFS)*, it allows for control over the supply voltage or frequency of hardware components (e.g., processors or memory). Decreasing voltage or frequency reduces the power consumption at the cost of longer application runtime. *Forced idleness* is another technique, that works by periodically forcing the application to pause execution, causing the whole processor to transition into a low-power sleep mode. However, this approach interferes with the application and its communication with external processes or the operating system, which can adversely impact performance, limiting the usefulness of the technique. Finally, *Running Average Power Limit (RAPL)* is a technology present in modern Intel processors, which combines automatic DVFS and clock throttling to maintain a user-defined upper limit for the power consumption. The integration of power monitoring and control inside the chip causes RAPL to be more accurate and faster at identifying and adapting to workload changes compared to plain DVFS.

## 2.3 Summary

The need to gather and analyze data streams in realtime to extract insights and detect emerging changes is increasing. Stream-processing systems enable carrying out these tasks in an efficient and scalable manner, by taking data streams through a network of operators. Simultaneously, awareness for sustainable computing is increasing and demands for more energy efficiency are made. Power capping can be used to limit the maximum energy consumption of computer systems. It is an important system level technique, that we use to effectively locate and efficiently leverage energy-saving opportunities in stream-processing applications in this work.

# 3 ARCHITECTURE

After having described the fundamentals of distributed data-stream processing and power management in computer systems, the following chapters will detail how our solution aims to reconcile the need for energy-efficient computing with the increasing importance of distributed realtime data-stream–processing engines. Fundamentally, the approach uses power capping to reduce the energy efficiency of distributed stream-processing applications without deteriorating the observable performance of the application being executed.

This chapter describes the underlying goals motivating and affecting the design decisions. It is concluded by detailing the functionality of each component and how they interact. The design and its guiding principles remain independent of any concrete streaming or clustering software used.

## 3.1 Design Goals

The main idea of our approach is to find and set power caps during runtime, where workload and resource availability can be queried dynamically. “Power cap” refers to the upper limit on the maximum power consumption, set using a power-capping technique. The solution should be able to reduce the energy consumption of the streaming cluster without decreasing the performance. For this reason, the ability to continuously measure the performance characteristics of the topology being executed and the impact of power-capping is one major requirement.

The solution should be able to automatically find the *ideal power cap* for each server in the cluster, that is, the lowest power cap that does not negatively affect the global application performance while maximizing energy savings. Additionally, the approach should not require any prior knowledge of the application, its performance characteristics, or the workload it processes to function. However, it requires some way to determine the current application performance and system workload during runtime. This is necessary for the solution to be able to elastically react to workload changes and to determine if any power caps set have caused the performance to deteriorate. For streaming topologies, the tuple throughput is a good indicator for the overall application performance.

Besides basing our solution on runtime elasticity, our approach also has to ensure a minimal impact on the resource utilization of the streaming cluster – that is, all components that need to run on the same servers that execute the actual application logic must not require excessive CPU and memory resources. Furthermore, our solution should not require any modifications to the code of the streaming application – it should transparently work in existing and future deployments. Finally, we aim for our design to be scalable with regards to the size of the cluster.

Our solution is limited to handling the case of a single distributed streaming-application running on the cluster. The approach does not handle multiple streaming topologies being executed on the same cluster, or additional non-streaming applications.

## 3.2 Design

The architecture of our solution follows the master–slave principle: a single *cluster energy-regulator* (the master) evaluates power caps for each server of the cluster based on performance information queried from a database of runtime metrics. A *local energy regulator* (the slave) is running on each server, receiving information about setting power caps from the cluster regulator. Finally, the streaming application is executed on each node of the cluster. The streaming application components are not running in isolation, but instead represent parts of the larger distributed data-stream–processing application. Each consists of the physical PEs and any auxiliary components required by the DSPE. It emits updates describing the performance state of the application to the metrics database. Figure 3.1 shows the architecture of our solution.

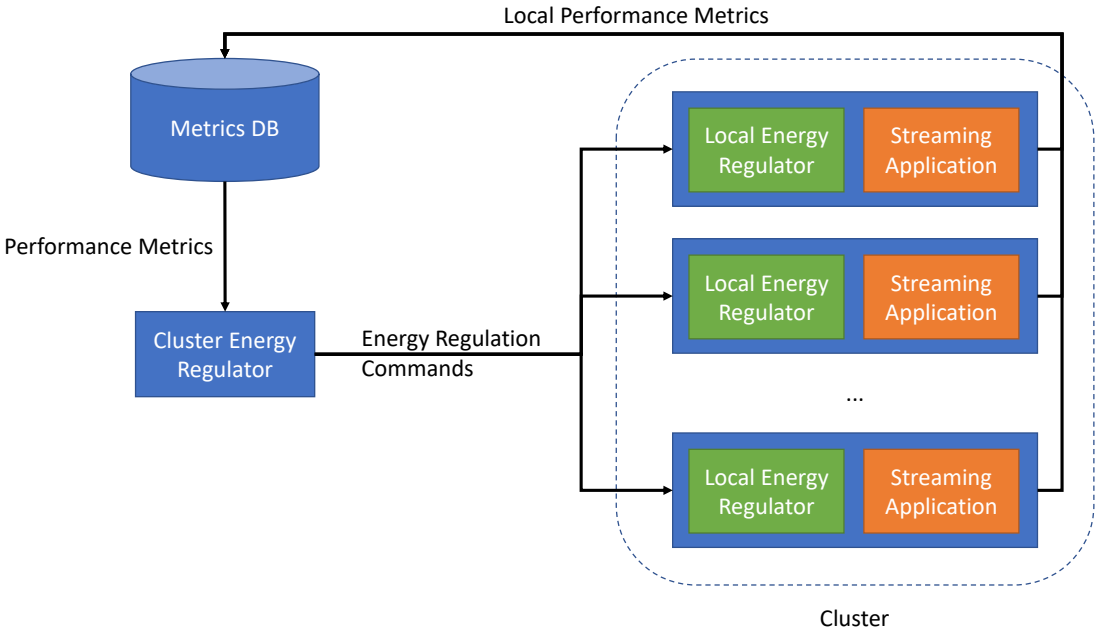



**Figure 3.1** – Architectural view of the elements composing our solution.

In the following, we will introduce and describe all of its components.

### 3.2.1 Cluster Energy Regulator

The *cluster energy regulator* is the central component of our design. It periodically queries the metrics database for runtime information about the cluster whose energy it is supposed to regulate. Using this information and any local state it maintains, it re-evaluates the power caps to be set for each server. The decision to change the power cap and by what extent is made by the *capping strategy*, a part of the control algorithm that decides to raise or lower the cap for each individual node of the cluster. The cluster regulator communicates with each server to notify them about the new caps to set by sending *energy-regulation commands* to the desired *local energy regulator*.

The most defining feature of our solution is the ability of the cluster energy regulator to maintain a global view of the cluster performance and power caps. This allows the cluster energy regulator to

monitor the impact of setting limits on the power consumption of an individual server on the total application performance and react appropriately by either lowering or raising the caps.

### Control Algorithm

The control algorithm is the core functionality of the cluster energy regulator. Fundamentally, it is a function that receives performance information of the cluster and the streaming application being executed as input and returns ideal power caps for each server of the cluster as output. It is run periodically to decide whether power caps need to be lowered or raised for each server. To determine whether changes to the power caps have caused performance to deteriorate, it first needs to establish a baseline view of the performance of the stream-processing application. The baseline view consists of the throughput observed, and power drawn by the cluster for a fixed workload when no power caps are set. The algorithm attempts to maintain this baseline throughput while simultaneously reducing the power consumption by applying power caps.

The task of finding the lowest power caps that do not cause the observed throughput to deviate from the baseline is executed by the capping-strategy algorithm. Fundamentally, it uses the topology throughput and the current energy usage of each server in the cluster as input and strategically lowers the power caps until the ideal power caps are found.

Figure 3.2 gives a high-level description of the control algorithm, illustrating the abstract processing steps. After determining the performance baseline for the workload, the cluster servers are run with the current (default) power caps set. After some time has elapsed, the algorithm queries the cluster performance state (including the tuple throughput). If the set power caps have not caused the performance to deteriorate, the power caps are lowered for one or more of the servers according to the capping strategy and continue executing the application with these newly set power caps. If the set power caps have caused the performance to deteriorate, they are reset to a value that previously did not cause any degradation. This is now the ideal power cap for that server. This process is repeated until ideal power caps for all servers have been found. The capping process is restarted every time the workload changes.

The algorithm relies on the back pressure and throughput information queried from the metrics database to detect changes in the workload as an indicator for lowering or raising the power caps. The presence of back pressure (see Section 2.1.4) is an indication that the current power caps are degrading performance more than required to handle the load, thus suggesting to raise the caps. A decrease in throughput without the presence of back pressure is an indication that the workload is decreasing, which suggests lowering the power caps.

To summarize, the control algorithm operates based on the following two principles to handle changing workloads:

1. **Expand:** If the throughput decreases, raise the power caps.

2. **Contract:** If the throughput remains consistent with regards to the baseline, lower the power caps.

In order to decrease the settling time of the capping and control algorithm, we employ a *capping cache*: it maps previously observed throughput values to power caps set for each server. If a workload change is detected, this allows the algorithm to first hit the cache and start out at an initial power cap close to one known to be able to handle the throughput.

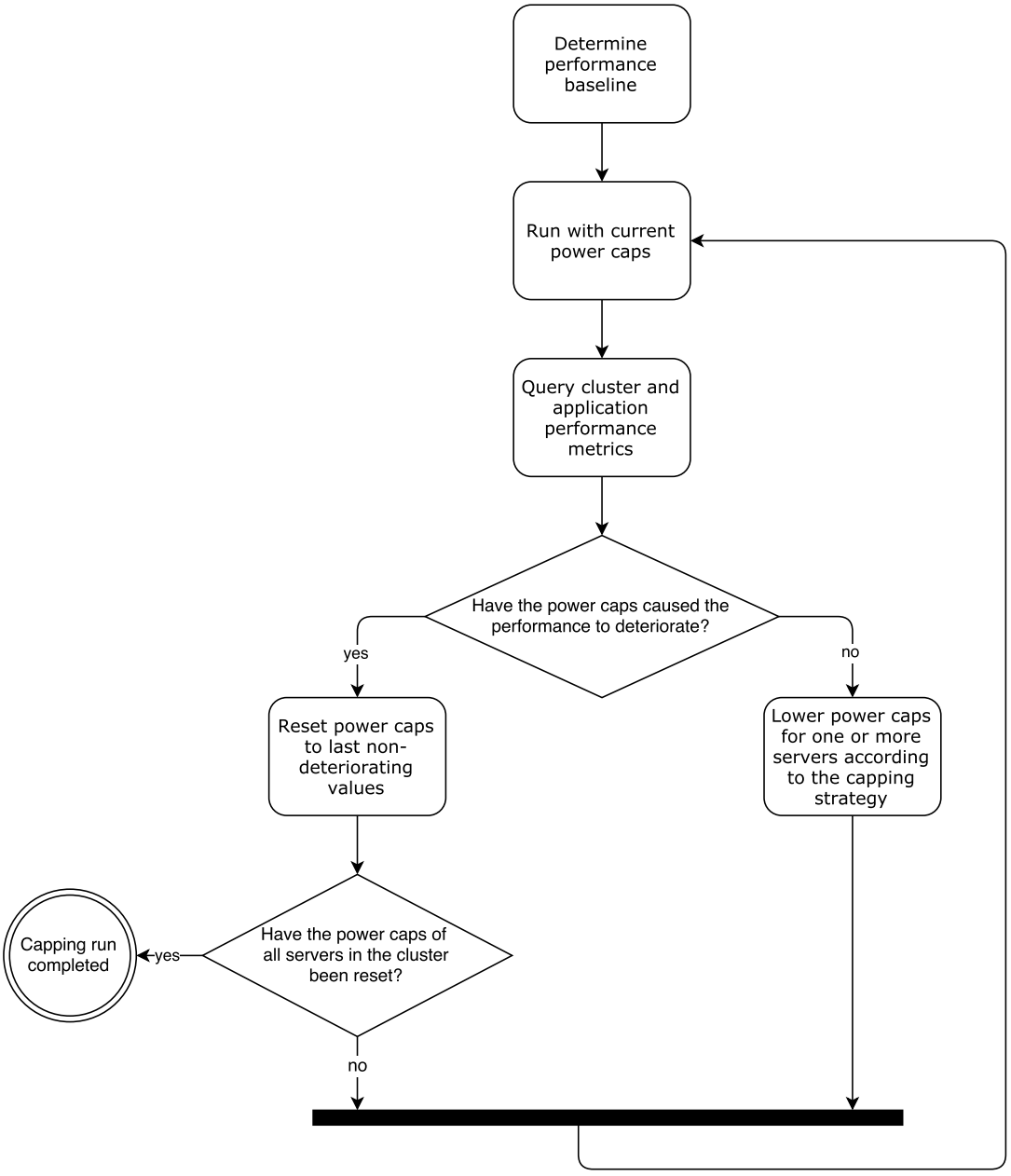



**Figure 3.2** – High-level description of the control algorithm.

### 3.2.2 Metrics Database

It is essential for the functionality of the control algorithm to be aware of the performance and workload state of the cluster and the streaming application being executed during operation. Only when these metrics are up-to-date and accurate, can the control algorithm determine the ideal power cap accurately and minimize the impact on application performance. The *metrics database* component is supposed to reflect this importance. It receives and stores runtime information about the DSPE application and each server of the cluster. For the capping process, the throughput of the streaming application, the energy usage of each server, and any back pressure observed are essential for its function.

Despite the name of the component, performance metrics do not have to be stored in an (external) database. However, in general, due to the nature of the data, it is ideally implemented as a time-series database. This allows the control algorithm to more easily query the performance state over a past observation period – for example, if power caps were set for 5 minutes and the impact on the application performance needs to be determined, a time-series database provides an easy interface to query the throughput measured for the past 5 minutes.

### 3.2.3 Local Energy Regulator

After the decision to cap a server of the cluster to a specific value, this command needs to be transferred and executed. For this reason, a regulator slave is running on each server. It listens for commands sent from the cluster energy regulator and, after receiving one, it sets the local power cap as requested. The regulator slave is a lightweight component as it does not maintain any local state or communicates with other servers in the cluster. It only interfaces with the power-capping mechanism used in its implementation and sets the power caps as instructed.

### 3.2.4 Streaming Application

The streaming application represents the physical PEs and any auxiliary components required by the DSPE. How the streaming application is distributed among the nodes of the cluster depends on the specifics of the engine and the configuration of the physical topology. In Figure 3.1, each streaming-application component is part of the larger, clustered stream-processing deployment. In order for the proposed solution to increase the energy efficiency to be effective, ideally each server in the cluster is involved in processing the total streaming workload. This means that at least one PE is executing on each server.

The distributed streaming application needs to be able to monitor its own performance and report it to the metrics database. The total topology throughput best reflects the overall application performance as a function of the workload and can only be determined at the application-layer.

## 3.3 Summary

The fundamental idea of our proposal is to leverage power capping to optimize the energy efficiency of distributed stream-processing applications in a transparent and non-invasive way. The building blocks and concepts of our solution have been discussed in the preceding sections in an abstract way. The next chapter will detail the implementation of these concepts and ideas on top of a cluster running production-grade software and hardware components.

# 4 IMPLEMENTATION

In this chapter, we describe how the solution presented in abstract terms in Chapter 3 is actually implemented in a production environment. The streaming engine of our implementation is Twitter Heron, running on a Mesos cluster. The power-capping mechanism we use is Intel RAPL.

## 4.1 Implementation Goals

The goal of the implementation was to fulfill the requirements outlined in Section 3.1 in a way that is applicable to a production-grade cluster environment. This implies that the choice of software components and their deployment reflects the state of the art in cluster-management and stream-processing software. Heron has been used at Twitter for realtime analytics since 2014, and has become the de-facto replacement for Storm due to its superior performance in benchmarks and ease of use. Similarly, Apache Mesos has been adopted and used by industry leaders ranging from Apple to Netflix and Uber. Due to its widespread use and out-of-the-box support for Heron, we chose Mesos as the cluster manager for the implementation. Intel RAPL is a highly effective and fast power-capping technique, as empirically confirmed by previous work and a comparison with other mechanisms [Pet+15]. RAPL is readily available and fully supported by the version of Linux we use.

Finally, the implementation must be able to integrate into the streaming cluster in a minimally-invasive way. That is, no modifications to Heron, Mesos, or the topology code should be necessary.

## 4.2 Apache Mesos

Apache Mesos is a platform for sharing cluster resources across diverse cluster-computing frameworks [Hin+11]. It does so by abstracting the underlying infrastructure and providing a common interface for accessing cluster resources to applications. The underlying cluster manager used is in general independent of the design we propose. We picked Mesos because it is well-supported by Heron and has been adopted by many organizations as the cluster manager for their deployments.

### 4.2.1 Architecture

Figure 4.1 shows the main components of Mesos and their interactions with elements of the Heron architecture (described in more detail in Section 4.3). A Mesos *agent* is running on each server of the cluster. The *master* daemon manages these agents. An application running on a Mesos cluster is called a *framework*. A framework consists of multiple tasks that are executed on one or more agents. When deploying topologies to the cluster, Heron is a framework running on top of Mesos.

The Mesos master enables sharing of resources (CPU, RAM, disk space) by making *resource offers* to the frameworks. Each offer contains an identifier for the originating agent as well as a list of resources and their offered amount. They are received by the framework schedulers (e.g., the Heron scheduler), which decide which of the offered resources to use. When a framework accepts the offer, it passes a description of the tasks it wants to run on them to the Mesos master. In turn, Mesos launches the tasks on the corresponding agents.

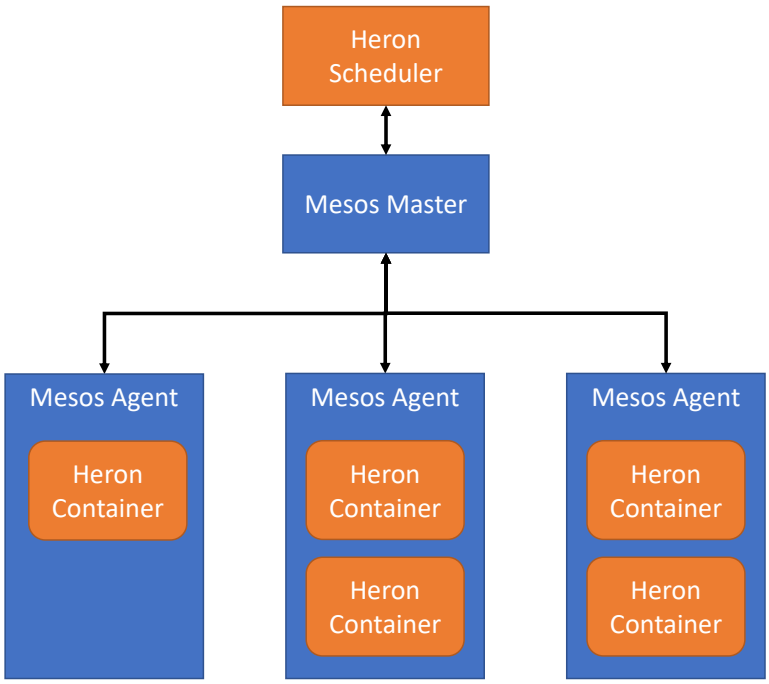



**Figure 4.1** – Architecture of Mesos components for a three-node cluster executing a Heron topology.

### 4.2.2 Resource Isolation

Mesos agents employ containerizers, which are used to run tasks in containers in order to isolate them from each other. They are used to control and restrict the resource usage of the tasks. For this implementation, Mesos is configured to use the default containerizer, which provides lightweight containerization and resource isolation of executors. It uses Linux-specific system-level techniques such as control groups (cgroups) and namespaces.

## 4.3 Twitter Heron

Heron is a soft-realtime, distributed, fault-tolerant, general-purpose stream-processing engine from Twitter. It was developed to address the limitations present in the first-generation stream-processing engine, Apache Storm, previously used at Twitter. Namely, the need arose for a system that provides better scalability, debugability, performance and is easier to manage within a shared cluster infrastructure [Kul+15]. Heron is used at Twitter as a de-facto replacement for Storm, as it

outperforms it in several benchmarks [Fu+17]. The API was designed to be backwards-compatible with Storm to ease the migration process for both Twitter and other adopters.

Heron is written in a mixture of C++, Java, and Python code. C++ and Java are being used to implement the main components, whereas Python is used to build the tools and interfaces facing the end-user. Topologies are written using either Java or Scala and are run on the JVM. Communication between the components across a cluster is achieved via asynchronous network communication over TCP/IP. Thus, Heron is only soft-realtime as it cannot provide strict upper bounds on the time it takes to process tuples and produce an output.

### 4.3.1 Heron Topologies

Much like any streaming topology, Heron topologies are organized as direct acyclic graphs processing streams of data. They consist of three basic components: *spouts* and *bolts*, connected via streams of *tuples*. Spouts are responsible for emitting and feeding tuples into the topology. They usually receive their data from an underlying streaming layer (e.g., a queuing system such as Kafka, Kinesis or ZeroMQ), read tweets from the Twitter API, or generate tuples on the fly – as is the case for most benchmarks. Heron bolts consume the stream of tuples emitted by the spouts and perform user-defined operations on those tuples, among them transformations and storage operations, aggregating multiple streams into one and emitting new tuples to other bolts within the topology.

### 4.3.2 Architecture

Fundamentally, Heron topologies are a collection of processes that are distributed and run on the servers of the cluster. Heron relies on the underlying cluster manager to provide isolation and resource provisioning.

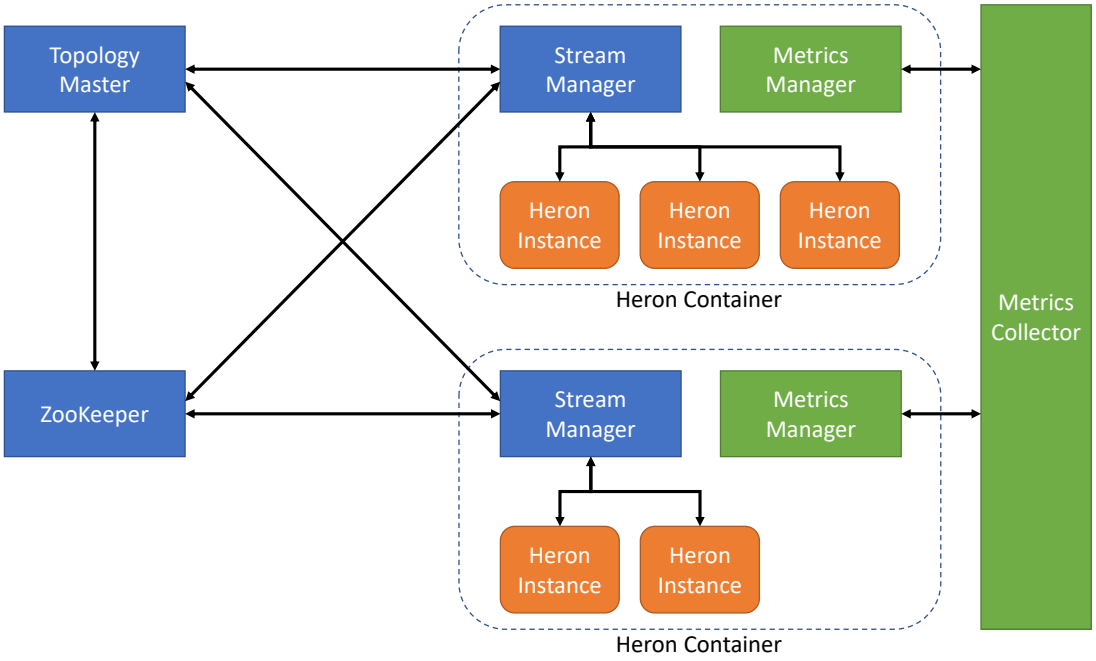



**Figure 4.2** – Architecture of Heron components.

A Heron topology is run as an isolated job consisting of several containers. The first container always runs the Topology Master (TM), the remaining containers each run a Stream Manager (SM), a metrics manager, and a number of Heron Instances (HIs). Figure 4.2 shows the architecture of a Heron topology. In the following, we will describe Heron components and core concepts.

#### 4.3.2.1 Topology Master

The TM is responsible for managing the streaming topology throughout its entire lifecycle, from the time it is initially submitted until it is ultimately stopped. Upon startup, the TM registers itself with the ZooKeeper daemon by creating an ephemeral node at a location defined by the name of the topology. For fault-tolerance, an additional TMs can be created in standby mode, however only one of them can be the main master at a time – a constraint guaranteed by the ZooKeeper node. The TM also acts as a gateway for topology performance metrics and was explicitly designed to not be part of the data processing path to prevent it from becoming a bottleneck.

The TM also creates and manages the logical and physical topology plan and relays it to the SM of each container.

#### 4.3.2.2 Heron Container

Every topology consists of multiple containers, each containing one or more HIs, a SM, and a metrics manager. Containers act as units of scheduling for the underlying cluster manager, requesting a number of CPU, RAM, and storage resources configured before topology submission. Heron containers are directly mapped to Mesos containers. Mesos uses the CPU, RAM, and storage resources assigned to each container to schedule it on whichever servers can fulfill these requirements.

#### 4.3.2.3 Heron Instance

Heron instances are JVM processes that execute the actual user code. Each instance either represents a bolt or a spout of the physical topology. Having a single process per spout or bolt makes it easier to debug or profile, since the topology writer can trace the events and logs originating from an HI. This design choice is a major benefit compared to an implementation where the work is hidden behind layers of abstraction or inside a process consisting of multiple spouts/bolts.

HIs are implemented using a two-threaded approach: a *gateway* and a *task execution* thread. The two threads communicate between themselves using two unidirectional queues. The gateway thread opens a TCP connection to the SM of the same container and controls the transfer of tuples to- and from the local SM. The gateway thread pushes tuples it received from the stream manager to the *data-in* queue of the task execution thread. This thread is now dequeuing tuples and executing the actual user code. It emits tuples via an outbound *data-out* queue, which is used by the gateway thread to send tuples to other parts of the topology.

These queues are at the core of the back-pressure mechanism. Both queues are bounded in size. The gateway thread stops reading from the local SM when the data-in queue exceeds this bound, causing the back-pressure mechanism to be triggered at the SM. When the amount of tuples causes the data-out queue to exceed its bound, the gateway thread cannot send more data to the SM and should stop feeding tuples to the execution thread.

#### 4.3.2.4 Stream Manager

The SM manages the routing of tuples between topology components. It is connected to all HIs within the same container and to all SMs of other containers. Tuples transferred from one HI to another within the same container are not routed via the SM but instead sent directly.

Besides functioning as a routing engine for tuples, SMs are responsible for propagating back pressure within the topology when necessary. In Heron, back pressure is detected when the tuple receive buffer of an instance is filling up faster than it can emit tuples. It is mitigated by first clamping down container-local spouts connected to the bolt under stress and eventually a special start back-pressure message is propagated to other SMs which in turn clamp down their spouts to reduce the new data that is injected into the topology. As the slow instance catches up with processing buffered tuples, a stop back-pressure message is sent to the SMs, causing them to resume processing data from local spouts again.

#### 4.3.2.5 Metrics Manager

The metrics manager collects general performance and custom, user-defined metrics from the SM and HIs in the container. The metrics include topology throughput, total time spent under back pressure per component, the execute latency of each bolt, CPU usage of the HI JVM processes, HI RAM utilization, and time spent doing garbage collection per minute. These metrics are reported to both the TM and any external collector.

The rate at which these metrics are sent from components to the metrics manager is configurable; the default value is once every 60 seconds. Metrics such as the throughput in tuple per second is calculated inside the sink bolts once every second, so before sending it to the metrics manager it has to be aggregated by calculating the mean value over a period of 60 seconds. For our implementation, we have set this reporting interval to 5 seconds to receive more granular and up-to-date information about the performance state of the streaming application.

### 4.3.3 Modifications to Twitter Heron

One goal of our design and implementation was to keep modifications made to the DSPE and cluster manager as minimal as possible. Our solution can generally be used and evaluated using the latest versions of Heron and Mesos respectively. However, we made some modifications to stock Heron [1] to achieve more fine-grained control over the resources used by the Heron containers and instances. In the following, we describe the internal components we modified and detail the modification itself.

#### 4.3.3.1 Packing Algorithm

When implementing a topology, the developer has to manually specify the total number of containers required, as well as the number of physical instances for each spout and bolt. When the topology is submitted, a packing algorithm places the instances in the containers. The default packing algorithm assigns instances to each container one by one in circular order. If the number of instances is a multiple of the numbers of containers, then each container receives an equal number of instances.

Based on this default implementation, we built a custom manual packing algorithm which allows us to explicitly specify the container an instance should be placed in. This is necessary because we need to create heterogeneous containers that only contained specific parts of the physical topology.

[1] Based on Heron source code [Twi17b] at commit 731783c9555a1ae84447d6dae8f2b6741a834d93 of the master branch.

In combination with the customized scheduler, this allows for the placement of specific instances on the selected servers.

This granular control over the dimensioning and placement of containers in the cluster is important for evaluating our approach in the face of heterogeneous workload distribution. When dimensioning the multi-lane topologies described in Section 5.3.2, we use the manual packing algorithm to pack containers such that the physical instances corresponding to each lane of the topology are placed in their own container. The scheduler then executed these containers on distinct physical servers. The power-capping implementation we use only allows us to cap entire servers (more specifically: their CPU power consumption), so this strategy allows us to evaluate the effectiveness of our solution in finding distinct power cap levels for a cluster where each server is processing a different workload.

#### 4.3.3.2 Scheduler

The Heron scheduler communicates with the underlying cluster manager to distribute and execute the topology among the physical servers. Since the underlying cluster manager we are using is Mesos, we had to modify the Mesos-specific Heron scheduler. By default, the scheduler places containers on whichever server first sent an offer to Heron and satisfied the resource requirements of the container. We modified the scheduler to become more deterministic in its placement of containers by ensuring that a container with a given identifier is always placed on the same physical server. Since the packing algorithm allows for the placement of HIs in specific containers, this custom scheduler eventually places the instances on whichever server we request. The scheduler also ensures that the evaluation results are reproducible for subsequent executions of the same topology.

#### 4.3.3.3 Stream Grouping

We implement a custom *container-local grouping* data-partitioning strategy based on shuffle grouping described in Section 2.1.3. If the target bolt has one or more instances in the same container, tuples will be shuffled to just those. Otherwise, it acts like a normal shuffle grouping. This implementation is similar to the *local or shuffle grouping* available in Storm. Note that the modifications to the Heron packing algorithm outlined in Section 4.3.3.1 were a prerequisite for the implementation of this grouping strategy, as Heron does otherwise not provide any interfaces to determine the instance–container mapping during runtime.

#### 4.3.3.4 Component CPU Resources

When configuring a Heron topology, the developer can manually specify the total number of instances of each component as well as the amount of RAM to be allocated for each instance. However, Heron does not provide a way to manually configure the number of CPU cores required for running each instance and instead defaults to one core. We address this inconsistency and implement a configuration parameter for the number of CPU cores required by each component. This allows us to increase the number of instances per container, which in turn increases the total throughput by making each server work to capacity.

### 4.3.4 Tuning Heron Topologies

When developing a Heron topology, the programmer has to manually configure the bolt and spout parallelism (number of instances), the amount of RAM and CPU cores for each instance, the total number of containers and the placement of instances in them. The goal of tuning a topology by

adjusting all these parameters is maximizing the possible tuple throughput and increasing resource efficiency. The information outlined in this section helps understand how the physical topologies of the streaming applications we used for the evaluation were created.

In our experience, the most essential part of tuning a topology is visualizing throughput, back pressure and custom metrics like the size of lookup tables or the activity of the garbage collector. Additionally, cluster performance metrics such as CPU and memory use offer important insights when configuring topologies for optimal resource utilization and maximal throughput. For this reason, we set up a Graphite [Pro17] time series database for storing Heron metrics, deploy Diamond [Dia17] to collect server system metrics and finally visualize this data using a Grafana dashboard [Lab17]. Grafana is a web-based data analytics and monitoring platform; we use it to visualize the time-series data stored in the Graphite database. Figure 4.3 shows the view of this dashboard.

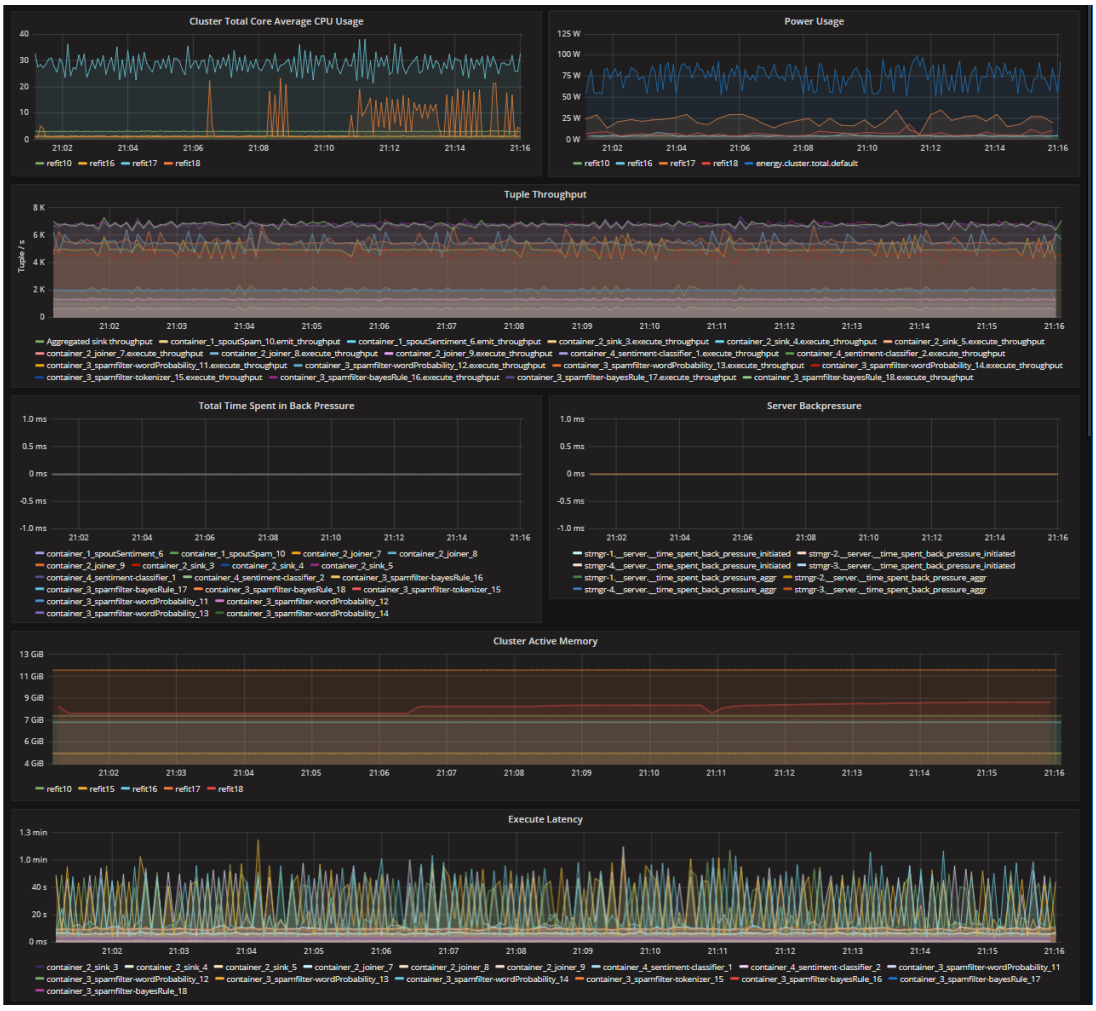



**Figure 4.3** – View of the Grafana Dashboard in the web browser. From top left to bottom: cluster server CPU usage, cluster and CPU power usage, topology throughput by PE instance, back pressure by component id, back pressure by stream manager, server RAM utilization, component execute latency.

Fundamentally, we followed these five steps when dimensioning topologies:

1. Configure the topology with an initial estimate of the resource requirements, based on input data size, component logic, and experience. Launch the topology with this configuration.

2. If any back pressure is reported, increase the component parallelism, container count, RAM or CPU appropriately to resolve it.

3. Increase the spout emit rate if the back pressure was resolved, decrease it if not

4. Repeat steps 2 and 3 until there is no back pressure - with the goal of maximizing the spout emit rate

5. At this point, the CPU and RAM usage are stable, the throughput is maximized, no back pressure is observed.

Besides following these steps, when implementing the topology it is important to avoid blocking operations (see Section 2.1.1). For example, Heron redirects any `stdout` output to log files on disk, so using console output for debugging should be avoided in production.

While these steps may seems simple and intuitive at first, it takes a significant amount of time to get a topology to maximize its data processing rate and optimize its resource utilization. The steps above describe the fundamental process of tuning a topology, but some more intricate details of Heron may further limit the potential performance if disregarded. For example, each container houses a single stream manager responsible for routing and (de-) serializing tuples. The stream manager is a single-threaded process and can turn into a bottleneck if its capacities are exceeded. This problem can be alleviated by increasing the number of containers – not necessarily changing the total instance count. This allows for tuples to be distributed among them and decreasing the individual workload.

The topologies we implement are not processing data from real-world sources, but instead tuples are synthetically created in the spouts by application logic at a configurable rate. It is important to not leave this spout emit rate unbound, but instead set it to a fixed rate and incrementally increase it as the other parameters are tuned to maximize throughput.

## 4.4 Cluster Energy Regulator

As described in Section 3.2.1, the cluster energy regulator is the key component of our solution. It is implemented as a .NET Core application written in C#. The program queries topology performance metrics from the Graphite time series database via the Graphite rendering API and uses it as input for the control algorithm.

The control algorithm uses the metrics provided to determine whether the topology workload has changed in either direction. If it has, all current power caps are removed, causing the servers to process the data uncapped. After a waiting period has elapsed, the capping strategy is executed with the goal of lowering the power caps to reduce the overall energy usage. The waiting period is mostly required if the workload increased compared to the last observation point: a relatively higher load at the same power caps may cause back pressure to occur. A waiting period before starting the capping procedure ensures that the topology has reached a state where the new throughput is stable and no back pressure is present.

### 4.4.1 Capping Strategies

The capping strategy is an algorithm executed every time a workload change has been detected, with the goal of finding the lowest possible power cap for each server in the cluster so that the workload can still be processed.

**Require:** $H$: set of all servers in the cluster
$t_{sleep}$: period to wait until re-observing the effects of setting the power cap
$\Delta P_{cap}$: value to decrement the power cap for each loop iteration
$\Delta T$: throughput deviation threshold. Used to indicate when the degradation in performance is no longer acceptable

```
1:  throughputBaseline = QueryMetricsDBForCurrentThroughput()
2:  for each host ∈ H do
3:    lastCapping[host] = ∞
4:    while true do
5:      currentPower = QueryMetricsDBForCurrentPowerUsage(host)
6:      newCap = currentPower - ΔP_cap
7:      PowerCapHost(host, newCap)
8:      Sleep(t_sleep)
9:      currentThroughput = QueryMetricsDBForCurrentThroughput()
10:     throughputChange = currentThroughput / throughputBaseline
11:     if throughputChange > ΔT then
12:       PowerCapHost(host, lastCapping[host])
13:       cappingResult[host] = lastCapping[host]
14:       break
15:     end if
16:     lastCapping[host] = newCap;
17:   end while
18: end for
```

**Algorithm 4.1** – Pseudocode implementation of the linear-capping strategy.

Algorithm 4.1 shows the pseudo-code implementation of a *linear-capping strategy*. The algorithm attempts to cap each server in the cluster one after the other, lowering the cap until it eventually deteriorates the throughput. The cap set just before this point is then used as the final capping result for this host and stored in the *cappingResult* map. The algorithm is called *linear*-capping strategy, as it iterates through the set of hosts in a linear (or sequential) fashion, observing the effects of setting the cap for each host in isolation.

The algorithm stops capping a host, when the observed throughput is lower than the baseline value recorded before the capping process began. Lowering the amount of power available to a CPU causes its performance to drop. When exceeding the idea power cap, this lowers the throughput of bolts or spouts executed on it. The performance degradation caused by power capping can result in bolts no longer being able to keep up with the workload and entering a back-pressure state. This in turn causes the spouts to stop emitting tuples, resulting in the total throughput to drop significantly. Naturally, throughput dips caused by back pressure are more severe than those caused by tuples simply being processed slower.

### Capping Order

The order in which the nodes of the cluster are capped generally does not have a significant effect on the total power savings. However, the first node chosen to be capped is able to completely exhaust the throughput deviation threshold, leaving less leeway to other servers. The power cap of the first server can be set as low as to deteriorate the throughput by the full $\Delta T$, thereby preventing the remaining servers from reducing the total throughput at all. This means that the first server to be capped potentially exhibits the largest energy savings.

### Parallel Capping

An alternative algorithm would be the *parallel-capping strategy*, where the order of the for-each- and while-loop are switched: in a single loop, all the power caps of the host are decremented at once. However, an additional step is required to find the host responsible, if the throughput is deteriorated: it is required to determine which of the servers capped in parallel was responsible for the throughput decrease. For this purpose, the algorithm tracks the power cap set in the current iteration (that may have caused throughput to drop) as well as the power cap set one iteration prior (where the throughput was not affected). It iterates over all hosts, sets their power cap to the one set in the previous iteration and check whether the throughput has recovered. If it has, the algorithm knows that the host which was just re-set was responsible, and that its ideal power cap was the one of the previous iteration.

### Settling Time

To evaluate the differences in settling time between the two capping strategies outlined above, we have formulated the runtime as mathematical expressions. Fundamentally, the settling time of the capping strategies are a function of the amount of power saved, the size of the capping step, and the sleeping duration between caps. $N$ is the number of servers to cap, $P^i_{savings}$ is the total amount of power saved when capping the $i$-th server (i.e., the difference between the power usage of the server when fully power capped and when running without caps), $\Delta P_{cap}$ is the value to decrement the power cap for each loop, and finally $t_{sleep}$ is the period to wait until re-observing the effects of decrementing the power cap. Both formulas assume a-priori knowledge about the total amount of energy saved.

$$t_{linear} = \sum_{i=0}^{N} \frac{P^i_{savings}}{\Delta P_{cap}} \cdot t_{sleep} \tag{4.1}$$

In the linear case (Equation 4.1), every time the power cap is decremented by $\Delta P_{cap}$, the algorithm waits for $t_{sleep}$ seconds before lowering the cap. This is repeated until the final cap is found, $\frac{P^i_{savings}}{\Delta P_{cap}}$ times. The total runtime is the sum of the individual capping duration for each server.

$$t_{parallel} = \sum_{i=0}^{N} (N-i) \cdot t_{sleep} + t_{sleep} \cdot \max\left(\frac{P^j_{savings}}{\Delta P_{cap}}\right) \tag{4.2}$$

For the parallel strategy, this total runtime differs as a result of capping multiple hosts simultaneously. Equation 4.2 describes the runtime of the parallel strategy. It consists of the time required to linearly cap the server with the largest energy savings ($t_{sleep} \cdot \max(\frac{P^j_{savings}}{\Delta P_{cap}})$), plus the time required to find the culprit responsible for the drop in throughput among the $N$ hosts ($\sum_{i=0}^{N}(N-i) \cdot t_{sleep}$).

Figure 4.4 shows the settling times calculated using the formulas of the linear and parallel capping strategies at three different absolute cluster energy savings. At smaller capping step, and particularly as the energy savings increase, the parallel strategy provides lower settling times.

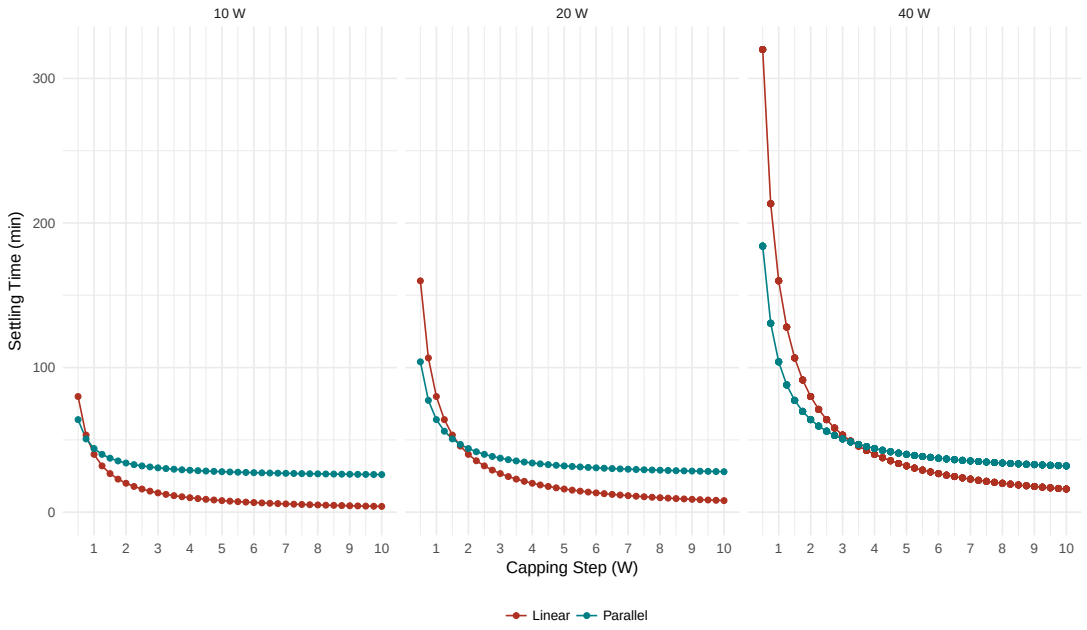



**Figure 4.4** – Settling time of the linear- and parallel-capping strategies at different capping steps ($\Delta P_{cap}$) for three aggregated energy savings ($\sum_{i=0}^{N} P^{i}_{savings}$ = 10 W (left), 20 W (middle) and 40 W (right)), $N = 3$ and $t_{sleep} = 5$ min

Generally, the number of possible capping strategies is nearly endless, differing in ease of implementation, accuracy and settling time. For this work, both the linear and parallel capping strategies provide sufficiently low settling time, considering our choice of parameters and size of the cluster used.

#### 4.4.1.1 Configuring Parameters

Both the control algorithm and the capping strategy used provide a variety of parameters that can be adjusted to reduce the settling time and increase the accuracy of finding ideal power caps. Below, we will focus on the three most relevant ones for the capping algorithm.

**Power-Capping Step**

$\Delta P_{cap}$ is the amount of energy to reduce the power cap by, during each iteration of both capping strategies described in Section 4.4.1. It is a means for describing the accuracy of the algorithm: if the power cap is limited linearly, then each iteration converges to the lowest power cap by $\Delta P_{cap}$. Setting this capping step to a larger value causes the final cap to be reached quicker, but the chances of overshooting the goal are higher. A small step guarantees that when the power cap is set below a value that can sustain the given throughput, it does not drop too significantly, as the throughput deteriorates proportionally with the amount of energy reduced below the ideal power cap.

#### Capping Evaluation Time

$t_{sleep}$ is the time the algorithm should wait after setting a new power cap before evaluating its effect on the topology throughput. It should be set to the amount of time it takes the topology to reach stable state after a capping or workload change occurred. It is used to mask any noise when calculating the mean throughput. It is both linked to the value of $\Delta P_{cap}$ and the accuracy of the throughput metrics available. Throughput metrics are periodically reported to the database, so a longer sleeping or decision period increases the accuracy of the reported effect of changing the power cap or workload. This is caused by the longer sleeping period counterbalancing any noise present in the reported throughput, as it is calculated based on more data points. The $t_{sleep}$ period also increases the settling time, as it delays each capping change.

#### Throughput Deviation Threshold

The throughput deviation threshold ($\Delta T$) is a configuration parameter that determines when the performance degradation caused by reducing the server power limit is no longer acceptable. For example, if $\Delta T$ is set to 5 %, and lowering the power cap by $P_{cap}$ during an iteration of the linear-capping strategy causes the throughput to dip by 6 %, then the capping process for the current server is stopped. When the capping strategy has completed execution, the reduction in throughput caused by power capping can at most be $\Delta T$. This threshold acts as an exit condition for the capping strategy but is not an upper limit for the throughput degradation caused while the algorithm is running: capping beyond the ideal power cap can still cause back pressure to reduce the throughput to 0, albeit at most for the time period indicated by $t_{sleep}$. Increasing $\Delta T$ can yield higher energy savings as the power caps can be reduced further without exceeding the threshold, however, this also means the performance degradation can potentially be larger.

In a pull-based stream-processing engine like Heron, a decrease in throughput does not necessarily mean that data or tuples are dropped. If the underlying streaming layer supports buffering or queuing of raw data before being polled by the spouts, then the size of these buffers simply increases as the throughput is limited via power capping.

### 4.4.1.2 Detecting Workload Changes

Changes in the workload are detected by querying the topology throughput and the back pressure reported by each component. If the throughput decreases compared to the last observation, then it is taken as an indication that the total workload has decreased. To avoid misreporting a workload change if the throughput decrease was caused by a power cap being set too low, the spout emit rates are queried as well and all must report the same decrease. Furthermore, it is recommended to set the throughput deviation threshold $\Delta T$ used by the capping strategy to a smaller value than the threshold used by the workload change detector, use a smaller value for $t_{sleep}$ and a reasonably small $\Delta P_{cap}$. This is an important aspect of the configuration to avoid the workload change detector interpreting a decrease in throughput caused by power capping as a natural decrease in workload caused by external factors. The workload change detector queries the mean throughput over its observation period and uses it to detect changes. Setting $t_{sleep}$ lower than this observation period results in dips in throughput caused by capping to be smoothed out over time as the caps are raised to non-deteriorating levels and does not affect the workload change detection.

If the throughput has increased, an increase in the workload is assumed. Setting power caps does not just limit the peak power consumption of the cluster, but also limits the peak performance available to applications. Consequently, the throughput may not increase beyond what is currently

possible given the set power caps. To be able to detect workload changes despite these limitations, the actual caps used for each server are offset by a few watts.

If the cluster energy regulator has detected a change in workload – in either direction –, it cancels the capping process running at the time, if any, and removes all power caps. After an interval has elapsed to determine the new baseline throughput and energy usages, the capping algorithm is executed based on these new observations.

### 4.4.2 Capping Cache

In a streaming application that relies on user-generated data for processing, it is very likely for workloads to repeat themselves over time. For example, during a soccer match, when a goal is scored, the number of tweets about it increases rapidly and unpredictably, but settles to previous levels after some time [OED12; Abe14]. This can be taken advantage of, by storing throughput and the associated power caps for the cluster in a cache. The capping strategy can use this cache to lookup the starting power caps for each host during the first iteration. If the exact throughput does not exist as an entry in the cache, the caps for the closest throughput larger than the lookup value can be used.

## 4.5 Local Energy Regulator

The local energy regulator is the slave component of our master–slave architecture. It is running on each server of the cluster and communicates with the master energy regulator.

### 4.5.1 Implementation

The local energy regulator is implemented as a .NET Core application running on each server of the cluster. It uses ASP.NET to implement a REST server that receives POST commands from the cluster energy regulator containing the power caps to set. When a command is received, the requested power cap is applied by invoking Intel RAPL via its sysfs interface. The cap is set on the package domain of the CPU, which includes all power consumers placed on the die. To ensure that the power cap “sticks” and the Linux kernel does not remove the cap for some reason, the commands to set a power cap are sent periodically by the cluster energy regulator. The regulator slave does not maintain any kind of internal state of the current cap, but simply executes the power-capping request every time it is received.

### 4.5.2 Intel RAPL

Intel RAPL (Running Average Power Limit) is an interface available in modern Intel CPUs providing mechanisms to enforce power consumption limits [Int17]. It can be used by applications to not only gain more deterministic control over the power consumption of the system, or manage thermal properties, but also to measure the energy usage of the processor and memory domain.

The interface exposes functionality to measure and cap the power usage of different parts of the machine – called *power domains*. Figure 4.5 depicts these domains and their logical placement on the CPU die. For our purposes, the *package* or *PKG* domain is the most relevant: it refers to the entirety of the CPU package including its cores, caches and additional components like an integrated graphics-processing unit.

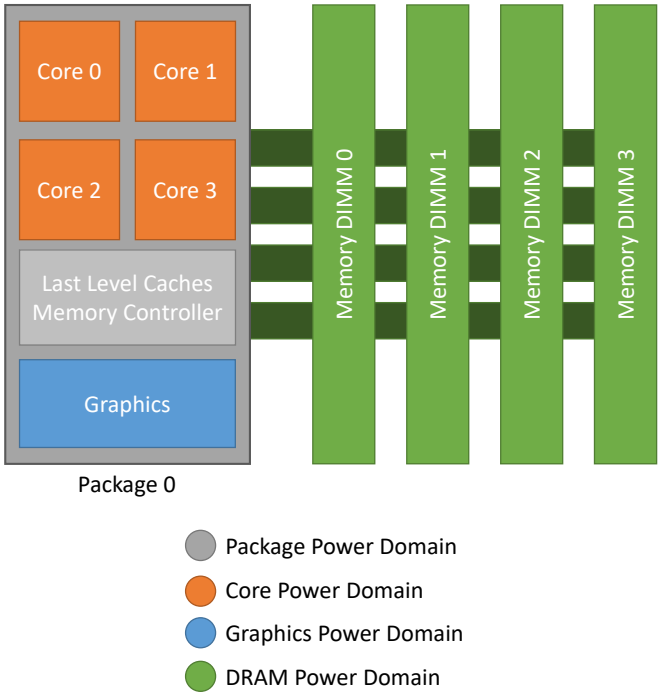



**Figure 4.5** – Intel RAPL power domains for which power monitoring and -control are available on a single-socket CPU. The figure is based on [Dim+12].

The local energy regulator uses RAPL to set power caps for the processor (the PKG domain) of the server it is running on. It is also used to measure the energy consumption of each server. We communicate with the RAPL subsystem using the power-capping framework provided by the Linux kernel [ker13]. The power-capping framework provides an interface between kernel and user space and exposes the RAPL power-capping device to user space via sysfs.

## 4.6 Summary

In this chapter, we described the implementation of our scheme for increasing energy efficiency in distributed stream-processing systems in detail. We detailed our solution architecture and how its components interact with Heron – the streaming engine on top of which we built our system. The cluster energy regulator represents the central component of our design: it leverages its view of the total cluster performance to orchestrate local energy regulator to apply power caps Capping strategies are used to incrementally reduce the power consumption of the cluster and form the core of our solution.

# 5 APPLICATIONS

In this chapter, we first recall the fundamentals of topologies and briefly describe how they are implemented using Heron. Afterwards, we present the two fundamental categories of streaming topologies we analyzed in this work. Their individual use cases, logical structure, and functionality are described in detail. We also consider the role of back pressure in the case of multi-lane topologies and its effect on application performance when power capping is applied.

## 5.1 Overview

Applications for data-stream–processing systems are called topologies and are organized as a directed acyclic graph of spouts and bolts. This graph provides the top-level abstraction that is submitted to the streaming cluster for execution. Spouts are injecting tuples into the topology by reading data from an underlying streaming layer, packaging it as tuples, and sending it to the adjacent bolts. Bolts are implementing functions that transform the tuples received from other bolts or spouts and emit new tuples themselves.

Topologies can be considered a representation of the computation to be performed on data injected by spouts. When designing a topology, the developer is aware of the underlying business data (e.g., Tweets, stock market transactions, or website impressions) and has a goal in mind (e.g, calculating tweet sentiment, finding the best time to buy a stock, or count website impressions by region). The developer now has to devise a set of transformations which process the input data and generate the desired result. Complex stream transformations require multiple steps and thus multiple bolts. Bolts can run arbitrary functions on input data, do streaming aggregations, do streaming joins, query or perform transactions on external databases, and more.

## 5.2 Heron Topologies

Heron topologies can be written in Java and are compiled into JAR files that are distributed among the servers of the cluster when the topologies are submitted. The Heron API provides interfaces for bolts and spouts that are implemented by user-defined classes encapsulating the processing logic. They override methods that are called when a tuple is available for processing or new data tuples can be emitted respectively. The API further provides a topology builder class, which is used to declare, configure, and build the topology before submission to the Heron cluster. It is used to connect processing elements among each other, define the number of containers and physical instances, and configure resource requirements. For each bolt, its connections to other bolts or spouts are declared by specifying the upstream bolt and the desired stream grouping. Before submission, the topology

builder also validates the integrity of the topology by, among others, ensuring the connectedness of the DAG formed by the logical topology.

To better illustrate the functionality of Heron from the perspective of a topology developer, two code snippets are presented in the following. The Listings 5.1 to 5.2 below show the simplified implementation of the `SentenceWordCount` topology (see Section 5.3.1.1) in Java using the Heron API. This topology splits a sentence emitted from the spout into words and finally counts the total number of occurrences for each word. Listing 5.1 shows the implementation of the `Tokenizer` bolt which receives a sentence and splits it. The `execute` method (line 4), is called whenever a new tuple is received by the bolt. The `tuple` parameter contains all attributes as a list of key-value pairs. The `Tokenizer` bolt accesses the sentence emitted from the spout by calling the `getString` method on the received tuple using "sentence" as the key (this key has previously been defined in the spout). The sentence is now split by the whitespace character (line 6) and each word is then emitted as a new tuple by invoking the `emit` method on the `BasicOutputCollector`. The `declareOutputFields` method (line 12) allows the bolt to specify the structure of the tuple it emits, more specifically, it is used to set the keys of the attributes emitted.

Listing 5.2 shows the complete implementation of the `main` method of the topology Java program. It is used to build the topology by connecting spouts to bolts and bolts to each other, configuring the number of physical instances, and setting the resource requirements of each PE. The `TopologyBuilder` class is used to build the topology by registering spouts and bolts via the `setSpout` and `setBolt` methods respectively. Each logical PE is identifiable via a user-specified string (e.g., `"sentenceSpout"`), this name is used to setup the groupings between the elements. For example, lines 15-16 show how a `WordCount` bolt is added to the topology using the name `"count"` and a requested physical instance count of two. It is connected to the upstream `Tokenizer` bolt named `"tokenizer"` using fields grouping on the attribute `"word"`. Per-component and per-container resources are set using the `Config` class. Finally, the topology is submitted to the streaming cluster with a call to `submitTopology` of the static `HeronSubmitter` class in line 36.

```
1  public static class TokenizerBolt extends BaseBasicBolt {
2
3      @Override
4      public void execute(Tuple tuple, BasicOutputCollector collector) {
5        String sentence = tuple.getString("sentence");
6        for (String word : sentence.split("\\s+")) {
7          collector.emit(new Values(word));
8        }
9      }
10
11     @Override
12     public void declareOutputFields(OutputFieldsDeclarer declarer) {
13       declarer.declare(new Fields("word"));
14     }
15   }
```

**Listing 5.1** – Code listing of the `Tokenizer` bolt.

```
public static void main(String[] args) {

    // the number of physical instances per logical PE
    int instanceParallelism = 2;

    TopologyBuilder builder = new TopologyBuilder();

    // the sentenceSpout instance parallelism is set to 1
    builder.setSpout("sentenceSpout", new RandomSentenceSpout(), 1);

    // connect the tokenizer bolt to the spout via shuffle grouping
    builder.setBolt("tokenizer", new TokenizerBolt(), instanceParallelism)
        .shuffleGrouping("sentenceSpout");

    // connect the consumer bolt to the spout via fields grouping on the
    // attribute "word"
    builder.setBolt("count", new WordCountBolt(), instanceParallelism)
        .fieldsGrouping("tokenizer", new Fields("word"));

    Config conf = new Config();

    // sets the total number of containers
    conf.setNumStmgrs(2);

    // component resource configuration
    Config.setComponentRam(conf, "sentenceSpout",
        ByteAmount.fromMegabytes(512));
    Config.setComponentRam(conf, "tokenizer",
        ByteAmount.fromMegabytes(512));
    Config.setComponentRam(conf, "count",
        ByteAmount.fromMegabytes(512));

    // container resource configuration
    Config.setContainerDiskRequested(conf, ByteAmount.fromGigabytes(3));
    Config.setContainerRamRequested(conf, ByteAmount.fromGigabytes(3));
    Config.setContainerCpuRequested(conf, 2);

    HeronSubmitter.submitTopology("SentenceWordCountTopology",
        conf,
        builder.createTopology());
}
```

**Listing 5.2** – Code listing of the Java topology application entry point.

## 5.3 Implemented Topologies

The topologies we implement for the purpose of later evaluating their energy-savings potential are chosen to reflect a broad spectrum of streaming use cases: from simple word counting over algorithmic trading to identifying spam and detecting denial-of-service attacks. Fundamentally, we discuss two types of topologies: single- and multi-lane. In single-lane topologies, each distinct tuple emitted by the logical spout yields one result being output at the end. In multi-lane topologies, the total logical topology is partitioned into two or more topology sub-graphs, whose streams are joined at a common downstream bolt at the end. Here, to output a final result, each tuple needs to be processed by the two sub-topologies.

Subsequently, we describe the topologies we develop for both types in detail, starting with two single-lane topologies, followed by two multi-lane topologies. For the latter, the impact of back pressure plays an important role, which we will discuss in detail.

### 5.3.1 Single-Lane Topologies

Single-lane topologies are characterized by having a single (logical) spout feeding tuples into the other bolts constituting the topology. Each distinct tuple emitted by the logical spout yields one result being output at the end. In single-lane topologies, no sub-graphs can be identified, whose processing of tuples needs to be awaited at a common bolt.

In the following section, we present the two single-lane topologies we implemented and used for the evaluation in Chapter 6. Note that each topology is completed using a sink bolt, it does not do any processing but is used to report the total topology throughput.

#### 5.3.1.1 SentenceWordCount

Word count topologies can be considered the ‘Hello World‘ of stream-processing engines; they can be found in the sample section of any major stream-processing framework – including Storm and Heron. The `SentenceWordCount` topology counts the total number of occurrences of each word in sentences provided as input by the spouts. Figure 5.1 shows the logical view of this topology. Sentences are created in the `sentenceSpout` synthetically by randomly picking ten words from a pre-generated list. The subsequent `tokenizer` bolt splits the sentences received from the spout by the whitespace character using a regular expression, thus extracting the actual words. The `count` bolt store every word in a local table that maps words to their total number of occurrences so far. This bolt also emits the received word and its counter to the final `sink` bolt.

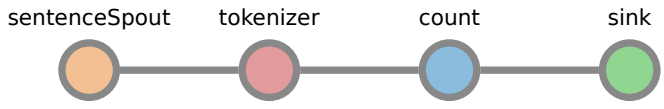



**Figure 5.1** – Logical topology of the SentenceWordCount application.

#### 5.3.1.2 BargainIndex

Volume-weighted average price or VWAP [BLN88] is a common calculation in financial trading – particularly algorithmic trading [Len11]. The VWAP is a trading benchmark that is calculated by adding up the number of shares bought per transaction multiplied by the share price over a specific

trading period and dividing it by the total shares traded over the period. It is the average price weighted by the trade volume.

$$\text{Volume-Weighted Average Price}_j = \sum_{i=1}^{N} \frac{(\text{Vol}_{ij}) \times (\text{Price}_{ij})}{\text{Vol}_j} \tag{5.1}$$

Equation 5.1 shows how the VWAP metric is calculated. Each transaction during the trading period $j$ is subscripted $i$. $\text{Vol}_{ij}$ is the number of shares in the $i$th trade, and $\text{Price}_{ij}$ is the price at which the $i$th transaction was performed. $N$ trades occurred during the trading period for which the VWAP is measured. The volume-weighted average price on any day represents the price a "naive" trader can expect to obtain.

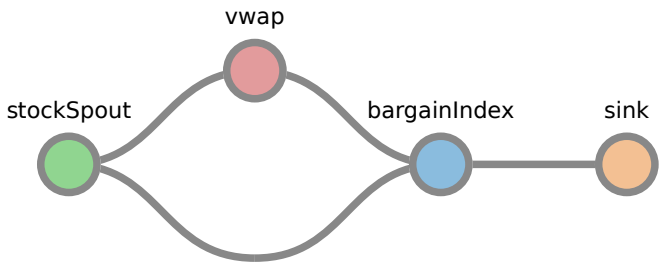



**Figure 5.2** – Logical topology of the BargainIndex application.

The `BargainIndex` topology calculates the VWAP for a stream of stock trades within a given time period (we use 1 min) and uses it to determine whether buying the stock at its current ask price reflects a bullish or bearish sentiment, that is, if the price of a buy trade is lower than the VWAP, it is a good trade. The opposite is true if the price is higher than the VWAP.

Our implementation shown in Figure 5.2 is based on the IBM InfoSphere Streams VWAP example [IBM17]. The `stockSpout` emits both stock quotes and trades to the `bargainIndex` and `vwap` bolts respectively at the same rate. The trading data emitted by the spout is based on actual information queried from the NASDAQ stock exchange on March 7th 2017 for the TSLA, AMD and GOOGL symbols. Based on this data, we simulate stock quotes ranging between 90 and 110 % of the actual current price with offered size ranging between 10 and 80 % of the volume. Note that the duration of any calculation in the topology is independent of any specific tuple value.

The `vwap` bolt calculates the VWAP by multiplying the stock price received from the upstream spout by the number of shares traded and then dividing by the total number of trades for the past minute. This bolt also updates a moving VWAP that is emitted for each trade if the trading period has not elapsed yet. The `bargainIndex` bolt stores the VWAP for each stock symbol calculated so far to determine if a stock quote received from the spout is a good or bad trade. Note that this bolt does not join or wait for tuples to be available from the two upstream nodes: if the VWAP has not been calculated yet for a stock symbol, the bolt simply emits a neutral sentiment.

### 5.3.2 Multi-Lane Topologies

Multi-lane topologies are characterized as having one or more spouts that send tuples along two or more lanes (paths). Those lanes are joined at the very end of the topology. It consists of two or more single-lane topologies forming multiple paths that are processing tuples independently.

This type of topology is useful when the same kind of data can be processed independently, but each processing result is part of a larger piece of information. A shared key or identifier must be

present in each tuple of the streaming paths to eventually be able to join the results. Each lane can either exist as a standalone topology, deployed independently and potentially even geographically separated, or they can exist as parts of a larger shared topology, deployed as a single unit.

The fundamental problem of joining two continuous streams of events exists, for example, in the advertisement platform provided by Google: when a user issues a search query (with a unique query id) and later clicks on an advertisement (with a query and advertiser id), two chronologically distinct and independently processed events occur. Google needs to eventually join both on the query id to bill advertiser appropriately. For this reason, they implemented Photon [Ana+13], a distributed system for joining multiple continuously flowing streams of data in real-time with high scalability and low latency. Our join-bolt implementation is significantly simpler: a hash-map storing the id and tuple until two tuples with the same id have arrived. Nevertheless, the Photon example proves the real-world existence of these topologies and inspired the `ClickAnalysis` topology.

If the lanes are placed on separate physical machines, they can be used to demonstrate how our approach deals with clusters experiencing heterogeneous load distribution. In the following section, we present the two multi-lane topologies that were implemented and used for the evaluation.

#### 5.3.2.1 TweetAnalysis

The purpose of the `TweetAnalysis` application is to analyze tweets to determine the sentiment and the likelihood of a given tweet being spam. The logical topology is shown in Figure 5.3. Each lane of the topology has its own spout that both emit the same tuples identified by a common id. The `joiner` bolt is used to join tuples from both lanes on this shared key. Once the join has completed, the information created in both lanes is emitted as a shared result to the sink.

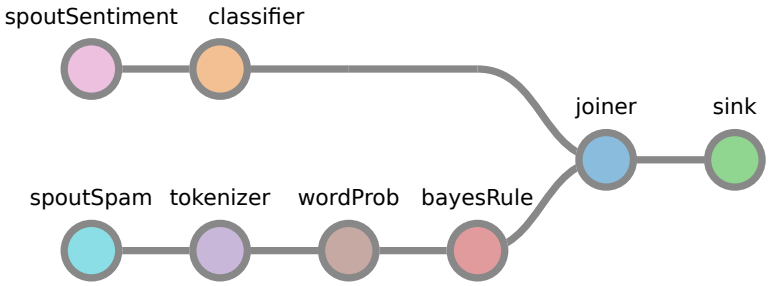



**Figure 5.3** – Logical topology of the TweetAnalysis application.

The text sentiment analysis is implemented using the AFINN-111 corpus [Nie11]. It is a simple text file listing 2477 words and phrases, manually labeled with a sentiment between minus five (negative) and plus five (positive).

Table 5.1 shows an excerpt from this corpus: in every line, the word is separated by a tab-character from the integer sentiment value. During initialization of the topology, this database is

**Table 5.1** – Excerpt from the AFINN-111 data set (Lines 407–409).

| | |
|---|---|
| ... | ... |
| clarity | 2 |
| clash | -2 |
| classy | 3 |
| ... | ... |

read from disk, parsed, and transformed into a hash map of word-sentiment pairs in each instance of the `classifier` bolt. When the `classifier` bolt receives input from a spout, it splits the tweet into words and looks up the sentiment for each one in the database. The final bias of the tweet is calculated as the sum of the individual word sentiments and emitted to the `joiner` bolt.

The spam identification sub-graph implements naive Bayes filtering [Sah+98] as a stream-processing application to label each tweet with its probability of being spam. The Bayes classification works by correlating the use of words with spam and non-spam (also referred to as "ham") text and applying Bayes' theorem to determine whether the text is spam. For the classifier to work correctly, it requires an initial training step, where a lookup table of words and their probability of appearing in spam messages (and their probability of appearing in ham messages) is constructed. We train the classifier using the TREC 2007 spam corpus [CL07], a set of 75 419 emails already labeled as either spam or ham. The first step of the topology is to split the tweet received from the spout into its words, which is done by the `tokenizer` bolt. Afterwards, the `wordProb` bolt looks up the probability of the received word being spam and emits this value to the next bolt. The final `bayesRule` bolt temporarily stores the received words until all words of a tweet have arrived and applies the Bayes theorem to calculate the probability of the tweet being spam.

To better illustrate the functionality of this elaborate topology, in the following we use the example text "Your chance to buy great replica watches online!" to show the processing steps executed. In reality, the tuples are processed in both sub-graphs simultaneously, but for demonstration purposes the processing steps in the sentiment lane are shown first, followed by the spam lane.

In the `spoutSentiment` lane, a tuple consisting of the text and an identifier is emitted:

```
(1, "Your chance to buy great replica watches online!")
```

This tuple arrives at the `classifier` bolt, is split into words, and the sentiment value for each word is looked up in the hash-map containing data from the AFINN-111 corpus. These values are then added up to create a sentiment score for the entire sentence: "chance" has a positive sentiment of 2, "great" has a positive sentiment of 3. For the other words, no entries exist in the table and a neutral sentiment of 0 is assumed. The `classifier` bolt now emits a tuple consisting of the id, the text, and the sentiment score:

```
(1, "Your chance to buy great replica watches online!", 5)
```

This is received by the `joiner` bolt and stored in a map with the key being the tuple identifier, until data from the second lane with the same id is received.

In the `spoutSpam` sub-graph, the same tuple consisting of the text and an identifier is created and emitted. In the `tokenizer` bolt, the text is split into words, and for each word a new tuple is emitted with the same identifier as the entire sentence and an attribute indicating the total number of words in the original sentence:

`(1, "Your", 8)`, `(1, "chance", 8)`, `(1, "to", 8)`, etc.

These tuples are now sent to the `wordProb` bolt, where the probability of the individual word being spam is calculated. This is done by looking up the spam probability for each word in the word map. "Your" does not exist in this map as it is too common in the English language to have any meaningful impact on the analysis. For words not in the list, we use a default, optimistic spam probability of 40 %. The original tuples are now augmented with this information:

`(1, "Your", 8, 40 %)`, `(1, "chance", 8, 61 %)`, ...,
`(1, "replica", 8, 97 %)`, `(1, "watches", 8, 95 %)`, etc.

The `bayesRule` bolt stores these tuples until all words corresponding to the original sentence are received, using the identifier and total number of words in the sentence. Once all tuples have arrived, the Bayes rule is applied to compute the likelihood of the entire text being spam given the probability of each individual word being spam. The bolt then simply emits the tuple id and the total spam probability.

```
(1, 99%)
```

Finally, once a tuple with the same id is received from both lanes in the *joiner* bolt, all tuple attributes are merged and the final result is emitted:

```
(1, "Your chance to buy great replica watches online!", 5, 99%)
```

In our implementation, this result is simply discarded at the `sink` bolt. In a production use-case, it may be stored in a database for later statistical purposes or displayed on a dashboard.

#### 5.3.2.2 ClickAnalysis

The ClickAnalysis topology depicted in Figure 5.4 reveals insights into the behavior and origin of users visiting and interacting with web pages. The spout emits an IP address identifying the user as well as a URL of the page the user visited.

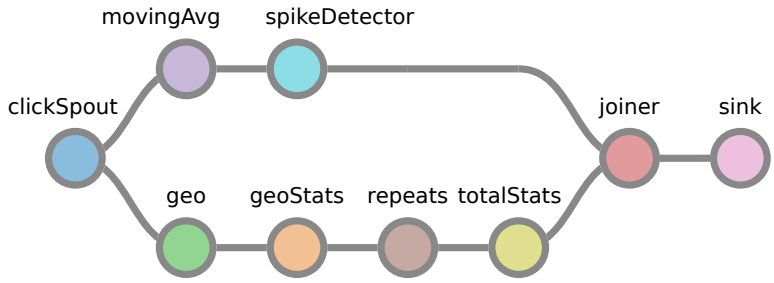



**Figure 5.4** – Logical topology of the ClickAnalysis application.

The first lane (the top one in Figure 5.4) determines whether a user is accessing the site significantly more than the average user. This information can be used to detect denial-of-service attacks. The second lane collects statistical information about the origin of the user, by using a database to lookup the geographical location corresponding to the IP-address. It also calculates how many users in total share the same geographical origin and the number of times he or she has visited the site and each URL in total. To determine the location of the user, we use the free version of the MaxMind GeoIP2 database [Max17]. It is provided as a binary file that is loaded into RAM in each instance of the geo bolt and used to look up the country code and city corresponding to a given IP address.

As previously done for the preceding topology, we will now use an example to better illustrate the functionality of this topology. The spout emits a tuple consisting of an id, IP address, and URL of the page visited. For example:

```
(1337, "198.51.100.123", "/hello/world.html")
```

The `movingAvg` bolt keeps track of the number of visits for the given IP address as well as a moving average of visits over a sliding window. The bolt increments its internal visit counter and

updates the current moving average for visits. It emits a tuple consisting of the id, visit count, and average visits during the time window:

```
(1337, 20, 30)
```

The next bolt, `spikeDetector`, now determines if the tuple visit count exceeds a threshold that indicates an unusual spike in activity for the given user. The user visit count, 20, does not exceed the moving average of visits, 30, so the bolt emits the following:

```
(1337, "FALSE")
```

`"FALSE"` simply indicates that no spike was detected.

The lower sub-graph first determines the country and city of origin in the `geo` bolt by looking up the IP address received in the input tuple in the GeoIP2 database. Country and city are added to the tuple and transferred to the `geoStats` bolt:

```
(1337, "198.51.100.123", "/hello/world.html", "DE", "Erlangen")
```

This bolt keeps track of the total number of visitors from a given country and city in its internal state. The emitted tuple contains the total number of visitors from the same country and the same city as the received tuple:

```
(1337, "198.51.100.123", "/hello/world.html", "DE", "Erlangen", 25, 11)
```

Here, 25 total users visited from an IP address located in Germany, 11 from the city of Erlangen. The `repeats` bolt simply determines if the user – identified by the IP address – has visited the same URL previously. It augments the tuple with a boolean value indicating if it was the first visit:

```
(1337, "198.51.100.123", "/hello/world.html", "DE", "Erlangen", 25, 11,
                                "FALSE")
```

Finally, the `totalStats` bolt returns the total visit and the total unique visit count for the URL:

```
(1337, "198.51.100.123", "/hello/world.html", "DE", "Erlangen", 25, 11,
                            "FALSE", 100, 70)
```

Similar to the processing in the `TweetAnalysis` topology, when tuples from both sub-graphs are received by the joiner, it merges the data and emits the final result to the sink.

#### 5.3.2.3 Back-Pressure Considerations

In multi-lane topologies, back pressure plays an important role: the joiner bolt in particular can become a bottleneck for the throughput, as its stream manager has to deserialize, and the bolt logic has to process, tuples from two or more streaming lanes at the same time. Additionally, if the lanes are processing tuples at significantly different rates, that is, if one lane has to expend significantly more work on each tuple, it is likely that back pressure is present at the slowest bolt of the overloaded lane. This causes the spouts to stop emitting tuples, which in turn degrades the total throughput further. A mismatch between the processing rates can also cause the joiner bolt to become overwhelmed as it has to store each tuple from the faster lane until a matching one from the slowest lane arrives. This causes any in-memory tables used for the join to expand in an uncontrolled fashion, eventually causing garbage collection to kick in and deteriorate performance.

Explicitly limiting the processing rate of the fastest lane in a multi-lane topology to that of the slowest one can be used to alleviate back pressure without affecting the total topology throughput. This can be achieved using a rate limiting mechanism such as token bucket. However, details regarding the implementation of a rate synchronization mechanism between sub-graphs in a multi-lane topology are beyond the scope of this work.

## 5.4 Summary

This chapter details the implementation and functionality of the streaming topologies we developed to evaluate the proposed scheme. The topologies were chosen to reflect a broad spectrum of use cases that can be found in production stream-processing deployments. The topologies are used as the basis for the evaluation and should demonstrate how our approach for improving the energy efficiency of streaming topologies works with a broad variety of applications.

# 6 EVALUATION

After detailing the fundamental design and its implementation in the preceding chapter, we now evaluate its effectiveness based on experimental results. We primarily focus on demonstrating the energy efficiency, overall energy conservation potential, and the performance impact of the approach. The setup of the cluster used and additions specific to the evaluation process are described in detail. The description of the streaming applications introduced in Chapter 5 is concluded here by specifying their physical topologies. Finally, we briefly demonstrate how changing the stream-grouping algorithm can also lead to energy savings.

## 6.1 Setup

This section describes the cluster and evaluation setup as well as the systems used to conduct, store, and evaluate the measurements. The implementation of our scheme for increasing energy efficiency described in Chapter 4 is augmented using a set of components necessary for the evaluation which are detailed in the following. This section also presents the physical topologies of the streaming applications described in Chapter 5 that are used throughout the evaluation.

Dayarathna and Suzumura [DS13] have compared the performance characteristics of three stream-processing engines: System S, S4, and Esper. Their benchmarking methodology was used as a foundation for the evaluation in this work and their observations regarding the scale and parallelism of PEs provided valuable insight for dimensioning the physical topologies in Section 6.1.4.

### 6.1.1 Cluster Setup

The setup of the cluster used for the evaluation is depicted in Figure 6.1. It consists of four machines: machine 0 is hosting the management and infrastructure components required by Mesos and Heron. Machines 1, 2 and 3 are hosting and executing the actual topology. When total cluster energy usage is referenced in this evaluation, it refers to the energy used by these three machines. Machine 4 hosts the infrastructure and software used to measure this total cluster energy usage.

Each machine is equipped with 16 GB of RAM and a single Intel Xeon E3-1275 v5 CPU (Skylake architecture) with 4 physical cores and 8 threads via Hyper-Threading. They are connected via a 1 Gbit/s Ethernet networking switch. Ubuntu 15.10 with Linux kernel version 4.2.0-42 is running on each machine. To avoid interference, any software components accessing the RAPL interface, such as the thermald temperature daemon, have been disabled. The Linux power governor has been set to *performance*.

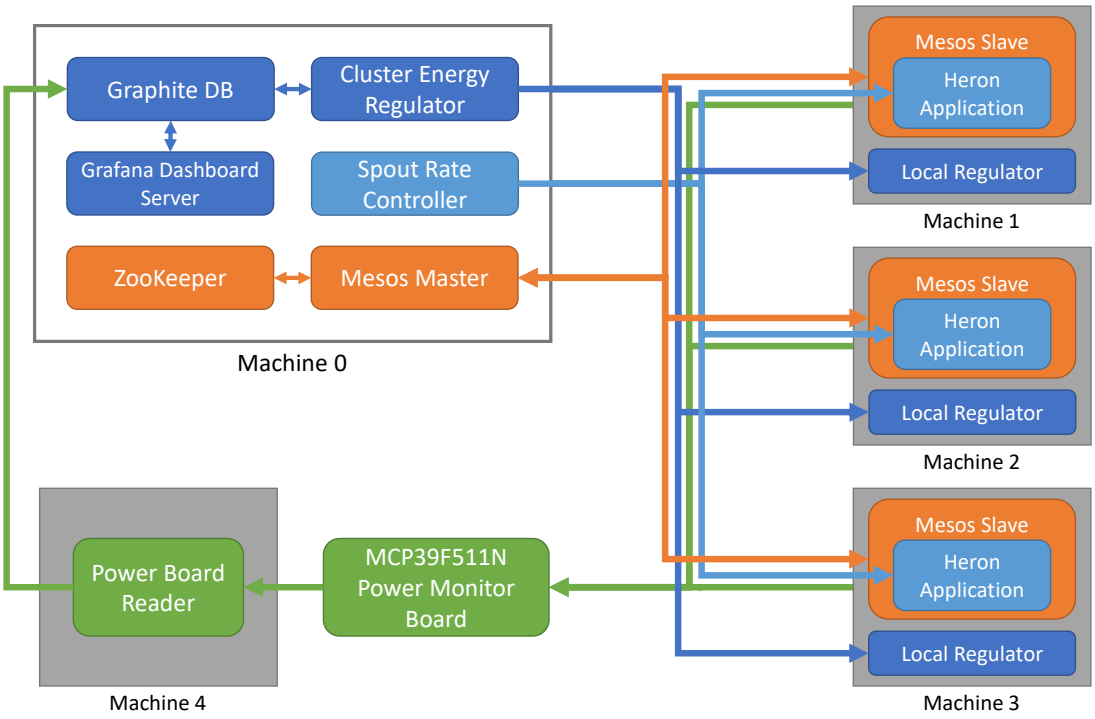



**Figure 6.1** – Cluster setup used for the evaluation.

The total cluster energy utilization when all CPUs are in idle state is 45 W, thus the static, non-CPU energy usage of each of the three machines is 15 W. This static energy usage is created by the non-CPU hardware components of the server, including RAM, the mainboard, cooling fans, and other peripherals.

The spout rate controller placed on machine 0 in Figure 6.1 is specifically used for the evaluation to control the rate at which tuples are emitted by the spouts of the topology and is used to simulate different workloads to be processed by the topologies.

### 6.1.2 Spout-Rate Controller

The rate at which tuples are emitted by the spouts in the topology is manually configured for the evaluation, in order to simulate different workloads. For this reason, we have introduced a *spout-rate controller* as an additional component running on machine 0. Figure 6.2 shows this controller and its connections to other elements of the cluster.

The spout-rate controller is implemented as a simple REST server written in Node.js. It stores the rate at which each spout should emit tuples in the spout rate database realized using MongoDB. The controller exposes a POST route to set and a GET route to retrieve the rate for the spout whose component name is specified as the URL-parameter. Each spout in the topology polls this rate in 10 s intervals from the service and emits tuples accordingly. Internally, each spout uses a rate limiter to ensure that on average no more tuples than indicated by the queried spout rate are emitted during any given second. We use the `RateLimiter` implementation that is part of the Google Guava set of common libraries for Java [Goo17].

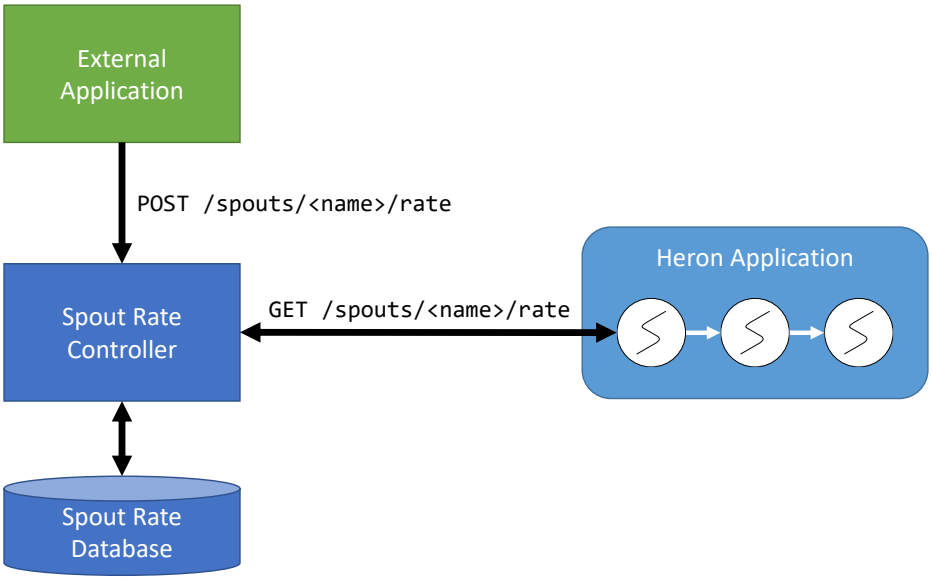



**Figure 6.2** – Spout-rate controller in context.

The external application setting the tuple emit rate for a spout can be any REST client to set it manually, or the cluster energy regulator to set the rate as part of an automated experiment during the evaluation. The spout rate REST service is running on a fixed IP address and listening on a fixed TCP port. This makes it easy for the spouts to discover the service as it is simply hard-coded into the topology code.

### 6.1.3 Measurements

For the evaluation, we measure the energy consumption of the 3-machine cluster executing the actual workload. We are using the MCP39F511N Power Monitor Demonstration Board by Microchip Technology Inc. [Mic15]. As opposed to just relying on the CPU power usage reported by RAPL, these measurements represent the total cluster energy usage, including the energy required by the power supply unit, mainboard, RAM, cooling fans, and peripherals. The monitoring board is connected via USB to an additional machine and reports the cluster energy usage at a 1 Hz sampling frequency. The sampled values are sent to the Graphite database and stored until they are used for the analysis below.

As described in Chapter 4, the topologies report their throughput and back-pressure data to the Heron metrics manager, which in turn sends them to the Graphite database for storage and analysis. Bolts and spouts of the topology export their metrics every 5 seconds, or at a rate of 0.2 Hz. The sink throughput metric represents the number of tuples that were processed in the past second and is thus created at a 1 Hz rate. Since this metric is only exported every 5 seconds, it is internally reduced to the mean of the past 5 throughput observations.


**Figure 6.3** – MCP39F511N Power Monitor Demonstration Board by Microchip Technology Inc. Image from [Inc17].

### 6.1.4 Physical Topologies

This section describes the physical topologies of the streaming applications first introduced in Chapter 5. In Figure 6.4 through Figure 6.7 the logical and the corresponding physical topology are shown. The outer rectangle marks the machine the Heron containers – shown as inner rectangles – are executed on. In each container, the colored squares represent Heron instances; their color corresponds to the spout or bolt of the same color in the logical topology. The total number of containers, instances, and their placement was chosen according to the guidelines for tuning a Heron topology described in Section 4.3.4.

For both types of topology, single- and multi-lane, the main goal was to maximize the throughput. For the multi-lane topologies, an additional constraint was placing each lane on an individual machine, to be able to power-cap the lanes independently.

For every topology, the meta PEs (spouts, sinks, and joiners) were always placed on Machine 1. Elements that include more specific application logic were distributed equally among the servers for the two single-lane topologies (Figure 6.4 and Figure 6.5). In case of the multi-lane topologies (Figure 6.6 and Figure 6.7), the processing bolts were placed such that each lane is run entirely on a single physical machine.

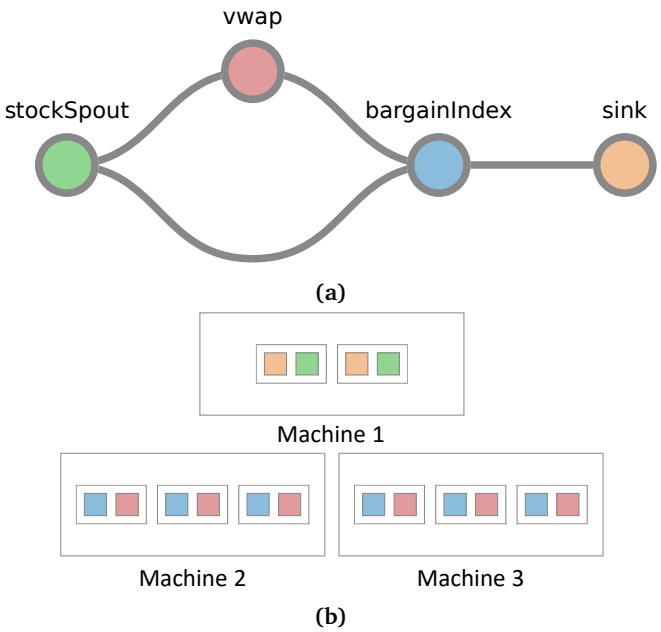



**Figure 6.4** – Logical (a) and physical topology distributed across the cluster (b) of the `BargainIndex` streaming application.

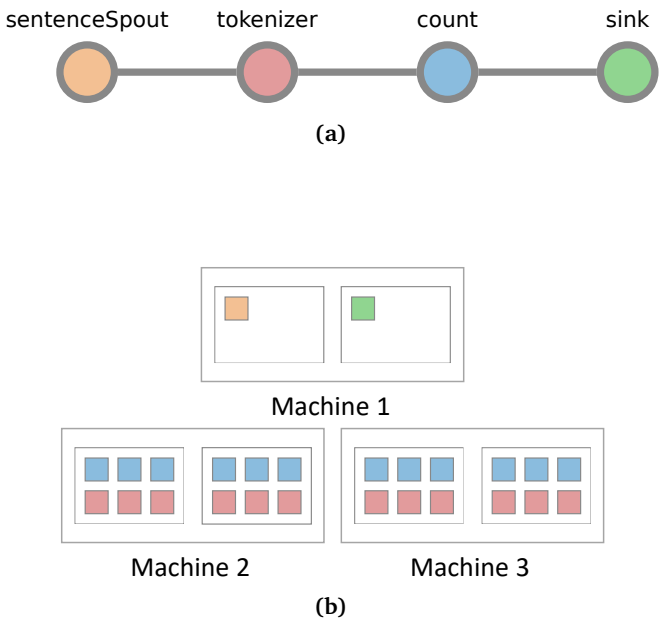



**Figure 6.5** – Logical (a) and physical topology distributed across the cluster (b) of the `SentenceWordCount` streaming application.

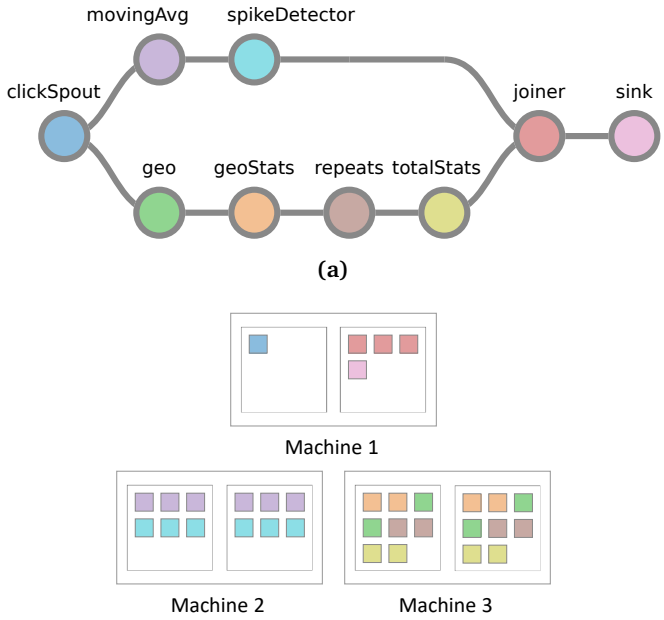



**Figure 6.6** – Logical (a) and physical topology distributed across the cluster (b) of the `ClickAnalysis` streaming application.

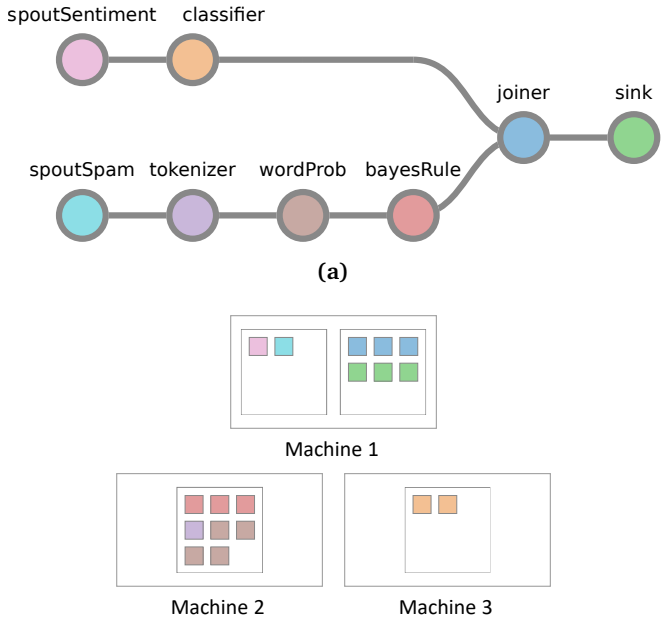



**Figure 6.7** – Logical (a) and physical topology distributed across the cluster (b) of the `TweetAnalysis` streaming application.

## 6.2 Control-Algorithm Properties

Describing the distribution of the physical instances on the servers of the cluster concludes the setup section. In the following, we conduct the actual evaluation and analyze the findings, starting with giving an overview of the runtime behavior of the proposed solution.

The control algorithm is the core feature of our solution for reducing the power usage of a cluster of servers executing a stream-processing topology. The algorithm fundamentally combines a capping strategy for lowering the power cap of each server with a continuous feedback loop that monitors the application health ensuring that power capping does not cause the performance to deteriorate. The control algorithm is continuously run on the cluster energy regulator and the power caps are applied via the local energy regulator on each server of the cluster.

In this section, we first look at the runtime behavior of the control algorithm, showing how the power caps are set during the capping process and how they affect the throughput and total cluster energy usage. This provides valuable insights into how the algorithm operates and the effect it has on the execution of the topology. Second, we show how adjusting the configuration parameters of the capping strategy impacts the settling time and accuracy of the algorithm.

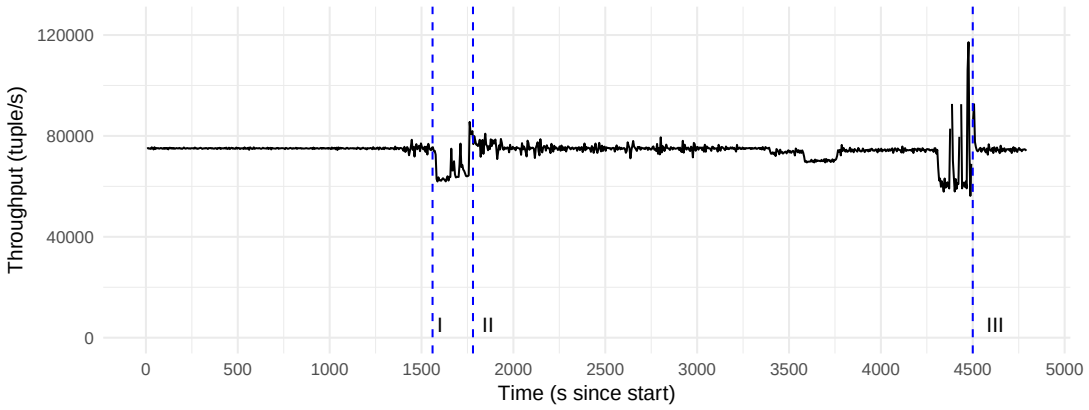
### 6.2.1 Runtime Characteristics

To provide insights into how the control algorithm affects the cluster energy usage and the topology throughput during the power-capping process, we execute the `BargainIndex` topology while running the control algorithm. The parameters are configured as follows: the throughput deviation threshold is set to 5 %, $\Delta P_{cap}$ is set to 1 W, $t_{sleep}$ is set to 2 min and the linear-capping strategy is used.

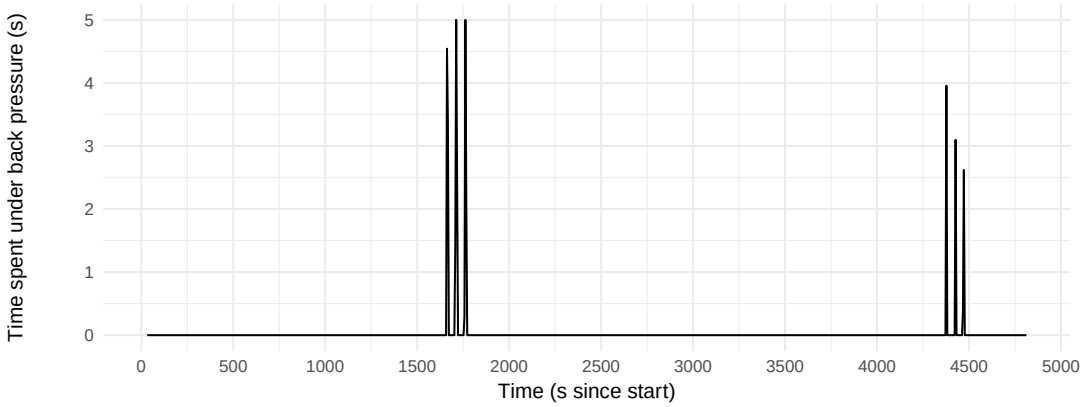
Figure 6.8 shows the topology throughput (Figure 6.8a), the presence of back pressure (Figure 6.8b), and the CPU package energy consumption of each server (Figure 6.8c) during the power-capping process.

Figure 6.8b shows the back pressure reported by the stream managers over time. The value indicates how much of the reporting interval of 5 seconds the topology has spent under back pressure. A value of 5 indicates, that in the past 5 seconds, the topology has been under back pressure constantly.

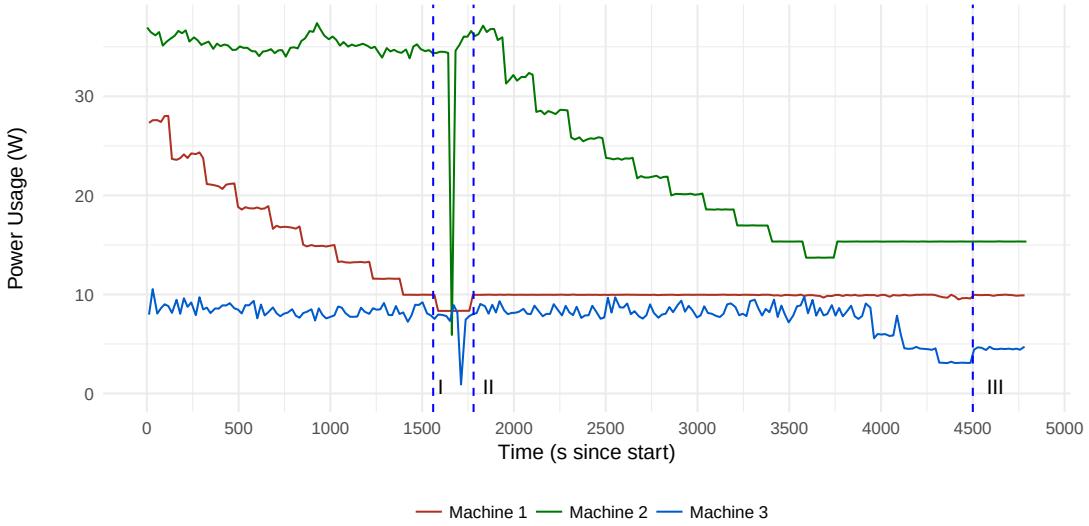
The capping process begins at 120 s with machine 1 (see Figure 6.8c). The first power cap is calculated by decrementing the CPU baseline package energy usage of 25.3 W by $\Delta P_{cap}$, thus the first power cap is 25.3 W during the first iteration of the capping strategy. Note that the measured power usage is slightly below this capping value, which is a result of the power cap being a strict *upper* limit, so measured values below it are to be expected. Figure 6.8c shows the steps with a height of $\Delta P_{cap}$ and a width of $t_{sleep}$ as the ideal cap is approached over time. The vertical line labeled with the roman numeral I indicates the point in time when the cap was set too low, causing the throughput to dip below 5 % of the baseline (see Figure 6.8a). Reducing the performance of the machine via power capping has caused back pressure to be present (see Figure 6.8b), causing the spout-emitted throughput to decrease. The reduced total workload causes the energy usage for machines 2 and 3 to drop as well. After $t_{sleep}$ has elapsed (indicated via the vertical line labeled II), the algorithm detects this dip and resets the power cap to the previously set one and determines that this cap is the final, ideal cap for this machine. The algorithm now repeats this process for the next two machines. At the time indicated by the vertical line labeled III, ideal caps for all servers have been found and the topology continues execution at minimal power usage until the end of the run.




**(a)** Total topology throughput over time.




**(b)** Presence of back pressure over time.




**(c)** The CPU package power usage as measured via RAPL for each server of the cluster.

**Figure 6.8** – Cluster throughput (a), back pressure (b), and server CPU power usage (c) throughout the entire capping duration of the `ClickAnalysis` topology set to a spout emit rate of approximately 75 000 tuple/s.

When the capping procedure is completed, the throughput decreased by 0.04 %, from 75 098 to 75 068. The total cluster power usage was reduced by 35.4 %, from baseline 134.0 W to 86.6 W after capping. Thus, the solution achieves a trade-off between a small and adjustable decrease in throughput with a significant reduction in power usage.

With one hour and 20 minutes, the total settling time for the given choice of parameters is quite large. However, the capping procedure represents a continuous improvement of energy efficiency as the total power usage is reduced, while the throughput remains consistent. Figure 6.9 describes this relationship between time and energy efficiency by showing how the throughput per watt (i.e., the amount of tuples that can be processed per second for each watt of power consumed) is linearly increasing over time.

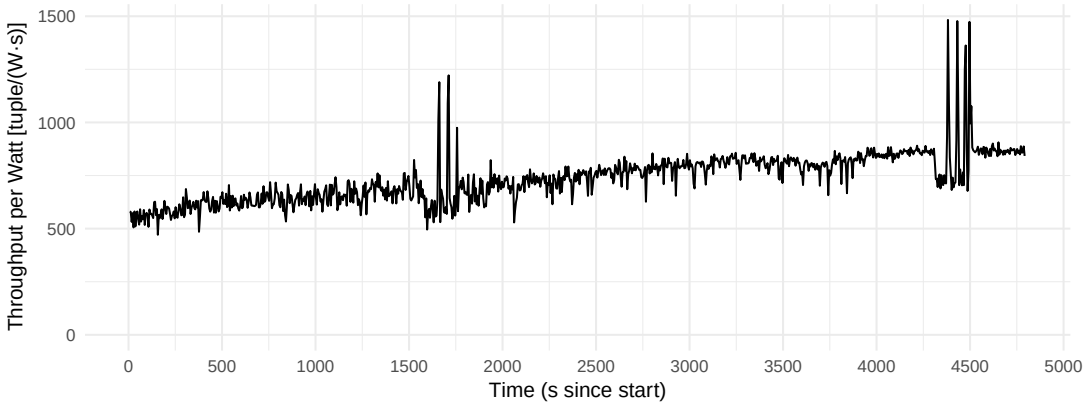



**Figure 6.9** – The total energy efficiency over time calculated as tuple throughput per watt.

Figure 6.10 depicts the total cluster power usage during the run. Besides visualizing the linear decrease in power usage, it also shows how power capping causes the measured values to smooth out in conjunction with the hard power limits set for each CPU.

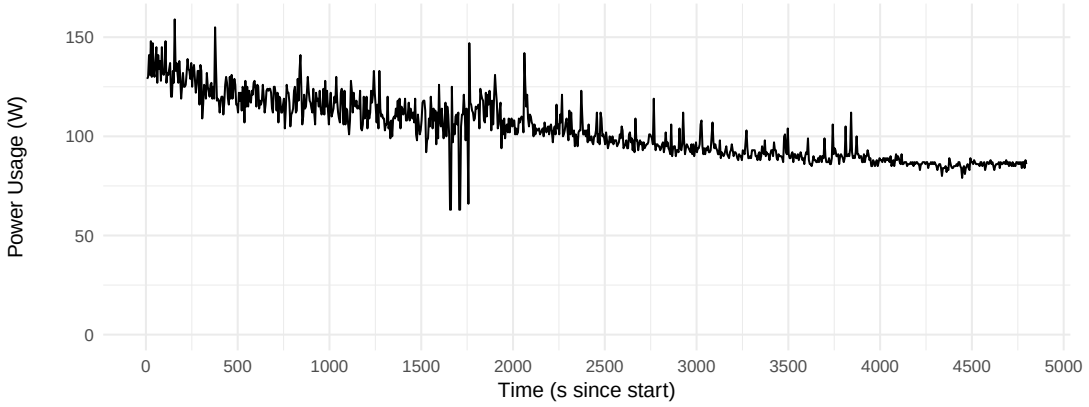
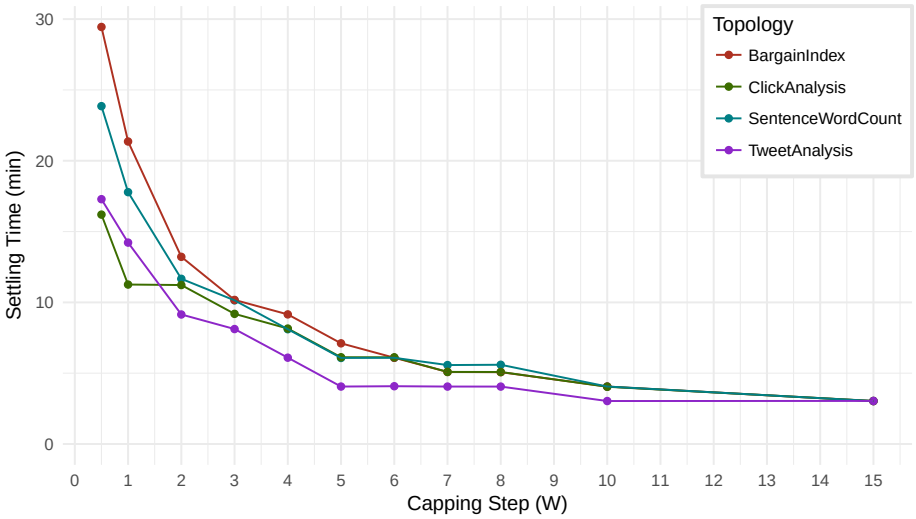



**Figure 6.10** – Total cluster power usage over time.

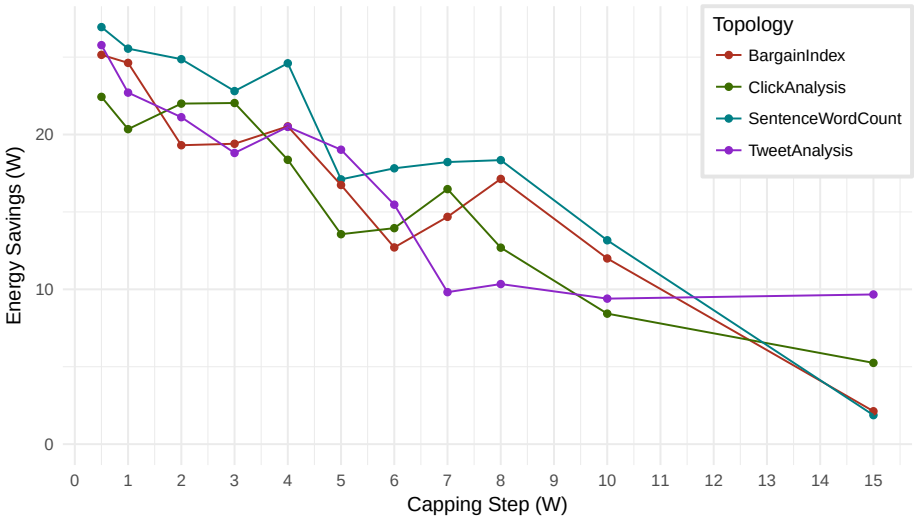
### 6.2.2 Impact of Configuration Parameters

In Section 4.4.1, the theoretical impact of the capping strategy configuration parameters on the settling time was discussed. To verify these findings, the linear capping strategy with increasing capping steps is applied to a topology running at a fixed throughput. The throughput was selected to ensure an equal absolute energy-saving of approximately 25 W for each topology. The results are obtained by repeating the experiment for each capping step three times, and calculating the mean value for the recorded settling time and energy-savings.

Figure 6.11 shows how the capping step affects the settling time of the control algorithm and energy-savings.




**(a)** Effect of different capping steps on the total settling time.




**(b)** Effect of different capping steps on the energy-savings.

**Figure 6.11** – Impact of different capping steps on the settling time (a) and cluster energy-savings (b).

The time until the algorithm finishes execution is naturally decreasing as a bigger capping step allows it to approach the minimal power cap per host quicker. However, Figure 6.11b shows how it also decreases the possible energy-savings (or accuracy) as bigger steps lead to more significant overshoot. This also results in larger throughput decreases during periods when the set power cap exceeds the minimum.

A low capping step in the range of a few watts is generally preferable as the only negative effect is a longer settling time, whereas the downsides of a larger step are both a decrease in energy saved and potentially larger deterioration of application performance. Throughout the evaluation, a capping step is set to 1 W, as it provides the best trade-off between settling time and accuracy for the hardware configuration and topologies used.

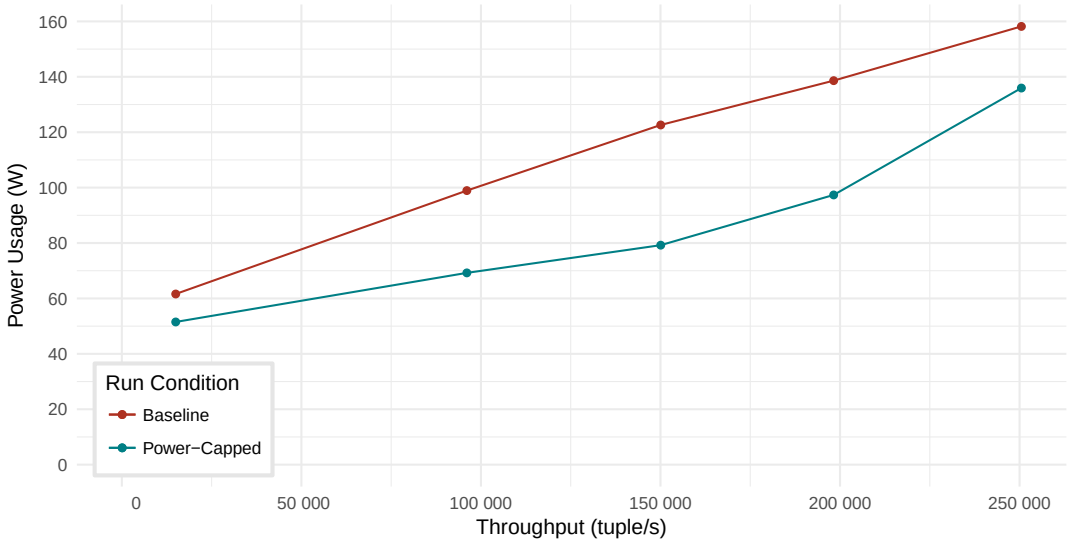
When choosing the capping step, it is also important to consider the CPU used and its degree of energy efficiency: if the CPU is inherently energy-efficient, power capping may only be able to achieve small additional power savings. In this case, using a small capping step is recommended. If the CPU is not very energy-efficient and power-capping is the only mechanism for power-saving used, a larger step can yield significant savings relatively quickly. In general, $\Delta P_{cap}$ should be chosen based on the expected power savings and, if any, constraints for the settling time must be considered.

## 6.3 Energy Savings for Various Workloads

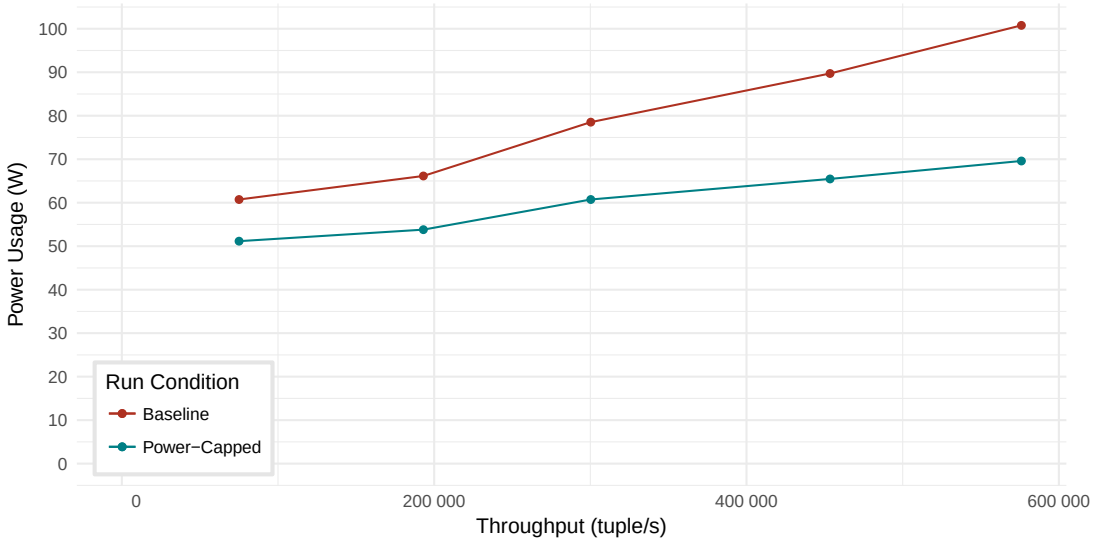
This section aims to give an overview of the energy-savings achievable with our solution.

For these experiments, each topology is executed at different workloads and the total cluster power usage measured during an uncapped (baseline) phase is compared to the energy usage after the control algorithm has finished executing (power-capped). For both phases, the total cluster energy usage is measured over a period of 10 min. The capping procedure is repeated five times for each topology and throughput, the plotted power usage represents the mean of these runs. The results for the single-lane topologies are depicted in Figure 6.12, for the multi-lane topologies, they are shown in Figure 6.13. The gap between the baseline and the power-capped line indicates the power saved by our proposed scheme for a given workload. For example, in Figure 6.12a at a throughput of 150 000 tuple/s, the baseline power usage is 122.6 W. Now, the control algorithm is executed to find the ideal power caps for each node in the server cluster, these power caps are set and the topology continues processing tuples for another 10 min. The power-capped energy usage is calculated as the mean usage over this period. In this example, it is 79.2 W, so our energy-saving scheme manages to reduce the total cluster power consumption by 43.4 W.

Our solution is able to reduce the power usage of each topology running on the cluster by an average of 26 %. In absolute terms, the energy saved ranges from 9.6 W (`SentenceWordCount` topology at 75 000 tuple/s) to 48.4 W (`ClickAnalysis` at 75 000 tuple/s). The latter represents a peak reduction in power usage of 35 %. As the throughput decreases, the total cluster power usage approaches values close to the idle usage of 45 W. The capping algorithm cannot cap beyond several watt above this hard lower bound without stopping tuples from being processed entirely. Therefore, at lower workloads the possible capping gains are comparatively lower. This behavior can best be observed in Figure 6.13a: the gap between baseline and capped power usage closes as the throughput – and consequently processing load for each server – decreases.

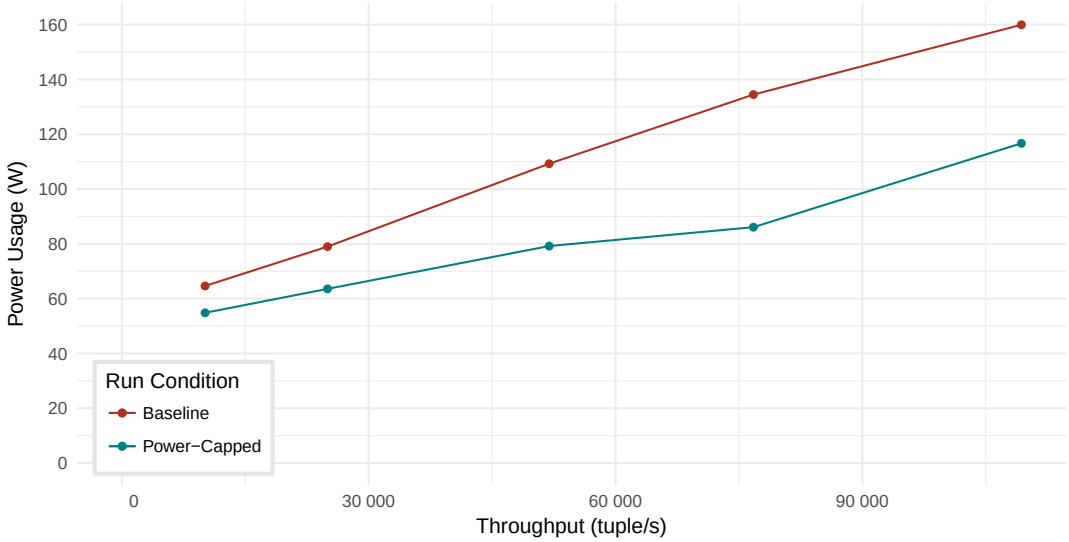



**(a)** BargainIndex

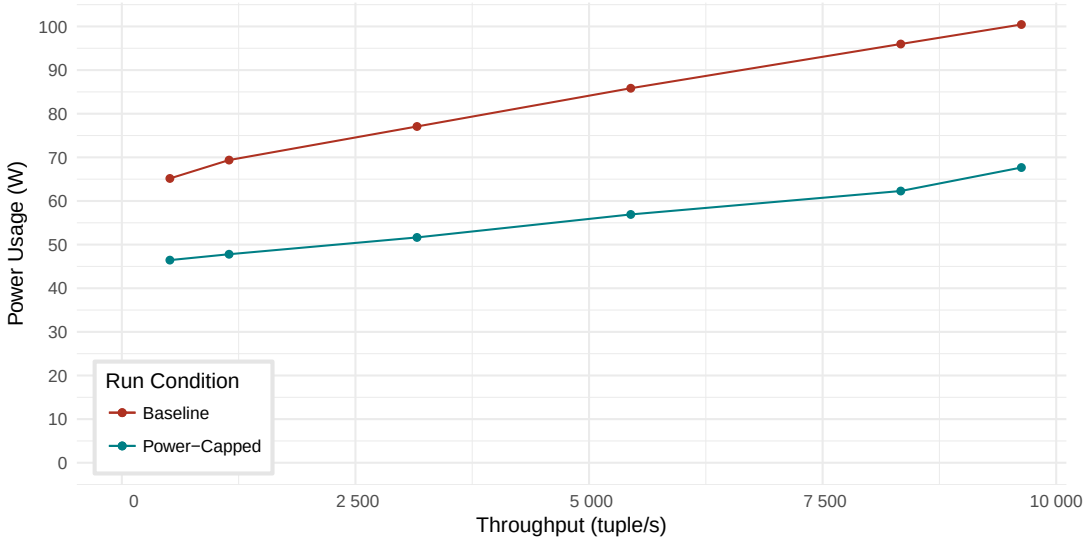



**(b)** SentenceWordCount

**Figure 6.12** – Total cluster power usage of the single-lane streaming topologies when uncapped (baseline) and capped at various throughputs. The throughput deviation threshold is set to 5 %, $\Delta P_{cap} = 1\,\mathrm{W}$, $t_{sleep} = 2\,\mathrm{min}$.




**(a)** ClickAnalysis




**(b)** TweetAnalysis

**Figure 6.13** – Total cluster power usage of the multi-lane streaming topologies when uncapped (baseline) and capped at various throughputs. The throughput deviation threshold is set to 5 %, $\Delta P_{cap} = 1\,\mathrm{W}$, $t_{sleep} = 2\,\mathrm{min}$.

## 6.4 Range of Energy Savings

The results of the previous section are summarized in Figure 6.14, which shows the range of the absolute power savings for each topology. While the `BargainIndex` and `ClickAnalysis` exhibit the widest range of savings, `SentenceWordCount` and `TweetAnalysis` have a much narrower range. In case of the latter topology, this is caused by the significant difference in workload between the

machines processing each lane. For example, at a throughput of 10 000 tuple/s, the machine on which the spam detection lane is executed, on average, draws 38.5 W of power in uncapped state, whereas the machine executing the sentiment analysis consumes only 4.4 W (the remaining machine uses 5.6 W). This effectively limits the most significant energy-savings potential to a single machine of the cluster. In case of the `SentenceWordCount` topology, the generally lower CPU load compared to other topologies limits the energy-savings potential.

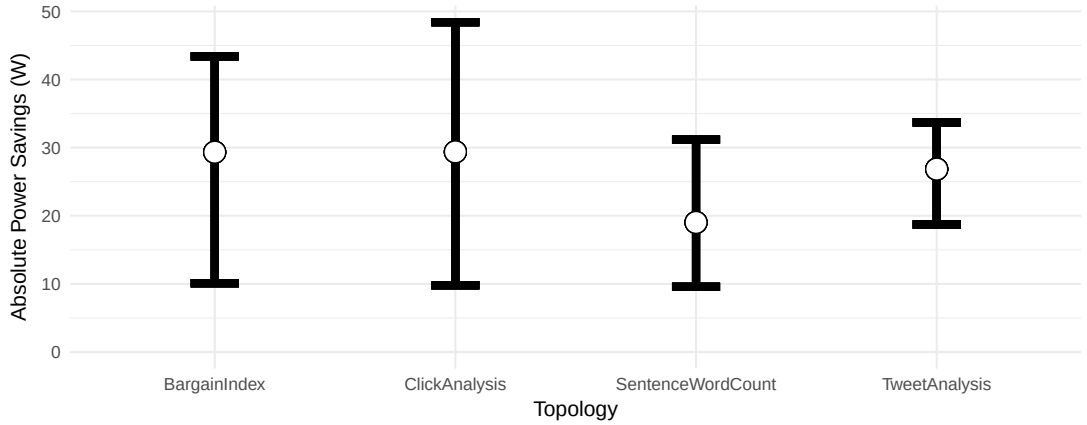



**Figure 6.14** – Range of the absolute power savings for each topology. The lower and upper whiskers correspond to the minimum and maximum, the circle to the mean energy savings.

## 6.5 Leveraging Container-Local Stream Grouping

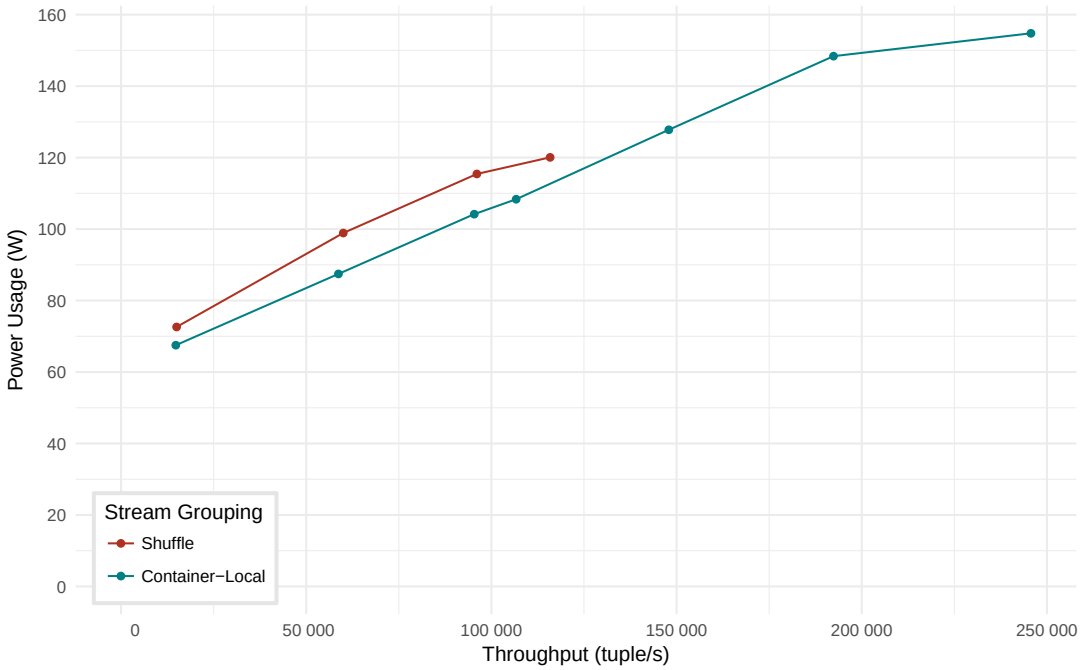
This section represents a semantic gap compared to the preceding ones, as it no longer uses power capping, the control algorithm and the associated infrastructure to transparently improve the energy efficiency, instead moving to the application layer and modifying topologies directly.

An additional method for indirectly reducing the energy consumption of a stream-processing application leverages awareness of the physical container placement in the stream-grouping strategy. More specifically, it replaces shuffle grouping as the data partitioning strategy with the container-aware implementation described in Section 4.3.3.3.

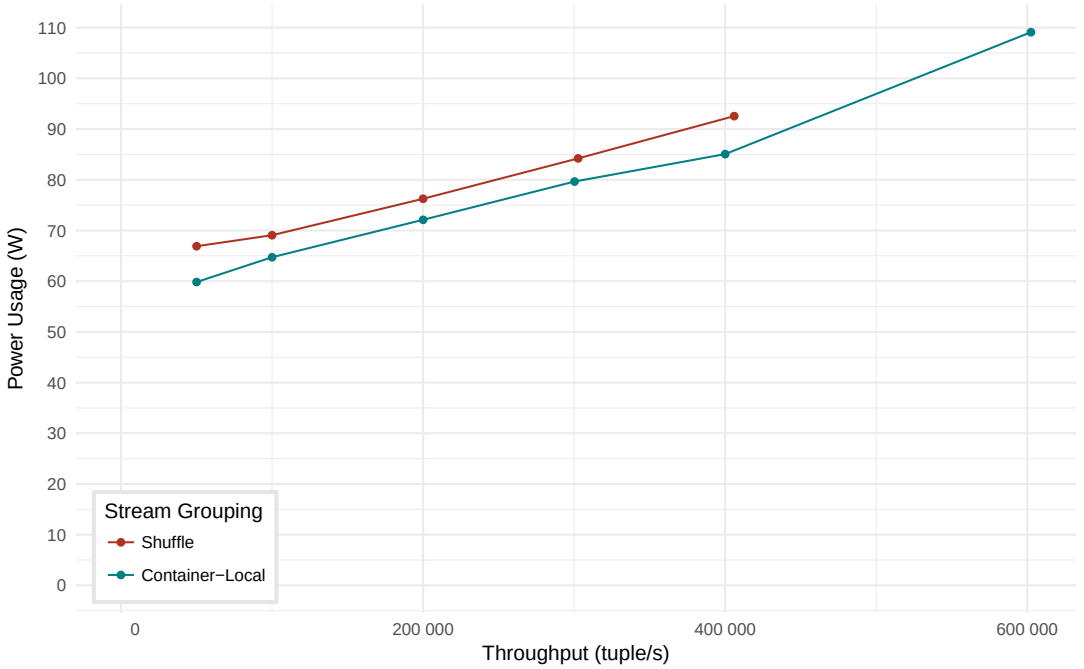
To evaluate the effects of container-local and shuffle stream grouping on the energy usage of the cluster, we executed the topology at various throughputs and measured the total cluster energy usage over a period of 10 min. Figure 6.15 depicts the result of these runs.

Using the container-local stream grouping algorithm in the `SentenceWordCount` topology increases the maximum throughput by 50 %, from 400 000 to 600 000 tuple/s. Additionally, it decreases the cluster energy usage by 5 W on average. In the `BargainIndex` topology, the maximum throughput more than doubles: from 110 000 to 270 000 tuple/s, while the energy usage at identical throughputs decreases by 8 W on average.

This experiment demonstrates how small changes in the application logic can lead to energy-savings without compromising the observable output of the topology. Leveraging container locality reduces the energy expenditure inflicted by the CPU overhead of serializing and transferring tuples over the network. Instead, the short-circuit routing of tuples within the same container by the stream manager is explicitly enforced.




**(a)** `BargainIndex` topology




**(b)** `SentenceWordCount` topology

**Figure 6.15** – Effect of the container-local and shuffle stream-groupings on the total cluster power usage at various throughputs for the `BargainIndex` (a) and `SentenceWordCount` (b) topology.

## 6.6 Summary

This chapter described the evaluation of our solution. We first detailed the cluster setup, tools used to measure the total energy usage, and the physical topologies of the applications used for the evaluation. We have shown how the capping step affects both the settling time and absolute energy savings for the topologies. Applying the power-capping-based scheme to increase the energy efficiency of the distributed stream-processing platform lead to a peak reduction in power usage of 35 % or 48.4 W in absolute numbers. Finally, we have shown how a minor modification to the application-level stream-grouping algorithm can lead to energy savings without using power-capping. In one case, it achieved a reduction in total cluster power usage of 8 W invariant of the workload processed. More impressively, it caused the throughput to more than double in the same case.

From these results, we conclude that the proposed scheme is effective at reducing the total power utilization of a distributed data-stream–processing cluster with minimal impact on the performance of the application.

# 7 LESSONS LEARNED

The preceding chapters have detailed and discussed the main contributions of this work. In Chapter 6, we have shown how the power-capping-based scheme is able to effectively and efficiently reduce the energy consumption in a distributed data-stream–processing cluster.

This work precedes an in-depth analysis of data-stream processing and the ways power capping could be used to improve its energy efficiency. A multitude of vectors were considered, some of which are described in the previous chapters, some of which can be found in the related work section (Chapter 8). In this chapter, we now briefly describe one such observation we made while working on the problem.

This chapter is intended to provide insights into the development of this work as well as motivate and inspire future work.

## 7.1 Overview

The fundamental idea – that eventually lead up to this work – was finding new methods that would allow us to utilize power capping to reduce the energy consumption of a distributed streaming cluster. One of the first experiments we conducted was executing a streaming application while lowering the power cap in small increments and observing how it affects the throughput of the topology. These experiments deliberately lowered the power cap even beyond levels that negatively affected the performance of the application being executed.

## 7.2 Experiment Setup

The setup was as follows: on a single server of the cluster (see Section 6.1.1 for details on the hardware used), we execute a topology tuned to fully stress all CPU resources. After an initial 35 min warm-up phase, we start power capping in 1 W-decrements, lowering the maximum power usage linearly. Each power cap is kept active for 5 min while the application throughput is continuously monitored and logged. The power usage is measured via the RAPL power monitoring interface for the CPU package.

## 7.3 Analysis

Figure 7.1 shows the outcome of the experiment for the `BargainIndex` topology described in Section 5.3.1.2. It shows that while the power usage decreases linearly, the resulting throughput

does not decline proportionally. This behavior implies that at some point in time (and at some power cap respectively) the gap between the two graphs maximizes. The power cap at which this occurs can be considered the most energy-efficient, as limiting the maximum CPU energy usage to this value when executing the application yields the highest throughput per watt.

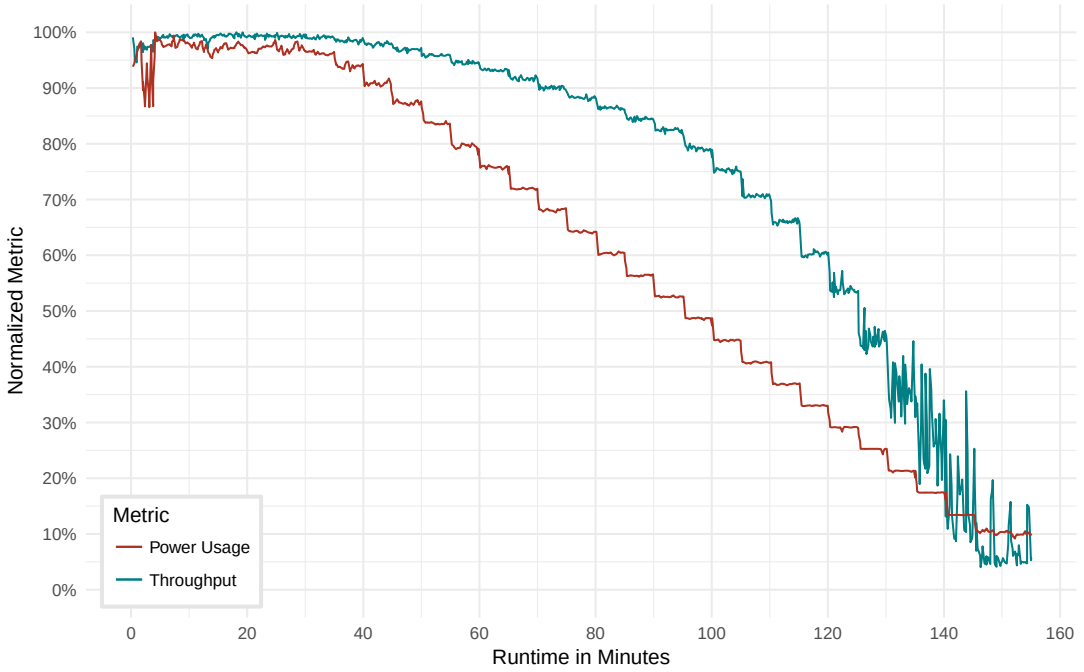
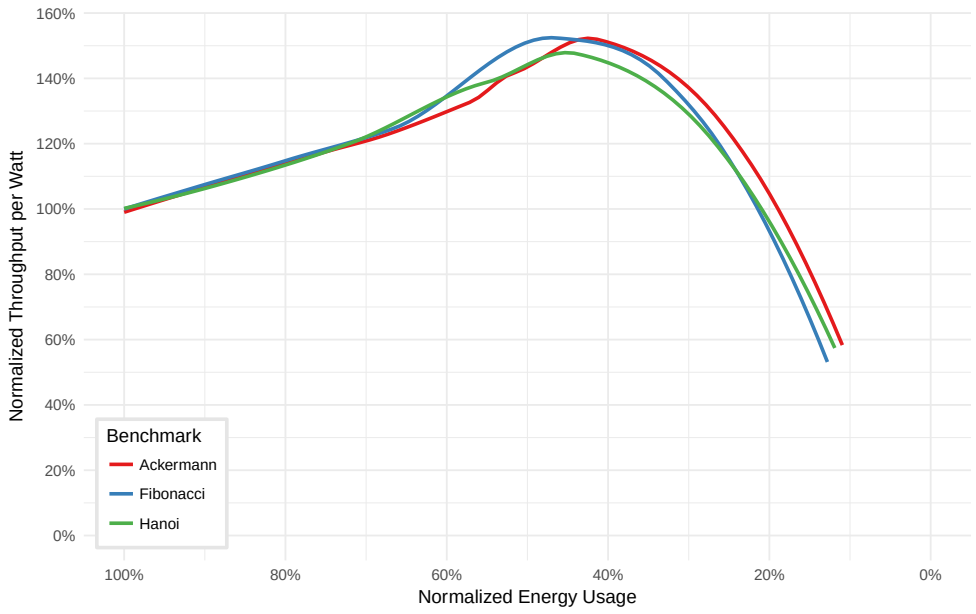



**Figure 7.1** – Normalized power usage and throughput over time for the `BargainIndex` topology executed on a single server. The initial maximum power usage was 50 W.

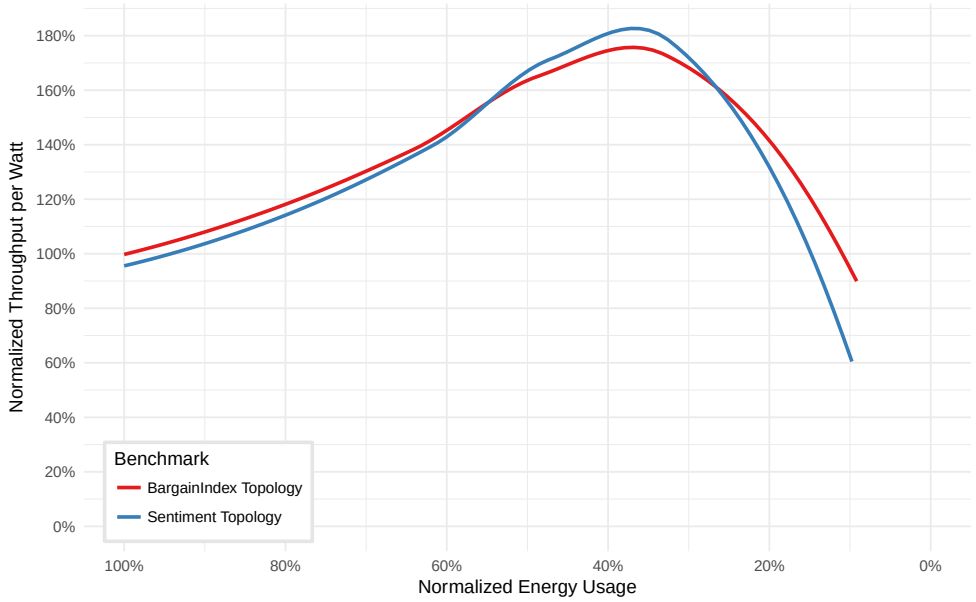
In the next step, we repeat the experiment outlined above not only for an additional streaming topology (the isolated sentiment part of the `TweetAnalysis` topology discussed in Section 5.3.2.1), but also for a set of benchmarks contained in the stress-ng [Kin17] tool. Again, each topology and benchmark is set to maximally stress the CPU, that is, use all available cores. To better evaluate the results, we merge the throughput and energy metrics into a single value: throughput per watt. Throughput is defined differently for the two types of benchmarks: for topologies, it is the rate at which tuples reach the final bolt (sink) in *tuple per second*. For the stress-ng benchmark, we use the *bogo ops/s* metric provided by the tool. It is a value describing how much overall work has been achieved by the benchmark per second. The results of this experiment can be seen in Figure 7.2.

The graphs shows that as the energy usage decreases due to power capping, the throughput per watt increases for all benchmarks. In case of the stress-ng benchmarks, we observe peak efficiency between 40 % and 50 % of the original power usage. Both topology benchmarks peak between 30 % and 40 %. Running an application at the power cap associated with this peak reduces the throughput, as can be seen in Figure 7.1. Here, at the power cap of approximately 45 %, the throughput is reduced to 70 % of its original, uncapped value. To restore the system to its full throughput again, an additional server could be added to process the remaining 30 % of the workload. By capping the original server to 40 % of its maximum power usage, a total of 30 W (60 % of the 50 W maximum power usage) is saved. The static energy consumption of a server is 15 W and running the `BargainIndex` topology with 30 % of the maximum throughput costs an additional

10 W. Adding these energy demands up and subtracting it from the 30 W of savings, the total power consumption has effectively been reduced by a total of 5 W.




**(a)**




**(b)**

**Figure 7.2** – Normalized throughput per watt at different power cap levels for stress-ng (a) benchmarks and Heron topologies (b). See [Kin17] for details on the individual stress-ng benchmarks.

## 7.4 Conclusion

In general, the benefit of setting one server to its most energy-efficient power cap for a given application and offloading the diminished throughput to one or more additional servers depends on three values: the energy saved by capping the initial host, the static power consumption of the additional servers, and the power required to process the remaining throughput. Only when the first value is larger than the last two values combined, are positive energy savings achieved. To improve the savings, hardware with lower static power consumption might be introduced into the cluster, however the details of such an implementation are beyond the scope of this work.

# 8 RELATED WORK

The more general topic of energy efficiency in modern distributed systems has been an important subject of research for the past years. In the face of ever-increasing power consumption and the associated economic and ecologic footprint, the goal of system design has been shifted towards energy efficiency. In the following, the proposed solution is classified among related work.

**Auto-optimization and Evaluation of Data-Streaming Systems**

In the category of data-stream–processing-specific work, Gedik et al. [Ged+14] have developed a scheme for addressing the profitability problem associated with auto-parallelization of general-purpose distributed data-stream–processing applications. Their focus is on exploiting parallelization opportunities in stream-processing applications, that is, automatically finding paths in the data flow graphs of a topology that could be run in parallel. This work is related to our approach, insofar as it uses topology throughput as an indicator for how the control algorithm affects performance. Additionally, they introduced SOSA properties as guiding principles for the algorithm – an approach we included in our work as well.

Floratou et al. [Flo+17] have proposed Dhalion, a framework running on top of Heron that can automatically reconfigure Heron topologies to meet throughput service level objectives. Their work introduces the notion of *self-regulating* streaming systems. These systems are *self-tuning*, meaning that they can automatically tune configuration parameters to achieve a stated quality of service. They are also *self-stabilizing*, that is, they react to spikes in workload by automatically scaling up, followed by scaling back down after the load stabilizes. Their approach is similar to ours in that it attempts to optimize resource utilization and thus, indirectly, energy efficiency. However, they do not consider energy as a resource whose utilization needs to be explicitly considered and no means to reduce it are applied. Additionally, we assume that topologies are already tuned/dimensioned to handle peak load, hazarding the consequences of over-provisioning. It stands to reason that a combination of both approaches could lead to further energy savings. Unfortunately, at the time of writing, their system is not available for public usage.

**Dynamic Reconfiguration via Power-Capping**

Eibel et al. [Eib+18] have proposed EMPYA, an energy-aware middleware platform that dynamically adjusts the system configuration to achieve the best energy–performance tradeoff based on the current workload. The platform was specifically designed for the Akka toolkit [Lig17], a framework for implementing applications following the actor-based programming paradigm. Their implementation relies on an initial profiling step to create a database of energy profiles, mapping performance

metrics to ideal power caps. It has also only been evaluated on a single machine, ignoring the distributed use case.

Finally, Schafhauser [Sch16] has proposed Qosos, an energy-aware cluster manager based on the Mesos framework. Similar to EMPYA, it optimizes energy efficiency by dynamically reconfiguring the system while maintaining performance. It also relies on an initial analysis step to generate a database of energy profiles. Our approach renders the static nature of these profiles obsolete, and instead, fully relies on data observed during runtime. Additionally, our solution leverages a global view of the cluster and its total observable performance to set power caps appropriately, whereas in their solution caps were applied by the equivalent of a local energy regulator in isolation. Finally, in their evaluation, they considered only the CPU energy savings achieved by capping, while we measure the total cluster energy usage.

# 9 FUTURE WORK & CONCLUSION

After having motivated, described, and evaluated our approach in the previous chapters, we now briefly describe the areas suitable for future research, before concluding this work in the final section.

## 9.1 Future Work

The approach outlined in this work is a capable and promising solution to the problem of improving energy efficiency in distributed data-stream–processing system. However, there are still areas for improvement and substantial future work can be derived from this contribution. In the following, some possible areas for future work are described briefly.

#### Improving Capping Strategies

This work relied on the linear capping strategy for finding ideal power caps and briefly outlined the parallel capping strategy. In general, strategies for finding the ideal power caps for a distributed set of servers and a given throughput provide substantial potential for improvements and innovations, with regards to decreasing the settling time and increasing accuracy. For example, one such approach could be moving away from continuously decrementing the power caps and instead applying a mechanism similar to a binary search: set the initial power cap to half the observed power usage, if this caused the throughput to deteriorate, raise the power cap by half and repeat until the power caps are found that no longer decrease the performance. Alternatively, machine learning could be used to deduce power caps for unknown workloads by observing the effect of power capping on application performance in a cluster.

#### Topology-Level Code Changes and Energy Savings

Changing the stream grouping algorithm has demonstrated how even small changes in application logic can lead to energy savings. Investigating this connection further and finding additional high-level modifications in topologies could be a promising topic for future work.

#### Generic Topology Performance Metrics

The approach presented in this work is heavily reliant on the total topology throughput being a constant multiple of the spout emit rate. However, topologies exist where tuples are only emitted to the sink after a certain time (e.g., a topology emitting the trending topics of the past hour), or after a certain number of tuples has been processed by an intermediary bolt. In order for our system to

function with these topologies, a more comprehensive metric for workload in streaming applications needs to be defined.

#### Coordinating Spout Emit Rates and Power Caps

A controller to manually set the spout emit rate has been used for the evaluation to simulate a range of workloads. This form of explicit rate limiting could be used to coordinate topology workloads and power caps. For example, a sudden spike in user activity could be buffered in the underlying streaming layer and only be processed at a rate and power cap that would optimize energy efficiency, achieving a trade-off between the total energy expenditure and the ability of the system to process data in realtime.

#### Heterogeneous Server Clusters

Finally, a significant portion of the total cluster energy is consumed by static (i.e., non-CPU) parts of the servers. The effect of shutting down entire servers on energy savings during periods of low utilization, or creating clusters composed of heterogeneous nodes, could be a promising approach. In Chapter 7 we have shown that power caps exist at which the energy efficiency is at a maximum. However, executing an application at this power cap deteriorates performance by a certain percentage. For example, assume that at its most energy-efficient power cap, the server is able to save a total of 20 W while only being able to process 70 % of the original workload. An additional server is needed to process the remaining 30 % of the workload. Ideally, this server is dimensioned such that its static power consumption and the energy required to execute the 30 % of the workload are significantly lower than the 20 W of power saved initially. A heterogeneous cluster could consist of servers that can process the majority of the workload at the most energy-efficient power cap, while the remaining workload is processed by servers with a low static energy footprint.

## 9.2 Conclusion

Batch-oriented systems have been used to process large quantities of data for decades, but they do not fulfill the need to process unbounded and continuous data as it is produced in many real-world applications. In comparison to these traditional systems, stream-processing engines consume data as it arrives and are a more natural fit. At the same time, a growing need for sustainable computing compels providers to develop more energy-efficient solutions.

In this work, we presented a scheme for increasing the energy efficiency of distributed data stream-processing applications. It uses power capping to reduce the energy usage of a cluster and is able to adjust the power caps at runtime, depending on the workload observed. The approach considers throughput as the central performance characteristic of the distributed topology and aims to reduce the total energy consumption without affecting it. The control algorithm incrementally lowers the maximum power limit of each host in a streaming cluster using Intel RAPL, while monitoring the throughput. It is able to achieve sufficient accuracy, has a short settling time, and avoids oscillation and overshoot. We described how the configuration parameters of the algorithm and the internally used capping strategy affect these SOSA properties.

We evaluated our approach on a cluster running Heron: a realtime, distributed stream-processing engine developed at Twitter. The four streaming applications used for the evaluation were chosen to be representative of realistic use-cases and used to evaluate the effectiveness of our approach. Our scheme reduces the total cluster energy usage by up to 35 %. We also briefly demonstrated how changing the stream grouping algorithm can lead to energy savings and improved throughput.

In conclusion, this work analyzed vectors for improving the energy efficiency in distributed stream-processing applications and demonstrated the effectiveness of power capping in lowering the energy consumption significantly.

# LIST OF ACRONYMS

| | |
|---|---|
| **DAG** | Directed Acyclic Graph |
| **PE** | Processing Element |
| **DSPE** | Distributed Stream-Processing Engine |
| **TM** | Topology Master |
| **SM** | Stream Manager |
| **HI** | Heron Instance |
| **CAPEX** | Capital Expenditure |
| **OPEX** | Operating Expenditure |

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF LISTINGS

# LIST OF ALGORITHMS

# REFERENCES


[Abe14] Emile Aben. *Internet Traffic During the World Cup 2014*. 2014. URL: https://labs.ripe.net/Members/emileaben/internet-traffic-during-the-world-cup-2014 (visited on 09/04/2017).

[Ana+13] Rajagopal Ananthanarayanan et al. "Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013.

[Apa17a] Apache. *Apache Hadoop*. 2017. URL: http://hadoop.apache.org/ (visited on 08/27/2017).

[Apa17b] Apache. *Storm Concepts*. 2017. URL: http://storm.apache.org/releases/1.1.0/Concepts.html (visited on 09/01/2017).

[BC12] Danah Boyd and Kate Crawford. "Critical Questions for Big Data: Provocations for a Cultural, Technological, and Scholarly Phenomenon." In: *Information, Communication & Society* (2012).

[BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2013. ISBN: 9781627050104.

[BLN88] Stephen A Berkowitz, Dennis E Logue, and Eugene A Noser. "The Total Cost of Transactions on the NYSE." In: *The Journal of Finance* (1988).

[Cho+16] J. Y. Choi et al. "Stream Processing for Near Real-Time Scientific Data Analysis." In: *2016 New York Scientific Data Summit*. 2016.

[CL07] Gordon V. Cormack and Thomas R. Lynam. *Spam Track*. 2007. URL: http://trec.nist.gov/data/spam.html (visited on 09/12/2017).

[Coo+14] Gary Cook et al. "Clicking Clean: How Companies Are Creating the Green Internet." In: *Greenpeace* (2014).

[Dia17] Diamond. *Diamond*. 2017. URL: https://github.com/python-diamond/Diamond (visited on 08/16/2017).

[Dim+12] Martin Dimitrov et al. *Intel® Power Governor*. 2012. URL: https://software.intel.com/en-us/articles/intel-power-governor (visited on 09/25/2017).

[DS13] Miyuru Dayarathna and Toyotaro Suzumura. "A Performance Analysis of System S, S4, and Esper via Two Level Benchmarking." In: *Proceedings of the 2013 International Conference on Quantitative Evaluation of Systems*. 2013.

[Eib+18] Christopher Eibel et al. "Empya: Saving Energy in the Face of Varying Workloads." In: *Proceedings of the 2018 IEEE International Conference on Cloud Engineering*. Under submission. 2018.

[Flo+17] Avrilia Floratou et al. "Dhalion: Self-regulating Stream Processing in Heron." In: *Proceedings of the 2017 VLDB Endowment conference* (2017).

[Fu+17] Maosong Fu et al. "Twitter Heron: Towards Extensible Streaming Engines." In: *Proceedings of the 2017 IEEE International Conference on Data Engineering*. 2017.

[Ged+14] Buğra Gedik et al. "Elastic Scaling for Data Stream Processing." In: *IEEE Transactions on Parallel and Distributed Systems* (2014).

[Goo17] Google. *Google Core Libraries for Java*. 2017. URL: https://github.com/google/guava (visited on 09/17/2017).

[Ham09] James Hamilton. "Cooperative Expandable Micro-slice Servers (CEMS): Low Cost, Low Power Servers for Internet-scale Services." In: *Proceedings of the 2009 Conference on Innovative Data Systems Research*. 2009.

[Hin+11] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *Proceedings of the 2011 USENIX Conference on Networked Systems Design and Implementation*. 2011.

[Hwa+05] J-H Hwang et al. "High-availability Algorithms for Distributed Stream Processing." In: *Proceedings of the 2005 International Conference on Data Engineering*. 2005.

[IBM17] IBM. *VWAP example*. 2017. URL: https://www.ibm.com/support/knowledgecenter/SSCRJU_4.2.1/com.ibm.streams.tutorials.doc/doc/VWAPExample.html (visited on 08/22/2017).

[Inc17] Microchip Technology Inc. *MCP39F511N Power Monitor Demonstration Board*. 2017. URL: http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=ADM00706 (visited on 09/22/2017).

[Int17] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals*. 2017. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (visited on 08/15/2017).

[Kam+13] Supun Kamburugamuve et al. *Survey of Distributed Stream Processing for Large Stream Sources*. Tech. rep. 2013.

[ker13] kernel.org. *Power-Capping Framework*. 2013. URL: https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt (visited on 08/15/2017).

[Kin17] Colin Ian King. *Stress-ng*. 2017. URL: http://kernel.ubuntu.com/~cking/stress-ng/ (visited on 08/06/2017).

[Kul+15] Sanjeev Kulkarni et al. "Twitter Heron: Stream Processing at Scale." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015.

[Lab17] Grafana Labs. *Grafana*. 2017. URL: https://grafana.com/ (visited on 08/16/2017).

[Len11] Marc Lenglet. "Conflicting Codes and Codings: How Algorithmic Trading Is Reshaping Financial Regulation." In: *Theory, Culture & Society* (2011).

[Lig17] Lightbend. *Akka*. 2017. URL: http://akka.io/ (visited on 09/06/2017).

[Lo+14] David Lo et al. "Towards Energy Proportionality for Large-scale Latency-critical Workloads." In: *Proceedings of the 2014 Annual International Symposium on Computer Architecuture*. 2014.

[LX08] Z. Liu and C.H. Xia. *Performance Modeling and Engineering*. Springer US, 2008. ISBN: 9780387793610.

[Max17] MaxMind. *GeoIP2 Databases*. 2017. URL: https://www.maxmind.com/en/geoip2-databases (visited on 09/10/2017).

[Mic15] Microchip. *MCP39F511N Power Monitor Demonstration Board User's Guide*. 2015. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/50002445A.pdf (visited on 08/24/2017).

[Net16] Netflix. *Evolution of the Netflix Data Pipeline*. 2016. URL: https://medium.com/netflix-techblog/evolution-of-the-netflix-data-pipeline-da246ca36905 (visited on 08/02/2017).

[Nie11] F. Å. Nielsen. *AFINN*. 2011. URL: http://www2.imm.dtu.dk/pubdb/p.php?6010 (visited on 07/22/2017).

[OED12] Guido van Oorschot, Marieke van Erp, and Chris Dijkshoorn. "Automatic Extraction of Soccer Game Events from Twitter." In: *Detection, Representation, and Exploitation of Events in the Semantic Web* (2012).

[PD10] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications." In: *Proceedings of the 2010 USENIX Symposium on Operating Systems Design and Implementation*. 2010.

[Pet+15] Pavlos Petoumenos et al. "Power-Capping: What Works, What Does Not." In: *Proceedings of the 2015 International Conference on Parallel and Distributed Systems*. IEEE. 2015.

[Pro17] The Graphite Project. *Graphite*. 2017. URL: https://graphiteapp.org/ (visited on 08/16/2017).

[Sah+98] Mehran Sahami et al. "A Bayesian Approach to Filtering Junk E-mail." In: *Papers from the 1998 Workshop on Learning for Text Categorization*. AAAI Workshop on Learning for Text Categorization. 1998.

[Sch16] David Schafhauser. "Design and Implementation of an Energy-Aware Platform for QoS-Oriented Applications." MA thesis. Germany: Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Dec. 2016.

[Sim13] Phil Simon. *Too Big to Ignore: The Business Case for Big Data*. John Wiley & Sons, 2013. ISBN: 9781118641866.

[SÇZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. "The 8 Requirements of Real-Time Stream Processing." In: *ACM Special Interest Group on Management of Data* (2005).

[Twi17a] Twitter. *Heron*. 2017. URL: https://twitter.github.io/heron/ (visited on 07/25/2017).

[Twi17b] Twitter. *Heron Source Code*. 2017. URL: https://github.com/twitter/heron (visited on 08/14/2017).

[Wal15] Russell Walker. *From Big Data to Big Profits: Success with Data and Analytics*. Oxford University Press, 2015. ISBN: 9780199378326.

[Win+16] Wolfram Wingerath et al. "Real-time Stream Processing for Big Data." In: *it-Information Technology* (2016).