Vrije Universiteit Amsterdam

Department of Computer Science, Faculty of Sciences
Master of Science in Computer Science

A Scalable and Hybrid Cloud Framework for Evaluating Streams of Sensor Data in Real-Time

Supervisors:
Prof. dr. ir. Henri Bal
Roshan Bharath Das

Second Reader:
Drs. C. Kees Verstoep

Candidate:
Gabriele Di Bernardo

November 2017

"Il termine utopia è la maniera più comoda per liquidare quello che non si ha voglia, capacità, o coraggio di fare. Un sogno sembra un sogno fino a quando non si comincia da qualche parte, solo allora diventa un proposito, cioè qualcosa di infinitamente più grande."

Adriano Olivetti

Abstract

Combining smartphone sensors and Internet-of-Things (IoT) enables new kinds of application scenarios. Cowbird is a framework that combines the evaluation of both smartphone sensors and IoT in the cloud. However, Cowbird does not scale well in large deployment contexts where thousands of IoT sensors can continuously generate high volumes of data that have to be processed. To this end, we extended Cowbird to support the sensing and evaluation of large streams of sensor data in real-time. We designed a distributed architecture based on the synergy of traditional computing systems and novel stream data processing technologies. The former is required for building a scalable IoT data sensing platform, while the latter are necessary for evaluating and processing large streams of sensor data. We show that in some conditions the data streams evaluation can be performed directly in the sensing platform. This approach avoids the introduction of extra costs and overheads caused by offloading the sensors data evaluation to a stream data processing engine.

Contents

Chapter 1

Introduction

Modern smartphones are characterized by significant processing power, advanced network capabilities and a variety of different on-board sensing devices such as accelerometer, GPS, gyroscope, ambient light sensors and many others. Mobile applications developers can take advantage of these sensors to build sophisticated applications that affect millions of people lives everyday.

Smartphone sensors create a multitude of opportunities for mobile application developers. Sensed data can be used for building applications such as monitoring the number of steps taken during the day and to remind the user to do physical movement if he/she has been sitting for too long. Sensor-based applications usually collect the data coming from different sensors and use the collected records to provide context-awareness to the end users. However, having many different applications that access the same smartphone sensor can lead to the redundant collection and storage of the same sensor data values. To this extent, the SWAN (Sensing with Android Nodes) framework [45] has been designed. The SWAN framework is a middleware between Android applications and the smartphone sensors. The SWAN framework provides a high-level abstraction to access sensor measurements avoiding data storage duplication. The SWAN framework is characterized by the SWAN-Song language that allows the creation of context expressions based on smartphone sensor data.

The emergence of the Internet-of-Things (IoT) brings sensors and actuators to the internet. The IoT adoption and deployment contributes to the dissemination of many small sensors over buildings or cities. The combination of smartphone sensors and IoT sensors can enable many new application scenarios. In particular, the local contexts of the user's smartphone can be extended with more global information coming from the surrounding IoT sensors. Hence, applications such as optimizing the biking route through the quietest streets or indicate the most sunny paths can

be created.

IoT sensors and actuators typically report the sensed data to the cloud. Thus, if a smartphone application wants to access IoT data it has to continuously poll a web endpoint to retrieve the fresh data values. This approach can be very *energy inefficient* for a smartphone. Furthermore, IoT sensors can generate large streams of data that hardly can be aggregated, computed and evaluated by a mobile device. To this end, Cowbird [33] was created. Cowbird is an energy efficient framework that helps developers build applications that combine smartphone sensor data and IoT. In particular, Cowbird offloads the sensing and communication capabilities to the cloud. Cowbird extends the original SWAN framework making it possible for an Android application to create SWAN-Song expressions that evaluate sensor data both on the phone and in the cloud.

Internet-of-Things sensors generate continuous streams of data. For example, a smart city with 10,000 sensors (e.g., humidity, sound, CO_2) transmitting at a rate of one measurement per second will generate 600,000 data points per minute, or 864 million measurements per day. To address such scenarios, many streaming technologies arose. In such streaming applications, the live data is ingested in a *stream processing engine* that evaluates the data records in real-time. Stream processing frameworks rely on scalable and high-performance computing architectures.

The current Cowbird cloud application is not designed to scale and it can be executed only on a single computing node. In this thesis, we present an extension of the Cowbird cloud framework designed to sense and evaluate large streams of sensor data in real-time. We organize the Cowbird cloud infrastructure into two layers:

- A distributed and scalable sensing layer.
- A high-performance streaming evaluation layer implemented over the Apache Flink processing engine.

For some type of SWAN-Song expressions we keep the evaluation in the sensing layer, close to the data generation. This strategy is adopted to avoid communication overheads between the two layers and thus to increase performance. Since the evaluation is performed by a combination of novel streaming technologies and traditional computing systems, we called our architecture *hybrid*.

Our contributions are as follows:

- We present the design and implementation of a distributed architecture for the Cowbird cloud framework. The architecture is composed of a distributed sensing layer and a streaming evaluation layer powered by a streaming engine.
- We implement an optimized evaluation strategy for the SWAN-Song expressions that can be used in streaming-oriented applications.
- We build an experimental application that makes usage of sound sensors. We use this application to test and evaluate our proposed Cowbird cloud architecture. We also describe a benchmark framework we built for evaluating stream processing engines. The framework helped us understanding which streaming engine among all the available platforms is the most suitable for our use case.

This thesis is structured as follows:

- **Chapter 2** provides background information about the SWAN framework and Cowbird.
- **Chapter 3** describes the large-scale architectures that can be used to analyze streams of data in real-time.
- **Chapter 4** gives an overview of the cutting-edge stream processing technologies. In this chapter, we also describe the framework we built for evaluating some of the available stream processing engines.
- **Chapter 5** describes the design and implementation details of the extend Cowbird architecture.
- **Chapter 6** reports the testing methodologies adopted and the evaluation results of the proposed distributed architecture.
- **Chapter 7** reports the related work in streaming architectures applied to the IoT scenarios.

Chapter 2

Background

In this chapter we present the existing Cowbird framework described within the SWAN project [65]. Cowbird is a flexible and energy efficient framework for building Android applications that use both smartphone sensors and IoT sensors[33]. Cowbird extends the SWAN framework [45] and it supports context-based expressions (queries) that can run both on a smartphone and in the cloud; with the Cowbird framework, developers can easily decide where to perform the computation according to the source of the data and the frequency of the updates.

2.1 The SWAN Framework

SWAN (Sensing With Android Nodes) is a framework for easily building context-aware applications for Android smartphones [45]. The framework provides application developers a high-level abstraction for accessing smartphone's sensors. In particular, the SWAN framework is designed to be executed as an Android background service and it acts as a middleware between applications and sensors: running applications can access it through a well-defined API. The SWAN framework reduces storage redundancy and code duplication caused by multiple sensor-based applications running simultaneously. In fact, it provides a centralized solution for collecting and storing data from sensors.

2.1.1 SWAN Sensors

One of the most important characteristic of SWAN is its *multi sensor support*; at the moment, SWAN supports more than 20 different types of sensors. Below, we describe two categories of sensors:

- *Hardware sensors.* The physical sensors installed on the devices (e.g. ac-

celerometer, GPS).

- *Software sensors.* Software *data sources* (e.g., calendar, rain prediction, Twitter) that can be installed on the devices or that can run somewhere else.

SWAN sensors are implemented as background services that are responsible for gathering the sensor's data. In case of external software sensors such as the rain prediction or Twitter, the SWAN sensor continuously polls from the web endpoint to get live data. The data accuracy can be improved by increasing the polling frequency.

From an architectural point of view all the SWAN sensors run in separate processes in order to remain unaffected if the application built on top of SWAN crashes: this design approach prevents data losses. The communication between SWAN sensors and the other components is performed through inter-process communication (IPC).

2.1.2 SWAN-Song

SWAN-Song [52] is a domain specific language for context expressions. SWAN-Song is used by applications to register expressions into SWAN. After registering an expression, SWAN evaluates it and sends back the result to the application.

The SWAN-Song language is a form of zeroth-order logic, which allows the description of a particular combination of contextual information using sensor based expression predicates. The predicates can be combined using math, comparison, and logic operators, and a specification of a *history window* and *history reduction* operator.

SWAN-Song Expressions & Predicates

A SWAN-Song expression is composed of two required components and three optional components. The required components are a sensor entity and a value path. The optional components are a list of one or more configuration parameters, a history window and a history reduction mode.

There are two types of SWAN expressions:

- *Value expressions.* Such expressions return a sensor value or a list of sensor values. For example, the following expression gets the current light intensity:

$$self@light : lux\{MEAN, 2000ms\} \tag{2.1}$$

In expression (2.1) *self* is the location identifier of the device, light is the sensor name, *lux* is the value path, *MEAN* represents the history reduction

mode and *2000ms* indicates the history window. The expression computes the average value generated by the light sensor in the last 2000 milliseconds.

- *Tristate expressions* Such expressions are useful for defining complex context conditions and they can take one of the following values: TRUE, FALSE or UNDEFINED. An expression is evaluated to UNDEFINED if the sensor queried in the expression is turned off or not available. As an example, the following expression(s) can be used to notify the user that he/she should exercise:

$$Morning = self@time : hour < 12 \tag{2.2}$$

$$Dry = self@rain : expected == 0 \tag{2.3}$$

$$Mobility = self@movement : total\{MAX, 3600000\} > 15.0 \tag{2.4}$$

$$Exercise = Morning \;\&\&\; Dry \;\&\&\; !(Mobility) \tag{2.5}$$

The *Morning* expression (2.2) uses the time sensor to check if it is earlier than 12:00 pm; the *Dry* expression (2.3), instead, checks the live rain prediction using the rain sensor and *Mobility* (2.4) uses the accelerometer sensor to check if the user has been moving in the past hour. The *Exercise* expression (2.5) combines the other TriState expressions to determine whether or not the context of the user is suitable for a workout.

Sensor predicates represent an array of *time-stamped values* because of the time windowing of sensor values. History operators are then applied on these values collected within the time specified time frame. SWAN-Song expressions support different history reduction modes, such as:

- ALL. This operator checks if *all* the time-stamped values, sensed within the time window, match the condition defined in the SWAN-Song expression.
- ANY. Selects any value that makes the comparison *true* as the value of the sensor. The ANY operator is also the default history reduction mode.
- MIN, MAX, MEAN, and MEDIAN, which will select or calculate the appropriate value.

2.1.3 Architecture

Android applications can register SWAN-Song expressions using the SWAN API. On registering a new expression, the Expression Evaluation Engine (see Figure 2.1)

is invoked. The Expression Evaluation Engine is responsible for evaluating SWAN-Song expressions: it interacts with the sensors in order to fetch the sensed data and computes the result of the expressions. Whenever new sensed data is available, it is sent to the Expression Evaluation Engine, which processes it and sends the result back to the application.

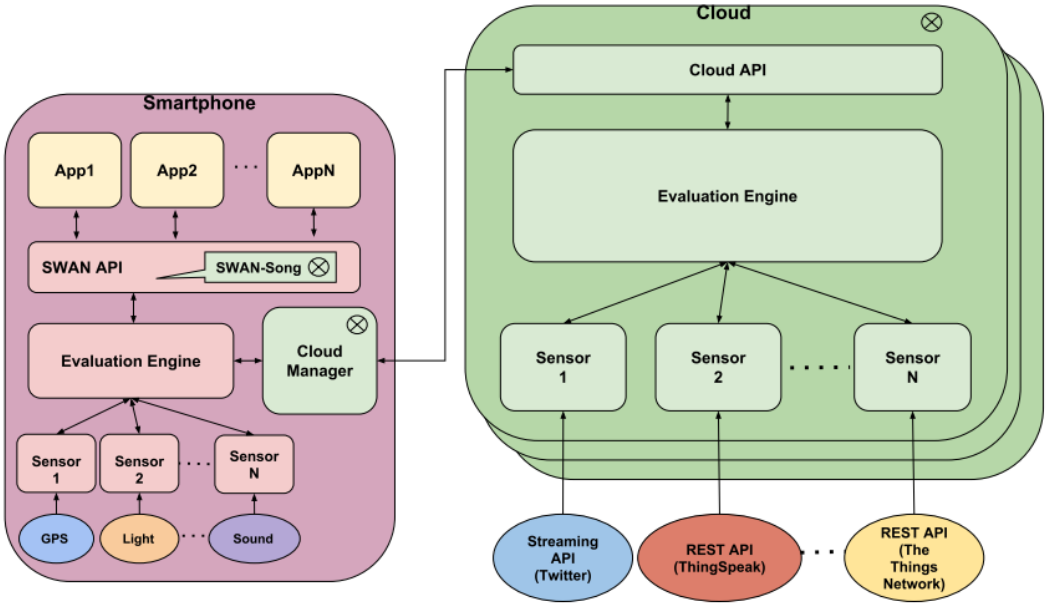

Figure 2.1: SWAN and Cowbird architecture. Cowbirds components are marked with ⊗. Source: [33]

2.2 The Cowbird Framework

The Cowbird framework has been designed as a cloud extension of the SWAN framework. In particular, it enables the combination of smartphone sensors and real-time web data: these data can be originally generated by IoT sensors or any data which can be retrieved by polling a web resource (e.g. Twitter feeds) [33]. Furthermore, the Cowbird framework reduces the overall communication between the smartphone and the cloud in order to save energy. In fact, the evaluation of phone sensors is performed locally while the evaluation of IoT sensors is computed in the cloud.

The description below has been adapted from [33] and contains textual overlap with the paper. Some of the core features and characteristics of Cowbird are [33]:

- *Flexibility.* The developers using the Cowbird framework can easily connect

to any cloud to offload the evaluation of the IoT sensors.

- *Usability.* Cowbird uses the SWAN-Song domain specific language to access both sensors in the phone and real-time data.

- *Portability.* The cloud part of Cowbird is programmed in Java. Hence, it can be easily ported to any device running a JVM.

- *Privacy.* Cowbird performs the evaluation of local sensors on the phone and evaluation of remote IoT sensors remotely. Thus, the sensitive data can be kept and processed locally on the phone.

- *Energy Efficiency.* Cowbird is designed to minimize the energy consumption of smartphones. Offloading the computation and the communication required to fetch the IoT real-time data (continuous polling from the web) to the cloud will preserve the smartphones battery life.

- *Mobile Data Cost.* Sending much data through cellular networks is costly, so Cowbird tries to optimize the communication between the phone and the cloud. The result of the evaluation in the cloud is only sent to the phone if there is a change from the previous result.

2.2.1 Cowbird Architecture Details

Cowbird enriches the SWAN framework designed to be executed on Android smartphones. In particular, Cowbird integrates novel capabilities to the existing SWAN services and it introduces new sensing services to be executed on the cloud.

The existing SWAN framework has been extended by modifying the SWAN-Song language in order to support *cloud-based expressions*. Furthermore, a new module named *Cloud Manager* has been introduced: it runs on the smartphone and it is responsible for managing the communication with one or more clouds [33].

At the cloud side, the original SWAN evaluation engine has been redesigned to be executed on the cloud. The cloud side of the framework has support for various IoT sensors (implemented as virtual sensors and executed by a different thread) and a mechanism to push data to the phone.

We discuss now the typical interaction flow between Cowbird and third-party app built on top of it. A third-party app uses the SWAN API to register a remote expression to the cloud. The expression is then sent to the evaluation engine service which will forward it to the cloud manager in the phone. The cloud manager checks

the location identifier of the expression and sends it to the specified URL. The cloud acknowledges the expression and starts the evaluation by activating the relevant sensor thread. The result of the evaluation from the cloud is received by the cloud manager and is further sent to the evaluation engine in the phone for local processing. The final result is then sent back to the app.

Remote Expressions in Cowbird

In Cowbird, the location identifier of SWAN-Song has been extended in order to support remote expressions. The developer can add the URL of its cloud in the configuration of the SWAN framework. The developer can then refer to that cloud using *cloud* as location identifier in its SWAN-Song expression. For example, in case of a sound sensor feed available in a certain channel of the ThingSpeak platform [66] we can write a SWAN expression such as:

$$cloud@thingspeak : field?\, channelid =' 1'\#field =' 1'\{MEAN, 20s\} > 50.0 \tag{2.6}$$

The expression 2.6 checks if the mean value over a period of 20 seconds is greater than 50.0 decibel (dB).

The developer can also refers to the cloud indicating the endpoint directly in the SWAN-Song expression. For example, the following SWAN-Song expression perform the same evaluation of expression 2.6:

$$url = http://cowbird.herokuapp.com \tag{2.7}$$

$$url@thingspeak : field?\, channelid =' 1'\#field =' 1'\{MEAN, 20s\} > 50.0 \tag{2.8}$$

The SWAN-Song expression 2.8 assumes to have a Cowbird cloud instance running on Heroku at the URL provided as location identifier of the expression.

A combination of expressions that use smartphone local sensors (accelerometer) and IoT sensors (live data coming from ThingSpeak) can be written as:

$$E1 = self@movement : total\{ANY, 1000\} > 15 \tag{2.9}$$

$$E2 = cloud@thingspeak : field?\, channelid =' 1'\#field =' 1'\{MEAN, 20s\} > 50.0 \tag{2.10}$$

$$CombinedExpression = E1 \;\&\&\; E2 \tag{2.11}$$

The evaluation of the local expression (E1) occurs locally on the Android smartphone. The evaluation of the remote expression (E2) occurs in the cloud and the result is sent to the phone. The CombinedExpression in 2.11 is evaluated on the phone based on the results from E1 and E2.

The evaluation of a SWAN-Song expression composed by two remote sub-expressions, instead, is completely evaluated in the cloud.

$$E1 = cloud@thingspeak : field?\, channelid =' 1'\#field =' 1'\{MEAN, 200s\} > 50.0 \tag{2.12}$$

$$E2 = cloud@thingspeak : field?\, channelid =' 2'\#field =' 1'\{MEAN, 200s\} > 50.0 \tag{2.13}$$

$$CombinedExpression = E1 \;\&\&\; E2 \tag{2.14}$$

The result of the combined expression in 2.14 is evaluated on the cloud; the combined result is then sent to the phone.

Cloud Manager

The description below has been adapted from [33] and contains textual overlap with the paper. The evaluation engine running on the smartphone registers an expression to the cloud manager using three parameters: the expression identifier, the expression itself and the location (URL) of the expression. The cloud manager makes a HTTP POST request to the cloud indicated by the expression with the id, expression and a Firebase token. The cloud manager runs a Firebase Message Service used to receive the result of the evaluation computed in the cloud. The result is received at the smartphone side as push notification through Firebase Cloud Messaging. Since Cowbird is Android-based, the Google Play Service already has socket connections to the Firebase server and multiple applications can use this service to minimize battery consumption. The message received from the server contains the result of the expression which can be either a new value or a new TriState (TRUE, FALSE, UNDEFINED) and the expression id. The Cloud manager sends this data to the evaluation engine service for local evaluation. After the local evaluation, the final result is sent to the app.

Cloud Instance

The description below has been adapted from [33] and contains textual overlap with the paper. The Cowbird cloud instance is implemented using the Play Framework [53] with Java. The Cloud API contains a controller that routes a specific URL to a functionality. The cloud instance supports requests to register expressions both from mobile clients and web clients. The Cowbird cloud instance currently supports various IoT sensors such as ThingSpeak, The Things Network, Twitter, News, Weather, Currency and Flight. Additional sensors can be easily added as a plugin. Sensors are implemented as Java threads.

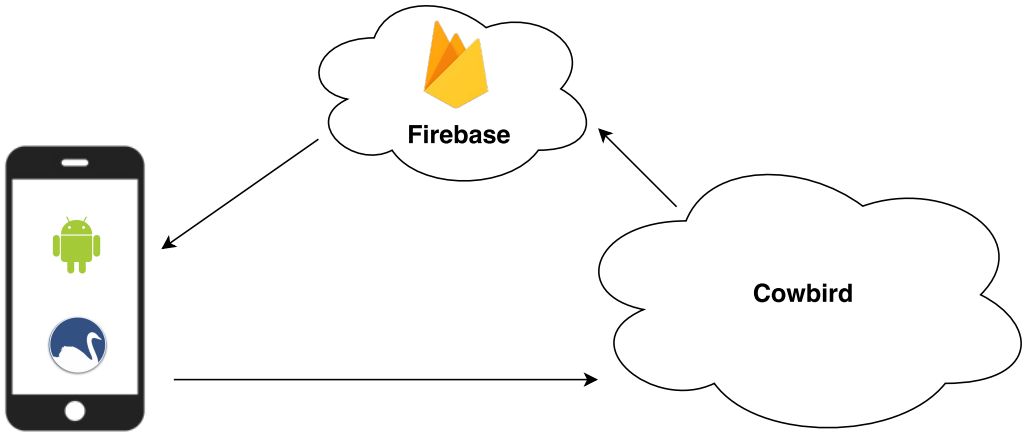

Figure 2.2: SWAN smartphone and Cowbird communication architecture.

On receiving a request from the phone, the Cloud API forwards it to the evaluation engine. The evaluation engine will start the relevant sensor thread that will keep polling external web data from an endpoint. The sensor data is sent back to the evaluation engine which will evaluate the data; expression evaluation performed in a multi-threaded fashion. The evaluation result is sent to the Cloud API that will forward it to the phone. The communication between the Cowbird cloud instance and the phone happens through Firebase push notification (see Figure 2.2).

2.2.2 Optimization

The process of sending back from the Cowbird cloud instance to the smartphone the result of a SWAN-Song Tristate expression has been optimized. The result is sent back only if there is a change in the evaluation (e.g., from FALSE to TRUE) and not every time a sensor value changes [33]. Minimizing the communication between the phone and the cloud reduces the energy consumption and preserves the smartphone battery life.

2.3 Cowbird Limitations

The Cowbird architecture presents some limitations. In particular, the current implementation of the cloud instance does not scale since it is designed to run on a single computing node. Now imagine a scenario where many expressions from different smartphones are offloaded to the cloud; thus, many threads are instantiated in order to fetch the sensor data. Multiple running threads can easily saturate the computing resources especially if each thread performs updates with a high

frequency. Furthermore, each sensor generates an unbounded and potentially infinite stream of data that has to be processed by the evaluation engine. The huge amount of data that has to be processed can overwhelm the computing resources (i.e. CPU, memory) and reduce the overall performance of the framework.

In order to evaluate the streams of data generated by the IoT sensors in real-time a distributed computing architecture is required. In the next chapters we will investigate how such infrastructure can be built and how the Cowbird cloud instance can be extended. We will focus on how Cowbird can accommodate SWAN-Song expressions evaluation on a large scale.

Chapter 3

Large-Scale Streaming Architectures

The internet traffic explosion started in the mid-90s induced the creation of massive datasets. Processing very large amounts of data, even though is conceptually straightforward, can be an onerous task. In fact, it means many computations that have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. In early 2000s, the developer had the burden to write complex applications responsible for parallelizing the computation, distributing the data, and handling possible failures. As a reaction to this complexity, Google in 2004 published a paper describing a novel programming model for processing large datasets named MapReduce [34]. MapReduce provides an abstraction that allows the developer to focus on the computation he/she wants to express hiding the details of parallelization, fault-tolerance, data distribution and load balancing in a framework. The MapReduce framework takes advantage of the Google File System [41]: a distributed file system that uses replication to provide availability and reliability on top of unreliable commodity hardware. The open source community, inspired by the Google MapReduce programming model, created Hadoop [15] and HDFS [58]. Hadoop release boosted the growth of the big data analysis field and created new scenarios and new types of applications that were very tough to realize with previously existing tools. From that moment on, a lot of effort has been put in realizing more efficient and high-performance technologies for collecting, aggregating and analyzing huge amounts of data.

In this chapter we will discuss some of the technologies and architectures that can be used in order to process large amounts of data generated by (IoT) sensors in real-time.

3.1 Batch-Mode Architecture

Google MapReduce and Hadoop are designed to be executed on top of clusters of computing nodes. The first releases of Hadoop integrated the cluster resource management with the data processing engine itself (Hadoop MapReduce). In 2012, Hadoop 2.0 was released [15] and it introduced YARN [69]; Apache Yet Another Resource Negotiator (YARN) decouples the programming model from the resource management infrastructure and the users' applications scheduling. In general, YARN is a distributed operating system that is capable of managing a cluster of machines and scheduling users' applications on the available resources. Furthermore, it provides an API that can be used to develop any generic distributed application. Hence, different computing frameworks that address different application requirements have been built on top of YARN (e.g., the general purpose Apache Spark [20], Apache Giraph [14] for iterative graph processing).

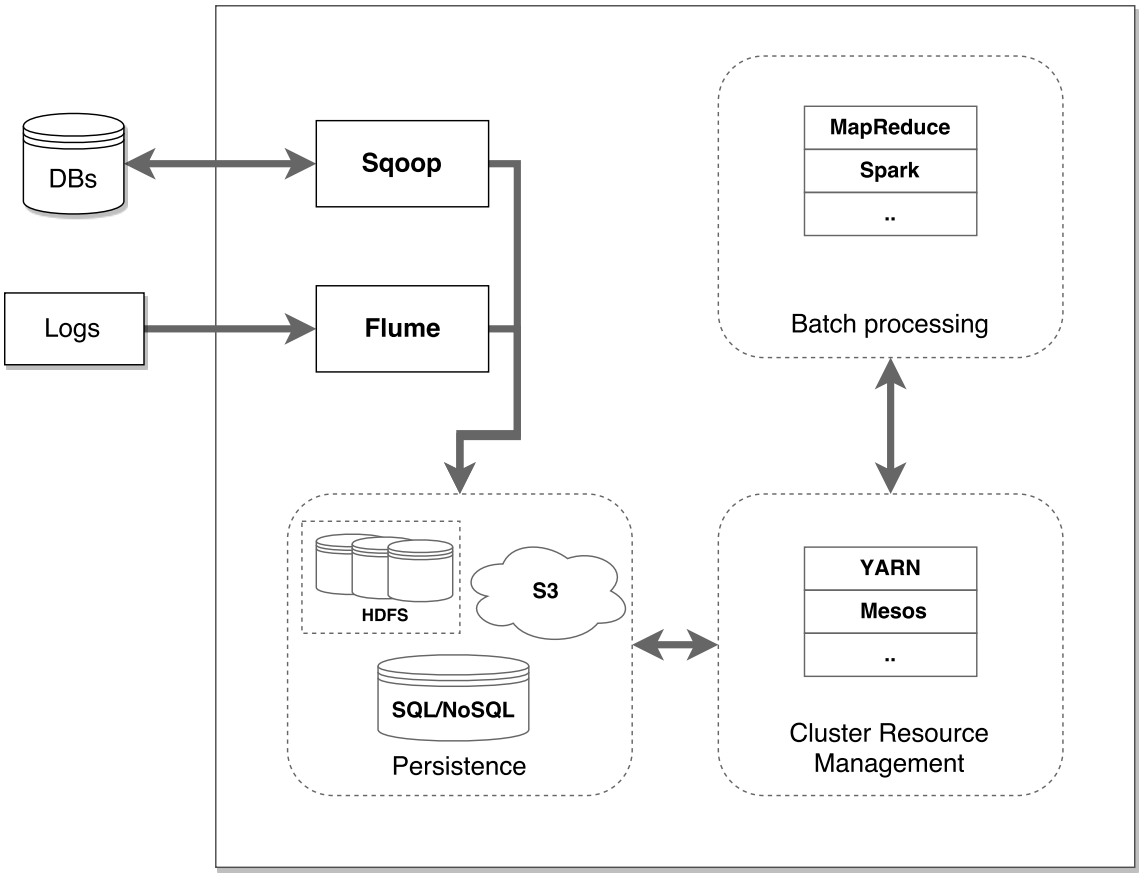

Figure 3.1: Classic big data batch architecture

A big data application usually has a well defined architecture, no matter the computing framework it uses and whether or not it runs on top of YARN or on another cluster management system (e.g., Apache Mesos [17], Kubernetes [48], in

the cloud using Amazon Elastic Map Reduce (EMR) [5]). The *classic* big data architecture for processing large amounts of structured or unstructured data is named *batch architecture* [47, 71, 55]. The typical big data batch architecture consists of the following components or layers (see Figure 3.1):

- *Data ingestion layer.* The data ingestion layer can be used to move data that can come from different sources into the persistence tier. The data ingestion layer can take advantage of special-purpose services such as Apache Flume [13] for log aggregation or Apache Sqoop [22] for interoperating with databases.

- *Data persistence layer.* The data persistence layer is responsible of storing and indexing final as well intermediate datasets. This tier can store large amounts of data using one of the next-generation database systems. In fact, the Dynamo [35] paper accelerated interest in NoSQL systems and many NoSQL databases emerged such as Apache Cassandra [9] or MongoDB [51]. Furthermore, the CAP theorem emerged as a way of understanding the trade-offs between consistency and availability of services in distributed systems when a network partition occurs. For the always-on applications, it often made sense to accept *eventual consistency* in exchange for a greater availability [71]. In big data application, database systems are often used when row-level access (e.g. CRUD operations) is required [71].

  In order to store large datasets, the data persistence layer can also use a distributed file system or object store, such as HDFS or Amazon S3 [7]. These systems offers lower cost per GB storage compared to databases and more flexibility for data formats [71]. However, they are best used when scans are the dominant access pattern, rather than CRUD operations.

- *Cluster resource management.* The cluster distributed operating system (e.g., YARN, Apache Mesos, Kubernetes).

- *Data Processing layer.* The data analysis jobs written in Hadoop MapReduce, Spark, or other tools submitted to the cluster resource manager.

A typical big data application can be built as a *workflow*, a directed graph of distributed computing jobs [60]. Each job reads one or more input datasets from the persistent storage and produces one or more output datasets. In MapReduce, if the application requires data reuse between computations (e.g., between two jobs) the intermediate result is also written to an external stable storage system. This can incur substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times. Recognizing this problem, researchers

have developed specialized frameworks for some applications that require data reuse. These frameworks only support specific computation patterns and perform data sharing implicitly for these patterns. Eventually, the general purpose computing framework Apache Spark [74] was released. Spark is the first system that allows a general-purpose programming language to be used at interactive speeds for in-memory data mining on clusters [74]. Spark is based on the concept of resilient distributed datasets (RDDs) [73] that enables efficient data reuse in a broad range of applications. The RDD abstraction allows users to explicitly persist intermediate results in memory, control the data partitioning to optimize the data placement, and manipulate them using a rich set of operators. Even though Spark increased the interactivity of many big data applications and tools, a typical job in batch-mode can last from few minutes to several hours. In fact, in classic batch-mode, data is collected and stored in the persistent storage layer and once in a while (e.g., every hour, once per day) the data is processed through one or more batch jobs. The high-latency imposed by the batch architecture is not suitable for real-time applications in which data should be computed as it is generated, in an incremental fashion.

3.2 Real-Time Streaming Architecture

Recently, many big data real-time applications emerged such as detecting fraudulent financial activity or optimizing e-commerce search results in real-time. These kinds of applications process unbounded datasets and potentially infinite *streams* of data, and they are often referred as real-time streaming applications. A real-time streaming application is usually backed by a distributed, high-performance and always available architecture. Figure 3.2 represents a real-time streaming architecture. This architecture is also called Lambda architecture [71, 46, 70]. The Lambda architecture is essentially composed by two computing *layers* or *paths* that can be realized using various big data technologies:

- *A batch (or offline) layer.* The batch layer aims at perfect accuracy by being able to process all available data when generating a computing result. This means it can fix any errors by recomputing based on all the available historical data.

- *A low-latency (or online) layer.* The purpose of this layer is to minimize latency by doing real time calculations as the data arrives into the system. The output of this layer does not guarantee accuracy and completeness as the one eventually produced by the batch layer. The result of this layer can be

replaced when the batch layer process a consistent output for the same input data (if necessary).

The Lambda architecture can also involve a third layer named *serving layer*. It combines the output from the batch and speed layers and it is used to respond to ad-hoc queries.

In the following sections we will discuss and highlight the core components of the real-time streaming architecture and what makes this architecture different from the batch-mode.

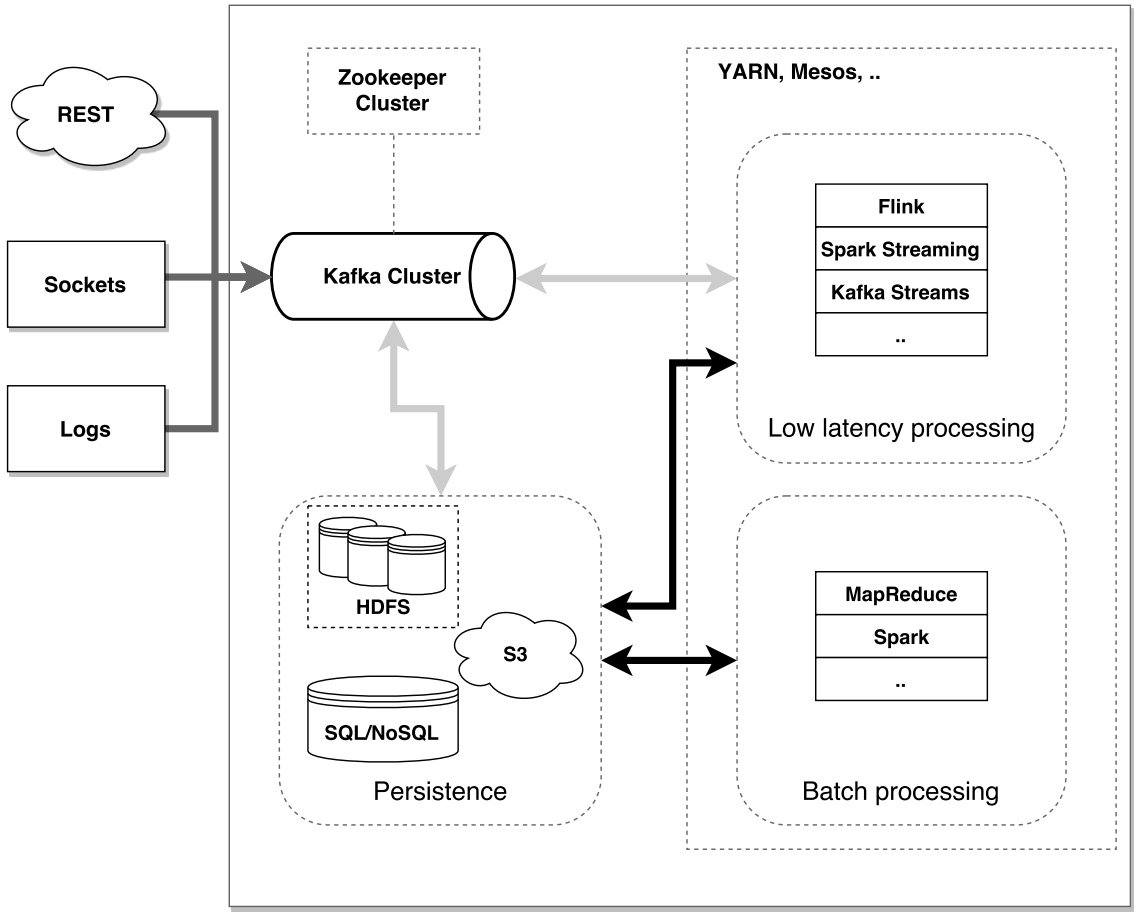

Figure 3.2: The Lambda architecture

3.2.1 Distributed Message Broker

Streams of data can arrive into the system over sockets from other servers within the environment or from outside. For example, data such as telemetry values from IoT devices or social network feeds coming from the Twitter real-time API [68] can be continuously *polled* and ingested into the system. The data could also flow into the system via REST or continuously updated log files [71]. These large streams of

data can be ingested in the system through a message queue. Message queues are *first-in, first-out* (FIFO) data structures, which is also the natural way to process data that come in streaming. Message queues/brokers organize the ingested data into user-defined *topics*: each topic has its own queue. Producers/publishers send information to the message queue at a specific topic; a consumer/subscriber receives messages every time there is an update in a topic it is a subscriber. This promotes scalability through parallelism (each topic has its own queue), and it also allows producers and consumers to focus only on topics of interest. A distributed, efficient and reliable message broker is the backbone of a high-performance streaming architecture [71]. It provides scalable, durable and temporary storage. Furthermore, it disseminates data to the different internal components as the data flow into the system. In Figure 3.2, the message broker is connected to processing or storage components of the architecture through gray arrows. The most popular message broker system used within a streaming architecture is Apache Kafka [16].

Apache Kafka

Apache Kafka is a massively scalable publish/subscribe message queue architected as a distributed transaction *log*. It benefited from years of production use and development at LinkedIn, where it started. In September 2015, the LinkedIn's deployment of Apache Kafka surpassed 1.1 trillion messages per day [71].

A Kafka topic is divided into *partitions*: messages within a partition are totally ordered. Topics partitioning is necessary in order to provide *horizontal scalability*: different partitions can reside on different machines and no coordination across partitions is required. The assignment of messages to partitions may be random, or it may be deterministic based on a key [47]. A Kafka partition is also known as a *log*, since it resembles a database's transaction commit log [47]: when a new message is published to a certain topic, it is appended to the end of the log. *Fault-tolerance* is guaranteed by replicating each partition across multiple Kafka nodes. One of the replicated partitions is elected as *leader* and it is responsible for handling all reads and writes of messages for that partition. All the updates to the partition are then synchronously propagated from the leader to a configurable number of replicas. When the leader fails, one of the *in-sync* replicas is chosen as the new leader [47].

In Kafka, a message is composed by a *key* and a *value*. When a message is published to a certain topic, the Kafka broker assigns an *offset* to that message, which is a per-partition monotonically increasing sequence number. A Kafka consumer application reads all the messages in a topic partition sequentially. For each partition, the client keeps track of the offset up to which it has seen messages, and it continuously

polls the brokers waiting for the arrival of new messages with a greater offset. The offset is periodically checkpointed to stable storage; if a consumer client application crashes and restarts, it resumes reading from its most recently checkpointed offset [47]. In Kafka, the messages are not deleted when read, so any number of readers can ingest all the messages in a topic. In fact, Kafka deletes the oldest message using a user-specified retention time (which by default is set to seven days) or a maximum number of bytes in the queue (the default is unbounded). A combination of the two approaches is also possible [71].

Kafka as a distributed system works in combination with Apache Zookeeper [24]. Zookeeper is used for tasks requiring consensus, such as leader election, and for storage of some state information.

3.2.2 Low Latency Processing

One of the core components of a real-time streaming architecture is the low latency streaming processing engine. These are scalable distributed systems that continuously process, aggregate and analyze unbounded streams of data. In the Lambda architecture, they serve as *speed layer*. The computations of these systems are usually modelled as Direct Acyclic Graph (DAG) topologies. Many streaming computing technologies have arosen such as, Apache Spark Streaming [21], Flink [10], Storm (and Trident) [23], Samza [18], Heron [42], Kafka Streams [43] and many others. All these systems usually run on top of a clustering system such as YARN or Mesos. However, these frameworks do have some differences in the underlying architecture with different performance in terms of throughput, latency and accuracy. We will discuss the peculiarities and architectural characteristics of these systems in the next chapter. For now, we can think of a streaming engine as a processing framework that is capable of processing huge volumes of data continuously as the data is flowing through the system. The latency of these systems can usually vary from hundreds of milliseconds to few seconds.

In a streaming environment, a low latency computing engine processes data that come from a distributed message broker such as Apache Kafka and then the output is written back to the broker. The stream processing results can also be saved to the persistent storage. Potentially, the input data of the streaming computation can also be ingested from the persistent storage, although this imposes higher latency than streaming through Kafka.

3.2.3 Batch Processing

A streaming architecture usually has also support for a traditional batch processing system. In fact, as we already mentioned, the core concept behind the so called Lambda architecture is to handle massive quantities of data by taking advantage of both batch and stream processing methods. A batch processing framework acts as *batch (or offline) layer*. This layer can adopt any traditional big data processing framework that are commonly used in the batch architecture and they are used to perform expensive calculations with data coming from the persistent storage. In fact, the data coming into the system through Apache Kafka can be consumed by a streaming application running on top of a streaming engine but it can also be stored in the persistent storage layer.

3.2.4 Persistence

In the Lambda architecture, the data persistence component is basically used as in the batch-mode architecture. In Figure 3.1, the black arrows show the connection from the persistence layer to the processing components of the architectures.

The persistence layer can also be used in the Lambda architecture as *serving layer* for combining the output from the batch and the speed layers.

Chapter 4

Comparing Stream Processing Frameworks for Cowbird

In the previous chapter, we have seen that the crucial component of a large-scale streaming infrastructure is a low latency stream processing engine. The big data streaming landscape is constantly growing and many distributed stream processing frameworks are available today. Since we want to extend the current Cowbird framework in a way it can support the processing of large amounts of data in real-time, we extensively studied and analyzed some of the stream processing frameworks currently available. We also built a small benchmark framework that helped us understanding which engine is the most suitable for our use case.

In this chapter, we will discuss the architectural characteristics and main features of the cutting-edge stream processing technologies. We will also describe the benchmark framework we built for evaluating some of the streaming engines described in the following sections. This framework helped us understanding which of the tested streaming engines is the most suitable for our Cowbird architecture.

4.1 Overview

Sensor monitoring, network traffic analysis and security threats detection are some of the applications that generate large amounts of data to be processed in real-time. These stream applications are characterized by an unbounded sequence of events that continuously arrive and need to be processed with a relative low latency. The advent of the Internet-of-Things (IoT) increases the need of real-time monitoring [50]. In fact, Gartner Inc. forecasts that more than 8 billion connected things will be in use worldwide in 2017 and will reach 20.4 billion by 2020 [39]. Hence, the amount of data generated cannot always be processed centrally.

A distributed architecture is desired in order to process such volume of data. In these systems, the input data is partitioned and treated in parallel. Thus, high availability and fail recovery are critical in stream processing systems. The stream processing platforms must provide resilience mechanisms against faults, such as delays, data loss, or out of order samples, which are common in a data stream. In low latency applications, the recovery should be quick and efficient, providing processing guarantees [50]. Streaming frameworks usually support one or more of the following processing guarantee semantics:

- *At-most-once.* Each input sample or message is processed at most one time. In case of failures, the sample/message is not reprocessed.
- *At-least-once.* Each input sample or message is processed at lest one time. In case of failures, the sample/message might be reprocessed multiple times and multiple results are delivered.
- *Exactly-once.* Each input sample or message is processed once and only once.

Stream processing systems have a highly optimized execution engine to provide real-time response for applications with high data-rates. In order to achieve a good processing performance and keeping latency within microseconds streaming platforms minimize communication overheads when data trasmission is required between distributed computations [50]. There are two methods of data processing in streaming processing systems [50]:

- *Micro-batch processing.* Micro-batch treats the stream as a sequence of smaller data blocks. For small time intervals, the input is grouped into data blocks and delivered to a traditional batch system to be processed.
- *Stream processing.* The stream processing analyzes and processes massive sequence of unlimited data that are continuously generated. In contrast to micro-batch, it is not limited by any unnatural abstraction.

Micro-batch engines have a higher latency in comparison to pure stream processing engines. However, fault tolerance and load balancing are much more efficient in micro-batch processors since, in case of failure, the (micro) batch assigned to the failing node can simply be reappointed to another available machine. Viceversa, in pure streaming engines, fault tolerance is performed for each processed message introducing extra overheads.

The streaming technologies landscape is very rich and there are many streaming engines available. In the following sections, we will briefly discuss the cutting-edge stream processing technologies available right now.

4.2 Apache Storm

One of the most established streaming engines is Apache Storm [23].It has been for years the Twitter's streaming backbone. This system has a pure streaming process engine with a very low latency. However, Storm has also a low throughput and at-least-once processing semantics. In order to add exactly-once semantics to Storm, Twitter created Trident: it provides a high-level abstraction on top of Storm that brings exactly-once semantics and *stateful* stream processing (the engine keeps an internal state of the data being processed) to the platform. This is made possible since Trident treats pure streams of data as sequences of micro-batches. Micro-batch incurs to extra latency, even though it increments the throughput. In reaction to these issues, Twitter in 2015 announced Heron [42]. Heron was developed internally at Twitter and it provides low latency and high throughput but it has support only for at-least-once and at-most-once processing semantics [38, 49].

4.3 Apache Samza

Apache Samza [18] is an open-source stream processing framework originally developed in conjunction with Kafka at LinkedIn [47]. A Samza job consists of a Kafka consumer, an event loop that calls the application code to process incoming messages, and a Kafka producer that sends output messages back to Kafka [47]. For each partition of an input topic of a Samza job, a computing unit named StreamTask is allocated. Each task instance consumes messages from just one partition; the task instances can run on the same node or they can be distributed across multiple machines. In Apache Samza, fault tolerance is guaranteed by Kafka. Samza does not implement its own network protocol for transporting messages from one operator to another; all the communications between the computing nodes depend on Kafka [47]. Since Apache Samza highly relies on Kafka, it is considered a full-streaming processing engine. However, it also supports batch processing [47]. Kafka initially supported only at-least-once semantics, thus Samza has the same degree of processing guarantee. However, Confluent, the company behind Kafka, recently announced the introduction of exactly-once semantics in Apache Kafka in its 0.11 release.

Samza implements durable state within streams of data using a key-value store abstraction. In particular, Samza uses the RocksDB [56] embedded key-value store, which provides low-latency and high-throughput access to data on local disk [47].

4.4 Apache Spark Streaming

Apache Spark Streaming [21] extends the Apache Spark framework by introducing the discretized streams (D-Streams) programming model [72]. The key idea behind D-Streams is to treat a streaming computation as a series of deterministic batch computations on small time intervals or windows. A discretized stream is just a series of RDDs grouped together; this series of RDDs contains data arrived within a certain time interval. D-Streams provide both *stateless* operators which act independently on each time interval, and *stateful operators*, such as aggregation over a defined sliding window, which operate on multiple intervals and may produce intermediate RDDs as stream state. Spark Streaming takes care of distributing the state in the cluster, managing it, transparently recovering from failures, and providing fault tolerance.

Fault tolerance is guaranteed in Spark Streaming in the same way it is implemented in Spark; a (micro) batch is simply reassigned to another available node in case of failure. In particular, D-Streams uses an approach called *parallel recovery*. The system periodically checkpoints RDDs state, by asynchronously replicating them to other nodes. When a node fails, the system detects the missing RDD partitions and launches tasks to recover them from the latest available checkpoint. Multiple fine-grained tasks can be launched concurrently in order to compute different RDD partitions on different nodes [72].

Spark Streaming supports exactly-once sematincs, it has a very high throughput but also a notable latency because of its micro-batching engine [50, 26, 19].

Spark Streaming also supports full batch processing. Thus, with Apache Spark it is possible to combine stream with batch processing within a single engine.

4.5 Apache Flink

Apache Flink [10] is an open-source stream processing framework. It is a full stream processing engine and it is based on its majority on the Google's Dataflow Model [64]. Flink supports a full-fledged and efficient batch processor on top of a streaming runtime. Flink executes batch programs as a special case of streaming programs, where the streams are bounded (finite number of elements) [29, 30]. Both batch and streaming Flink programs eventually compile down to a common representation called the dataflow graph [29, 30]. The dataflow graph is executed by Flink's runtime engine and it is a directed acyclic graph (DAG) that consists of stateful operators and data streams that flow through these operators: the result

stream of an operator is then consumed by another operator in the graph. Since dataflow graphs are executed in a distributed fashion, operators are parallelized into one or more parallel instances called *subtasks* and streams are split into one or more *stream partitions* (one partition per subtask). Stream operators in Flink can be both stateful and stateless.

Flink supports the notion of *windowing* as in the Google's Dataflow model. Windows split the stream into *buckets* of finite size, over which Flink can apply computations. Flink supports four types of windows when dealing with unbounded data:

- *Tumbling Windows.* Tumbling windows have a *fixed* size and do not overlap (e.g., five minutes windows, hourly windows or daily windows).
- *Sliding Windows.* Sliding windows are defined by a window size and slide period (e.g., hourly windows starting every minute). The period may be less than the size, which means the windows may overlap.
- *Session Windows.* A session window groups elements by sessions of activity. Session windows do not overlap and do not have a fixed start and end time, in contrast to tumbling windows and sliding windows. Instead a session window closes when it does not receive elements for a certain period of time (i.e a gap of inactivity) occurred.
- *Global Windows.* Global windows assigns all elements marked with the same key to the same single global window. Since a global window does not have a natural end at which Flink could process the aggregated elements, this windowing mechanism works by specifying a *trigger* that indicates when the elements grouped in the window are ready to be processed.

Windowing mechanisms are time-based. Apache Flink supports three different notions of time in streaming programs:

- *Processing time.* Processing time refers to the system time of the machine that is executing a certain operation. When a streaming program runs on processing time, all the time-based operations (like time windows) will use the system clock of the compute node that runs the respective operator. Processing time is the simplest notion of time and requires no coordination between streams and machines. It provides the best performance and the lowest latency.
- *Event time.* Event time is the time that each individual event occurred on its producing device (e.g., when a data record is produced by a smart city sensor). This time is typically embedded within the data records before they

enter Flink and that event timestamp can be extracted from the record. For example, an five minutes event time window will contain all records that carry an event timestamp that falls into that five minutes, regardless of when the records arrive into the system and in what order they arrive. *Event time* gives correct results even on out-of-order events or late events. In *event time*, the progress of time depends directly on the data, not on any system clocks. *Event time* is implemented through *watermarks*. Watermarks flow as part of the data stream and carry a timestamp t. A Watermark(t) declares that event time has reached time t in that stream, meaning that there should be no more elements from the stream with a timestamp $t' \leq t$ (i.e., events with timestamps older or equal to the watermark). *Event time* programs must specify how to generate*Event Time Watermarks*.

Event time processing often incurs a certain latency, due to the arrival of late events and out-of-order events.

- *Ingestion time*. Ingestion time is the time that events enter Flink. Ingestion time sits conceptually in between event time and processing time.

  Compared to processing time, it is slightly more expensive, but gives more accurate results. Ingestion time uses stable timestamps (assigned once at the source), thus, different window operations over the data records will refer to the same timestamp. In processing time, viceversa, each window operator may assign the record to a different window (based on the local system clock or on any transport delay).

  Compared to event time, ingestion time programs cannot handle any out-of-order events or late data, but the programs don't have to specify how to generate watermarks.

In Apache Flink, fault tolerance is guaranteed though a mechanism called Asynchronous Barrier Snapshotting (ABS) [29]. In ABS, Flink takes a snapshot of the state of operators, including the current position of the input streams at regular intervals. In particular, barriers (control records) are injected into the input streams (similar to what happens for event-time watermarks) and they separate the stream to the part whose effects will be included in the current snapshot and the part that will be snapshotted later. When an operator receives a barrier it writes its internal state to durable storage (e.g., using RocksDB, directly to HDFS). Once the state has been backed up, the operator forwards the barrier downstream. Eventually, all operators will take a snapshot of their state and a global snapshot will be completed.

In case of failures, all the operators are reverted to their respective states taken during the last successful snapshot. Each operator then restarts consuming the input streams starting from the latest barrier for which there is a snapshot. ABS provides exactly-once semantics and it is completely decoupled from the mechanism used for reliable storage, allowing state to be backed up to different type of systems.

Apache Flink has been demonstrated [26, 19] to be extremely high-performance. It has a high-throughput and the lowest latency among all the others streaming frameworks.

4.6 Kafka Streams

Recently, the Streams API [43] has been added in Apache Kafka. It is a Java library that can be used to build highly scalable, fault-tolerant and distributed streaming applications. The main difference Kafka Streams has in comparison to other streaming frameworks is that it does not need a processing cluster or a special-purpose infrastructure. In fact, Kafka Streams applications are normal Java applications and they can be packaged, deployed, and monitored like any other Java application. However, Kafka Streams can be also deployed on a cluster system like any other large-scale stream processing engine.

Kafka Streams relies on the Kafka cluster for load balancing, fault tolerance and to guarantee stateful operations. In fact, if multiple instances of the same Java application use the same Kafka cluster, Kafka Streams recognizes them and distribute the computation expressed via the Streams API among the available nodes. Such scenario is very common in micro-services architecture, where multiple instances of the same application are replicated over many machines in the cloud. Kafka Streams provides exactly-once semantics with low latency within a light-weight Java library that can be embedded in any application with the only requirement to have an up-and-running Kafka cluster.

4.7 Comparison

Choosing the right streaming processor can be very tough since there is no silver bullet for any use case. So we defined a set of properties that the streaming engine must have in order to be integrated with Cowbird. We then selected three systems that fulfill our requirements, and we performed further performance evaluations of these. The properties the streaming engine must have in order to be integrated with Cowbird are:

- *Open source.* The system has to be open sourced.

- *High-performance.* We want a streaming engine that is capable of processing a high-volume of SWAN expressions in a short time frame. Hence, we are interested in a streaming engine with a high *throughput* and very low *latency*.

- *Exactly-once stream processing semantics.* Even though, for many IoT use cases the exactly-once semantics is not strictly required, we want our system to support the real-time analysis of data coming from critical sensors where correctness is a priority. Furthermore, a streaming framework that support exactly-once processing can also be set to a less stringent semantics.

- *Stateful operators.* We want to keep an internal state of the streams of data generated by sensors related to a SWAN expression.

	Storm	Storm + Trident	Heron	Samza	Spark Streaming	Flink	Kafka Streams
Model	Native	Micro-batching	Native	Native	Micro-batching	Native	Native
Latency	Low	High	Low	Low	High	Low	Low
Throughput	Low	Medium	High	High	High	High	High
Guarantees	At-most-once, At-least-once	Exactly-once	At-most-once, At-least-once	At-most-once, At-least-once	Exactly-once	Exactly-once	Exactly-once
Stateful Operators	No	Yes	Yes	Yes	Yes	Yes	Yes

Figure 4.1: Streaming engines overview.

An overview of the main features of the streaming frameworks discussed in the previous sections can be found in Figure 4.1. We decided to evaluate Apache Spark Streaming, Flink and Kafka Streams. Apache Spark Streaming and Flink can be both good candidates because of their performances [50, 26, 19] and their maturity. They are also very flexible since they can be used as a single engine for both streaming and batch processing. Hence, they could be used as a single processing engine in the Lambda architecture. We chose also to test Kafka Streams because this system can be a good candidate for light-weight stream processing and we did not find any available benchmark of this system.

4.8 Benchmarking Streaming Engines

In this section we will portray the streaming engines benchmark framework we designed. We will report the results of the tests we performed on Apache Spark, Flink and Kafka Streams.

4.8.1 The Benchmarking Framework

[26, 19] designed a streaming engines benchmark that resembles their application scenario. Similar to what they have done, we designed a framework for evaluating streaming engines that recall our use cases. This approach can help us understanding which streaming framework is the most suitable for Cowbird. We tested Apache Spark Streaming (version 2.1.0), Apache Flink (version 1.2.0) and Kafka streams (Kafka version 0.10.2.0)

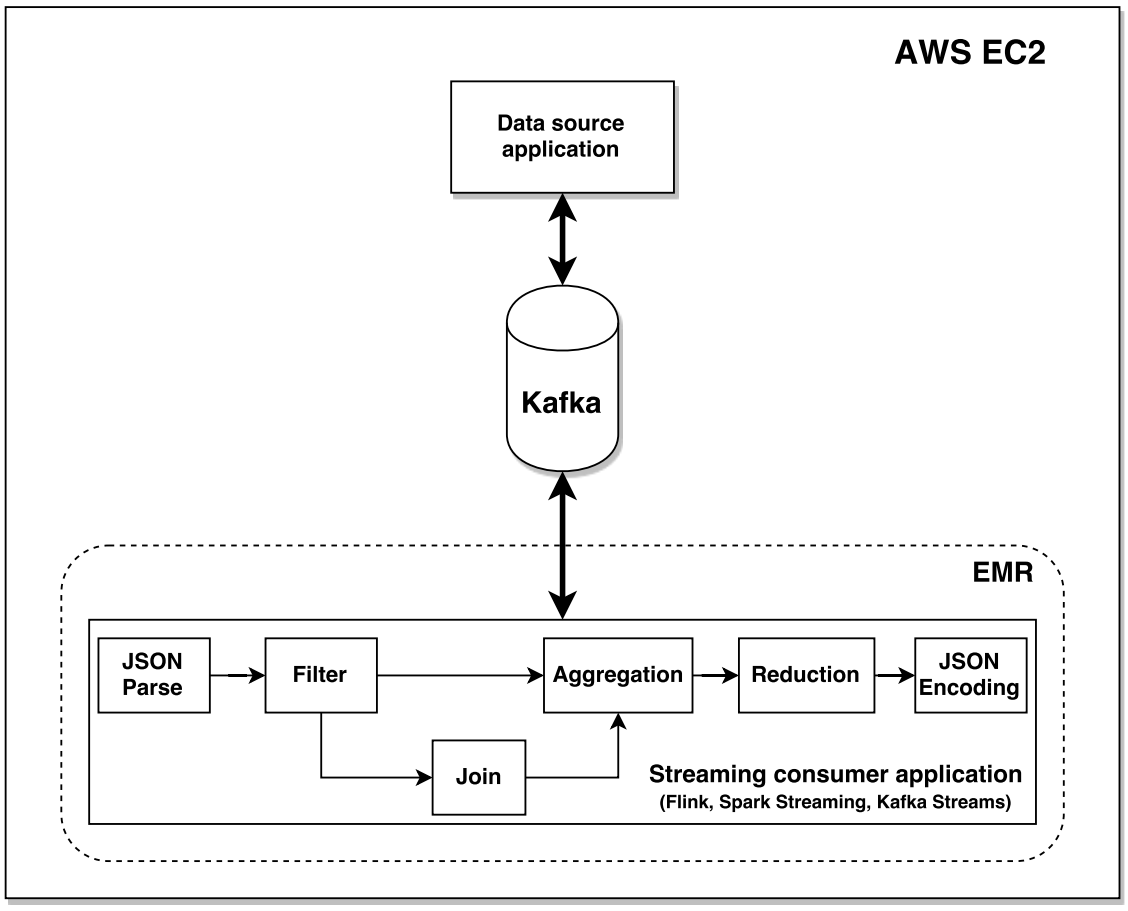

Figure 4.2: The streaming engines benchmark architecture.

Our benchmark framework contains (see Figure 4.2):

- A data source application that generates contents for a Kafka topic; each message generated contains an id that identifies a source, a double value, a

reduction operation (i.e., MEAN, SUM, MAX and MIN), and the number of values required for applying the reduction operation (e.g., 1000 values coming from a source with a certain id). The data source application generates a certain number of events for a certain source and sends those to the Kafka broker. The app can be tuned with some parameters provided by the user (e.g., number of sources, number of messages each source generates, the reduction operation to be performed on the data).

- A streaming consumer application that runs on the streaming engine and applies the reduction operations. The application emits on Kafka the result for a certain source along with the time spent to compute the reduction operation. The time spent for computation is calculated as the difference between the time at the reduction evaluation and the timestamp of the first message received by Kafka from a certain source.

A consumer application has been designed for each of the three systems that have been tested. The consuming applications keep an *incremental state* for each stream marked with a certain identifier. The incremental state stores only the partial result of the data that have been ingested so far into the system; this approach avoids to store all the records of data generated. The incremental state is calculated according to the reduction operator indicated for that stream. For example, in order to calculate the MEAN, the state will store the sum of the data values and the number of records that have flowed into the streaming engine. All the messages exhanged through the Kafka broker are in JSON format. The implementation of such benchmark framework can be found at [59]

4.8.2 Setup

The evaluation has been done in a relatively simple setup on Amazon Web Services (AWS) [8]. The streaming engines are executed through EMR [5] on top of a small cluster composed of one master and two workers (m3.xlarge 4 vCPU, 15 GiB memory, 80 SSD GB storage each).

Apache Kafka is executed in one broker instance along with Apache Zookeeper (m4.xlarge 4 vCPU, 16 GiB memory).

The streaming frameworks are executed using the default settings and we focused on writing correct, easy to understand programs without optimizing each streaming implementation to its full potential.

4.8.3 Results

We stressed all the three systems with different loads of messages containing double values. We use double values since many sensors produces measurements in such data format. In all the tests we performed the input messages have been processed using the MEAN history reduction mode. The batch duration in the Spark streaming was set to 1 second. In our tests, we focused on measuring how the tested systems react in terms of latency in situations of high input load.

Figure 4.3 and Figure 4.4 show the latency for various numbers of input messages per second. The results we obtained for Spark Streaming framework and Flink is similar to what was achieved by [26, 19].

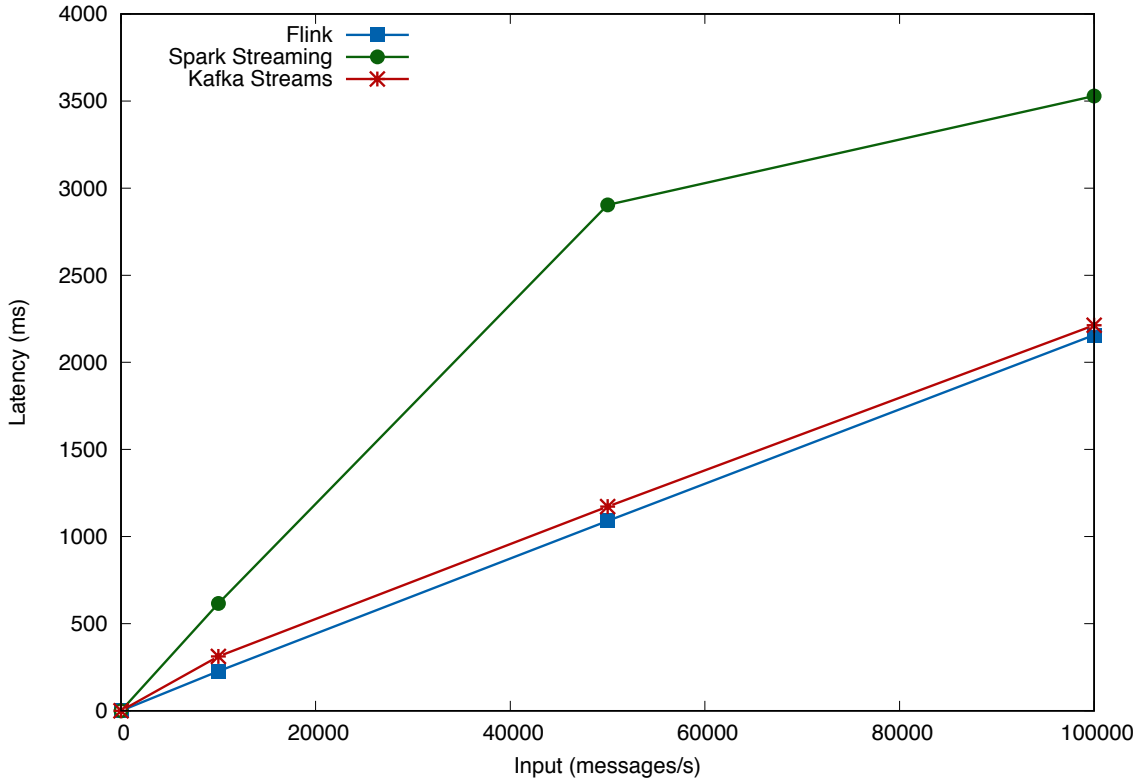

Figure 4.3: Latencies evaluation up to 100K messages/s

Kafka Streams performs very well and with a reasonable small input obtained performance similar to Apache Flink. Both Kafka Streams and Flink respond quite linearly. On the other hand, in Figure 4.3 Spark Streaming behaves in a stepwise way; this could be a direct result from its micro-batching design. On drastically increasing the amount of input messages per second, all the three systems start suffering from the backpressure effect (the system is receiving data at a higher rate than it can process) also because of the limited setup configuration.

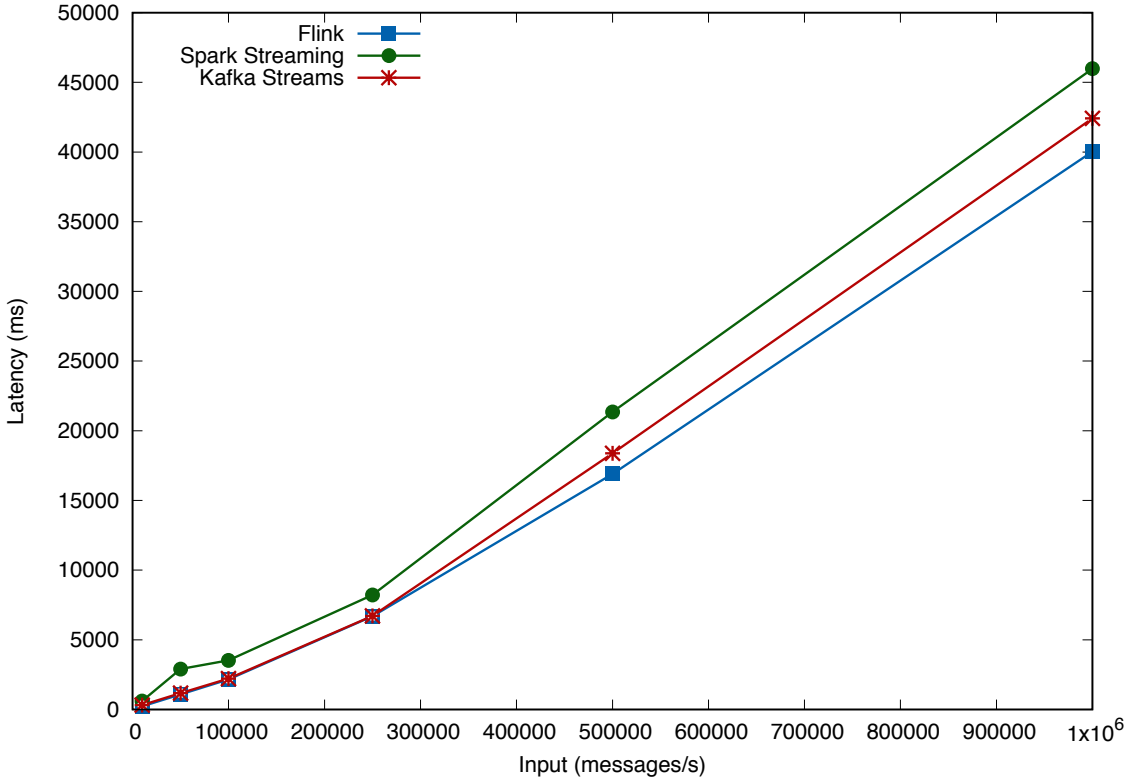

Figure 4.4: Latencies evaluation up to 1M messages/s

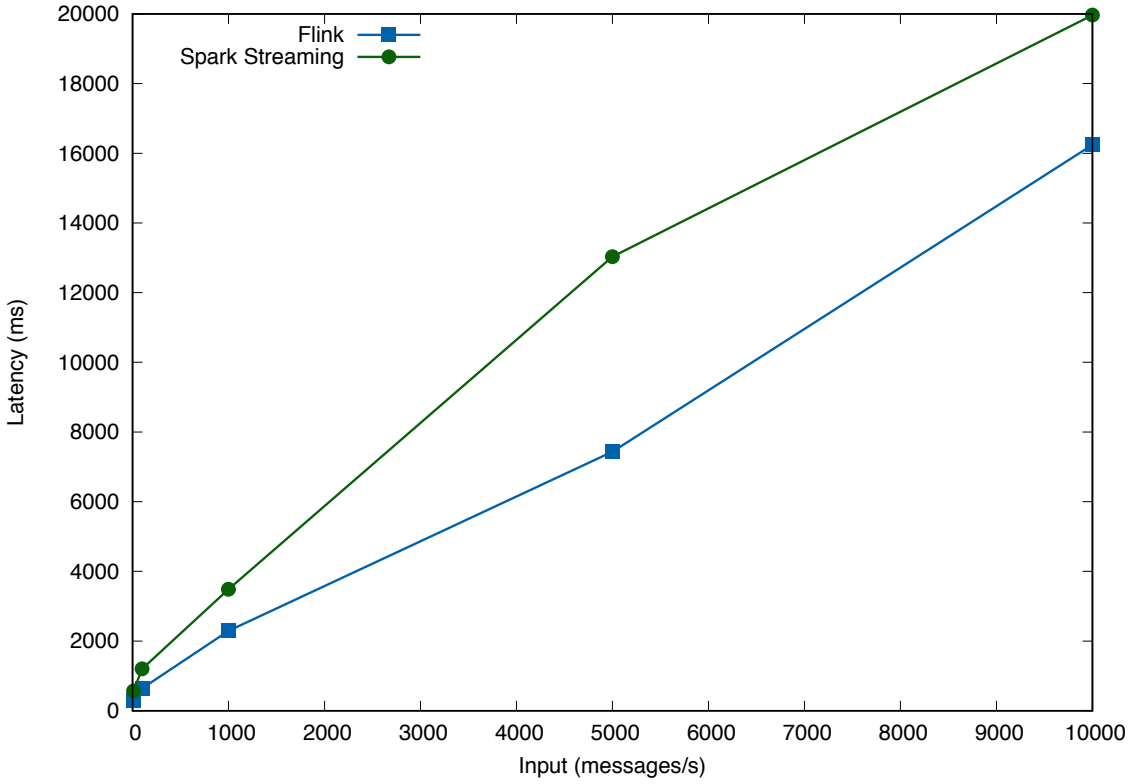

Figure 4.5: Two streams JOIN-REDUCTION evaluation

Figure 4.5 shows the evaluation result for join operations in Spark Streaming and Flink. In particular, the incoming streams of data (we set a window size of 1 second in Flink) are continuously joined with a set of 500MB pre-generated messages. The resulting streams of messages are then processed as usual. We notice that Flink has lower latency compared to Spark Streaming. We did not evaluate the join operation in Kafka Streams because such operator is not available for streams of data but only for tables.

The benchmark uses a naive setup and for this reason it can only give a sneak peek of the actual performance capabilities of the analyzed systems. Some of the achieved results are unacceptable or highly impractical for a real-time streaming application. However, this benchmark, even though executed on a relatively small scale, helped us understanding how the tested streaming engines can behave. Furthermore, in order to have a more detailed portrait of the tested engines fault tolerance and resources utilization should be evaluated as well.

According to the results we obtained, Apache Flink is the streaming framework that better fits our uses cases. Nonetheless, it is important to outline that our naive benchmarking application is much less complex than the Cowbird framework. Furthermore, Cowbird evaluates sensors data records over a history time window while our benchmarking application processes a certain number of data values generated by a certain source without considering any windowing parameter.

Chapter 5

Design & Implementation

In the previous chapter we described some of the available stream processing frameworks. We evaluated some of them and eventually we decided to integrate Apache Flink in Cowbird. In this chapter we are going to illustrate how SWAN-Song expressions evaluation can be ported to Apache Flink. In general, we will portray how we extended Cowbird in order to accommodate data sensing and SWAN-Song expressions evaluation on a large scale. We will discuss the design choices we made and the technical challenges we tackled.

5.1 Hybrid-Cowbirds

The existing Cowbird framework is designed to be executed on a single node machine. We designed a distributed architecture for supporting the real-time evaluation of a massive number of SWAN-Song expressions in Cowbird. We extended the Cowbird cloud architecture (see figure 5.1) by adding two main components:

- *Cowbird Fog.* A distributed layer responsible of receiving SWAN-Song evaluation requests from smartphones, polling sensors data and computing expressions. It is designed to bring expression evaluation as close as possible to the data generation in order to reduce latency. The Fog layer can decide to offload the SWAN-Song expression evaluation to the Streams layer according to some expression parameters (e.g. sensor data generation frequency, SWAN-Song expression time window). If an expression is offloaded to the Streams layer, the activated sensors threads in the Fog layer will *asynchronously* publish their sensed data to a Kafka broker that mediates communications between the Fog and the Streams layer.

- *Cowbird Streams.* A high-performance and scalable layer for evaluating SWAN-

Song expressions in real-time on a cluster using Apache Flink. It evaluates sensor data provided by the Fog layer and streamed through the Kafka broker. The result of the evaluation is sent back to Kafka and then made available to the Fog.

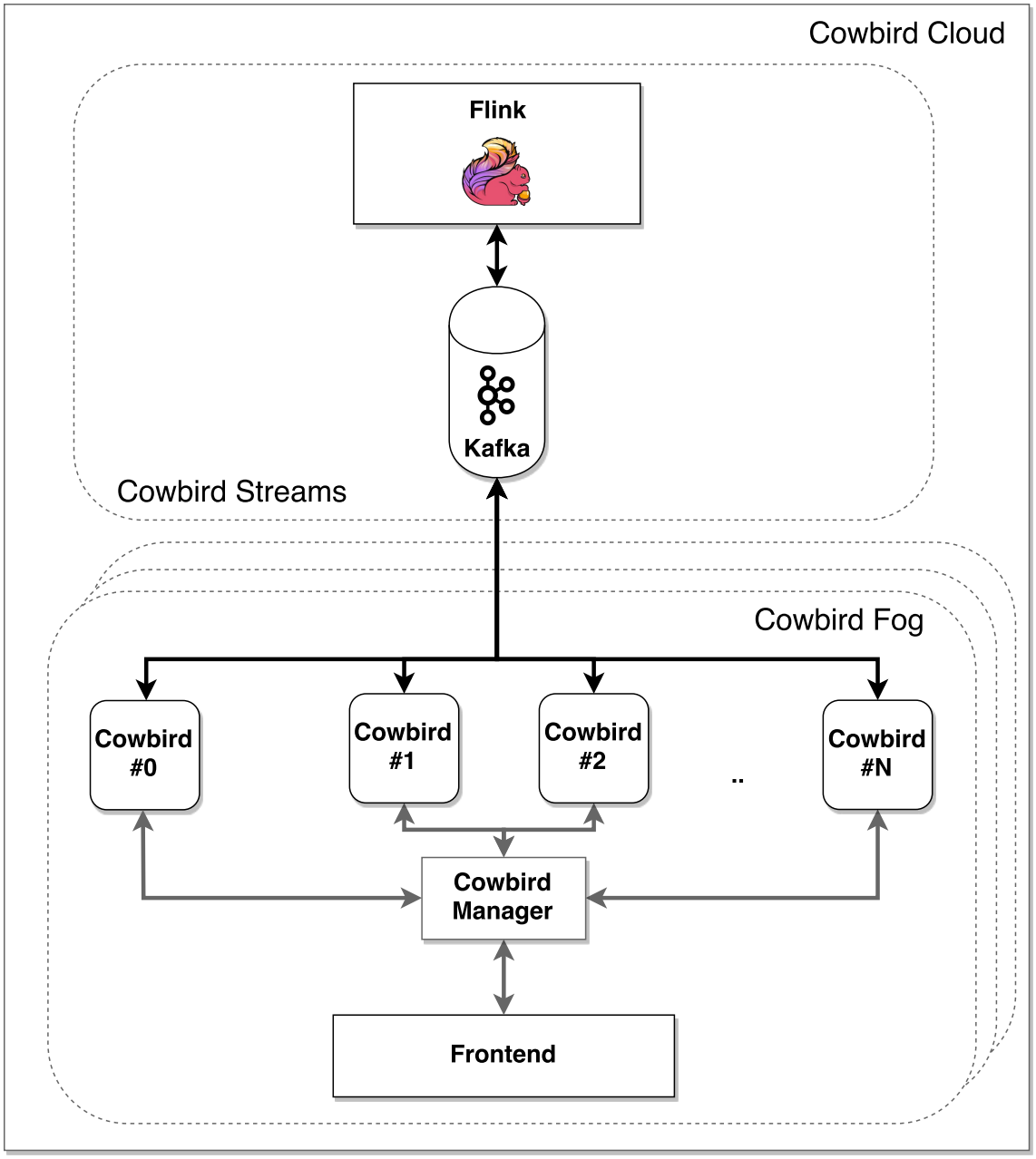

Figure 5.1: Hybrid-Cowbirds cloud architecture

The Hybrid-Cowbirds model differs from the other streaming architectures described in Chapter 3 and Chapter 7. Hybrid-Cowbirds is a *hybrid* architecture that combines large-scale modern streaming analysis techniques with more traditional processing systems. We added the Fog layer for performing some relative simple evaluation without involving an actual cluster system. The Fog layer can reduce

latency bringing computation close to the data generation and to the users (and their smartphones). The Fog layer could be geographically distributed in order to provide high availability and high performance while the Streams layer will be a more centralized entity used for more complex evaluations that the Fog layer cannot handle. Furthermore, the Fog layer is extremely portable since it could be deployed on any device that runs a JVM.

The remainder of this chapter will describe the internal details of the Hybrid-Cowbirds architecture.

5.2 Cowbird Fog

In Chapter 2, we have seen that Cowbird makes large usage of threads for polling sensor data from external endpoints. This approach allows Cowbird to poll many different sources concurrently and in an asynchronous fashion. In Cowbird, a multi-threaded mechanism is adopted for the SWAN-Song expressions evaluation. However, having many active threads on a single computing node can incur in a certain system overhead. Furthermore, when polling live data, fetching data values might be delayed by multi-threading. It might be the case that new data records are available but Cowbird needs to wait for the relative sensor thread to become active again. A delay in fetching the data will induce a delay in processing the expression. In reaction to these issues, we decided to *scale horizontally* the Cowbird cloud instance.

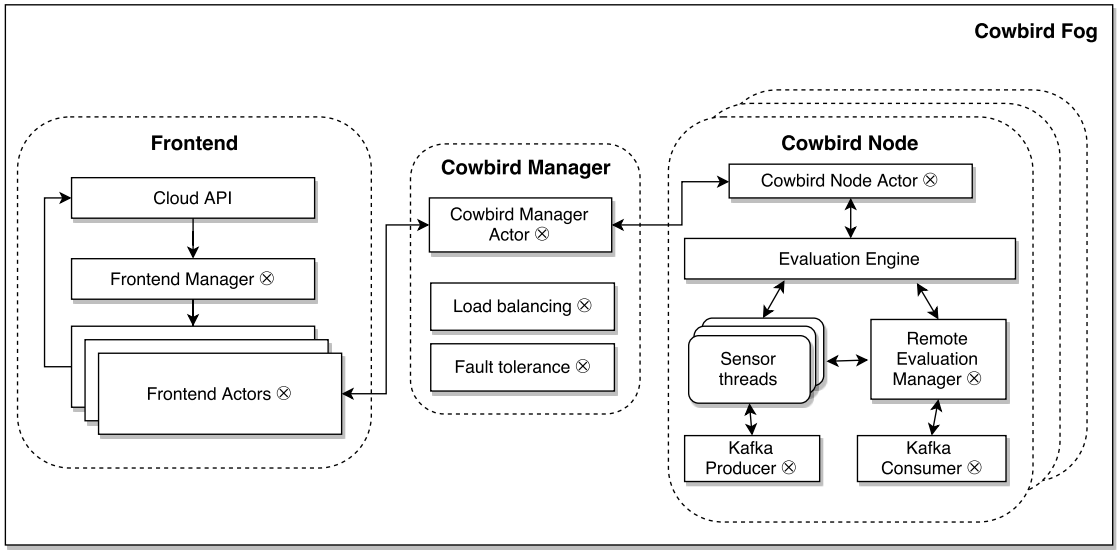

Figure 5.2: Cowbird Fog internal details. New components and features are marked with ⊗.

The Cowbird cloud instance is built using the Play framework with Java. For

these reasons, we decided to scale out Cowbird using Akka [3]. Akka is a set of open source libraries for building highly concurrent, distributed, and resilient message-driven applications using the *actor model* [2]. The *actor* is the primitive building block that forms the basis of the actor model. An actor is a concurrent entity that keeps an internal state that is not shared with other actors. The actor propagates data or events with other concurrent entities explicitly via asynchronous messages without blocking. Each actor processes the receiving messages one by one and it modifies its internal state. An actor is usually mapped to a thread; multiple actors can actually share the same system thread. The actor model can scale from processor cores to networks.

In the original Cowbird implementation all its functionalities such as receiving SWAN-Song evaluation requests, sensing sensor data and computing the expressions are abstracted as a single instance. In the new distributed architecture, we structured the Cowbird Fog layer as follows (see Figure 5.2):

- *Frontend.* The Frontend is in charge of receiving the SWAN-Song expressions evaluation requests from smartphones. The Frontend mainly consists of a Play framework controller used in the original Cowbird implementation that routes a specific URL to a functionality. The Frontend is managed by the Frontend Manager. Every time a SWAN expression evaluation request is received at the Frontend, the Frontend Manager will spawn a *Frontend Actor* responsible of the communication between the Frontend and the Cowbird Manager. In particular, the Frontend Actor registers the SWAN-Song to the Cowbird Manager and it will be notified when a new result value for the expression is available. The system also supports multiple active Frontend instances.

- *Cowbird Node.* The Cowbird Manager assigns SWAN-Song expressions to be evaluated to a Cowbird Node. On receiving an expression, through the evaluation engine, the Cowbird Node will start the relevant sensor thread that will keep polling external web data from an endpoint. When the Cowbird Node computes new results for a registered expression, it will send back the results to the Cowbird Manager; the result is sent back to the manager with the same communication optimization strategy described in Chapter 2. We assume that the evaluation of the expression changes with low frequency and for this reason we direct all the result through the Manager. However, with a small change the result could be sent back directly to the Frontend. This optimization is possible since every component in the architecture is backed by an Akka actor. Messages between Akka actors can be exchanged directly even though they are running on different machines. The Cowbird node runs

a *Remote Evaluation Manager* that determines whether or not a SWAN-Song expression should be offloaded for evaluation to the Cowbird Streams layer. A SWAN-Song expression (or just a part of it) is offloaded to the Streams layer if it refers to a sensor marked as *high-frequency* or if the expression has to be evaluated over a time window that is longer than a certain configurable threshold (e.g. 10 minutes).

The Cowbird Node integrates a Kafka *producer* and a Kafka *consumer* that coordinate the communication with the Streams layer.

Cowbird Nodes can be dynamically added or removed for horizontal scalability.

- *Cowbird Manager*. The Cowbird Manager is responsible for the coordination of the workload distribution among all the Cowbird nodes in the system. When a new SWAN-Song expression evaluation request is received from the Frontend, the manager assigns it to the *least-busy* Cowbird node in the network (i.e., the Cowbird Node with the minimum number of active threads). The Cowbird Manager is also responsible for monitoring the entire Fog layer. The manager detects failing Cowbird Nodes and Frontend(s). In reaction to these scenarios, the manager can redistribute the workload to another active node in the system or it can stop the sensor data sensing and expression evaluation in case of a failing Frontend. The Cowbird Manager can be deployed in *high-availability mode* to guarantee a certain level of fault-tolerance. In this scenario, when the manager fails one of the Cowbird Node will be elected as the manager.

The above described components can be deployed all three in a single machine or they can be distributed over many computing nodes for achieving better performance. Many deployment optons are possible and thus the Fog layer is extemely versatile. This is possibile thanks to the Akka framework flexibility. The scenario where different Fogs are distributed geographically for performance and availability is also possible. Hence, Cowbird cloud could meet the concept of *fog computing* [27].

5.3 Cowbird Streams

The Cowbird Streams layer is the high-performance module of our new Cowbird implementation. As we already mentioned, it consists of a Kafka cluster and an Apache Flink application. The Kafka cluster is responsible for receiving the sensed data from the Fog layer; the Flink application processes the incoming streams and writes the result of the evaluations back to Kafka.

The Cowbird Streams layer needs the Fog layer in order to exploit its sensing capabilities.

We ported the SWAN-Song evaluation mechanism to Apache Flink. In particular, we implemented the core SWAN-Song functionalities using the Flink API. In order to bring SWAN-Song evaluations on top of Flink, we need to stream the registered expressions along with the sensed data from the Fog layer through Kafka. The Fog layer exchanges *messages* with the Flink job running the SWAN logic. All the messages are exchanged in JSON format through the Kafka broker. We designed three different type of messages:

- *Sensor message*. Every time a sensor thread that belongs to a SWAN-Song assigned for evaluation to the Streams layer generates a data record it is encapsulated in a sensor message. The sensor message contains the data value and the identifier of the expression it belongs to. Each SWAN-Song expression has an unique identifier assigned by the Cowbird Manager.

- *Control message*. The control message is sent from the Cowbird Node when it wants to register/deregister a SWAN-Song expression to the Streams layer. It contains the identifier of the expression and the type of history reduction along with the window size.

- *Result message*. The result message contains the result of the evaluation. It is sent from the Flink job every time a new result is available for a certain expression.

When the Cowbird Node receives a new expression to evaluate from the Cowbird Manager, the remote evaluation engine decides if the expression or part of it should be evaluated by the Streams layer. If this is the case, the Cowbird Node will register the SWAN-Song through a control message to the Streams layer. The evaluation engine will then start the relative sensors threads; the sensed data instead of being stored and evaluated locally will be sent asynchronously to the Flink job running on Cowbird Streams. The Cowbird Node will be notified when a new result for the registered expression is available.

5.3.1 SWAN-Song Evaluation on Cowbird Streams

Implementing the entire SWAN logic on top of a streaming engine can be very inefficient. SWAN expressions are parsed by the Cowbird Fog and parsing them again would introduce extra overheads. Thus, we designed the system in a way that the Fog layer can offload only some *well-known* types of SWAN expressions

(or subexpressions of a more complex expression) to the Streams. The Fog layer will then be in charge to compute the final result of an expression with the partial results computed by the Streams layer. In particular, our Streams layer is capable of computing the following types of SWAN-(sub)expressions: SWAN simple value expression, SWAN simple tristate comparison expression and SWAN complex tristate comparison expression.

SWAN Simple Value Expression

This is the simplest kind of SWAN-Song expression that can be evaluated by the Cowbird Streams layer. Such expressions only contain the basic SWAN predicates but they don't support the ALL or ANY reduction mode. An example of such expressions can be:

$$cloud@thingspeak : field?\, channelid =' 1' \# field =' 1' \{MEAN, 3600000\} \quad (5.1)$$

If a SWAN simple value expression is part of a more complex expression, the final result would be computed by the Fog layer when the result of the simpler expression is evaluated on the Streams and then sent back. For example, consider the following expression:

$$cloud@thingspeak : field?\, channelid =' 1' \# field =' 1' \{MEAN, 3600000\} > 50.0 \quad (5.2)$$

In SWAN-Song (5.2) only the left-side of the expression would be offloaded to the Streams layer. The final comparison (if the MEAN value is greater than 50) would be computed on the Fog layer when the result of the left-side expression is received back.

SWAN Simple TriState Comparison expression

The SWAN simple tristate comparison expression is a comparison expression that consists of a SWAN-Song expression with ANY or ALL history reduction mode and a constant value. For example, consider the following expression:

$$cloud@thingspeak : field?\, channelid =' 1' \# field =' 1' \{ANY, 3600000\} > 50.0 \quad (5.3)$$

The expression (5.3) is completely evaluated on the Streams layer and when a result is available it is pushed back to the Cowbird Fog layer. This approach is essential in order to prevent the Cowbird Streams layer to return a long list of time-stamped values.

In order to maximize efficiency and performance some shortcuts have been implemented in the evaluation of such expression. In particular, in case of a simple comparison expression that has the ANY history reduction mode the expression is evaluated immediately when a data record that makes the expression TRUE flows into the Streams layer (without evaluating also the other data values generated within the window history length). On the other hand, a simple comparison expression that has the ALL history reduction mode is immediately evaluated if a record that makes the expression FALSE enters the system.

SWAN Complex TriState Comparison Expression

The SWAN complex tristate comparison expression represents a more complex type of comparison expression. It consists in comparison expressions that involve the ANY or ALL history reduction modes. For example, consider the following expressions:

$$cloud@mysoundsensor1\{ANY, 3600000\} > cloud@mysoundsensor2\{ALL, 2400000\} \tag{5.4}$$

The expression (5.4) is completely evaluated on the Cowbird Streams layer (if offloaded).

5.3.2 SWAN-Song Streaming-Oriented Evaluation

The new Cowbird implementation supports the same set of operations provided by the previous generation of Cowbird cloud. In fact, offloading the three types of (sub)expressions described above to the Streams layer and letting the Fog layer compute and aggregate the partial results (if applicable) leads to the same exact result of the traditional SWAN evaluation.

The SWAN features are implemented on Apache Flink using its low-level operations API (i.e. ProcessFunction). The Flink implementation keeps an internal state for each stream of sensors data using the RocksDB key-value store [56]. This state backend can store very large state that exceeds memory and it spills to disk efficiently. All the SWAN features are implemented in Apache Flink using *processing time* in order to guarantee real-time performance even if the data records are delayed. However, event time can be configured by the user.

We designed two different implementations of how the Cowbird Streams processes SWAN-Song simple value expressions. One implementation, that we called *core implementation*, reproduces exactly the way SWAN-Song expressions are processed in the original SWAN framework. In particular, for each expression the

system collects all the sensor values generated within the history time window and the history reduction mode is then applied. This approach is very precise and it allows the system to continuously generate an expression result based on the values stored that belong to the history time window. However, this approach is not really *streaming-oriented*; in fact, streaming applications should incrementally aggregate partial results while new data records enter the system. In reaction to this, we designed a *streaming-oriented* SWAN-Song evaluation that can be executed on Cowbird Streams. The philosophy of this implementation is to store the partial results of each stream of data related to a SWAN-Song expression instead of all the data values generated by the sensor within a certain history time window. For example, if we want to compute the MEAN value of the data generated by a sound sensor in the last five seconds we just need to count how many data occurrences have been generated by the sensor and the sum of the data values. Another example can be an expression with the MAX history reduction mode set. In this scenario, we just need to store a partial maximum value and every time a greater value enters the system we can simply replace the partial maximum. This approach can prevent the system to leave the ground to the back pressure effect and it drastically reduces the amount of storage required by each stream for keeping its internal state.

The MAX, MIN, MEAN history reduction modes can very easily adopt this streaming-oriented evaluation approach. However, the MEDIAN operator requires all the data records in order to compute the median value. We decided to design a MEDIAN operator implementation that approximates the median of the data streams. In some contexts, such as biology or genetics, robust statistical procedures are required to process data. Applications that use these kinds of sensitive data are not really suitable for adopting a MEDIAN approximatated value. However, in scenarios such as enviromental monitoring, agriculture and industrial monitoring applications [67] an approximation could be a very attractive alternative. In fact, these types of applications are usually characterized by a large number of sensing devices and they can tolerate a lower level of accuracy. In the literature, there are many approaches designed to approximate sensor values [31, 32]. We approximated the MEDIAN history reduction mode using the remedian algorithm [57]. This algorithm has been proven to be very efficient [28] and it only requires $\mathcal{O}(\log n)$ space to compute the median value. This technique can be used for both batch and stream processing.

The remedian algorithm requires k arrays of size b that are continuously reused. The data enter at the first array that is filled with the first b observations. Then, the median of these b observations is stored in the first element of the second array.

The first array is then used again for the second group of b observations, the median of which will be put in the second position of array 2. After some time array 2 is full too, and its median is stored in the first position of array 3, and so on. When the k-th array is complete its median becomes the final estimate. If the input data cannot fill the entire matrix, the estimation of the median is calculated as a *weighted median* of the stored data. In particular, the n_1 numbers in the first array have weight 1, the n_2 numbers in the second array have weight b, and the n_k numbers in the last array have weight b^{k-1}.

In our SWAN-Song streaming oriented evaluation implementation, the k and b value of the remedian matrix can be configured by the user. However, we set as default values k=15 and b=11 as described in [57]. With these values our system is able to process the (re)median value of a stream of data that has up to 11^{15} observations with just a 11×15 matrix.

The SWAN-Song streaming-oriented evaluation is much more efficient than the traditional implementation and it is more suitable for real-time streaming applications. However, the new implementation introduces a lower degree of accuracy for the MEDIAN operator. Furthermore, the SWAN-Song streaming evaluation operates only over *fixed* or *aligned* time windows (i.e., the reduction is applied only across all the data ingested during the window of time in question). Since the reduction is calculated continuously aggregating incoming data within a time frame, an expression result is emitted at the end of each time window (e.g., with a 5 seconds window size, a result is emitted every 5 seconds).

The SWAN-Song streaming-oriented evaluation is particularly suitable for applications that can tolerate a certain lack of accuracy and that operate at regular time intervals. Mission critical applications, where a better level of accuracy is required, could instead use the traditional implementation. Both the implementations are available in the Cowbirds Streams layer and they could be deployed together allowing different type of sensors to use the most convenient evaluation strategy.

5.4 Summary

In this chapter we described how we extended the existing Cowbird cloud instance for supporting a massive volume of SWAN-Song expression evaluations. We called the new Cowbird cloud architecture *hybrid* since it adds an extra computational layer to the traditional streaming architecture.

The Cowbird Fog module brings the evaluations closer to the sensor data generation. The Fog layer is also extremely scalable since Cowbird nodes can be dynam-

ically added and removed at run time. Furthermore, Fog(s) could be put closer to the user to improve availability and performance.

The Cowbird Streams is the high-performance component of the architecture. The Streams layer is powered by Apache Flink that we demonstrated to be the streaming framework that best meets our scenario. The Cowbird Streams supports two different SWAN-Song evaluation strategies: a *core implementation* that exactly reproduces the SWAN framework and a more *streaming-oriented* implementation. Both implementations can be adopted according to the type of sensor which the data is being evaluated. The Cowbird Streams is designed to accommodate *long-running* SWAN-Song expressions or expressions characterized by very *high-frequency* sensors.

The Hybrid-Cowbirds implementation is available at [37].

Chapter 6

Evaluation & Experimental Results

In this chapter we will show some of the results obtained during the experiments performed with the Hybrid-Cowbirds implementation outlined in Chapter 5. In particular, we created a SWAN-Song based experimental application that uses the architecture we designed for analyzing streams of data coming from sound sensors.

6.1 Hybrid-Cowbirds Evaluation

In this section, we describe some of the tests we performed on our hybrid and distributed architecture outlined in Chapter 5. We describe the approach we used and we report the results we obtained.

In our tests we focus on evaluating the performance in terms of *latency* and *throughput* of the hybrid platform we built. We want to compare the performance achieved by our system when a certain number of SWAN-Song expressions are exclusively offloaded to the Fog layer and when, instead, they are offloaded in both Fog and Streams layer. In general, we expect that after a certain threshold of workload a combination of the two approaches should be required especially if the Fog layer has been assigned with more modest computing resources than Streams layer. Furthermore, the Streams layer offers performance and flexibility of a streaming engine that should outscore the Fog layer.

6.2 The Test Application

For our tests, we simulate an application that continuously evaluates data coming from different sound sensors arranged in different geographical locations in order to

detect noisy areas or anomalous sounds. Sound sensors generate data values in the human hearing range; the overhead introduced by the data generation should not have a particular impact on the evaluations. Moreover, in a real Cowbird deployment scenario, the framework could encounter higher overheads when sensing data from an external web source due to the network connectivity.

The realized test application just measures if the MEAN value produced over a certain time frame is greater than a certain decibel threshold. The sound sensor produces a *float* value. For different locations, we set a TriState expression to gather and evaluate sound sensor data. Such an expression is very similar to the SWAN-Song below:

$$self@sound : value\{MEAN, 5000\} > 95.0 \tag{6.1}$$

6.3 Testing Environment

We deployed the Hybrid-Cowbirds architecture on the SURFsara infrastructure [61] with a relative small setup. We used HPC Cloud [63] for the Cowbird Fog layer and the Hadoop cluster [62] for running Apache Flink.

We executed the application using the following test configuration:

- 8 nodes (16 vCPU, 16 GiB memory) running the Cowbird Fog layer (1 Frontend, 1 Manager, 6 Cowbird nodes).
- 1 Kafka instance node (16 vCPU, 32 GiB memory, 50 GiB disk).
- 16 YARN containers (8 vCPU, 48 GiB memory) running Apache Flink.

6.4 Tests and Results

We performed several tests and we evaluated the performance of the Hybrid-Cowbirds architecture. We are interested in understanding if the realized architecture is effectively capable of evaluating a considerable number of SWAN-Song expressions in real-time. In our tests, we registered a certain volume of SWAN-Song expressions both in Cowbird Fog and Streams layer. Then, we measured the time required by the system to compute a result for a certain SWAN-Song expression (i.e., *latency*). We considered latency as the time required by the Cowbird Cloud to compute a result for an expression. We did not consider the further communication overhead required to send the result back to the Android smartphone since we are interested only in evaluating the performance of the cloud platform.

In particular, during our experiments we performed the following evaluations:

- *Cowbird vs Hybrid-Cowbirds.* In this setup, we want to test the benefits that a distributed architecture could bring to the Cowbird platform. In this experiment, we compared the original Cowbird framework with the new hybrid architecture.

- *Sensing Frequency Tests.* One of the parameters we considered during these experiments is the *sensor frequency* that is the rate at which a sensor thread produces (receives) fresh data values. For some mission critical applications data accuracy is crucial and a certain polling frequency is required. During these tests, we performed different evaluations in scenarios characterized by sensors that produce data that at various rates. In these experiments, we want to evaluate how the sensing frequency can affect the performance of the cloud platform.

- *Scalability Test* In this test, we evaluated a certain number of SWAN-Song expressions reducing the amount of resources allocated at the Fog layer. We expect to obtain worse performance than the original setup (described in Section 6.3) when evaluating the exact same workload. This test can help us understanding the effects of scaling out the sensing platform layer.

- *Two Sensors Expressions.* In this experiment, we evaluate a certain number of SWAN-Song expressions that are characterized by a combination of two different sensors. In particular, we want to measure how having multiple active threads for a single SWAN-Song expression can affect the realized hybrid architecture.

- *Streaming-Oriented vs Core Implementation* We extensively tested the streaming-oriented evaluation implementation realized for the Streams layer in order to measure the benefits that it can bring in terms of space consumption. We also compared the two implementations and measured their performance when a large number of expressions is offloaded to the Streams layer.

The remainder of this section will describe more in details the experiments we performed on the realized hybrid architecture and the results we obtained.

6.4.1 Cowbird vs Hybrid-Cowbirds

In this test, we compare the original Cowbird framework against the new hybrid architecture. Figure 6.1 shows the latency when up to 64 SWAN-Song expressions are evaluated in the original Cowbird cloud framework and in the new Fog and

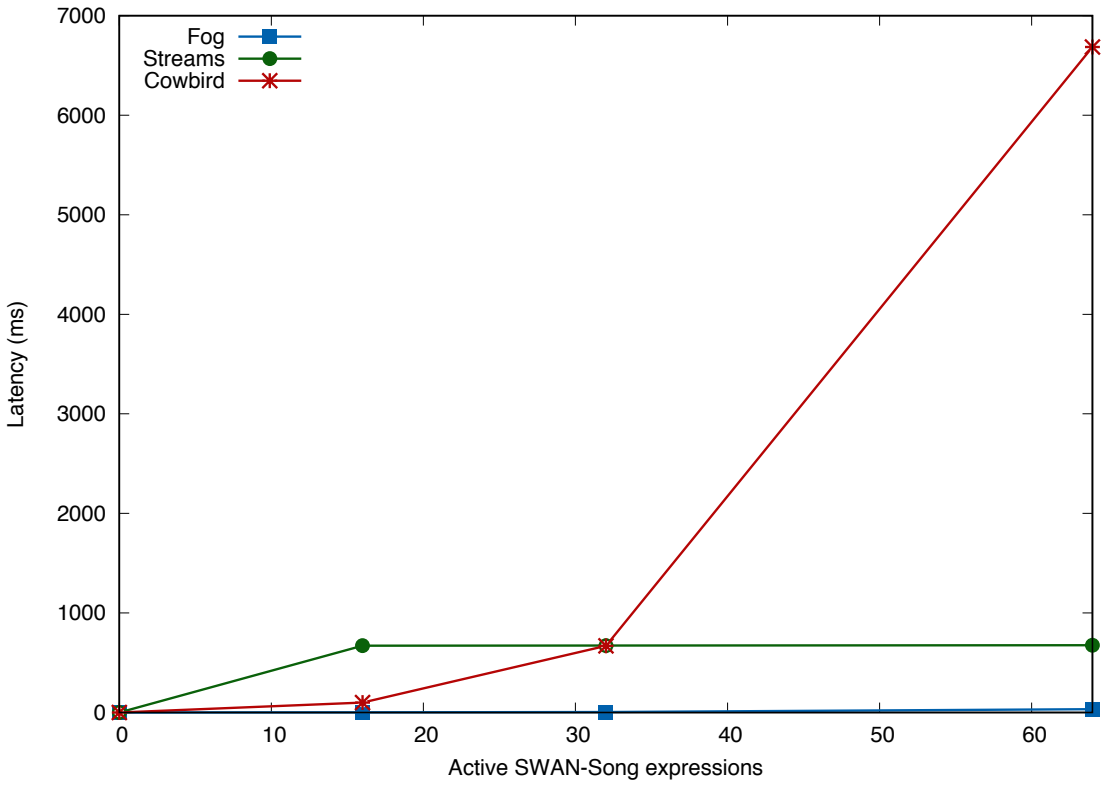

Figure 6.1: SWAN-Song expressions evaluation with sensors that produce fresh data every second. The expressions have the MEAN history reduction mode set and a time window of 5 seconds.

Streams layer. The SWAN-Song expressions evaluated are based on a sound sensor that emits a new data record every second; the history window is 5 seconds. The original Cowbird framework is running on a machine with the same characteristics of a Cowbird Node of the Fog layer. We can notice that the original Cowbird framework with just 64 active SWAN-Song expressions has a latency that is not tolerable for a streaming application.

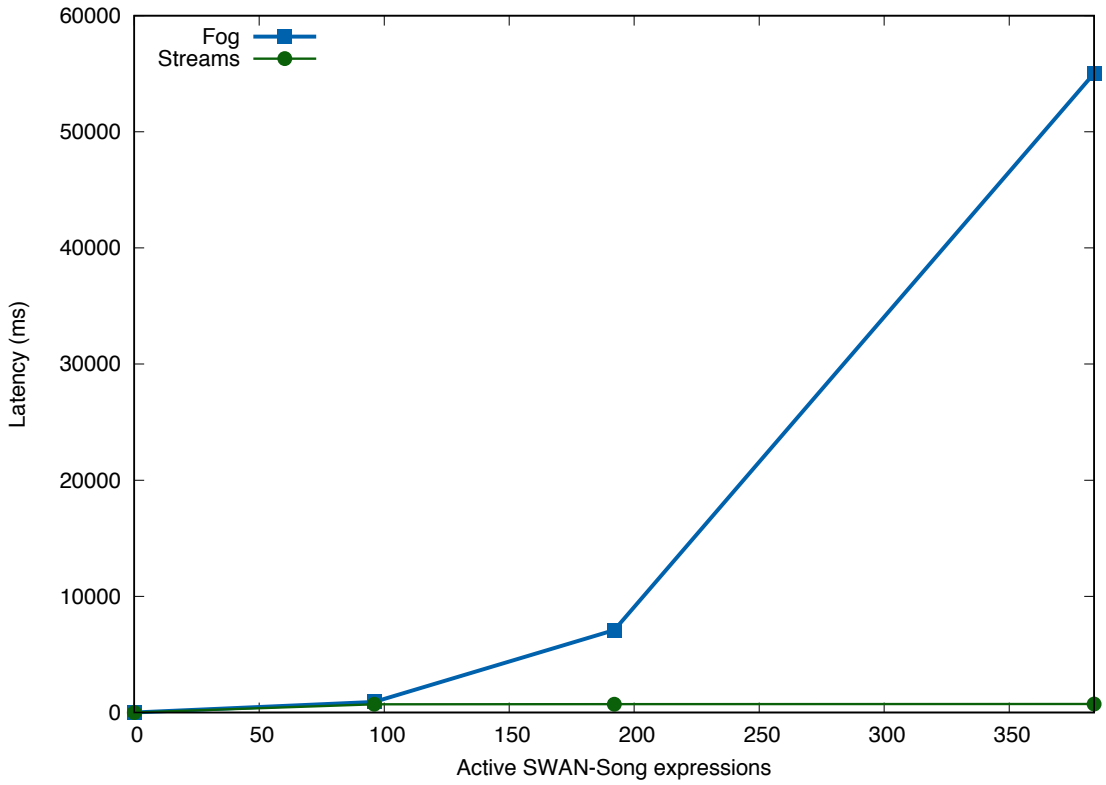

Figure 6.2: SWAN-Song expressions evaluation with sensors that produce a fresh data record every 500 ms. The expressions have the MEAN history reduction mode set and a time window of 5 seconds.

The SWAN-Song expressions evaluated over the Fog layer, instead, have a very low latency (less than 35 milliseconds). This performance is achieved because the sensing and the expression evaluations are distributed over different computing nodes.

The expressions evaluated in the Streams layer have a certain overhead due the data transmission and communication required between the Fog and the Streams layer through the Kafka broker.

6.4.2 Sensing Frequency Tests

In these tests, we offload a certain number of SWAN-Song expressions characterized by different sensing frequencies.

Figure 6.2 shows how the Hybrid-Cowbirds reacts when different loads of SWAN-Song expressions are offloaded to the system. The evaluated expressions use a sensor that produces a new data record every 500 ms and have a history window size of 5 seconds. The evaluation has been done in both the Fog and the Streams layer using

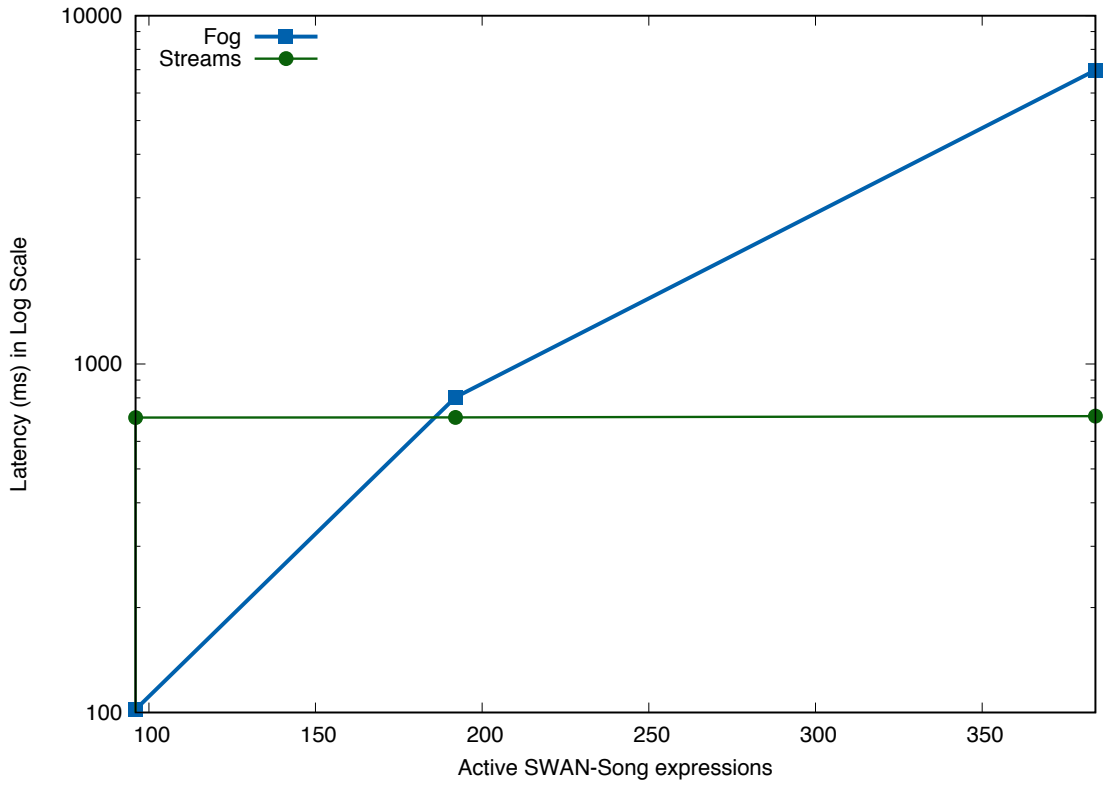

Figure 6.3: SWAN-Song expressions evaluation with sensors that produce a fresh data record every second. The expressions have the MEAN history reduction mode set and a time window of 5 seconds.

the core evaluation implementation.

Increasing the number of active SWAN-Song expressions in the system drastically boosts the latency when the evaluation is performed entirely in the Fog layer. In fact, many active sensors that produce fresh data records with a high-frequency slow down the SWAN-Song expression evaluations. Active threads that simultaneously need (finite) computing resources for fetching new data records or for computing SWAN-Song expressions have to wait for their turn to access the CPU. Furthermore, each sensor thread has to report the sensed data to an expression evaluation thread. Such communication requires synchronization between the sensing and evaluation threads that introduces a certain overhead. Hence, extra latency is involved with the increase of many concurrent active sensor threads.

Offloading the same load of SWAN-Song expressions to the Streams layer leads to way better performance. In fact, even though the number of sensing threads per Cowbird Node increases, each produced value is asynchronously sent through the network to the Streams layer. The Streams layer is capable of evaluating 384 active SWAN-Song expressions with a latency of ∼700 milliseconds.

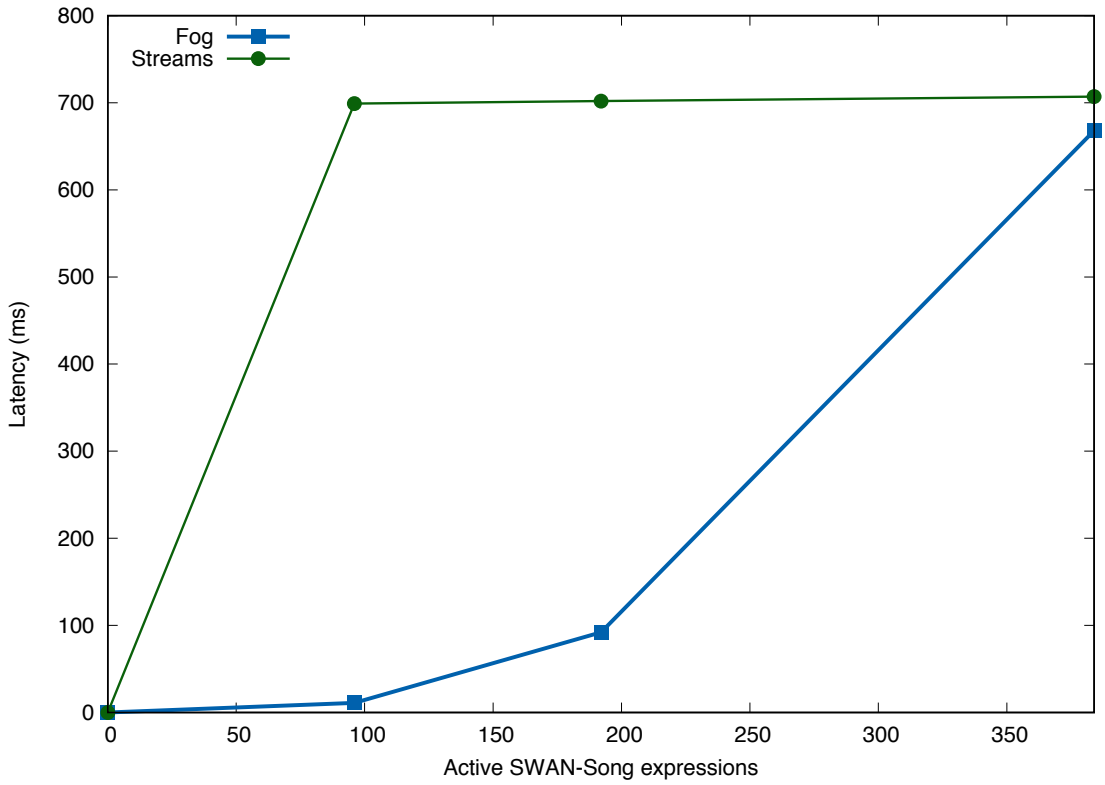

Figure 6.4: SWAN-Song expressions evaluation with sensors that produce a fresh data record every 2 seconds. The expressions have the MEAN history reduction mode set and a time window of 5 seconds.

Figure 6.3 shows how the system performed with SWAN-Song expressions that use sound sensors that produce data with a lower frequency than the previous test case (i.e. new data record every 1 second). For a better understanding of the plotted data, Figure 6.3 reports the achieved results in logarithmic scale.

In this experimental scenario, the Fog performs relatively better than the Streams layer up to a certain threshold of workload. A lower sensor frequency allows the Fog to be more flexible and efficient in comparison to the previous scenario. Figure 6.4 and 6.5 show that even lower frequency (a new date record every 2 and 5 seconds respectively) sensors reduce latency required to produce an evaluation result in the Fog layer.

We can conclude that with very high-frequency sensors the SWAN-Song evaluation could be offloaded to the Streams layer even though it involves extra communication overheads. By offloading the evaluation to the Streams layer, the Fog layer can be used exclusively to sense sensor data. However, from Figure 6.3 and Figure 6.5 we can notice that it also makes sense to offload the expressions evaluation to the Streams layer when the number of active sensor threads drastically increases.

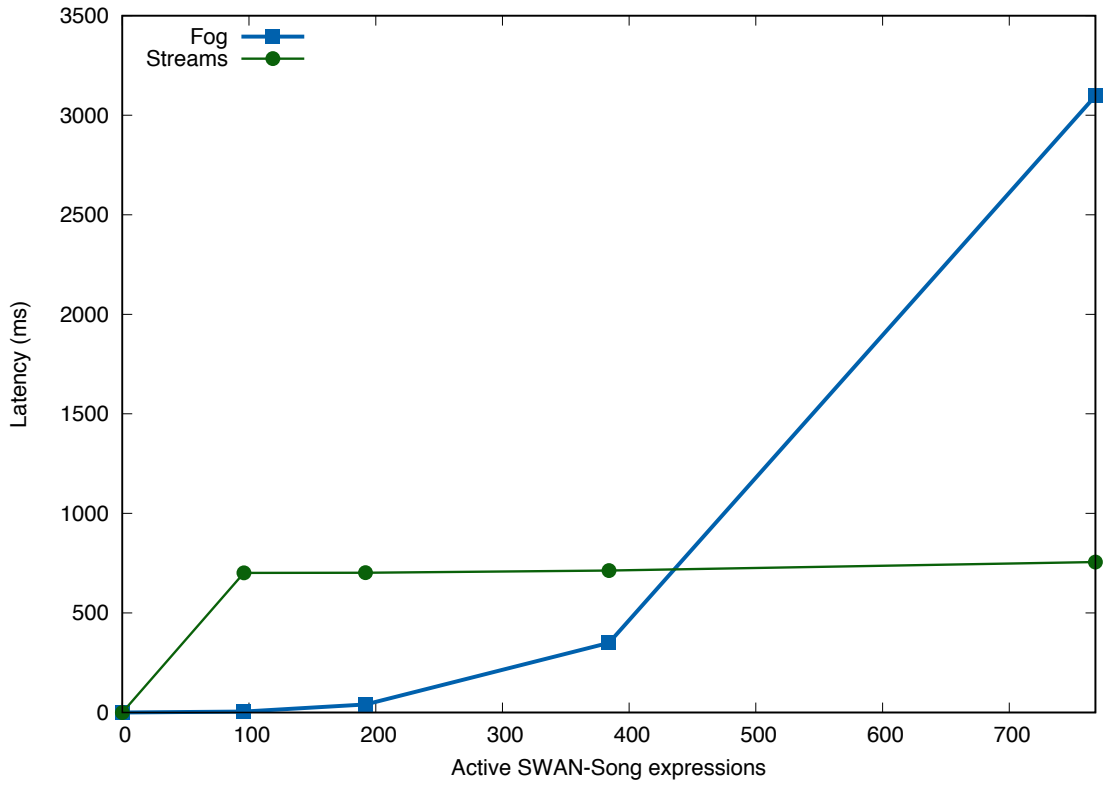

Figure 6.5: SWAN-Song expressions evaluation with sensors that produce a fresh data record every 5 seconds. The expressions have the MEAN history reduction mode set and a time window of 10 seconds.

6.4.3 Scalability Test

Figure 6.6 shows the results obtained on deploying the SWAN-Song expressions that use sound sensors that produce a new data record every 5 seconds. The evaluation is performed over a time window of 10 seconds. In this test, we deployed the Fog layer with only 5 nodes. We can see that the Fog layer performs worse than what is shown in Figure 6.5 with the same workload. Cowbird Nodes can be added at run time to *scale out* the Fog layer and accommodate bigger loads of data sensing and evaluation.

6.4.4 Two Sensors Expressions

We also tested our Hybrid-Cowbirds framework with more complex SWAN-Song expressions. Figure 6.7 reports the latency obtained when the system evaluates SWAN-Song expressions composed by two different sensors. For clarity, Figure 6.7 reports the achieved latency using logarithmic scale on the y-axis.

In this experiment, we combined in a SWAN-Song expression the sound sensor with a light sensor that provides information regarding the intensity of the light for

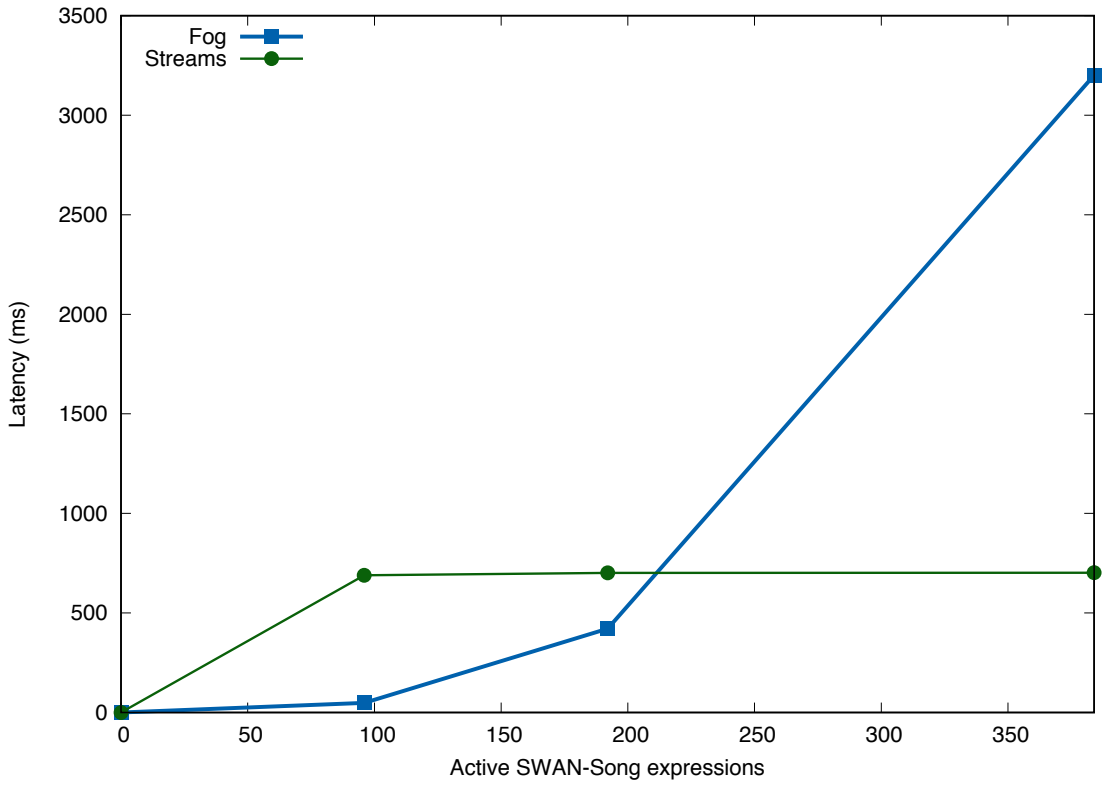

Figure 6.6: SWAN-Song expressions evaluation with sensors that produce a fresh data record every 5 seconds over a time window of 10 seconds. The Fog layer uses 5 computing nodes (3 Cowbirds Nodes).

a certain area. We designed the expression in order to detect quiet and bright areas. Such an expression is very similar to the SWAN-Song below:

$$self@sound : value\{MEAN, 5000\} < 55.0 \;\&\&\; self@light : value\{MEAN, 5000\} > 220.0 \tag{6.2}$$

We offloaded the SWAN-Song expression (6.2) on our hybrid architecture setting the sensor frequency to a data record per second and the history window to 5 seconds. From Figure 6.7 we can notice that the latency obtained by the Fog layer is much higher than what is shown in Figure 6.3. In both the test cases the sensors data are generated with the same frequency but in this second experiment each expression evaluation has to allocate two different sensing threads.

From Figure 6.7 we can also notice that the evaluation in the Streams layer, instead, remains pretty similar to the latency depicted in 6.3. With up to 384 SWAN-Song expressions the latency of the Streams layer does not encounter any break down.

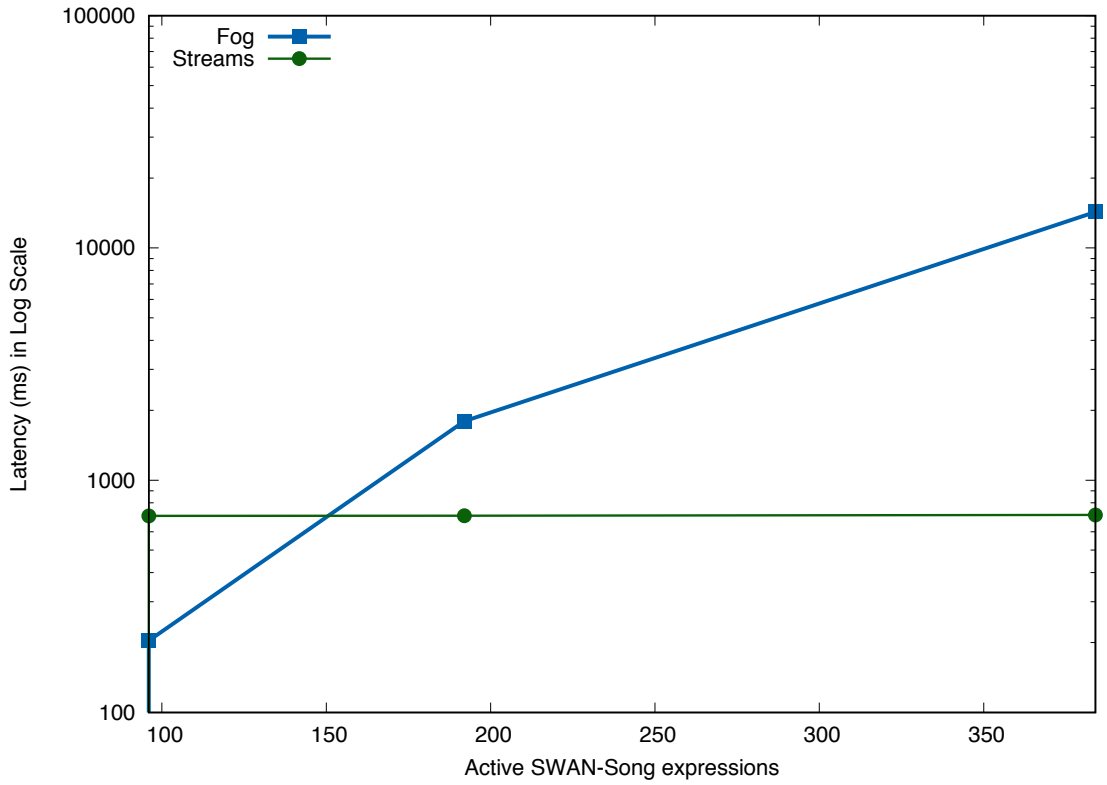

Figure 6.7: SWAN-Song expressions evaluation with two sensors that produce a fresh data record every second over a time window of 5 seconds.

6.4.5 Streaming-Oriented vs Core Implementation

We performed tests with both the SWAN-Song evaluation strategies implemented for the Streams layer: the *streaming-oriented* and the *core* implementation. As outlined in Chapter 5, the streaming implementation does not keep all the data records generated by sensors but instead it stores only the partial result for each SWAN-Song expression.

Streaming-Oriented Storage Efficiency

We tested the two implementations evaluating sound sensors data with the SWAN MEDIAN history reduction mode over a history window of one hour. Figure 6.8 shows the size of the RocksDB checkpoint directory used by Apache Flink in the Streams layer. Flink periodically records the *state* of the streaming it handles for fault tolerance purposes. After one hour of execution we can notice that the size of the directory used by the *streaming-oriented* evaluation implementation is ∼20 times smaller. This can be a good indication of the storage efficiency achieved by the streaming-oriented evaluation implementation.

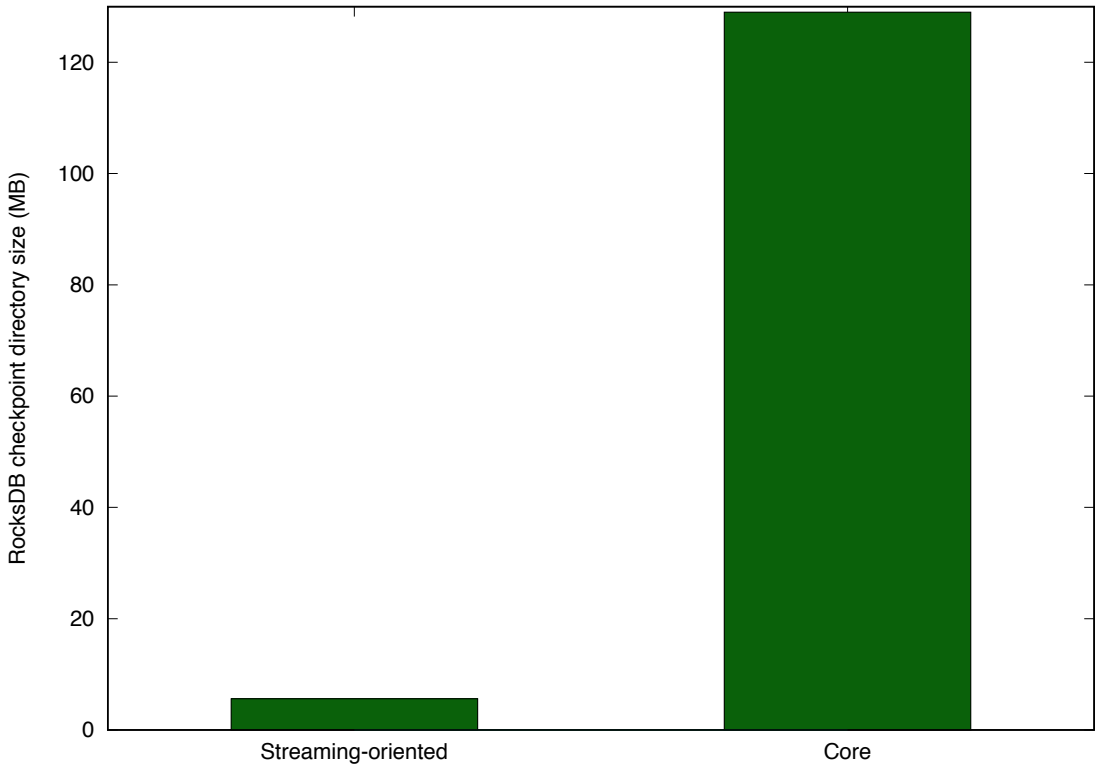

Figure 6.8: RocksDB checkpoint directory size after one hour evaluation of 768 SWAN-Song expressions that use a sound sensor that produces a data record every second.

Evaluating a Large Number of SWAN-Song Expressions in Cowbird Streams

We also measured the latency with a very high number of active SWAN-Song expressions evaluated in the Streams layer. Figure 6.9 shows a comparison of the latency measured with up to 5,000 SWAN-Song expressions deployed on the Cowbird Streams layer. In this scenario, over a test period of 1 hour each Cowbird Node transmitted over the network ∼1.5 GB. The evaluations are performed using both the *streaming-oriented* and the *core* implementation. The expressions use sensor threads that generate new data values every 1 second and have a history window of 5 seconds. We can notice that the two implementations react similarly in terms of latency. However, with a high load of expressions the streaming-oriented implementation seems to perform slightly better than the core evaluation strategy. One possible explanation could be a better streaming state management of the streaming-oriented evaluation strategy. As we have seen the streaming-oriented evaluation optimizes space consumption. Hence, when a result is computed only the partial result has to be decompressed from the RocksDB state backend. Instead, using the core implementation all the data values involved in the evaluation have to

be compressed/decompressed from the state backend. Furthermore, the core evaluation generates more communication traffic within the system. In the core evaluation strategy, expression results are computed continuously as fresh data enter the system. Instead, the streaming-oriented implementation uses *fixed* or *aligned* window mechanisms.

In this test scenario, we achieved a *throughput* of 5,000 events/s with a latency of ∼1.1 seconds. When offloading more than 5,000 expressions we noticed delays in the SWAN-Song expressions evaluation. In particular, we experienced a high CPU usage and a slow down of the outbound network traffic. This leads to an increment of the size of the outgoing message buffers and then to a bigger heap consumption. We also observed a decrease of the Kafka broker performance because of the active traffic between the Fog and the Streams layer. This is a clear symptom that the Fog layer should be scaled-out along with the Kafka instance.

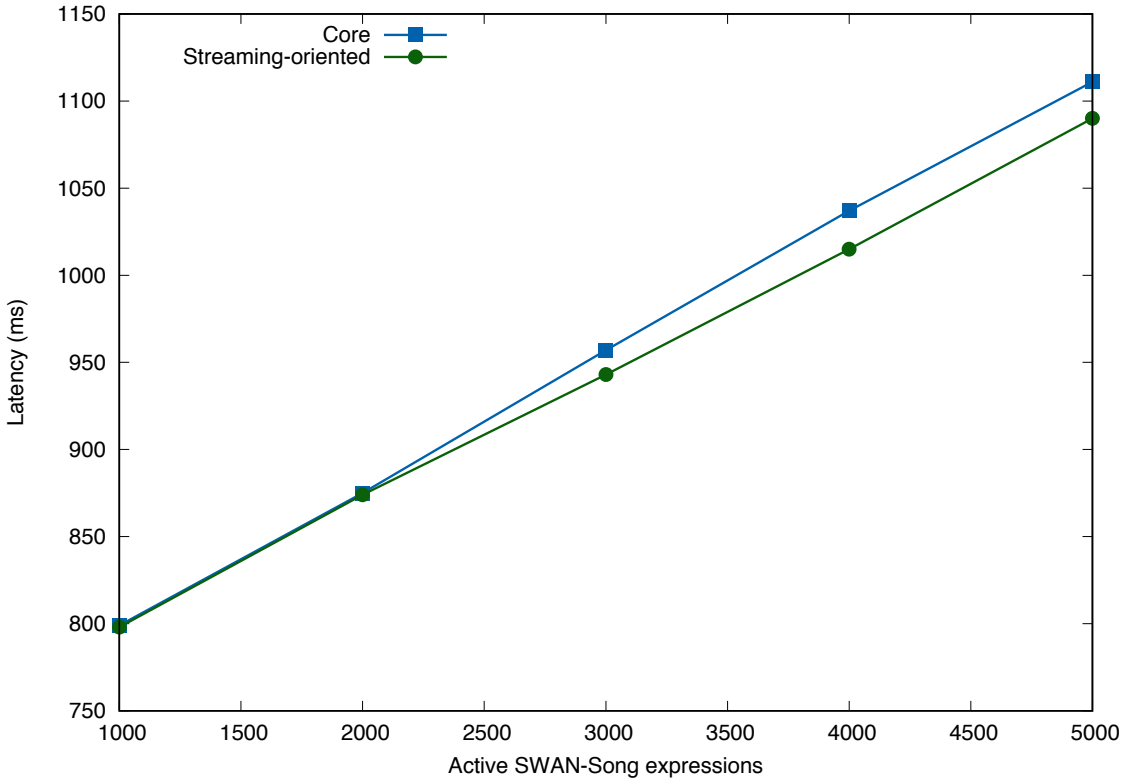

Figure 6.9: SWAN-Song expressions evaluation with sensors that produce a fresh data record every second. The expressions have the MEDIAN history reduction mode set and a time window of 5 seconds.

6.5 Discussion

In this section we described some of the tests we performed on our Hybrid-Cowbirds architecture. From the results of our experiments emerge that in some scenarios it makes sense to combine the data sensing and the SWAN-Song evaluation in the Fog layer. As we have seen in the *Scalability Test* described in Subsection 6.4.3, the Fog layer can scale out easily in order to accommodate an increasing number of SWAN-Song expression evaluation requests. However, experimental results outlined in Subsection 6.4.2 highlight that after a certain threshold, the Fog layer cannot handle anymore the burden of evaluating the sensed data in scenarios characterized by *high-frequency* sensors. The Streams layer along with the sensing capabilities provided by the Fog layer can guarantee real-time performance for data evaluations.

Furthermore, not only SWAN-Song expressions characterized by high-frequency sensors can be offloaded to the Streams layer. Some applications require sensor data collected over long time windows such as hours or even days. Such scenarios could certainly benefit from the Streams layer. In the original design of the SWAN framework, all the sensor values are kept in memory. Hence, the memory heap could be easily saturated if many long-running SWAN-Song expressions are allocated.

Long-running SWAN-Expressions could be offloaded to the Streams layer that runs on a cluster that is usually characterized by Terabyte of memory and Petabyte of disk.

The Kafka broker is the backbone of the Hybrid-Cowbirds architecture. In actual deployment, it should be characterized by multiple nodes and it should scale-out along with the Fog layer in order to prevent it from becoming a communication bottleneck between the Fog and the Streams layer as happened in our *Evaluating a Large Number of SWAN-Song Expressions in Cowbird Streams* test (Subsection 6.4.5).

We tested the *streaming-oriented* evaluation realized for the Streams layer in our *Streaming-Oriented vs Core Implementation* tests (Subsection 6.4.5). We demonstrated that it is more efficient than the core implementation when storing the data streaming state. This could bring benefits when evaluating long-running expressions characterized by a relatively big internal streaming state.

In a Fog computing context, the tested setup is not very realistic. In fact, all the architecture components run in the same SURFsara data center. As already mentioned in Chapter 5, the architecture could be geographically distributed putting

the Fog layer closer to the user. In such scenario, sensor evaluation capabilities will be close to the data generation and to the users. This approach makes possible the definition of a general purpose platform for evaluating smartphone and IoT sensors that would be ready to meet the Fog computing concept. Such a platform would be indeed characterized by low latency and high availability.

Chapter 7

Related Work

In this chapter, we briefly describe some streaming architectures applied to the Internet-of-Things scenario that are available in the literature.

[44] proposes an architecture called IoTCloud designed for real time robotics applications such as autonomous robot navigation. This architecture is structured over three layers:

- *Gateway layer.* This layer is responsible for managing *wireless sensor networks*. A gateway pushes sensor data to a message broker.
- *Publish-subscribe messaging layer.* The system supports different message brokers such as RabbitMQ [54], Apache ActiveMQ[1], and Apache Kafka [16].
- *Cloud-based big data processing layer.* The platform uses Apache Storm as low-latency stream processing engine.

[46] proposes a framework built in the Amazon cloud for processing large amounts of data coming from smart city sensors. This architecture, designed on top of EMR, uses Amazon Kinesis [6] for data ingestion; Kinesis is also used, in combination with Amazon Lambda [25], for the speed processing layer. The batch layer is realized through MapReduce jobs executed on EMR. Amazon S3 is used as the persistent storage layer.

[70] brings the Lambda architecture to the AllJoyn framework [4]. Alljoyn is an open source project for the development of a universal software framework aimed at the interoperability among heterogeneous devices, dynamic creation of proximal networks and execution of distributed applications. However, AllJoyn does not scale well because it does not support communications among devices belonging to

different broadcast domains (i.e., belonging to different subnets) and it does not provide any feature for the storage and real-time analytics of huge amount of data. [70] overcomes this limitation integrating the AllJoyn framework with Apache Storm for real-time stream processing and MongoDB for the massive storage.

[36] proposes a framework for combining real-time data coming from smart farm sensors (i.e. temperature, humidity, amount of precipitation) with historical data. Historical data is stored in a MongoDB noSQL database while the data received from the sensor based systems is ingested through a message broker and processed by Apache Storm. They also propose a new scheduler implementation for Storm that employs a more efficient mechanism for using available resources in the cluster.

[40] describes a streaming based processing infrastructure for high throughput and low latency IoT real-time analytics services. They propose a data adaptive mechanism for heterogeneous data stream integration. In particular, they developed a framework for schema description, flow based processing definition, and user defined operators. These mechanisms allow to build a common IoT service for all kinds of real-time analytic applications that use different types of IoT data formats. They implemented the system using Apache Spark streaming. IoT events are captured by Kafka and then ingested into the Spark cluster through it.

From an architectural perspective, all the above described systems, are applications of the Lambda architecture. The above mentioned works focus on immediately ingesting the generated sensor data into stream processing technologies. In our work, we extended this concept bringing part of the computation close to the data generation. Unlike all the aforementioned systems, our architecture does not use the traditional concept of Lambda or streaming architecture but for some type of SWAN-Song expressions we perform the evaluation where the data is generated. Computing the sensor data close to their sources reduces extra-overheads induced by transmission to a cluster system or by fault-tolerant mechanisms offered by distributed message brokers.

Our approach is made possible by the design of Cowbird. In fact, the Cowbird implementation uses polling threads to fetch sensors data into the system. Hence, part of the computation can be accomplished by the systems responsible for polling the data when it is possible. The design of Cowbird makes it possible to poll basically any type of data from any web resource. This is also another substantial difference with other available IoT streaming architecture implementations that are more specialized on particular scenarios.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

We designed and implemented a distributed architecture for the Cowbird framework. The distributed architecture promotes the sensing and evaluation of streams of sensor data in real-time. The architecture is composed by a distributed and scalable sensing layer and by a stream processing platform that is capable of evaluating large loads of SWAN-Song expressions with a low latency. Furthermore, when it is possible, the system combines sensing and evaluation features in the Fog layer avoiding extra communication costs. The architecture is extremely versatile and both its components can easily scale in situations of high input loads.

We built an experimental application that monitors the level of noise emitted by IoT sound sensors. This application helped us testing the designed architecture and measures its performance in a relatively simple setup. We conclude that the SWAN-Song expressions evaluation can be performed by both the architectural layers depending by the scenario. SWAN-Song characterized by *high-frequency* sensors or expressions that have a long history time window should be offloaded to the Streams layer.

8.2 Future Work

As future work, a real-time adaptive heuristic for offloading expression evaluations from the Fog layer to the Streams layer can be developed. This heuristic can be executed by the Remote Evaluation Manager running in the Fog layer with the goal to minimize latency in the SWAN-Song expressions evaluation. This algorithm can also select the most suitable evaluation strategy: *core* or *streaming-oriented*. To

this end, the streaming-oriented evaluation implementation could be realized also for the Fog layer.

The feasibility of adopting the realized architecture in a fog computing scenarios should be investigated as well. In fact, the Fog layer of our hybrid architecture could be deployed on the *edge* while the Streams could be executed in a cloud environment.

In addition, Cowbird Node internal components can be re-engineered as Akka actors instead of native Java threads. The Akka actor model could be used not only for handling communication between different nodes but also for threads management in the Cowbird Node instance. This approach could reduce synchronization overheads between sensors threads and expressions evaluation mechanisms. In fact, the communication between actors happens through asynchronous exchanged messages. Moreover, Akka maps running actors to a pool of active threads. Hence, resource utilization could be improved when many active sensors threads are allocated.

As a further improvement, sensed and evaluated data in the cloud could be shared among different SWAN users that require the same type of sensor data. Sharing sensed and derived sensor data would enhance resource utilization avoiding storage and computing workload duplication.

In order to increase performance and efficiency in the communication between the Fog and the Streams layer, a message binary format could be used instead of JSON.

Another possible improvement can be extending the SWAN-Song semantics to make it benefit from the capabilities brought by Apache Flink. Apache Flink has a vast ecosystem of libraries such as Complex Event Processing (CEP) [11] or machine learning [12]. SWAN-Song can be enriched with these features in order to offer a more rich API to mobile developers that want to use sensors data in their apps.

Bibliography

[1] *ActiveMQ*. http://activemq.apache.org. Accessed October 2017.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, June 1985.

[3] *Akka*. http://akka.io. Accessed September 2017.

[4] *AllJoyn Framework*. http://allseenalliance.org/framework. Accessed September 2017.

[5] *Amazon EMR*. http://aws.amazon.com/emr. Accessed September 2017.

[6] *Amazon Kinesis*. http://aws.amazon.com/kinesis. Accessed September 2017.

[7] *Amazon S3*. http://aws.amazon.com/s3. Accessed September 2017.

[8] *Amazon Web Services (AWS)*. http://aws.amazon.com. Accessed September 2017.

[9] *Apache Cassandra*. http://cassandra.apache.org. Accessed September 2017.

[10] *Apache Flink*. http://flink.apache.org. Accessed September 2017.

[11] *Apache FlinkCEP*. http://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/cep.html. Accessed October 2017.

[12] *Apache FlinkML*. http://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/ml/index.html. Accessed October 2017.

[13] *Apache Flume*. http://flume.apache.org. Accessed September 2017.

[14] *Apache Giraph*. http://giraph.apache.org. Accessed September 2017.

[15] *Apache Hadoop*. http://hadoop.apache.org. Accessed August 2017.

[16] *Apache Kafka*. http://kafka.apache.org. Accessed September 2017.

[17] *Apache Mesos*. http://mesos.apache.org. Accessed September 2017.

[18] *Apache Samza*. http://samza.apache.org. Accessed September 2017.

[19] *Apache Showdown: Flink vs. Spark*. http://jobs.zalando.com/tech/blog/apache-showdown-flink-vs.-spark. Accessed September 2017.

[20] *Apache Spark*. http://spark.apache.org. Accessed September 2017.

[21] *Apache Spark Streaming*. http://spark.apache.org/streaming. Accessed September 2017.

[22] *Apache Sqoop*. http://sqoop.apache.org. Accessed September 2017.

[23] *Apache Storm*. http://storm.apache.org. Accessed September 2017.

[24] *Apache Zookeeper*. http://zookeeper.apache.org. Accessed September 2017.

[25] *AWS Lambda*. http://aws.amazon.com/lambda. Accessed September 2017.

[26] *Benchmarking Streaming Computation Engines at Yahoo!* http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at. Accessed September 2017.

[27] F. Bonomi, R. Milito, J. Zhu, et al. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16.

[28] D. Cantone and M. Hofri. "Further analysis of the remedian algorithm". In: *Theoretical Computer Science*. Vol. 495. Elsevier, July 2013.

[29] P. Carbone, S. Ewen, S. Haridi, et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *Bulletin of the Technical Committee on Data Engineering*. IEEE, Dec. 2015.

[30] P. Carbone, S. Ewen, G. Fóra, et al. "State Management in Apache Flink : Consistent Stateful Distributed Stream Processing". In: *Proceedings of the VLDB Endowment*. Vol. 10. Aug. 2017.

[31] D. Chu, A. Deshpande, J.M. Hellerstein, et al. "Approximate Data Collection in Sensor Networks using Probabilistic Models". In: *Proceedings of the 22nd International Conference on Data Engineering, 2006. ICDE '06.* IEEE, Apr. 2006.

[32] J. Considine, F. Li, G. Kollios, et al. "Approximate aggregation techniques for sensor databases". In: *Proceedings. 20th International Conference on Data Engineering, 2004.* IEEE, Apr. 2004.

[33] R. Das, N. Bozdog, and H. Bal. "Cowbird: A Flexible Cloud-based Framework for Combining Smartphone Sensors and IoT". In: *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2017 5th IEEE International Conference on*. IEEE, 2017. DOI: 10.1109/MobileCloud.2017.14.

[34] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. Dec. 2004.

[35] G. DeCandia, D. Hastorun, M. Jampani, et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM, Oct. 2007.

[36] A. Dincu, E. Apostol, C. Leordeanu, et al. "Real-Time Processing of Heterogeneous Data in Sensor-Based Systems". In: *10th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*. IEEE, July 2016.

[37] *Distributed Cowbirds*. http://github.com/gdibernardo/cowbird-distributed. Accessed October 2017.

[38] M. Fu, S. Mittal, V. Kedigehalli, et al. "Streaming@Twitter". In: *Bulletin of the Technical Committee on Data Engineering*. IEEE, Dec. 2015.

[39] *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. http://www.gartner.com/newsroom/id/3598917. Accessed September 2017.

[40] Y. Ge, X. Liang, Y. C. Zhou, et al. "Adaptive Analytic Service for Real-Time Internet of Things Applications". In: *International Conference on Web Services*. IEEE, July 2016.

[41] S. Ghemawat, H. Gobioff, and S. Leung. "The Google file system". In: *19th ACM Symposium on Operating Systems Principles*. ACM, Oct. 2003.

[42] *Heron*. http://twitter.github.io/heron. Accessed September 2017.

[43] *Kafka Streams*. http://kafka.apache.org/documentation/streams. Accessed September 2017.

[44] S. Kamburugamuve, L. Christiansen, and G. C. Fox. "A Framework for Real Time Processing of Sensor Data in the Cloud". In: *Journal of Sensors*. Hindawi, Apr. 2015.

[45] R. Kemp. "Programming frameworks for distributed smartphone computing". In: *Ph.D. dissertation*. Vrije Universiteit Amsterdam, 2014.

[46] M. Kiran, P. Murphy, I. Monga, et al. "Lambda Architecture for Cost-effective Batch and Speed Big Data processing". In: *BIG DATA '15 Proceedings of the 2015 IEEE International Conference on Big Data (Big Data) Pages 2785-2792*. ACM, Oct. 2015.

[47] M. Kleppmann and J. Kreps. "Kafka, Samza and the Unix Philosophy of Distributed Data". In: *Bulletin of the Technical Committee on Data Engineering*. 2015.

[48] *Kubernetes*. http://kubernetes.io. Accessed September 2017.

[49] S. Kulkarni, N. Bhagat, M. Fu, et al. "Twitter Heron: Stream Processing at Scale". In: *SIGMOD '15 Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, June 2015.

[50] M. A. Lopez, A. G. Pastana Lobato, and O. C. M. B. Duarte. "A Performance Comparison of Open-Source Stream Processing Platforms". In: *Global Communications Conference (GLOBECOM), 2016 IEEE*. IEEE, Dec. 2016.

[51] *MongoDB*. http://www.mongodb.com. Accessed September 2017.

[52] N. Palmer, R. Kemp, T. Kielmann, et al. "SWAN-song: a flexible context expression language for smartphones". In: *PhoneSense '12 Proceedings of the Third International Workshop on Sensing Applications on Mobile Phones*. ACM, Nov. 2012.

[53] *Play Framework*. http://www.playframework.com. Accessed August 2017.

[54] *RabbitMQ*. http://www.rabbitmq.com. Accessed October 2017.

[55] R. Ranjan. "Streaming Big Data Processing in Datacenter Clouds". In: *IEEE Cloud Computing (Volume: 1, Issue: 1, May 2014)*. July 2014.

[56] *RocksDB*. http://rocksdb.org. Accessed September 2017.

[57] P. J. Rousseeuw and G. W. Bassett Jr. "The Remedian: A Robust Averaging Method for Large Data Sets". In: *Journal of the American Statistical Association*. Vol. 85. 1990.

[58] K. Shvachko, H. Kuang, S. Radia, et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. May 2010.

[59] *Streaming benchmark*. http://github.com/gdibernardo/streaming-engines-benchmark. Accessed October 2017.

[60] R. Sumbaly, J. Kreps, and S. Shah. "The "Big Data" ecosystem at LinkedIn". In: *International Conference on Management of Data (SIGMOD)*. ACM, July 2013.

[61] *SURFsara*. http://www.surf.nl/en/about-surf/subsidiaries/surfsara. Accessed September 2017.

[62] *SURFsara's Hadoop service*. http://userinfo.surfsara.nl/systems/hadoop/description. Accessed September 2017.

[63] *SURFsara's HPC Cloud service*. http://userinfo.surfsara.nl/systems/hpc-cloud. Accessed September 2017.

[64] T.Akidau, R.Bradshaw, C.Chambers, et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: VLDB, 2015.

[65] *THE SWAN PROJECT*. http://www.cs.vu.nl/SWAN. Accessed August 2017.

[66] *Thingspeak server*. http://thingspeak.com. Accessed August 2017.

[67] D. Tulone and S. Madden. "PAQ: Time Series Forecasting for Approximate Query Answering in Sensor Networks". In: *Proceedings. Third European Workshop, EWSN 2006*. Springer, Feb. 2006.

[68] *Twitter - Filter realtime Tweets*. http://developer.twitter.com/en/docs/tweets/filter-realtime/overview. Accessed October 2017.

[69] V. K. Vavilapalli, A. C. Murthy, C. Douglas, et al. "Apache Hadoop YARN: yet another resource negotiator". In: *SOCC '13 Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, Oct. 2013.

[70] M. Villari, A. Celesti, and M. Fazio. "AllJoyn Lambda: An architecture for the management of smart environments in IoT". In: *2014 International Conference on Smart Computing Workshops (SMARTCOMP Workshops)*. IEEE, Nov. 2014.

[71] D. Wampler. *Fast Data Architectures for Streaming Applications - Getting Answers Now from Data Sets that Never End*. O'Reilly, Sept. 2016.

[72] M. Zaharia, T. Das, H. Li, et al. "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters". In: *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX, June 2012.

[73] M. Zaharia, M. Chowdhury, T. Das, et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, Apr. 2012.

[74] M. Zaharia, M. Chowdhury, M. J. Franklin, et al. “Spark: cluster computing with working sets”. In: *HotCloud’10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. ACM, June 2010.

Acknowledgement

I would like to say thank you to my supervisors Henri and Roshan. Thank you for the advice you gave me during these months! Because of you I learned a little bit more of what doing research actually means. It has been an incredible experience working with you. Furthermore, I would like also express my gratitude to Kees for the valuable feedback he gave me during the thesis writing.

I would like to say thank you to my mom, dad, Kevin and Ludovica for all the support they gave me in the past two years. You always believed in me and you constantly gave me the strength to push forward throughout my studies. Thank you for supporting me in every choice I made. I will always be grateful for what you did for me.

Thank you.