Design and Implementation of a Tool to Collect Execution- and Service-Data of Big Data Analytics Applications

Bachelor's Thesis

for obtaining the academic degree
Bachlor of Science (B.Sc.)

at

Beuth Hochschule für Technik Berlin
Department Informatics and Media VI
Degree Program Mediainformatics

1. Examiner and Supervisor:	Prof. Dr. Stefan Edlich
2. Examiner:	Prof. Dr. Elmar Böhler
Submitted by:	Markus Lamm
Matriculation number:	s786694
Date of submission:	06.09.2016

Ackknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Contents

1 Introduction

1.1 Motivation

In preparation of this thesis, I first got in contact with Prof. Dr. Stefan Edlich on January 29th this year and presented an own idea for a topic of a bachelor's thesis. At this time, I was visiting my last course at Beuth Hochschule, the software-project which is spread over two semesters aiming to design and implement a software application in cooperation with software related companies based in Berlin, which was Lieferando in my case, an online food order service. During this project, I got in touch with a lot of technologies like Apache Kafka, Apache Spark, Cassandra, Elasticsearch and Consul, all together well known to me as 'buzzwords' from technology-blogs and magazines.

Because I was interested to learn a bit more about that "big-data-streaming-thing" and especially how to build software using stream processing frameworks, I decided my thesis to be in this big data context and created a working title "Design And Implementation Of A 'Data Processing Pipeline'" To Transform Continous Monititoring Data Streams". The basic idea was to aggregate data from REST RabbitMQ endpoints, send this raw data to a stream processor and create a model which fits the monitoring domain and store this data in a storage system which enables further data analytics.

During the following email correspondence, Prof. Dr. Stefan Edlich he suggested me to get the data from the the streaming platforms components itself, instead of a RabbitMQ queue as my idea suggested. So he presented one of his own topics which was quite congruent to my own idea with the given title *Design and Implementation of a Tool to Collect Execution- and Service-Data of Big Data Analytics Applications*, which I finally choosed to be the one to work out.

This topic is located on germans biggest big data research project "Berlin Big Data Center", which Prof. Dr. Stefan Edlich is a member of. Within the project, a program will be developed, which collects and stores relevant data of streaming platforms like Apache Flink, Apache Kafka or Apache Spark, with the overall aim to build a software that is able to "learn", based on the data that will be collected by the system that is proposed in this thesis.

Apache Flink is a "new player" in the plurality of stream processing frameworks. It was initialized by researchers of the Technische Universität Berlin, Humboldt Universität Berlin and Hasso-Plattner Institut Potsdam in 2008 and has emerged from the research project described above. On the 12th of January 2015 Flink became a top level project of the Apache Foundation. In the meantime, the development of Flink is driven by a grown community (216 contributers, 22.08.2016) and a wide range of companies that are actively using it.

1.2 Objective

The main goal of the thesis is a working software system to ingest and store data that can be collected from Apache Flink and Apache Kafka. It will be examined, which data is available and can be collected at all, what data is relevant and how to collect from source systems.

Furhermore, the collected data must be stored in a persistence system to become available for possible consumers like visualization applications, analytical processes or as a data source for applications from the context of Machine Learning for example.

This thesis will not be a deep introduction into big data, stream processing or covers deeper details of the internals of Apache Flink and Apache Kafka. To understand the context this frameworks are located in, the underlying concepts will be explained only briefly.

1.3 Structure of thesis

After a short introduction to the topics and the main goals of the present thesis in this chapter, Chapter 2 discusses the context of big data, stream processing, introduces Apache Flink and Apache Kafka as representatives of widely used stream processing frameworks. In preparation of Chapter 3, both Representational State Transfer (REST) and the Java Management Extensions (JMX) as possibilities of remote data access in distributed systems will be discussed.

Chapter 3 examines Apache Flink and Apache Kafka regarding to the provided data both of the systems. The different sources for the data collection will be described, as well what data should be collected and stored in a persistence system regarding to its relevance and data quality. According to the results of the data analysis, the functional and non-functional requirements of the system being developed will be introduced at the end of the chapter.

Based on the requirements elaborated in Chapter 3, Chapter 4 introduces the software solution by giving a detailed conceptional overview of the software components involved and discusses implementation details for selected items.

In chapter 5 we'll see how to setup the technical environment for the usage of the prototype to verify the correct functionality related to the requirements defined in Chapter 4.

The last Chapter 6 covers a conclusion and gives a resumee of the present work.

2 Basic Concepts

After a short introduction to the terminology of Big Data and Big Data Analytics Applications, the concept of stream processing and Apache Flink and Apache Kafka as streaming data frameworks will be explained. Representational State Transfer (REST) as an architecture paradigma for distributed software systems as well as the specification for managing and monitoring Java applications named Java Management Extensions (JMX) will be discussed at the end of the chapter.

2.1 Big Data

In the past decade the amount of data being created is a subject of immense growth. More than 30,000 gigabytes of data are generated every second, and the rate of data creation is accelerating[Nat15]. People create content like blog posts, tweets, social network interactions, photos, servers continuously log messages, scientists create detailed measurements, permanently.

Through advances in communications technology, people and things are becoming increasingly interconnected. Generally titled as machine-etcto-machine (M2M), interconnectivity is responsible for double-digit year over year data growth rates. Finally, because small integrated components are now affordable, it becomes possible to add intelligence to almost everything. As an example, a simple railway car has hundreds of sensors for tracking the state of individual parts and GPS-based data for shipment tracking and logistics[Pau12].

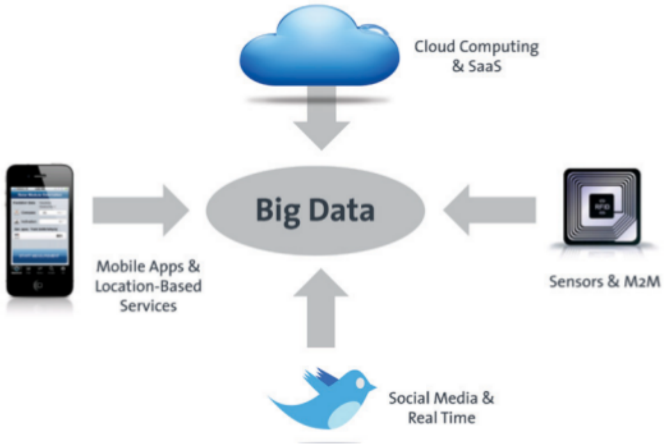

Figure 2.1: Sources of Big Data[Jör14b]

Besides the extremely growing amount of data, the data becomes more and more diverse. It exists in its raw and unstructured, semistructured or in rare cases in a structured form. [Jör14b] describes, that around 85 percent of the data comes in an unstructured form, but containing valuable information what makes processing it in a traditional relational system impractical or impossible.

According to [Nat15] [Pau12], Big Data is defined by three characteristics:

Volume The amount of present data because of growing amount of producers, e.g. environmental data, financial data, medical data, log data, sensor data.

Variety Data varies in its form, it comes in different formats from different sources.

Velocity Data needs to be evaluated and analyzed quickly, which leads to new challenges of analyzing large data sets in seconds range or processing of data in realtime

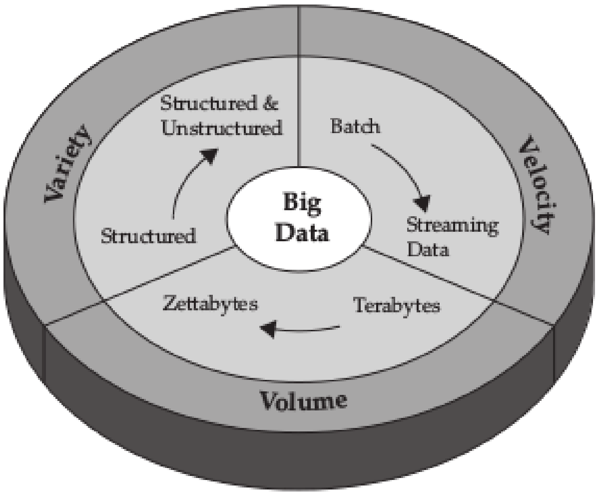

Figure 2.2: The three 'V's of Big Data[Pau12]

A possible definition for Big Data could be derived as follows: *"Big Data refers to the use of large amounts of data from multiple sources with a high processing speed for generating valuable information based on the underlying data."*

Another definition comes from the the science historian George Dyson, who was cited by Tim O'Reilly in [ORe13]: *"Big data is what happened when the cost of storing information became less than the cost of making the decision to throw it away."*

According to [Nat15] the term "Big Data" is a misleading name since it implies that pre-existing data is somehow small, which is not true, or that the only challenge is the sheer size of data, which is just one one them among others. In reality, the term Big Data applies to information that can't be processed or analyzed using traditional processes or tools.

2.2 Big Data Analytics Applications

Big Data Analytics describes the process of collecting, organizing and analyzing large volumes of data with the aim to discover patterns, relationships and other useful information extracted from incoming data streams [Nat15]. The process of analytics is typically performed using specialized software tools and applications for predictive analytics, data mining, text mining, forecasting and data optimization.

The analytical methods raise data quality for unstructured data on a level that allows more quantitative and qualitative analysis. With this structure it becomes possible to extract the data that is relevant for more detailed queries to extract the desired information.

The areas of applications may be extremely diverse and ranges from analysis of financial flows or traffic data, processing sensor data or environmental monitoring as explained in the previous chapter.

The illustration below summarises the six-dimensional taxonomy [Jör14a; Gro14] of Big Data Analytics Applications.

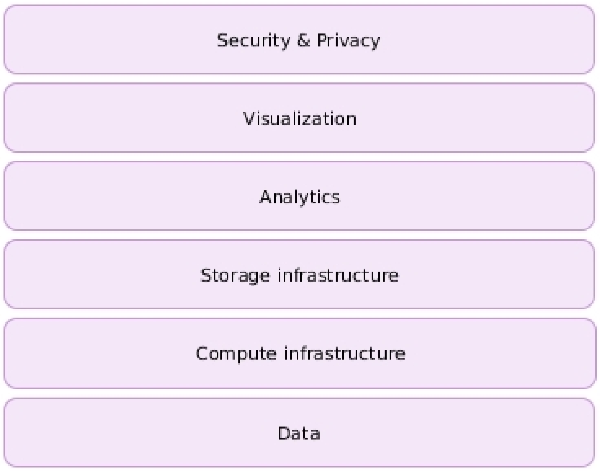

Figure 2.3: Taxonomy of Big Data Analytics Applications [Gro14]

The following section will discus the topic stream processing, which is part of the "Compute infrastructure" layer shown in the figure above.

2.3 Stream Processing

Computing infrastructures on big data currently differ on whether the processing on streaming data will be computed in batch mode, or in real-time/near real-time. This section is focussed on processing continous data streams in real-time/near real-time and introduces Apache Flink and Apache Kafka as representants of streaming frameworks.

According to [Kle16], stream processing is the computing of data continuously, concurrently, in real time and in a record-by-record fashion. In a stream, data isn't as treated static tables or files, but as a continuous infinite stream of data extracted from both live and historical sources. Various data streams could have own features, for example, a stream from the financial market describes the whole data. In the same time, a stream for sensors depends on sampling (e.g. get new data every 5 minutes).

The general approach is to have a small component that processes each of the events separately. In order to speed up the processing, the stream may be subdivided, and the computation distributed across clusters. Stream processing frameworks like Apache Flink and Apache Kafka primarily addresses parallelization of the computational load, an additional storage layer is needed to store the results in order to be able to query them.

This continous processing of data streams leads to the benefits of stream processing frameworks:

- Accessibility: live data can be used while the data flow is still in motion and before the data being stored.
- Completeness: historical data can be streamed and integrated with live data for more context.
- High throughput: large volumes of data can be processed in high-velocity with minimal latency.

To introduce a more formal expression, a data stream is described as an ordered pair (S, T) where:

- S is a sequence of tuples.
- T is a sequence of positive real time intervals.

It defines a data stream as a sequence of data objects, where the sequence in a data stream is potentially unbounded, which means that data streams may be continuously generated at any rate [Nam15] and leads to the following characteristics:

- the data arives continous
- the arrival of data is disordered
- the size of the stream is potentially unbounded

After this short introduction to the basics of stream processing, the following sections covers a short introduction of the streaming frameworks Apache Flink and Apache Kafka.

2.3.1 Apache Flink

As described in the documentation [Fli16], *"Apache Flink is an open source platform for distributed stream and batch data processing. Flink's core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. Flink also builds batch processing on top of the streaming engine, overlaying native iteration support, managed memory, and program optimization."*

The main components of Flink applications are formed by streams and transformations, in which streams define intermediate results whereas transformations represent operations computed on one or more input streams with one or more resulting streams.

To illustrate the main components of a Flink application, the following code from [Fli16] shows a working example of a streaming application, that counts the words coming from a web socket in 5 second windows:

```
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = ↩
            StreamExecutionEnvironment.getExecutionEnvironment();
        DataStream<Tuple2<String, Integer>> dataStream = env
                .socketTextStream("localhost", 9999) *(1)
                .flatMap(new Splitter()) *(2)
                .keyBy(0) *(2)
                .timeWindow(Time.seconds(5)) *(2)
                .sum(1); *(2)

        dataStream.print(); *(3)
        env.execute("Window WordCount");
```

```java
    }

    public static class Splitter implements FlatMapFunction<String, Tuple2<String, Integer>> {
        @Override
        public void flatMap(String sentence, Collector<Tuple2<String, Integer>> out) throws Exception {
            for (String word: sentence.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }
```

Code snippet 2.1: Basic Apache Flink streaming application

On execution, Flink applications are mapped to streaming dataflows, consisting of streams and transformation operators (3) where each dataflow starts with one or more sources (1) the data is received from and the resulting stream will be written in one or more sinks (3). to.

The dataflows of Apache Flink are in inherently parallel and distributed, by splitting streams into stream partitions and operators into operator subtasks, which are execute independently from each other, in different threads and on different machines or containers.

For the distributed processing of dataflows, Flink defines two type of processes:

1. **JobManagers:** The master process, at least one is required. It coordinates the distributed execution and is responsible for scheduling tasks, coordinate recovery on failures, etc.

2. **TaskManagers:** Worker processes, at least one is required. It executes the tasks, more specifically, the subtasks of a dataflow, and buffer and exchange the data streams.

A basic Flink cluster set up with a single JobManager and TaskManager on Docker will be introduced in Chapter 5 Evaluation and serves as source to collect data from, as well as a streaming component for processing collected data.

In addition, Apache Flink provides a client, which is not part of the runtime. It is used as a part of Java/Scala applications to create and send dataflows to the JobManager. The client will be used in the software component "CollectorDataProcessor" and introduced in Chapter 4 Architecture and Implementation.

2.3.2 Apache Kafka

Apache Kafka is publish-subscribe queuing service rethought as a distributed commit log [Kaf16], supporting stream processing with millions of messages per second, durability of messages through disk storage and replication accross multiple machines in clustered environments. It is written in Scala, was initially developed at LinkedIn and follows the distributed character of Big Data Analytics Applications by it's inherent design.

This excerpt from the paper [Jay11] the team at LinkedIn published about Kafka describes the basic principles:

A stream of messages of a particular type is defined by a topic. A producer can publish messages to a topic. The published messages are then stored at a set of servers called brokers. A consumer can subscribe to one or more topics from the brokers, and consume the subscribed messages by pulling data from the brokers. (. . .) To subscribe to a topic, a consumer first creates one or more message streams for the topic. The messages published to that topic will be evenly distributed into these sub-streams. (. . .) Unlike traditional iterators, the message stream iterator never terminates. If there are currently no more messages to consume, the iterator blocks until new messages are published to the topic.

A common use case for Apache Kafka in the context of stream processing is the buffering of messages between stream producing systems by providing a queue for incoming and outgoing data. According to the explanation of the concept of data sources and sinks in the Apache Flink section above, Apache Kafka is heavily used as an input source, as well as output sink for the processing dataflow in Apache Flink applications.

The following figure shows a typical use case for a data pipeline that typically start by pushing data streams into Kafka, consumed by Flink applications, which range from simple data transformations to complex data aggregations in a given time window. The resulting streams are written back to Kafka for the consumption by other services or the storage in a persitent medium.

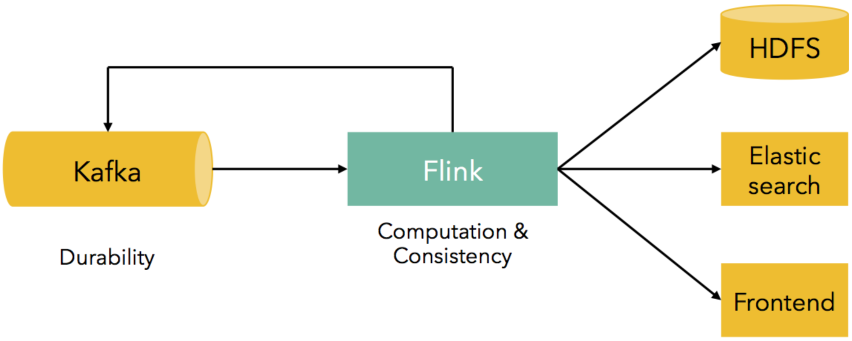

Figure 2.4: A typical Kafka-Flink pipeline[Rob]

Chapter 5 Evaluation describes the Docker setup for a single Kafka node that is part of the software solution in addition of provisioning data for collection.

2.4 Representational State Transfer (REST)

In his doctoral dissertation from 2000 titled "Architectural Styles and the Design of Network-based Software Architectures", Roy Thomas Fiedling introduced the term Representational State Transfer (REST) as core set of principles, properties, and constraints defining an "architectural style for distributed hypermedia systems"[Fie00].

The purpose of REST is focused on machine-to-machine communication and provides a simple alternative to similar procedures as Simple Object Access Protocol (SOAP) and the Web Services Description Language (WSDL). But REST is not a standard or technology. It should be more considered as reference for the development of applications that use the existing internet infrastructure based on Hypertext Transfer Protocol (HTTP) and corresponding HTTP verbs (GET, POST, PUT, DELETE, et al) for the exchange and manipulation of data, which is uniquely identified by Universal Resource Identifiers (URI) in the form of Uniform Resource Locators (URL).

Applications that follow the architectural style of REST are generally referred to as "Restful" web services and must meet the following characteristics, et al, according to [Fie00]:

1. **Client-Server architecture:** Clients and servers are separated by a uniform interface to facilitate portability. For example, a user interface is not concerned with data storage because it is internal to the server. On the other hand, the server is not concerned with the user interface or state. As long as the interface is not altered, the separation of concerns enables the components to evolve independently and thereby the improves scalability of the entire system.

2. **Stateless:** The communication between clients and server must be stateless. Each request from any client contains all the information necessary to service the request, and session state is held in the client.

3. **Uniform Interface:** The uniform interface between the interacting components is a fundamental characteristic of REST architectures and subjects to the following constraints:

    a) **Identification of Resources:** Resources describe any information that is originated on the server and can be be identified using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, based on the requested representation, the server may send data from its database as JSON data or HTML web page, what is diffent to the server's internal representation.

    b) **Manipulation of Resources through Representations:** The modification of the resource is performed by using the representation. If the representation and attached metadata is available, clients are able to change the state of the resource by modifying or deleting the resource using the HTTP verbs (GET, POST, PUT, DELETE, et al) in corresponding requests.

Chapter 4 Architecture and Implementation will apply these principles to enable the exchange of data between distributed software components by using an uniform interface based on HTTP.

2.5 Java Management Extensions (JMX)

JMX was created in 1998 as a Java Specification Request 3 (JSR-003), at that time still under the name Java Management API 2.0 and emerged with the participation of big companies such as IBM and Borland. Meanwhile, the specifications in the JSR-160 and JSR-77 contribute significantly to the term JMX. JSR-003 introduces the Java Management Extensions, also called the JMX specification as "architecture, the design patterns, the APIs, and the services for application and network management and monitoring in the Java programming language" [Inc16], "an isolation and mediation layer between manageable resources and management systems [Hea03]. In other words, JMX provides an programming interface between ressources and management systems based on the Java Virtual Machine(JVM) and is part of the core Java plattform since version 5. The following section explains the basic terms in preparation to the software solution discussed in Chapter 4.

The central point of a general JMX architecture is a **Manageable Resource**. A Manageable Resource can be any Java based application, service or device and applies both to the configuration and the monitoring of resources. In the Java world, Servlets, Enterprise JavaBeans (EJB) other JVMs are typical examples of Manageable Resources.

Java objects that implement a specific interface and conforms to certain design patterns according to the specification are called **MBean**s. The management interface of a resource is the set of all necessary information to gain access to the attributes and operations of the Managed Resource.

The **MBeanServer** represents a registry for MBeans in the JMX architecture. The MBean server is the component that provides the services for querying and manipulating MBeans. All management operations performed on the MBeans are done through the MBeanServer interface.

The **MBeanServerConnection** is a specialization of the MBeanServer interface, that provides a common way to access a MBean server regardless of whether it is remote, namely, accessed through a connector, or local, and accessed directly as a Java object.

The address of a connector is defined by the **JMXServiceURL** which clients can use to establish connections to the connector server. Taken from Chapter 4 Architecture and

Implementation, the url *"service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi"* enables the remote access to Apache Flink and Apache Kafka for collecting data according to the topic of this thesis.

An **ObjectName** uniquely identifies an MBean within an MBean server. Applications use this object name to identify the MBean to access query data from. The class represents an object name that consists of two parts, a domain name, an unordered set of one or more key properties. The ObjectName *"java.lang:type=Runtime"* as an example enables access to the management interface for the runtime system of the Java virtual machine.

2.6 Summary

The Chapter Basic Concepts explained the main characteristics of Big Data and Big Data Analytics Applications. To match the challenges that emerge with the immense growth of the data volume, the multiplicity of data sources and formats as well as the requirement of processing data in realtime, Apache Flink and Apache Kafka as widely used frameworks for processing streaming data had been introduced, as well as REST as a reference model for machine-to-machine communication based on HTTP. A short introduction to the JMX interface as a way to collect data from remote systems forms the end of the chapter.

3 Requirements and Specification

After a short introduction to the basic terms and Apache Flink and Apache Kafka in context of big data and stream processing, this chapter describes Data Quality and presents common criteria to measure the quality of data. Based on this criteria, available data for both of the systems will be inspected to build the foundation for the functional and non-functional requirements to be defined at the end of the chapter. The the main focus of the coming section is the analysis of available data that will be collected by the software solution and not a deeper exploration according to the relevance and quality of data.

3.1 Data Analysis

"You can only control what you observe and measure."[Chr07]. Even though logfiles, both provided by Apache Flink and Apache Kafka, are usefull for tracing problems in software systems, problems can be tracked and potential sources of error can be identified much earlier by collecting and storing system and application data at runtime to describe the state of the entire system at a given point in time.

Due to the distributed character of Apache Flink and Apache Kafka, where a system is composed of several interacting components, the examination of log data is not an adequate choice to gain insight into a distributed system containing several components.[Les14].

Runtime data to collect can be divided in three levels of abstraction:

1. **Business data:** The highest level of abstraction, often refered to as Key Performance Indicators(KPI), these data expresses direct business related values and usually have very little reference to technical details. As an example, the number of sales in an online shop.

2. **Application data:** On the middle level of abstraction, application data already contains many more technical details and refers to specific applications, like the number of GET requests and their corresponding HTTP Status response codes of a REST-based service.

3. **System data:** The lowest level of abstraction, data provided by the underlying systems an application is running on such as cpu, memory, network, or system utilization.

Based on Apache Flink and Apache Kafka, the following section discusses the data provided by both of the systems and tries a classification based on the abtraction levels.

3.1.1 System data

System data refers to the data provided by the computer system on the lowest level of abstraction and allows observation of system-related data. On unix-based systems, a variety of system tools is well known to system administrators to monitor the performance of servers, like vmstat (memory utilization), ifstat (network usage) or iostat (system input/output) [Höb00].

Another existing tool is called "DStat Versatile Resource Statistics Tool" and is described as follows: *"Dstat is a versatile replacement for vmstat, iostat, netstat and ifstat. Dstat overcomes some of their limitations and adds some extra features, more counters and flexibility. Dstat is handy for monitoring systems during performance tuning tests, benchmarks or troubleshooting. Dstat allows you to view all of your system resources in real-time, you can eg. compare disk utilization in combination with interrupts from your IDE controller, or compare the network bandwidth numbers directly with the disk throughput (in the same interval)."*[Wie16] Apache Flink and Apache Kafka, where a system is composed of several interacting components Dstat is a command line tool, the following figure shows the immediate output of running the application with the argument "–full", which expands more detailed information about multiple cpus and network interfaces:

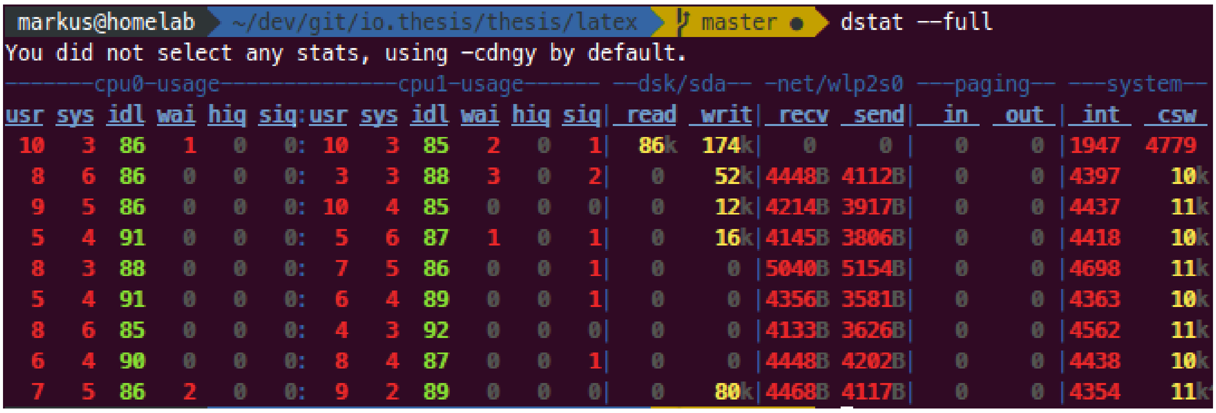

Figure 3.1: Output "dstat –full"

Additionally, Dstat provides multiple parameters to specify the data to be displayed, e.g. –cpu, –disk, –net, and many more. Used in combination, the data can be grouped in the following categories according to the parameters:

Category	**Dstat parameters**
cpu	("–cpu", "–top-cpu-adv", "–top-cputime", "–top-cputime-avg")
disk	("–disk", "–disk-tps", "–disk-util")
net	("–net", "–socket", "–tcp", "–udp")
io	("–io", "–top-io-adv", "–lock", "–fs")
memory	("–mem", "–top-mem", "–page", "–swap", "–vm")
system	("–sys", "–load", "–ipc", "–unix")
process	("–proc", "–proc-count", "–top-latency", "–top-latency-avg")

Table 3.1: Dstat data categories

Although the parameters are mostly self-explanatory, a list containing short descriptions for each of the parameter used in Chapter 4 Architecture and Implementation is available in Appendix. TODO Based on the data in the extracted categories, Dstat can be considered as a source that gives a fairly complete picture of the state of a system.

Dstat is a tool only available for unix systems, and therefore not available for Windows or Macintosh. Since Apache Flink and Apache Kafka are operated on Unix systems in most cases, this fact can be neglected because this tool offers a wide range of data to describe the system state a certain point of time.

3.1.2 Application data

Every application running on the Java Virtual Machine, can be accessed via JMX as discussed in Chapter 2 Basic Concepts. According the specification, every implementation of the JVM contains implementations for a basic set of management interfaces, that enables the access separate parts of JVM related data, located in the package "java.lang.management" [Ora16].

Management interface	**JMX ObjectName**
ClassLoadingMXBean	java.lang:type=ClassLoading
OperatingSystemMXBean	java.lang:type=OperatingSystem
RuntimeMXBean	java.lang:type=Runtime
ThreadMXBean	java.lang:type=Threading
MemoryMXBean	java.lang:type=Memory
BufferPoolMXBean	java.nio:type=BufferPool,name=*
GarbageCollectorMXBean	java.lang:type=GarbageCollector,name=*
MemoryManagerMXBean	java.lang:type=MemoryManager,name=*
MemoryPoolMXBean	java.lang:type=MemoryPool,name=*

Table 3.2: "Default" JMX JVM data

There's a difference in the way of data access between the object name containing an asterisk "*" and the one the ones that doesn't. The asterisk indicates the existence of multiple MBeans for a given query string, the result of a query for the object name "java.lang:type=GarbageCollector,name=*" results in multiple data sets according to existing garbage collector names.

This "default" set of management interfaces provides a deep insight into JVM data, is available for Apache Flink and Apache Kafka and will be included in the software solution in Chapter 4.

Apache Flink

Apache Flink provides application data via its monitoring API, a RESTful API, see Chapter 2 Basic Concepts, that delivers JSON data based on HTTP GET requests. It can be used to query general cluster information and status and statistics of running and completed jobs. The dashboard that comes with Apache Flink uses this monitoring API, but is designed to be used also by custom monitoring tools. The monitoring API runs as part of the JobManager and listens at post 8081 by default. All requests are of the sample form http://hostname:8081/jobs, below a list of available REST resources that will be used to fetch cluster- and job-related data for Apache Flink in Chapter 6 Implementation, see Appendix A for the corresponding JSON responses.

API path	**Description**
/config	Server setup
/overview	Cluster status
/jobs	Job ids by status running, finished, failed, canceled.
/jobs/jobId	Job details, dataflow plan, status, timestamps of state transitions
/jobs/jobId/exceptions	Exceptions that have been observed by the job
/jobs/jobId/config	User-defined execution config used by the job

Table 3.3: HTTP monitoring endpoints Apache Flink

Appendix A provides a list with sample JSON response according to this REST endpoints. TODO

Since version 1.1.0, Apache Flink also provides a rudimentary metrics system that exposes basic data for the Java Virtual Machine, the JobManagers and TaskManagers are running on. This data includes inter alia cpu usage or memory consumption, as well as basic information about running jobs. According to the "default" JVM data described in Table 3.2 and the data provided by the monitoring REST api, the metrics system in its current version represents just an excerpt of the data that will be collected anyway.

Apache Kafka

In addition to the standard interfaces and MBeans that come with the implementation of the JVM, Apache Kafka provides a set of managed resources providing application specific metrics concerning the Kafka domain, reaching from global broker metrics, global connection metrics to metrics per topic like in- and outgoing byte rates for example. Based on the requirement to collect as much data as possible, the data of all provided resources will be collected, the complete list of MBeans observed for Apache Kafka is available in Appendix A.

3.1.3 Data Quality

The following introduces to the basics of the term "Data Quality" for inspection the data sources described above based on common quality criteria.

Data quality refers to the quality of data as it is provided by measurements and describes the ability of data to represent the mapping from an empirical system("the real world" we operate in) to the numeric system correctly with the main goal to satisfy a given need or objective. It can not be expressed quantitatively, but [Det13] and [Chr07] introduce multiple common criteria for measuring the quality of data, from which a selection is made to check the data previously described on their quality:

1. **Correctness:** The data correspond to the entities of the real world, that is, the data represent the reality.
2. **Consistency:** Recorded data sets does not show discrepancies, logical contradictions or errors when compared among themselves.
3. **Reliability:** The origin of the data is traceable and the sources are trusted.
4. **Completeness:** The required information is available and no data values are missing or in an unusable state.
5. **Accuracy:** Expresses the mapping from the empirical system to the intended numerical system. Recorded values conform to actual values,
6. **Timeliness:** All data records correspond to the current state of the modeled world and thus are not outdated. The data are the actual properties of an object from a timely manner.
7. **Redundancy-free:** The data does not contain any duplicates, where a duplicate is meant to be data describing the same entity in the real world.

3.1.4 Results

According to the examinations of system and application data available for Apache Flink and Apache Kafka, the following matrix of data sources results for both of the systems:

	Apache Flink	**Apache Kafka**
System data (Dstat)	X	X
JVM data (JMX)	X	X
Application data (JMX)	X	X
Application data (REST)	X	-

Table 3.4: Data source matrix

Regarding the criteria of Data Quality, the following statements can be made:

1. The extracted data sources provide snapshot of system and application related data at a given point of time and thus are not outdated. They meet the criterion **Correctness**.

2. The data sources are well known and trusted, therefore they are **reliable**.

3. The Consistency of data needs to be evaluated be comparing data sets over a period of time, but will be assumed because the data sources are based on known applications and specifications.

4. The data is **accurate**, they represent the "real world" with a description of the state of the underlying system.

5. The data is **timely**. Dstat, JMX and REST data describe the state at the point when the data is queried.

6. The data is not redundancy-free! The "default" JMX data provided by the implementation of the JVM and the application data for Apache Flink available since version 1.1.0 both contain cpu, memory, garbage-collector, et al, data based on the management interfaces listed in Table 3.2. The same applies for the data available via Apache Flink's monitoring API and the application metrics regarding jobs and Job-/TaskManager information.

To summarise the results of the data analysis, there're three different sources for collecting system and application data for Apache Flink and Apache Kafka. The Dstat system tool for unix-based system provides technical data on the lowest level of abstraction. On a higher level, application data is provided by JMX for both of the systems containing application-related data, like broker metrics for Apache Kafka or job information for Apache Flink, but also "default" JVM data for all Java based systems. The JMX metrics system of Apache Flink exists since version 1.1.0 (Release date 9th of August 2016) and represents a current feature, which is not fully developed yet according to the JIRA of Apache Flink. The full set of Flink related application data is exposed via its HTTP monitoring API.

According to the criteria introduced above, a certain level of Data Quality can be assumed, because the data sufficiently matches these criteria except. The only exception is the criterion that requires data to be free of redundancies. The metrics for Apache Flink are just a very small excerpt of the JVM and application related data that will be collected anyway according to Table 3.2 and Table 3.3.

3.2 Functional Requirements

The main goal of the thesis is a working software system to collect system and application data available for Apache Flink and Apache Kafka. Furhermore, the collected data must be stored in a persistence system to become available for possible consumers like visualization applications, analytical processes or as a data source for applications from the context of Machine Learning for example.

From this, three main functional requirements derive:

1. Collection of data from Apache Flink and Apache Kafka source systems.

2. Storage of collected data in a persistent medium.

In preparation of this thesis my supervisor Prof. Dr. Stefan Edlich once said *"Sie sammeln alles, was nicht bei drei auf dem Baum ist"*, which leads to the requirement to collect as much data as possible. As a result, available data sources for has been inspected in the previous secion and the software solution must provide a mechanism to collect data from the sources discussed in the data source matrix shown in Table 3.4.

The collected data must be brought into format, that enables transmitting and storing structured data. Because nowadays usually the JavaScript Object Notation (JSON) is used for transmitting and storing structured data, the data collected on Apache Flink and Apache Kafka must be transformed into its JSON representation, before the data will be transmitted to the storage system. Another advantage of JSON is that it independent of any programming language, parsers for any language are available.

At the time of this thesis, the concrete usage scenarios of the collected data are not yet known. As is not yet known when the data is needed, the process of data collection must provide a mechanism to collect the data "on demand" and therefore an interface to start and to stop the collection process. This results in the advantage that the data is collected only when necessary to minimize the consumption of resources on the source systems as well as the memory consumption in the storage system.

In addition to the collection on source systems, data must be must be stored on a centralized storage system to become available for potential data consumers, which are unknown at present. The data must be held in a common format to be accessible to a variety of consumers. Since JSON is a standard format for exchanging data, the collected data will be in JSON format, therefore the storage system must be able to store and to query JSON data.

This results in the following functional requirements:

- Collect Dstat system utility data for both systems
- Collect "default" JVM data using JMX for both systems
- Collect application data using JMX for both systems
- Collect application data using REST for Apache Flink
- Data must be transfered into a JSON model
- Data must be stored in JSON format

3.3 Non-Functional Requirements

In addition to the functional-requirements that describe what the software solution is supposed to accomplish, the next section introduces the non-functional requirements, sometimes refered to as "quality attributes" of a system, that essentially specify how the system should behave and represent constraints upon the systems behaviour.

1. **Performance:** the data collection implementation doesn't cause a negative impact regarding system resources like cpu or disk usage on the systems. On architectural level, the collected data should be available in the database in real-time / near real-time. A duration of 2 seconds from leaving the source system until it arrives in the storage system is set as a criterion.

2. **Extensibility:** The collector architecture should be easy to adapt for other source systems like Apache Spark,independent of programming languages used for the collection and storage of data.

3. **Scalability:** Apache Flink and Apache Kafka systems are composed of several interacting components, they operate in clustered environments. The software must be scallable in a way, where it does not matter whether to collect data from a single system or multiple nodes in a cluster. Similarly, the consumers of the data are still unknown. Therefore there is a requirement that consumers can be integrated in the infrastructure easily. Furthermore, it has be ensured at the level of implementation that the software solution scales for further data sources, that might arise in the future, like additional resources in Apache Flink's monitoring API.

4. **Portability:** The solution must not be dependent on which storage technology is used. A change of the database may not affect the data collection process.

5. **Simplicity:** Data consumers and producers should be independent in the way that no specialized operations for interchanging data are required.

3.4 Summary

Summarize results, demarcation criteria

4 System Architecture

TODO: Distributed system -> distributed collection with clients, software solution is a streaming platform itself, cloud environments, client server architecture, client-management with service-discovery, explain", communication via REST, Publish(client) -> Topic <- Subscribe Logstash,

continuous, distributed, time-series "data feed", difference raw and aggregated[Kle16], events as "immutable facts", why? arch uses both, raw because of unknown data consumers, flink-job-index for demonstration of SP.

Pipeline: Collect-Transport-Persistence, see [Les14]

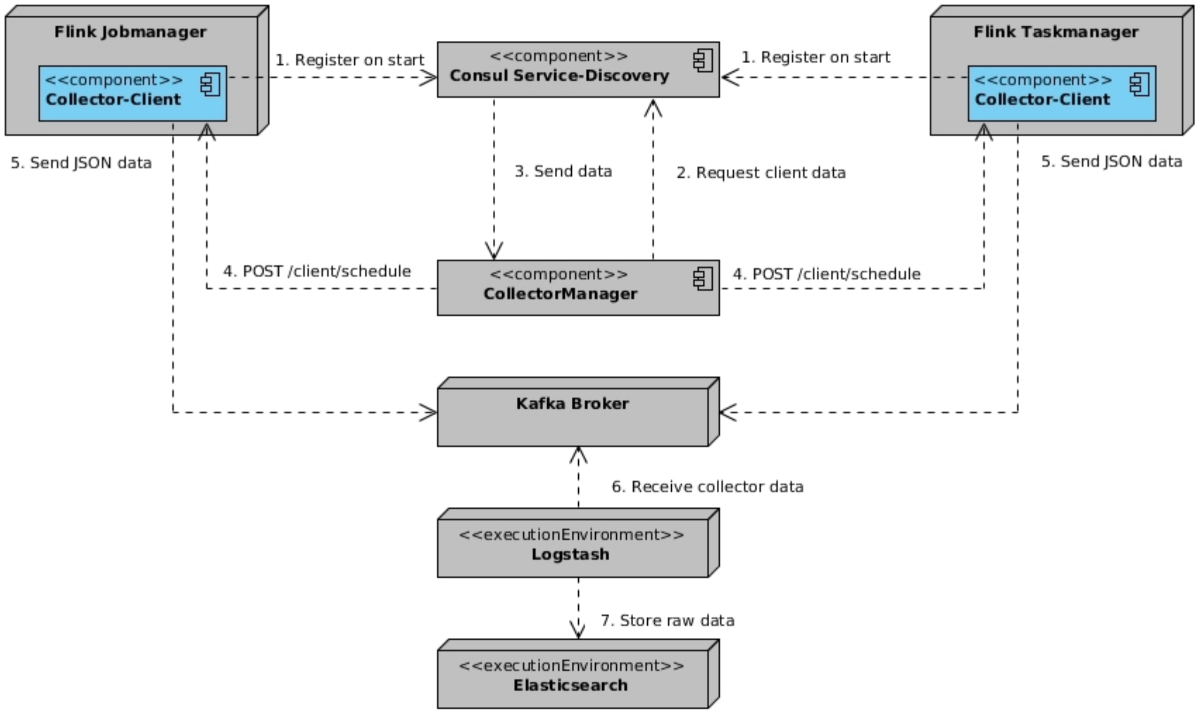

Figure 4.1: Component diagram

4.1 CollectorClient

The CollectorClient tier is our entry point for bringing data into the system , module to gather the data streams from data sources, installed on source systems, a small service that needs to be installed on source systems

data layer

4.2 CollectorManager

Clients need to be scheduled -> management component for Collector clients, gives overview and enables start/stop of registred CollectorClients, uses Consulas client-discovery backend, why? First experiments with implementing a "client-registry", heartbeats, needed another db for storing clients, Consul because of registry for components in distributed systems

4.3 Message-Broker

The Message-Broker integrates distributed collector-client to an overall system by providing a message queue for transmitting data between them. The CollectorClients can publish its data to the queue and the other applications can asynchronously read it from the queue at any time. Message Broker systems persist incoming messages in an special type of a message queue, named topic.

Queueing, see Marz15, compute layer Transport, "Event-Log", see [Kre13] Collect the streams and make them available for consumption

4.4 Indexer

Receive messages from Message-Broker, route data, create ES index, why, describe context BDAA, compute layer

4.5 Persistence

ES as search index for time-series based data, easy visualization with Kibana, why?, searchable, common in BDA, storage layer

4.6 Technologies

The following section introduces the infrastructure components used to realize the software architecture described above.

- Apache Kafka message broker WHY
- Logstash indexer WHY
- Elasticsearch search index WHY

4.7 Summary

Summarize architecture, discuss benefits and disadvantages.

5 Implementation

After Chapter 4 "System Architecture" has shown an overview of the main components of the software solution and explained the interaction between them, this chapter explains implementation details for all major components of the system architecture, including several collector implementations, the "CollectorClient" and "CollectorManager" component.

TODO: spring-boot, spring-kafka, eclipse jetty servlet container

For the implementation of the software components, Java in its version 8 had been chosen. In this current version, it supports more functional elements in form of lambda expression, the processing of collections as Streams as well as an Optional type for handling optional values respectively null values. According to the functional requirements discussed in Chapter 3 "Requirements and Specification", any other programming language could had been used, as long it is possible to provide and consume a RESTfull API and integration for the Producer and Consumer API of Apache Kafka is available.

5.1 Collectors

TODO: In artifarct "collectors-parent", explain SampleCollector concept:

```java
private static Map<String, SampleCollector> jvmSampleRegistry(final MBeanServerConnection mBeanServerConnection) {
    final Map<String, SampleCollector> registry = Maps.newHashMap();
    registry.put(ClassloadingSampleCollector.SAMPLE_KEY, new ClassloadingSampleCollector(mBeanServerConnection));
    registry.put(BufferPoolSampleCollector.SAMPLE_KEY, new BufferPoolSampleCollector(mBeanServerConnection));
```

```
    registry.put(GcSampleCollector.SAMPLE_KEY, new GcSampleCollector(mBeanServerConnection));
    registry.put(MemorySampleCollector.SAMPLE_KEY, new MemorySampleCollector(mBeanServerConnection));
    registry.put(MemoryPoolSampleCollector.SAMPLE_KEY, new MemoryPoolSampleCollector(mBeanServerConnection));
    registry.put(OsSampleCollector.SAMPLE_KEY, new OsSampleCollector(mBeanServerConnection));
    registry.put(RuntimeSampleCollector.SAMPLE_KEY, new RuntimeSampleCollector(mBeanServerConnection));
    registry.put(ThreadSampleCollector.SAMPLE_KEY, new ThreadSampleCollector(mBeanServerConnection));
    return registry;
}
```

Code snippet 5.1: Sample registry for "JvmCollector"

TODO: Collector results aggregated of sample collector results, extensibility for further sample sources

```
@Override
public CompletableFuture<CollectorResult> collect() {
    LOG.debug("Entering AbstractCollector collect()");
    final CompletableFuture<CollectorResult> collectorResultCF = CompletableFuture.supplyAsync(() -> {
    checkRegistry();
    final List<CompletableFuture<Map<String, Object>>> sampleResultCPList =
        getSampleRegistry().values().stream()
                    .map(SampleCollector::collectSample)
                    .collect(Collectors.toList());

    return CompletableFuture.allOf(sampleResultCPList.toArray(
        new CompletableFuture[sampleResultCPList.size()]))
            .thenApply(aVoid ->
                sampleResultCPList.stream().map(CompletableFuture::join)
                    .collect(Collectors.toList()))
            .thenApply(sampleResults -> {
```

```
                final Map<String, Object> dataMap = Maps.newLinkedHashMap();
                sampleResults.forEach(dataMap::putAll);
                final CollectorResult collectorResult =
                    new ↩
                        CollectorResult(getCollectorType().name().toLowerCase(), ↩
                        dataMap);
                LOG.debug("Finished AbstractCollector collect()");
                return collectorResult;
            }).join();
        });
        LOG.debug("Immediately return from AbstractCollector collect()");
        return collectorResultCF;
}
```

Code snippet 5.2: The "collect"-algorithm in "AbstractCollector"

5.1.1 CollectorType

TODO: Distinguishes collectors, meta information not neccessarily required

```
public enum CollectorType {
    JVM_JMX,
    DSTAT,
    FLINK_REST,
    FLINK_JMX,
    KAFKA_BROKER_JMX
}
```

Code snippet 5.3: Collector types

5.1.2 CollectorResult

Data events as "immutable facts" with state state of system at time, on host, at port, the type of collector and data. Usage of Jackson for JSON serialization.

```java
public class CollectorResult {
    @JsonProperty("client-timestamp")
    private final LocalDateTime clientTimestamp;

    @JsonProperty("client-host")
    private final String clientHost;

    @JsonProperty("client-port")
    private final Integer clientPort;

    @JsonProperty("instance-id")
    private final String instanceId;

    @JsonProperty("collector-type")
    private final String collectorType;

    private final Map<String, Object> data;
}
```

Code snippet 5.4: CollectorResult

5.1.3 JvmCollector

Collects data according to Table 3.2, standard set of management interfaces available for JVM data will be used.

5.1.4 DStatCollector

Dstat parameters, see Table 3.1 in Chapter "Basic Concepts"

```java
private static final String[] DSTAT_COMMAND = {"dstat", "-t",
    "--cpu", "--top-cpu-adv", "--top-cputime", "--top-cputime-avg",
    "--disk", "--disk-tps", "--disk-util",
    "--net", "--socket", "--tcp", "--udp",
    "--io", "--top-io-adv", "--lock", "--fs",
```

```
    "--mem", "--top-mem", "--page", "--swap", "--vm",
    "--sys", "--load", "--ipc", "--unix",
    "--proc", "--proc-count", "--top-latency", "--top-latency-avg",
    "--full",
    "--float", "1", "0"};
```

Code snippet 5.5: Dstat program parameters in "DstatCollector"

Dstat process with the given parameters results in string containing three lines, where only the third line ist required to to gather data of.

All collector implementations use a sample registry to achieve more flexibility in what data to collect:

```
private static Map<String, AbstractDstatSampleCollector> ↩
    defaultSampleRegistry() {
    final Map<String, AbstractDstatSampleCollector> registry = ↩
        Maps.newHashMap();
    registry.put(CpuSampleCollector.SAMPLE_KEY, new CpuSampleCollector());
    registry.put(DiskSampleCollector.SAMPLE_KEY, new DiskSampleCollector());
    registry.put(IoSampleCollector.SAMPLE_KEY, new IoSampleCollector());
    registry.put(MemorySampleCollector.SAMPLE_KEY, new ↩
        MemorySampleCollector());
    registry.put(NetSampleCollector.SAMPLE_KEY, new NetSampleCollector());
    registry.put(ProcessSampleCollector.SAMPLE_KEY, new ↩
        ProcessSampleCollector());
    registry.put(SystemSampleCollector.SAMPLE_KEY, new ↩
        SystemSampleCollector());
    return registry;
}
```

Code snippet 5.6: Sample registry in "DstatCollector"

Java ProcessBuilder to create Dstat process and read output using an InputStreamReader provided by the Process object.

```
final ProcessBuilder processBuilder = new ProcessBuilder(DSTAT_COMMAND);
processBuilder.redirectErrorStream(true);
```

```java
final Process process = processBuilder.start();
try (BufferedReader processOutputReader =
    new BufferedReader(new InputStreamReader(process.getInputStream()))) {
        final String dstatResult = processOutputReader.lines()
            .map(String::toString)
            .collect(Collectors.joining(System.lineSeparator()));
        final int exitCode = process.waitFor();
}
```

Code snippet 5.7: ProcessBuilder in "DstatCollector"

The result will be splitted by System.lineSeparator in AbstractDstatSampleCollector. Implements Function<String, Map<String, Object», a functional interface that takes on input and creates a result on apply.

```java
@Override
public Map<String, Object> apply(final String dstatResult) {
    final String[] lines = dstatResult.split(System.lineSeparator());
        if (lines.length != 3) {
            throw new DStatCollectorException();
        }
    return createResultMap(applyData(lines));
}
```

Code snippet 5.8: Split Dstat input in "AbstractDstatSampleCollector"

DstatCollector iterates through sample registry and invokes "collect" for all registred sample collector implementations. Usage of futures for non-blocking code, block as less as possible to increase performance on source systems.

```java
final List<CompletableFuture<Map<String, Object>>> dStatDataFuturesList =
    sampleRegistry.values().stream()
        .map(collector ->
            CompletableFuture.supplyAsync(() ->
                collector.apply(dstatResult))).collect(Collectors.toList());
```

Code snippet 5.9: Iterate sample registry in "DstatCollector"

All Dstat sample collectors are based on regular expressions, the third line of the result is splitted, and the data of interest extracted:

```
CPU_USAGE_PATTERN = Pattern.compile("" +
    "(\\d+(\\.\\d+)?)(\\s*)" +
    "(\\d+(\\.\\d+)?)(\\s*)" +
    "(\\d+(\\.\\d+)?)(\\s*)" +
    "(\\d+(\\.\\d+)?)(\\s*)" +
    "(\\d+(\\.\\d+)?)(\\s*)" +
    "(\\d+(\\.\\d+)?)");

private static Map<String, Object> parseCpuUsage(final String rawData, ↩
    final String cpuName) {
    return Optional.ofNullable(rawData).map(raw -> {
        final Matcher matcher = CPU_USAGE_PATTERN.matcher(raw.trim());
        final Map<String, Object> cpuUsageMap = Maps.newLinkedHashMap();
        if (!matcher.matches()) {
            LOG.warn("Unable to parse 'CpuUsage'");
        } else {
            try {
                cpuUsageMap.put(CPU_NAME_KEY, cpuName);
                cpuUsageMap.put(CPU_USAGE_USER_KEY, ↩
                    Float.valueOf(matcher.group(1)));
                cpuUsageMap.put(CPU_USAGE_SYSTEM_KEY, ↩
                    Float.valueOf(matcher.group(4)));
                ...
            } catch (NumberFormatException ex) {
                LOG.warn("Unable to parse 'CpuUsage'");
            }
        }
        return cpuUsageMap;
    }).orElse(Maps.newHashMap());
}
```

Code snippet 5.10: Extract sample date in "CpuSampleCollector"

At the end of Dstat collection, the futures of the different sample collectors will be aggregated to one CollectorResult future, that will be returned to the caller of the collect() method of DstatCollector.

```
dStatDataFuturesList
    .stream()
        .map(CompletableFuture::join)
            .collect(Collectors.toList()))
                .thenApply(dstatSamples -> {
                    final Map<String, Object> data = Maps.newLinkedHashMap();
                    dstatSamples.forEach(data::putAll);
                    final CollectorResult dstatCollectorResult = new ↩
                        CollectorResult(CollectorType.DSTAT.name().toLowerCase(), ↩
                        data);
                    LOG.debug("Finished {} collecting", CollectorType.DSTAT);
                    return dstatCollectorResult;
                    }).join();
```

Code snippet 5.11: Aggregation of sample results in "DstatCollector"

5.1.5 FlinkRestCollector

5.1.6 KafkaBrokerJmxCollector

5.1.7 AbstractCollector

5.2 CollectorClient

5.2.1 CollectorWorker

5.2.2 KafkaOutboundWriter

5.2.3 ScheduleController

5.2.4 ClientMetadataController

5.3 CollectorManager

5.3.1 CollectorClientInstanceService

5.3.2 MetadataRestClient

5.3.3 IndexController

5.4 Summary

Maybe Spring alternatives, Lagom, VertX, Play?

Maybe collector as agent, Instrumentation instead of separate service

Alternatives REST, maybe (Web-)Sockets

Possible secururity risk because remote JMX, firewalls

6 Test and Evaluation

6.1 Local test environment

Unit- and Integration tests, IT needs docker infrastructure, usually mocked JMX, REST data, lack of time, test separation via naming *Test and *IT via Maven Failsafe, check test coverage,

6.2 Docker environment

Short Docker intro, benefits in microservice environments, describe setup for components (docker-compose.yml), describe modifications made for Apache Flink and Apache Kafka to enable JMX remote access

6.3 Observations

CollectorDataProcessor: module to analyze the data streams creating derived streams and persist flat data -> data transformation, analytics layer

Kibana dashboard, show visualization of CollectorDataProcessor data

6.4 Discussion

6.5 Summary

TODO

7 Conclusion

TODO

7.1 Summary

7.2 Outlook

List of Figures

List of Tables

List of Source Codes

Bibliography

[Chr07] Reiner Dumke Christof Ebert. *Software Measurement, Establish - Extract - Evaluate - Execute*. Spring-Verlag Berlin Heidelberg New York, 2007. ISBN: 978-3-540-71648-8.

[Det13] et al Detlef Apel Wolfgang Behme. *Datenqualität erfolgreich steuern - Praxislösungen für Business-Intelligence-Projekte*. dpunkt.verlag GmbH, Heidelberg, 2013. ISBN: 978-3446420564.

[Fie00] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures, University of California*. 2000. URL: `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`.

[Fli16] Flink. *Apache Flink Documentation*. 2016. URL: `http://flink.apache.org` (visited on 08/18/2016).

[Hea03] Leigh Williamson Heather Kreger Ward Harold. *Java TM and JMX, Building Manageable Systems*. Addison-Wesley, 2003. ISBN: 978-0672324086.

[Höb00] Valentin Höbel. *Systemdiagnose von Vmstat über Netstat bis Dstat*. 2000. URL: `http://www.linux-magazin.de/Ausgaben/2012/05/Dstat-Co`.

[Inc16] Sun Microsystems Inc. *JavaTM Management Extensions (JMXTM) Specification,version 1.4*. 2016. URL: `http://download.oracle.com/otndocs/jcp/jmx-1_4-mrel4-eval-spec/`.

[Jay11] Jun Rao Jay Kreps Neha Narkhede. *Kafka: a Distributed Messaging System for Log Processing*. 2011. URL: `http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf` (visited on 08/21/2016).

[Jör14b] u.a Jörg Bartel Dr. Bernd Pfitzinger. *Big Data im Praxiseinsatz – Szenarien, Beispiele, Effekte*. 2014. URL: https://www.bitkom.org/Publikationen/2012/Leitfaden/Leitfaden-Big-Data-im-Praxiseinsatz-Szenarien-Beispiele-Effekte/BITKOM-LF-big-data-2012-online1.pdf (visited on 08/06/2016).

[Kaf16] Kafka. *Apache Kafka Documentation*. 2016. URL: http://kafka.apache.org (visited on 08/18/2016).

[Kle16] Martin Kleppmann. *Making Sense of Stream Processing*. First edition. Sebastopol, CA 95472: O'Reilly Media, Inc., 2016. ISBN: 978-1-491-94010-5.

[Kre13] Jay Kreps. *The Log: What every software engineer should know about real-time data's unifying abstraction*. 2013. URL: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying (visited on 08/18/2016).

[Les14] Tammo van Lessen. "Wissen, was läuft - Mit Laufzeitmetriken den Überblick behalten". In: *Javamagazin* 10000.11 (2014), pp. 48–52.

[Nam15] Dmitry Namiot. "On Big Data Stream Processing". In: *International Journal of Open Information Technologies* 1 (2015), pp. 48–51.

[Ora16] Oracle. *Java Platform, Standard Edition 8 API Specification*. 2016. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/management/package-summary.html.

[ORe13] Tim O'Reilly. *George Dyson's Definition of "Big Data"*. 2013. URL: https://plus.google.com/+TimOReilly/posts/Ej72QmgdJTf (visited on 08/18/2016).

[Pau12] et al Paul C. Zikopoulos Chris Eaton. *Understanding Big Data - Analytics for Enterprise Class, Hadoop and Streaming Data*. United States of America: The McGraw-Hill Companies, 2012. ISBN: 978-0-07-179053-6.

[Wie16] Dag Wieers. *Dstat: Versatile resource statistics tool*. 2016. URL: http://dag.wiee.rs/home-made/dstat/.

Image resources

[Gro14] Cloud Security Alliance - Big Data Working Group. *Big Data Taxonomy*. 2014. URL: https://downloads.cloudsecurityalliance.org/initiatives/bdwg/Big_Data_Taxonomy.pdf (visited on 08/18/2016).

[Jör14a] u.a Jörg Bartel Axel Mester. *Big Data Technologien – Wissen für Entscheider*. 2014. URL: https://www.bitkom.org/Publikationen/2014/Leitfaden/Big-Data-Technologien-Wissen-fuer-Entscheider/140228-Big-Data-Technologien-Wissen-fuer-Entscheider.pdf (visited on 08/06/2016).

[Nat15] James Warren Nathan Marz. *Big Data - Principles and best practices of scalable real-time data systems*. Shelter Island, NY 11964: Manning Publications Co., 2015. ISBN: 978-1-617-29034-3.

[Rob] Kostas Tzoumas Robert Metzger. *Kafka + Flink: A practical, how-to guide*. URL: http://data-artisans.com/kafka-flink-a-practical-how-to/.

A

A.1 Diagrams

A.1.1 Use Case diagram

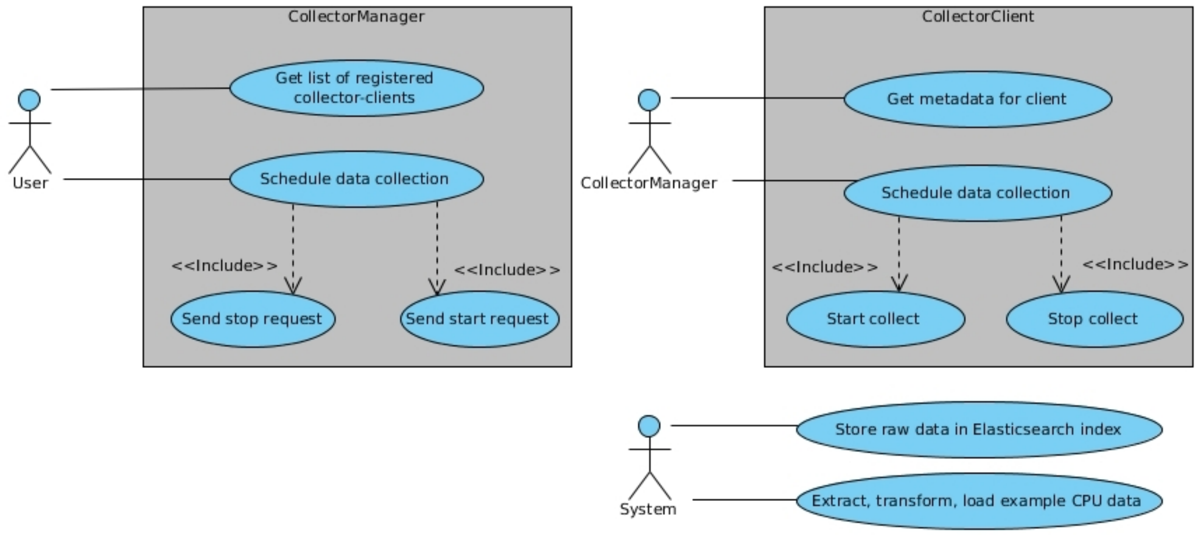

Figure A.1: Use Case Diagramm

A.1.2 Class diagrams

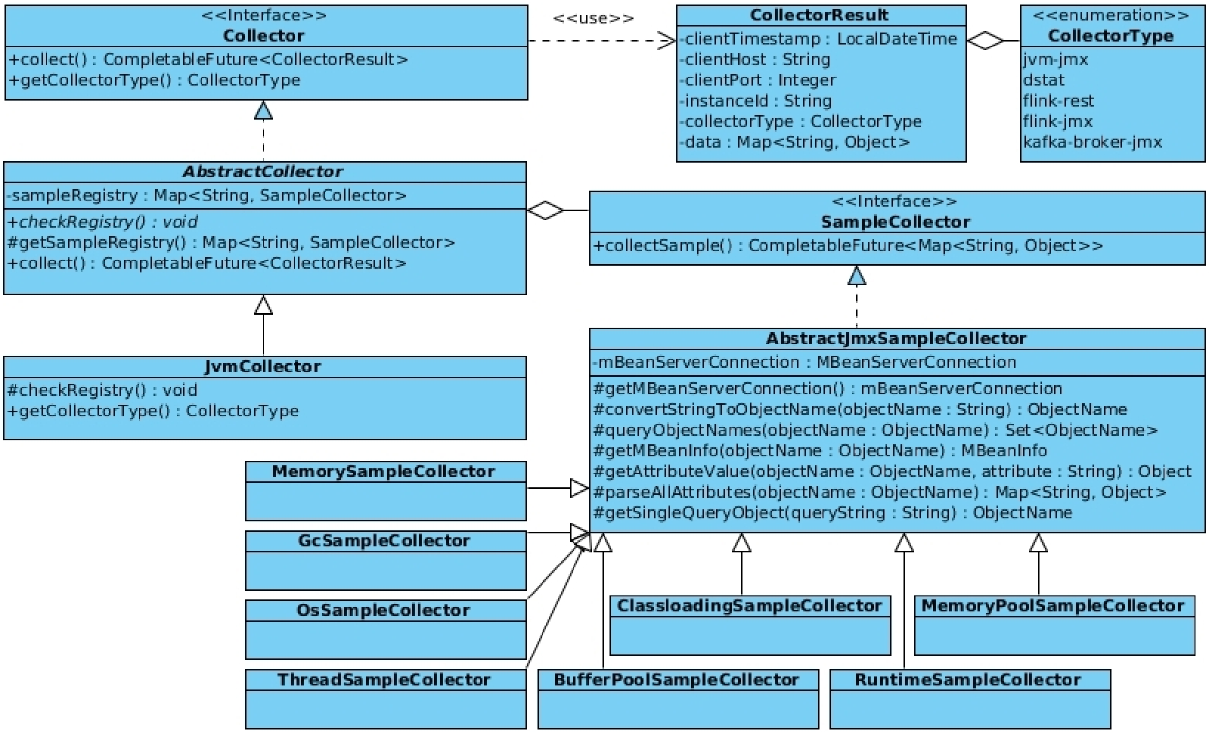

Figure A.2: Class diagram 'JvmCollector'

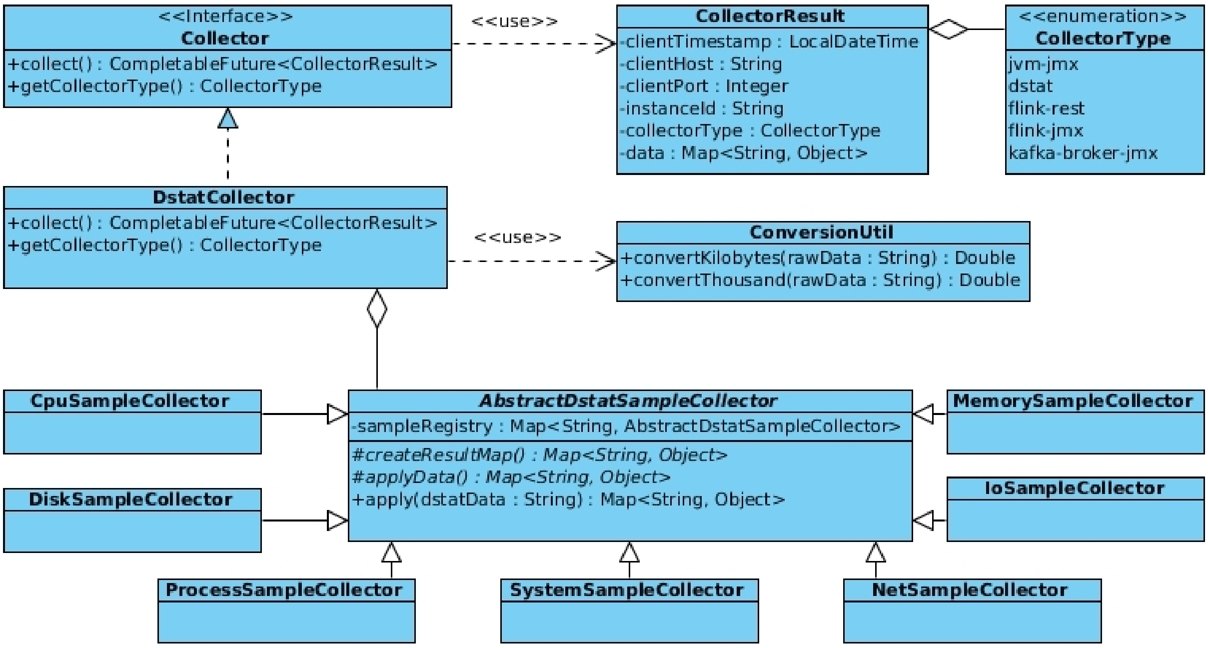

Figure A.3: Class diagram 'DStatCollector'

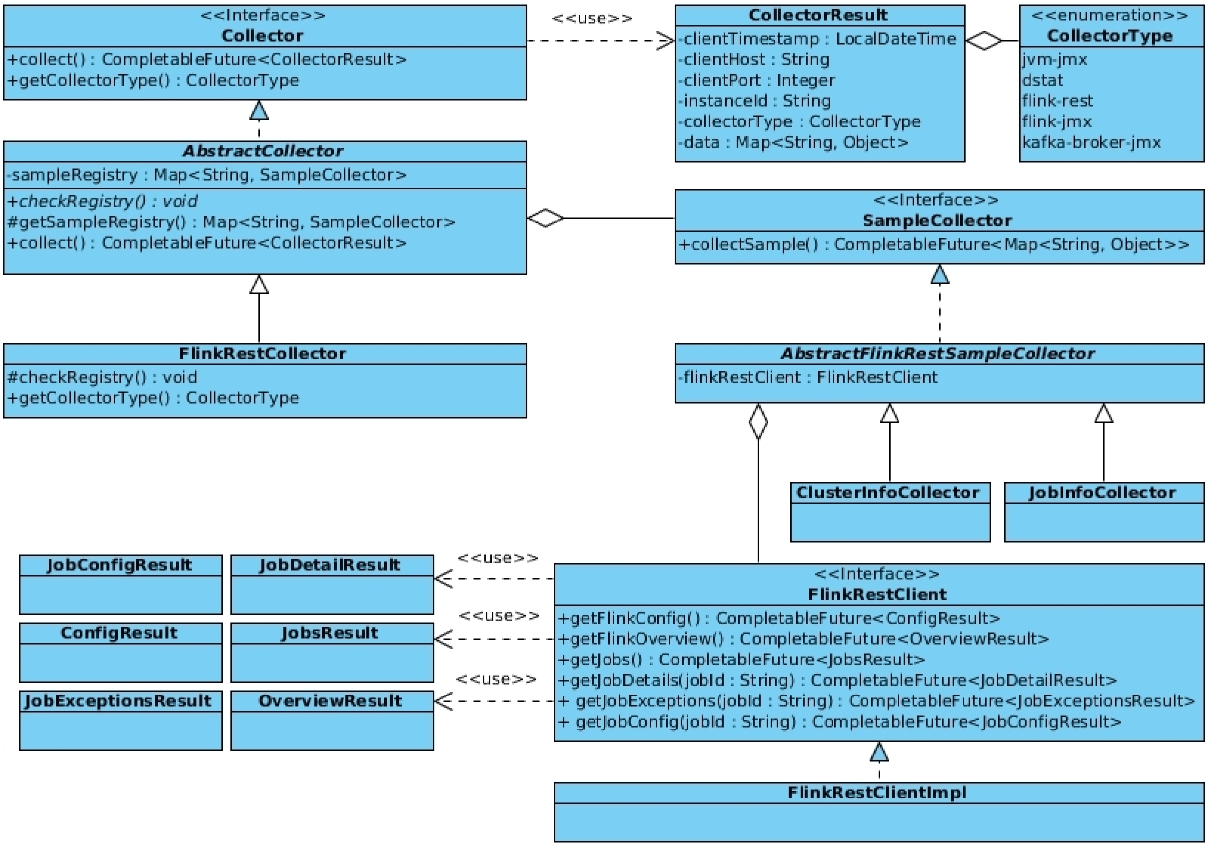

Figure A.4: Class diagram 'FlinkRestCollector'

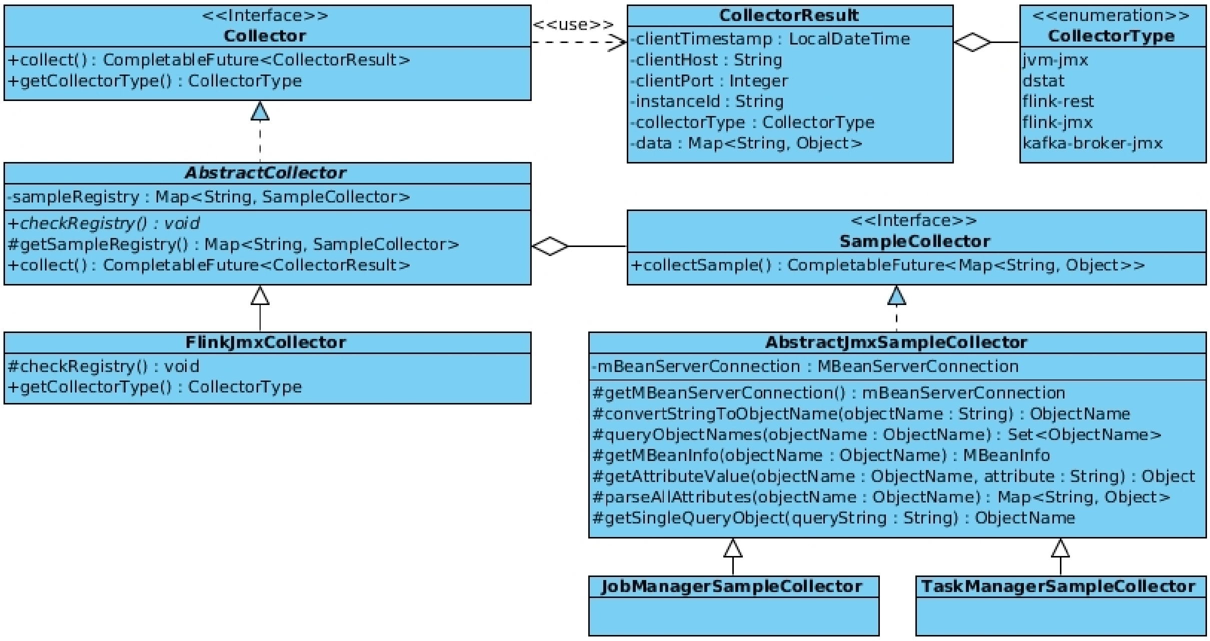

Figure A.5: Class diagram 'FlinkJmxCollector'

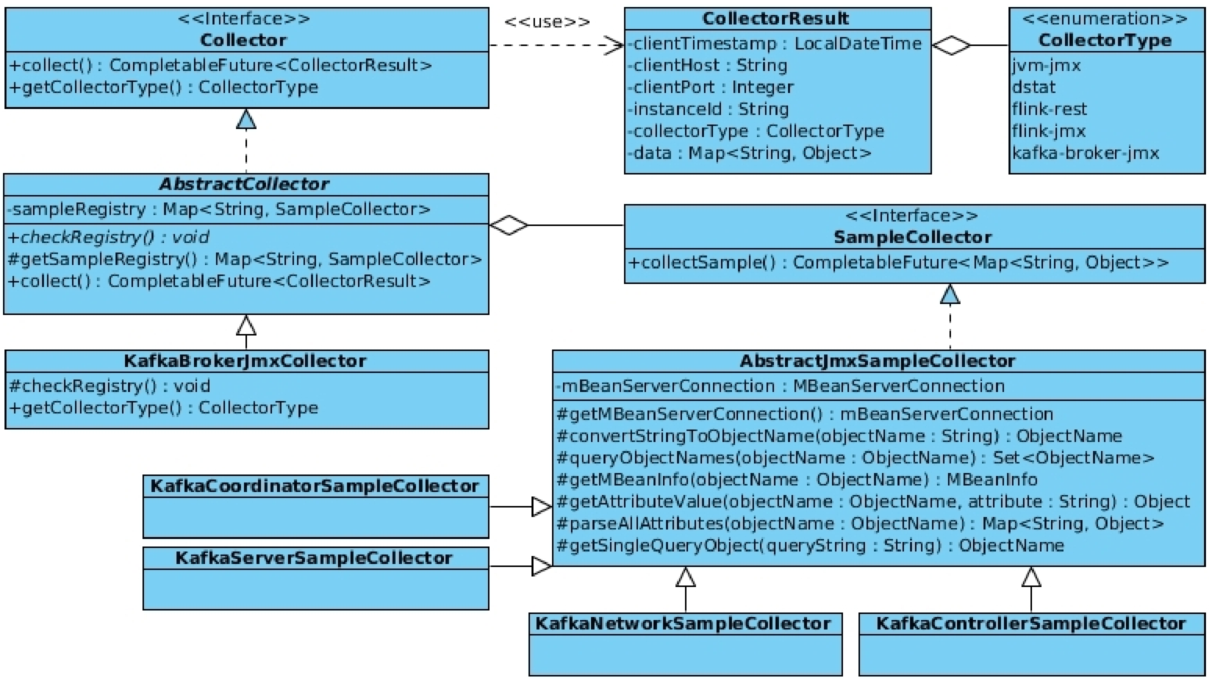

Figure A.6: Class diagram 'KafkaBrokerJmxCollector'

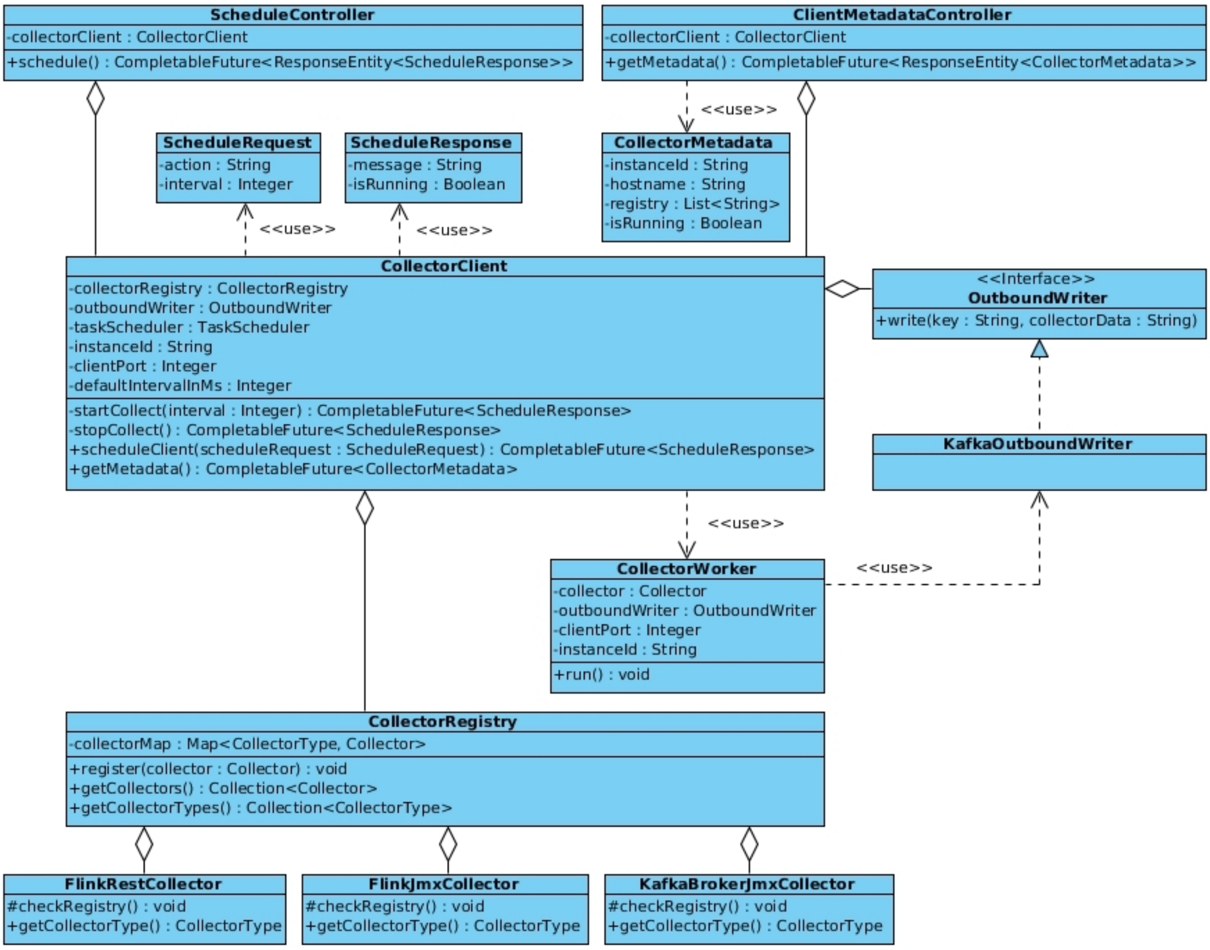

Figure A.7: Class diagram 'CollectorClient'

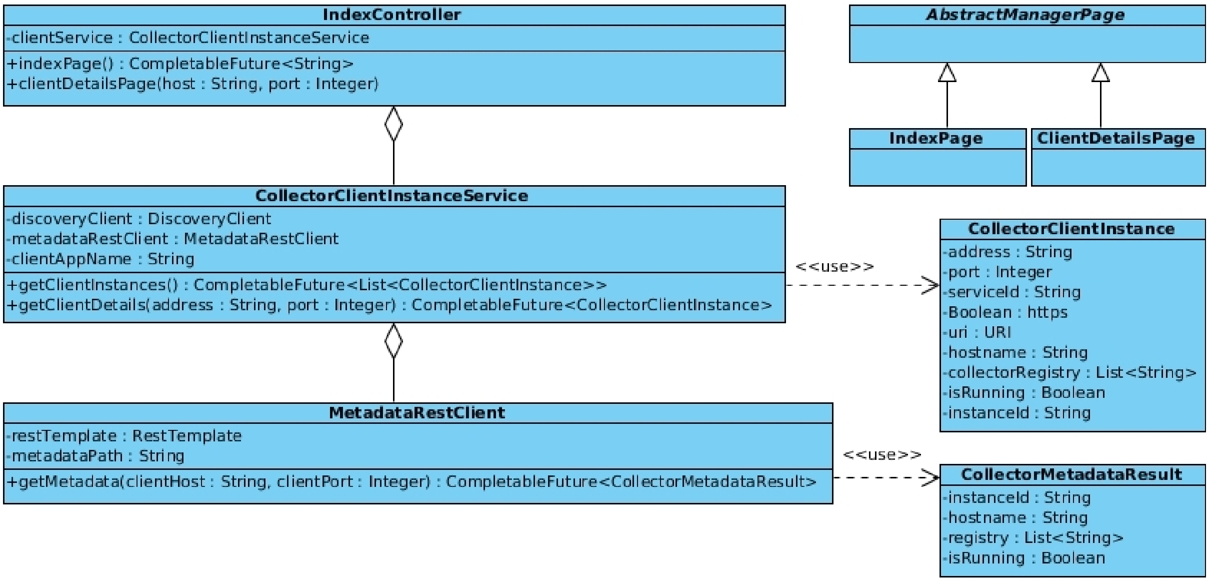

Figure A.8: Class diagram 'CollectorManager'

A.1.3 Sequence diagrams

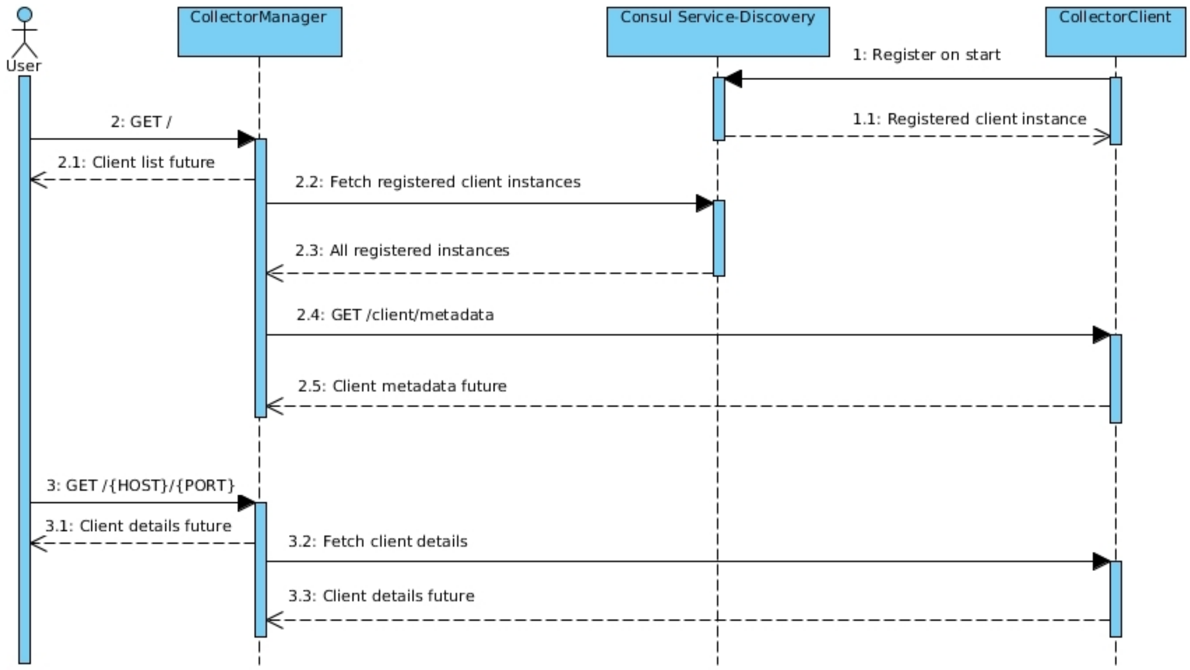

Figure A.9: Sequence diagram 'Client discovery'

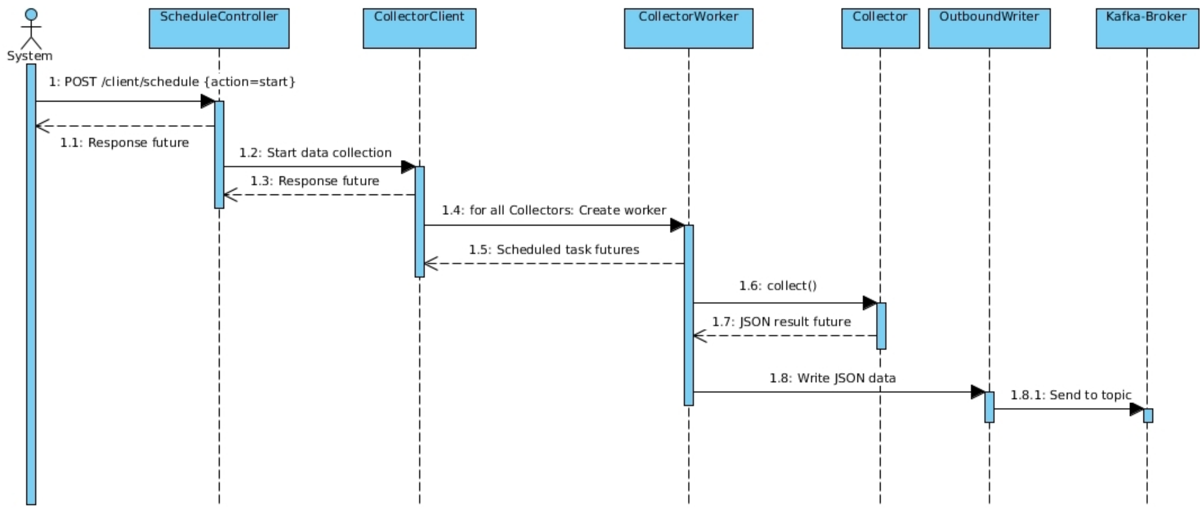

Figure A.10: Sequence diagram 'Client scheduling'

A.1.4 Component diagram

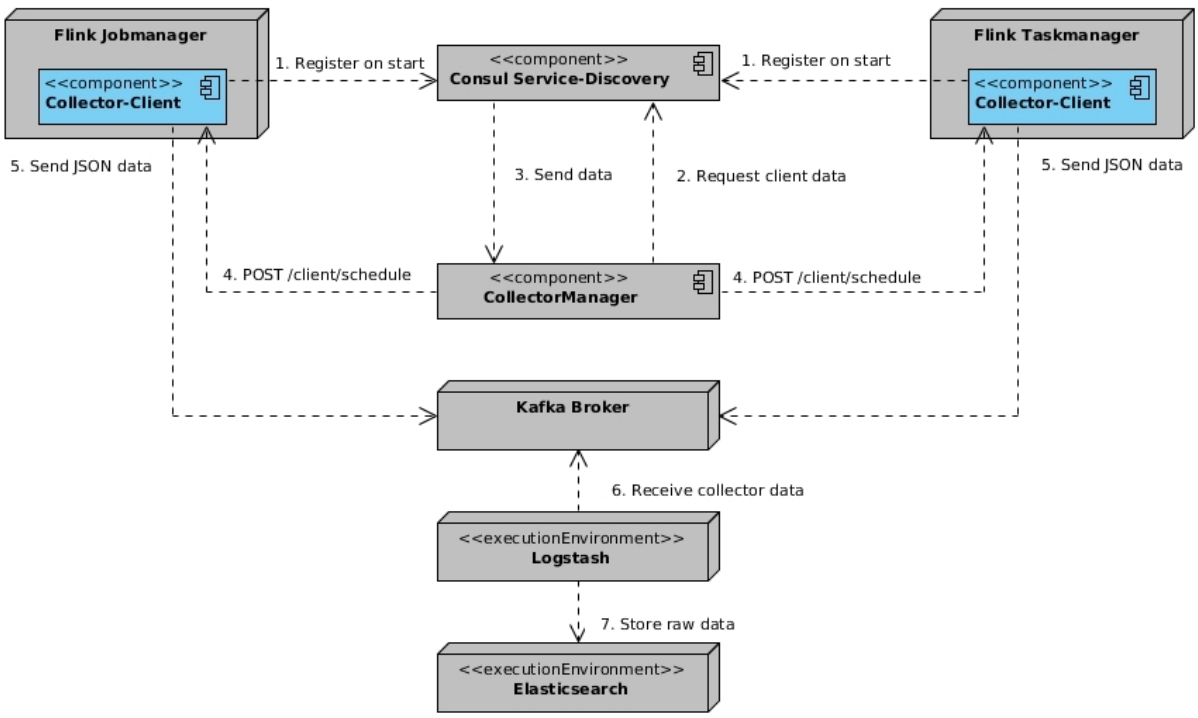

Figure A.11: Component diagram

A.1.5 Deployment diagram

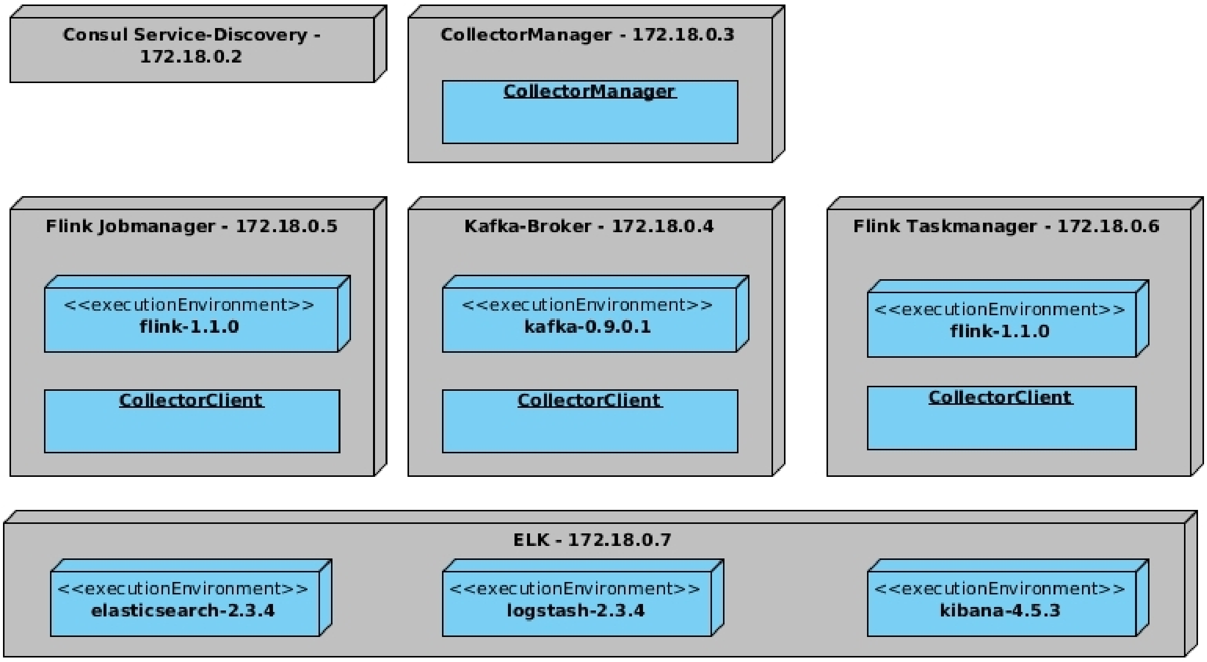

Figure A.12: Deployment diagram

A.2 Apache Kafka MBeans version 0.9.0.2

JMX ObjectName
kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs
kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec
kafka.controller:type=KafkaController,name=ActiveControllerCount
kafka.controller:type=KafkaController,name=OfflinePartitionsCount
kafka.controller:type=KafkaController,name=PreferredReplicaImbalanceCount
kafka.network:type=Processor,name=IdlePercent,networkProcessor=*
kafka.server:type=socket-server-metrics,networkProcessor=*
kafka.server:type=controller-channel-metrics,broker-id=*
kafka.server:type=ReplicaManager,name=IsrExpandsPerSec
kafka.server:type=ReplicaManager,name=IsrShrinksPerSec
kafka.server:type=ReplicaManager,name=LeaderCount
kafka.server:type=ReplicaManager,name=PartitionCount
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent
kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec
kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=*
kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec
kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=*
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=*
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=*
kafka.server:type=BrokerTopicMetrics,name=BytesRejectedPerSec
kafka.server:type=BrokerTopicMetrics,name=BytesRejectedPerSec,topic=*
kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec
kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=*
kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec
kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=*
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=*
kafka.coordinator:type=GroupMetadataManager,name=NumGroups
kafka.coordinator:type=GroupMetadataManager,name=NumOffsets

Table A.1: Collected Kafka MBeans

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Stadt, den xx.xx.xxxx

Max Mustermann