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 University, 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 fetch 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 similar 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 choosed to be the one to work out at last.

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 will be 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 (218 contributers, August 29th 2016, see [**FlinkG16**]) and a wide range of companies that are actively using it.

1.2 Objective

The main goal of the thesis is the design and implementation of a working software system to ingest and store data that can be collected from Apache Flink and Apache Kafka and represents the potential data providing component for the self-learning system described above. 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 topic and the main goals of the present thesis in this chapter, **??** discusses basic concepts of Big Data, Stream Processing and introduces Apache Flink and Apache Kafka as representatives of widely used streaming frameworks. In preparation of **??**, both Representational State Transfer (REST) and the Java Management Extensions (JMX) as possibilities of remote data access in distributed systems will be discussed.

?? 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 elaborated requirements, **??** introduces the system architecture by giving a detailed conceptional overview of the components involved, whereas **??** discusses implementation details for selected items.

?? introduces the local test environment and gives an detailed introduction to setup the technical environment for and the usage of the prototype to verify the correct functionality related to the requirements defined in **??**.

The last **??** 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[**Marz15**]. 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[**Ziko12**].

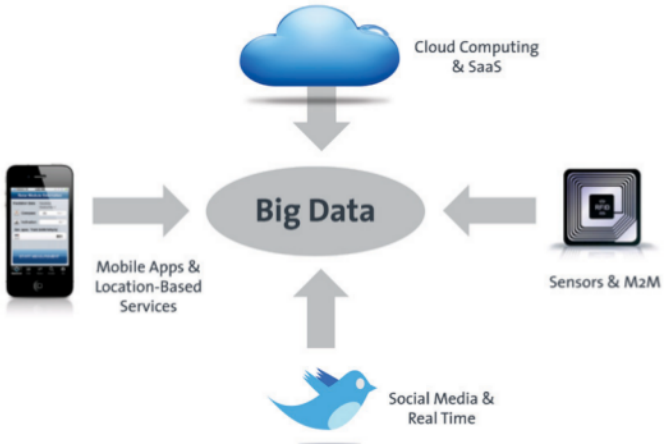

Figure 2.1: Sources of Big Data[**Bitk12**]

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. [**Bitk12**] 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 [**Marz15**] [**Ziko12**], 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

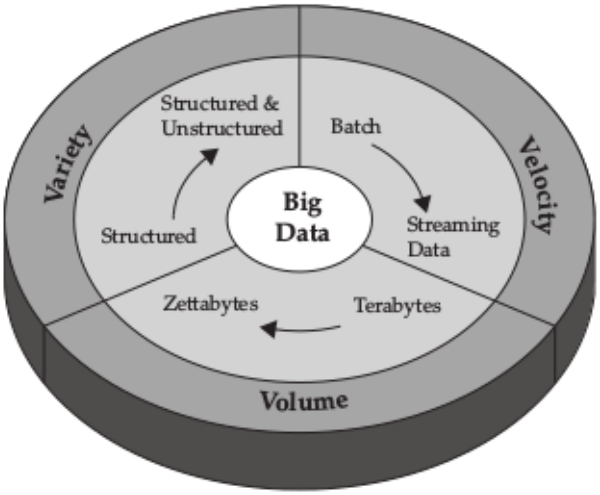

Figure 2.2: The three 'V's of Big Data[**Ziko12**]

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 [**Dys13**]: *"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 [**Marz15**] 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 [**Marz15**]. 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 **??**.

The illustration below summarises the six-dimensional taxonomy [**Bitk14**; **Csa14**] of Big Data Analytics Applications.

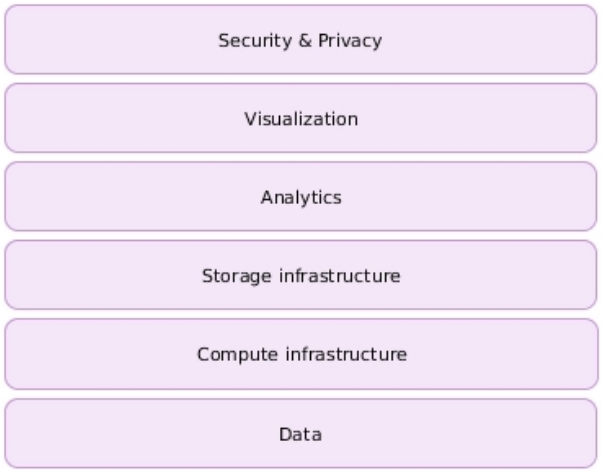

Figure 2.3: Taxonomy of Big Data Analytics Applications [**Csa14**]

The following **??** will discuss the topic stream processing, which is part of the "Compute infrastructure" layer shown in **??** 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 [**Klepp16**], 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 their 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 [**FlinkD16**], *"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 [**FlinkD16**] shows a working example of a streaming application that counts the words coming from a web socket in 5 second windows:

```
1  public static void main(String[] args) throws Exception {
2          StreamExecutionEnvironment env = ↩
               StreamExecutionEnvironment.getExecutionEnvironment();
3          DataStream<Tuple2<String, Integer>> dataStream = env
4                  .socketTextStream("localhost", 9999) *(1)
5                  .flatMap(new Splitter()) *(2)
6                  .keyBy(0) *(2)
7                  .timeWindow(Time.seconds(5)) *(2)
8                  .sum(1); *(2)
9
10         dataStream.print(); *(3)
11         env.execute("Window WordCount");
```

```
12      }
13
14      public static class Splitter implements FlatMapFunction<String, ↩
            Tuple2<String, Integer>> {
15          @Override
16          public void flatMap(String sentence, Collector<Tuple2<String, ↩
                Integer>> out) throws Exception {
17              for (String word: sentence.split(" ")) {
18                  out.collect(new Tuple2<String, Integer>(word, 1));
19              }
20          }
21      }
```

Code listing 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 **??** 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 **??**.

2.3.2 Apache Kafka

Apache Kafka is publish-subscribe queuing service rethought as a distributed commit log [**Kafka16**], 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 [**Neha11**] 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.

?? 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 persistent medium.

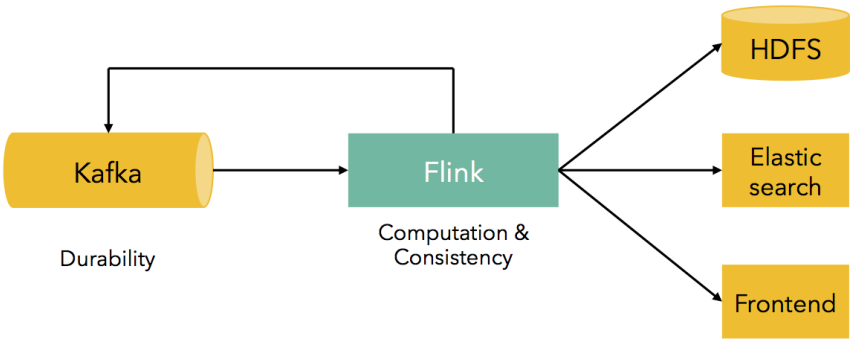

Figure 2.4: A typical Kafka-Flink pipeline[**Dartisans15**]

?? describes the Docker setup for a single Kafka node that is part of the software platform 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"[**Field00**].

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, inter alia, according to [**Field00**]:

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.

?? 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 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"* [**Sun13**], *"an isolation and mediation layer between manageable resources and management systems."* [**Kreg03**]. 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 archictecture discussed in **??**.

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 **??**, the url `service:jmx:rmi:///jndi/rmi://localh`

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

This chapter 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 in **??**, this chapter explains 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 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."[**Ebert07**]. 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.[**VanL14**].

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 sections discuss 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) [**Hoeb12**].

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)."*[**Wieers16**]

Dstat is a command line tool, **??** shows the immediate output of running the application with the argument "–full", which expands more detailed information about multiple cpus and network interfaces:

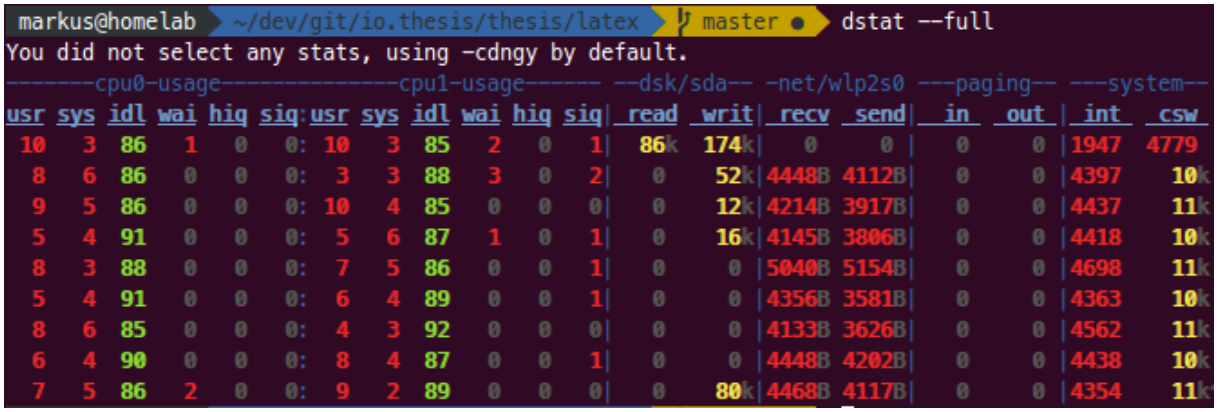

Figure 3.1: Output "dstat –full"

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

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 **??**. 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" [**Javadoc16**].

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-related data of the underlying application, is available for Apache Flink and Apache Kafka and will be included in the software solution in discussed in **??**.

Apache Flink

Apache Flink provides application data via its monitoring API, a RESTful API following the principles introduced in **??**, 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 **??**.

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 responses 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 **??** and the data provided by the monitoring API in **??**, 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 of 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 [**Daqua13**] and [**Ebert07**] 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.

According to the examinations of system and application data available for Apache Flink and Apache Kafka in **??** and **??**, 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 system and application 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 **??**. The same applies for the data available via Apache Flink's monitoring API in **??** 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. 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 **??** and **??**.

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 tansferred and 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, the following three main functional requirements derive:

1. Collection of data from Apache Flink and Apache Kafka source systems.
2. Transport of collected data to the storage system.
3. Storage of collected data in a database.

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 **??**.

The collected data must be brought into format, that enables the transport and storage of 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 send to the storage system. Another advantage of JSON is that it independent of any programming language, parsers for any languages 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
- The collection process must be able to be started/stopped
- Data must be converted into a JSON model

- Data must be transfered to the storage system
- 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 source 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

In preparation for the software platform that will be introduced in **??**, this chapter discussed data sources available for Apache Flink and Apache Kafka, which led to the conclusion to collect system data using the Dstat system tool to gather data related to cpu, disk, memory resources and so on. The Java Management Extensions (JMX) will be used, to access application data for both of the systems. In addition, Apache Flink provides a monitoring API, which will be queried to gain access to detailed information regarding cluster status and jobs.

4 System Architecture

After the context in which the solution is settled has been described in **??**, **??** inspected Apache Flink and Apache Kafka according available sources of data and evaluated the data based on defined criteria of data quality. As a result, the functional and non-functional requirements had been defined. According to the main goals of this thesis, the collection of system and application data from Apache Flink and Apache Kafka and storage of collected data in a centralized persistence system, the following sub-problems had been identified:

1. Both source systems operate as distributed systems which consists of multiple instances of Apache Flink Job- and TaskManagers or Apache Kafka brokers, producers and consumers respectively. The software solution proposed must enable the collection of data from multiple application instances to gain an overall picture of the entire system.

2. The process of data collection mustn't cause a negative impact on source systems. Since Apache Flink and Apache Kafka are both systems processing large amounts of data very fast, the utilization of system resources like cpu and memory of the collecting system must be kept as low as possible.

3. As a result, the data must be transfered to the storage system with as less input/output operations and as fast as possible, to become available for possible data consumers immediately.

4. The system architecture must provide a mechanism for the integration of possible data consumers which are unknown at the point of this thesis.

The following chapter introduces the *"Collector-Platform"* as a pipeline of data collection, transport and storage with the purpose to collect data of distributed Apache Flink and

Kafka systems and make it available for consumers that are potentially unknown at present as fast as possible. The components will be discussed at architectural level, whereas implementation details follow in **??**.

4.1 The "Collector-Platform"

According to the requirements in **??**, the *"Collector-Platform"* is responsible to perform the following operations, which describe the pipeline the data in running throught on each source system. Irrespective of the source system, the sequence remains the same for Apache Flink and Kafka or any other source system the *CollectorClient* is installed on, it just differs in the implementations of data collection:

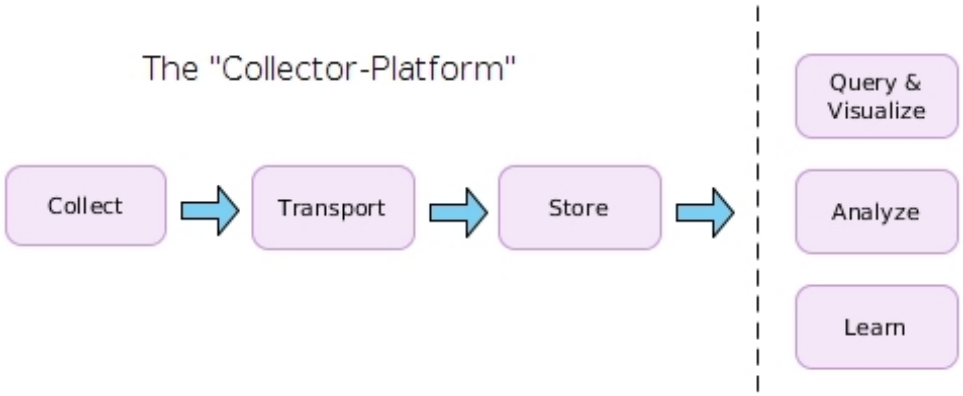

Figure 4.1: Collector Pipeline, [**VanL14**]

The the first step is the collection of data defined in **??** on the Apache Flink and Apache Kafka source systems. This includes collection of system data with the Dstat system utility as well as application specific data provided by JMX or REST respectively. Once the the corresponding data had been retrieved and converted into a structured JSON format, the data must leave source systems in the second step to reach the storage system at last, where they become available for visualization tools, analytical processing or as a source for Machine Learning applications. **??** introduces a sample application to demonstrate the possibilities for the integration of data consumers using the Stream Processing capabilities of Apache Flink.

The "Collector-Platform" consist of the following components and will be discussed in the corresponding sections:

- CollectorClient
- CollectorManager
- Consul ClientRegistry
- Kafka Message-Broker
- Logstash-Processor
- Elasticsearch Index

The following diagram shows an overview of all components involved and the interactions between them to fulfill the requirements of data collection, transport and storage on the example of one Apache Flink JobManager and TaskManager as source systems.

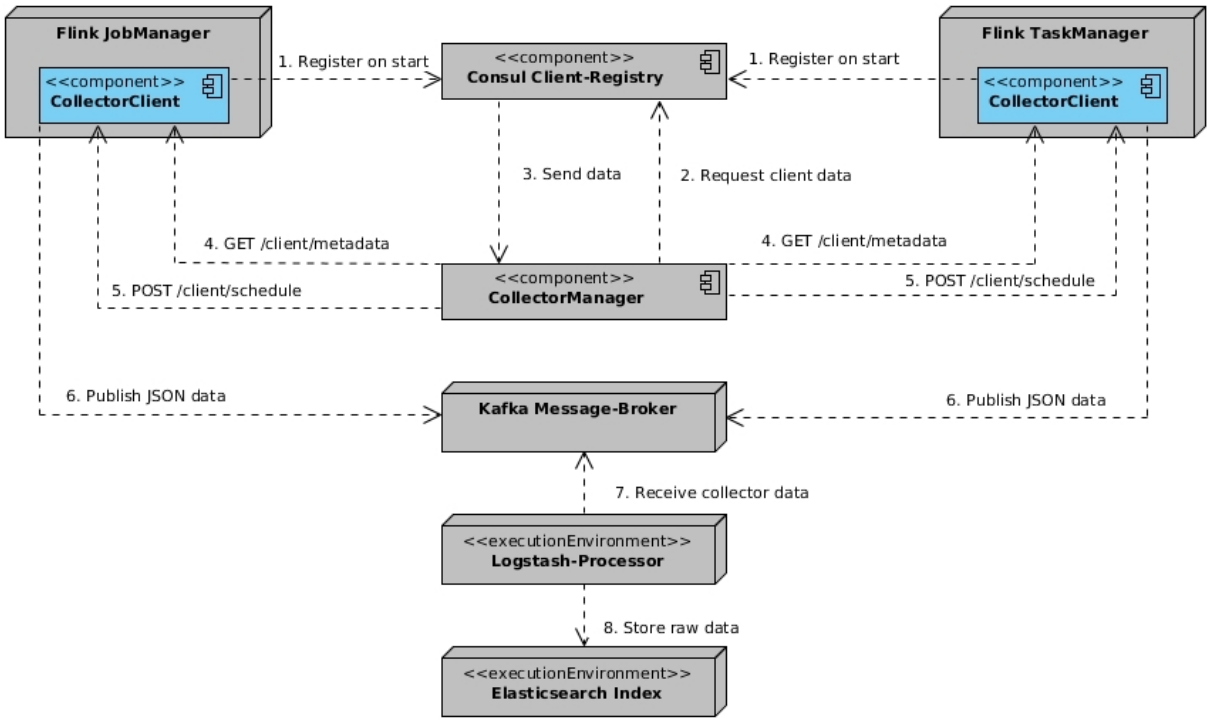

Figure 4.2: The "Collector-Platform"

The *CollectorClient*s installed on source systems, registers itself on application start with a central client registry, using an unique identifier `collector-client` **(1)**. This registry manages all registred clients and provides an interface that allows the *CollectorManager* to fetch network location (IP address and port) data for all clients that is required to address the individual clients. The *CollectorManager* uses this interface to display a list of all registred *CollectorClients* in a basic HTML user interface **(2,3)**. In addition, *CollectorManager* fetches additional meta information of the client **(4)**, like its host name and registered *Collector*s, provided by the implementation of the *CollectorClient*.

In the next step, the process of data collection can be triggered using the *CollectorManager*s user interface **(5)**. Therefore, the *CollectorManager* sends a HTTP `POST` request with an appropriate JSON request body and receives a status response that containing the status of the *CollectorClient* that can be `RUNNING` or `STOPPED`, according to the action that is triggered by the web user interface.

Now, the *CollectorClient* is active and collects data corresponding the *Collector* implementations registered in the clients internal registry, see **??**. Once all data of a single *Collector* implementation has been collected and transformed into a structured JSON format, the result will be enriched with a client timestamp and the clients identification given by the IP address and the port the client is availabe at, and published to the *Kafka Message-Broker* **(6)**.

The *Kafka Message-Broker* is the next step in the data pipeline and is the integration point for possible consumers of data received by distributed *CollectorClient*s. Due to the publish-subscribe principle Apache Kafka is based on, potential consumers just need to subscribe to the `collector-outbound-topic` for the data to become accessible.

The Logstash-Processor is the primary consumer of the data produced by *CollectorClient*s. It receives outgoing *CollectorClient* data by subscribing to the `collector-outbound-topic` topic, the clients publish the data into **(7)**. The Logstash-Processor performs two essential tasks: it creates the required indexes for storing the data in Elasticserach and converts incoming data into an internal message format that allows the data to be stored in Elasticsearch.

The Elasticsearch Index ist the last step in the *"Collector-Platform"* **(8)**. It is the data sink where all collected data will be stored at. The data is now available for querying or

visualization.

For a deeper understanding, the following sections introduce the components and their corresponding functions described above. Chosen frameworks and technologies will not be discussed in much detail, but it will be explained why each technology or approach has been chosen.

4.2 The "Collector" implementations

This set of classes builds the core for collecting data on source systems and takes the responsibility for fetching system data (**??**) with the Dstat system tool as well as application data for Apache Flink and Apache Kafka using the JMX interface (**??**) and via REST for Apache Flink only (**??**). As shown in **??**, all implementations are based on the *Collector* interface, what enables the handling of different realizations of data collection in a uniform way.

4.3 CollectorClient

The CollectorClient is main entry point for bringing data into the platform and belongs to the "Data" layer according to **??**. It is a simple Java application and will have to be installed on source systems. The module uses the concrete *Collector* implementations described above, according to the source system and depending on which *Collector*s are registered in the client. The client contains and internal registration to enable a mechanism to add or remove `Collector` implementations without the need to change the implementation of the *CollectorClient*.

For the scheduling purposes of the collection process, the client provides a REST resource that enables the start and stop of data collection from the HTML user interface provided by the *CollectorManager* component. This endpoint is available under the location `http://{client-host}:{client-port}/client/schedule` and expects HTTP `POST` requests with a request body containing JSON data with an appropriate *ScheduleRequest* for starting/stopping the data collection process.

Once started, the collection process that will be triggered by the client itself in a configurable interval, which is 5 seconds by default, the resulted data will be send to an Apache Kafka topic called `collector-outbound-topic`. The name of the topic explains the purpose of the topic, it acts as the data sink for *CollectorClient*s and separates the clients as the producing compontent from data consumers.

Besides the scheduling REST resource, the client contains an endpoint `http://{client-host}:{client-port}/client/metadata` that provides metatdata for the individual client and is required for displaying more detailed information about the client in the *CollectorManager*.

Corresponding to the `Collector` implementations registered in an internal registry, the client creates a continous stream of `Collector` data, a stream of "immutable facts" concerning the the system and application state of the underlying source system at a given point of time. Every data event created by the client contains the client's timestamp, the IP address and the port thus the origin of the data can identified. Appendix A contains the full JSON results for all available *Collector* implementations.

The client follows the approach to operate in a process that is separated to the JVM process the source system is running in. To enable the *CollectorClient*s to access to the management interfaces introduced in **??** on the remote JVM of Apache Flink and Apache Kafka, the source system must to be configured to allow the connection from a remote JMX client. **??** explains the adjustments that had been made on the example of Apache Flink. This in turn brings the disadvantage of potential security risks that occur by allowing remote connections on a given port.

An alternative would have been to use the capabilities of dynamic bytecode intrumentation provided by the core of Java. Because Apache Flink and Apache Kafka both are based on the JVM, it would have been possible to instrument the Flink or Kafka process itself, to perform the data collection in the same process. This approach was not pursued in this thesis, because the instrumentation causes a performance impact, that can be avoided by separation of collection in an individual process.

4.4 CollectorManager

Due to the fact that Apache Flink and Apache Kafka are composed of several interacting components and thus building a distributed system, the *"Collector-Platform"* builds a distributed system as well. Based on the requirement to provide a mechanism to schedule the collection process in distributed *CollectorClient*s, the *CollectorManager* represents the management component for the distributed system of clients by providing an overview of registered clients in the platform and the possibility to start/stop individual clients using an HTML based user interface.

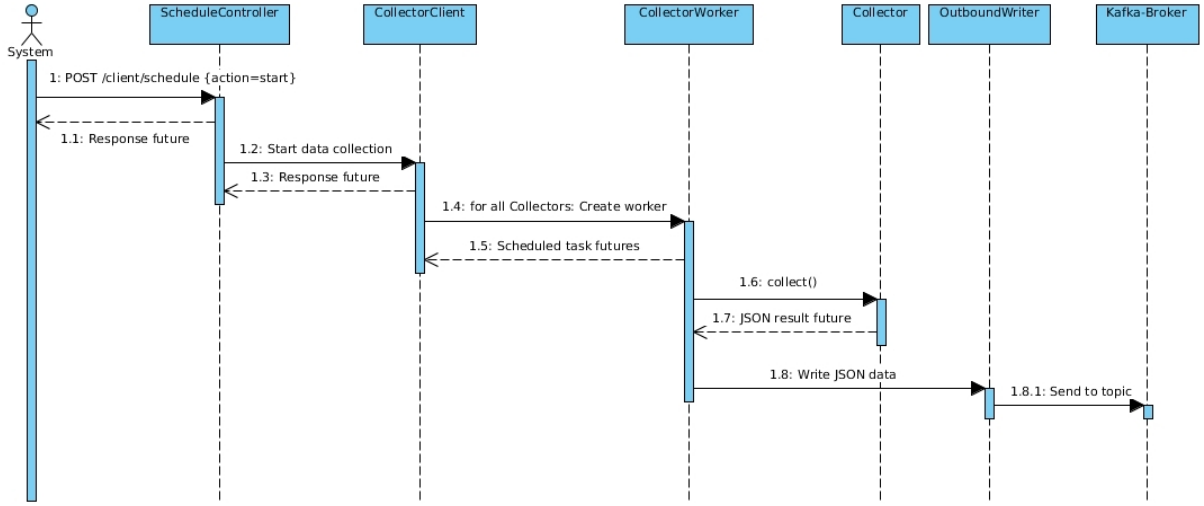

Figure 4.3: Sequence diagram 'Client scheduling'

The sequence diagram above presents the process of starting an individual *CollectorClient*. The client receives a HTTP `POST` request that contains an appropriate *ScheduleRequest* for starting the client. If the request is syntactically correct, the client starts data collection as discussed above.

4.5 Consul Client-Registry

The *Client-Registry* is a dedicated web service and acts as a central registry for all *CollectorClient* instances in the platform and makes the information about registered

clients available for the *CollectorManager*. For the realization of a public registry for distributed clients, the platform uses Consul, a service-registry, that enables the discovery of web services based on unique identifiers instead of host address and port information. Since the *CollectorClient* is a web service itself, clients register with the service name `collector-client` when the client application is started. The *CollectorManager* queries the Consul node for existing instances of services named `collector-client` which results in a list of registred client instances, that contains the required information like host and port for using the scheduling and metadata endpoints in the managers user interface.

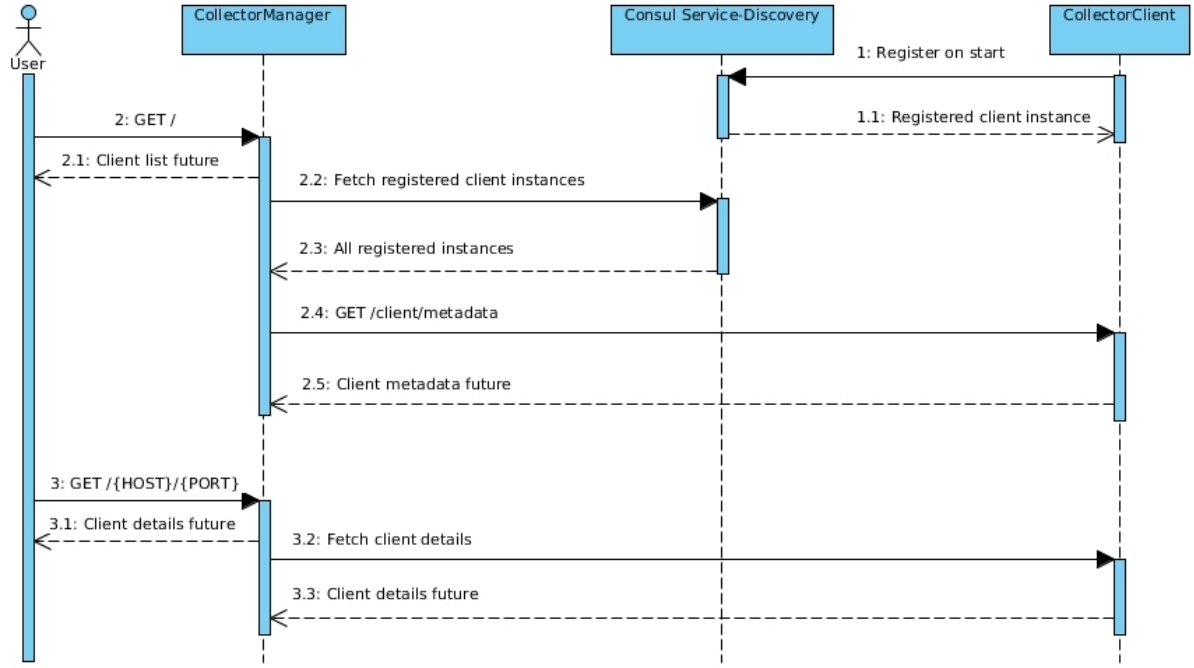

Figure 4.4: Sequence diagram 'Client discovery'

Using an central registry instance for discovering *CollectorClient* instances decouples the *CollectorManager* from the need to know detailed client information required for the localization of the service. Due to the fact that client addresses and ports are dynamically assigned and may change in distributed cloud environments, the service registry provides a scalable and flexible way to discover the providers of a given service name, `collector-client` in this case.

4.6 Kafka Message-Broker

The *Kafka Message-Broker* integrates distributed *CollectorClient*s to an overall system by providing a message queue for transporting data from clients to the *Logstash-Processor*. The *CollectorClient*s can publish its data to the queue and the other applications can asynchronously read it from the queue at any time. The usage of Apache Kafka as a publish-subscribe queuing service allows a separation of *CollectorClient*s as publishers and the *Logstash-Processor* and other potential as subcribers by buffering messages between stream producing and consuming systems.

4.7 Logstash-Processor

The *Logstash-Processor* receives the messages that arrive in the `collector-outbound-topic`. Although is is often thougth that Logstash is only capable of processing log files, it is more a data pipeline that enable the processing and transformation of log and other event data from a variety of systems. According to this thesis, it receives custom JSON data from clients and creates the indexes for storing the received JSON data in the *Elasticsearch Index*.

4.8 Elasticsearch Index

Elasticsearch is distributed, scalable, and highly available full-text search engine based on the Apache Lucene search index. It provides an HTTP web interface for managing, storing and quering schema-free JSON documents and is a widely used storage system of Big Data Analytics Applications. The search engine, the Logstash-Indexer and the Kibana visualization tool in combination are often refered to as "ELK" and provides a an end-to-end stack for central data collection and analysis as well as a graphical presentation of raw and analyzed data.

Elasticsearch allows the storage of collected data in its raw JSON format, what was a decisive criterion in the process of choosing an appropriate database for the *"Collector-Platform"*. Due to the fact that potential consumers of data produced by *CollectorClient*s

are unknown at the time of the thesis, the JSON format provides an universal structure that can be processed by any application.

4.9 Summary

This chapter introduced the *"Collector-Platform"* as data pipeline for collecting and storing system and application data on Apache Flink and Apache source systems. The pipeline starts on the *CollectorClient* instances, small web services that need to be installed on source systems, the data that will be collected depends on the `Collector` implementations registered in the client. Each `Collector` implementation produces its own custom result formatted in JSON that will be published to a *Kafka Message-Broker*. The broker provides a data queue that decouples data producers and consumers. Moreover using a queing system prevents the requirement of specialized operations for exchanging data and helps to decrease the performance impact on Apache Flink and Apache Kafka source systems.

The *Logstash-Processor* is a subcriber to the `collector-outbound-topic`, thus it receives collected data, creates required indexes and transforms the data to be stored in Elasticsearch.

To enable the scheduling of the data collection process, the *CollectorManager* provides an access point to start and stop *CollectorClients* using a REST resource available for each client. The REST paradigma, discussed in **??** provides an uniform interface for machine-to-machine communication and the exchange of data between participants in a distributed software system based on the existing HTTP infrastructure.

A centralized *Consul Client-Registry* as mechanism to discover *CollectorClient*s independently of concrete network locations has been discussed in the context of the platform.

Whereas this chapter just introduced an architectural overview of the components involved, the next **??** discusses details of the implementation and the technologies used for the realization of the *Collector-Platform*.

5 Implementation

The last **??** has shown an overview of the main components of the *"Collector-Platform"* and explained the interaction between them, this chapter discusses implementation details for all major components of the system architecture, including several collector implementations, as well as the "CollectorClient" and "CollectorManager" component.

The system architecture of the *"Collector-Platform"* consists two software components to be implemented, the Elasticsearch database, Apache Kafka message broker and the Logstash indexer to be configured to fulfill the requirements discussed in **??** and to realize the proposed system solution in **??**. The required configurations for Kafka, Elasticsearch and Logstash will be explained in **??** during the evaluation based on the prototype application stack .

The main component, the *CollectorClient* must provide a REST interface for starting/stopping the collection process on source systems. Furthermore, it must be possible to fetch metadata about each client. The *CollectorManager*, the second software component uses this interface for providing a basic web based UI that lists registered clients, shows detailed information and allows the scheduling the collection process separately for each client. It follows that two web applications are required, the client application must also be able to send data to a Apache Kafka, hence the usage of the Procucer API of Kafka must be supported.

The choice for implementing the web applications fell on Spring Boot in the current version 1.4.0. Taken from the reference documentation [**SpringB16**], "Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss". Features of Spring Boot include the ability to create standalone web applications conaining an embedded servlet container, what makes

the deployment of war files obsolete and as well as multiple integrations for different applications platforms including Apache Kafka in the subproject spring-kafka.

The following code shows a full example of an web application that provides a simple HTTP endpoint returning "Hello World". It creates an executable jar file containing an embedded Apache Tomcat servlet container and can be started from the command line, what means that there is no dedicated Tomcat instance required to deploy a war file to:

```
1  @Controller
2  @EnableAutoConfiguration
3  public class SampleController {
4
5      @RequestMapping("/")
6      @ResponseBody
7      String home() {
8          return "Hello World!";
9      }
10
11     public static void main(String[] args) throws Exception {
12         SpringApplication.run(SampleController.class, args);
13     }
14 }
```

Code listing 5.1: Spring Boot "Hello World"

The Spring framework provides usefull default configurations, thereby making it possible to create a simple REST-based web service with just a few annotations. The result is a completely self-contained executable jar, created with the Maven Buildmanagement tool. Executable jars (sometimes called “fat jars”) are archives containing the compiled classes along with all of the jar dependencies that the code needs to run. This produces the disadvantage that the memory requirement of the resulting executable increases. But this was ignored while making the decision for the framework because the presence of sufficient memory space on Apache Flink and Apache Kafka sources systems had been assumed.

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 expressions, the processing of collections as Streams as well as an Optional type for handling optional

values respectively null values, all features which were used often in the implementation of the *CollectorClient* and *CollectorManager* components. Java is the main programming language of Spring Boot, but also supports Groovy and Scala which did not come into consideration due to the authors lack of experience in these programming languages.

The software-solution uses Maven as Build- and Dependencymanagement tool, and is divided into the main modules:

- collectors
- collector-client
- collector-manager
- collector-data-processor

The following sections explains the content and and discusses implementation details for each of these modules separately with the exception of the "collector-data-processor" module that will be introduced in **??** to demonstrate the ease of data consumer integration into the *"Collector-Platform"*.

5.1 Collectors

The collector module contains the implementations for the required collectors based on **??**. It also defines a "collector-commons" module containing basic artefarcts of the collector domain, common classes and interfaces, that are used and required by the individual `Collector` implementations.

5.1.1 Base Domain

The next class diagramm shows the basic structure on the example of the class `JvmCollector` that will be introduced in **??**:

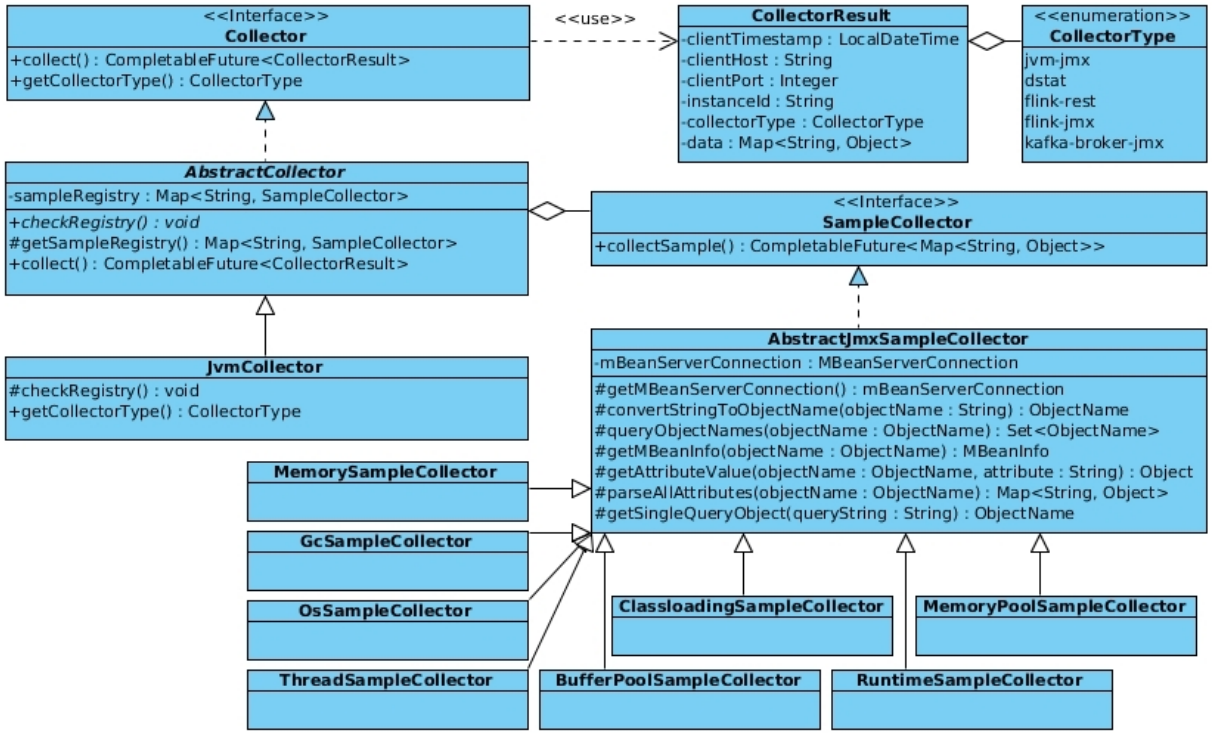

Figure 5.1: Class diagram 'JvmCollector'

This overview describes the collector domain, that is equal to all implementations and consists of following parts:

1. `CollectorType`: Enumeration, meta information, that distinguishes collectors implementations
2. `CollectorResult`: Data container, the result of a single invocation of the collect() method from the Collector interface. Describes a data event, an immutable fact" at a given time, containing the host and port the client is running on, the type of collector and the requested data.
3. `Collector`: The main interface for implementations, defines the protocol for data collection.
4. `AbstractCollector`: Abstract class, provides the implementation of the main collect() method.

5. `SampleCollector`: An interface for all classes that collect samples defined by the group the data belongs to.

On the example of `JvmCollector`, the following section explains the basic design principles that is common to the main collector domain more in detail wheras only relevant implementation details will be discussed in subsequent implementations, because the process of collecting data is similar in all.

5.1.2 JvmCollector

The implementation of the `Collector` interface for fetching "default" JVM data according to **??**. This basic set of different management interfaces build the foundation for the implementation for collecting JVM related data like memory, garbage collector or thread information.

To keep the main implementation as small as possible and following the "Separation Of Concerns" pronciple, the data was divided into different "sample groups", which resulted in one implementation each for collecting memory, garbage collector, thread data, et al.

A these `SampleCollector` implementations provide a `collectSample()` method, that fetches data from a different data group, here on the example of the class `MemorySampleCollector`.

```
1  private static final String OBJECT_NAME = "java.lang:type=Memory";
2
3  @Override
4  public CompletableFuture<Map<String, Object>> collectSample() {
5      return CompletableFuture.supplyAsync(() -> {
6          final Map<String, Object> memoryResultMap = Maps.newLinkedHashMap();
7          memoryResultMap.put(SAMPLE_KEY, ↩
               parseMemory(getMemoryMXBean(OBJECT_NAME)));
8          return memoryResultMap;
9      });
10 }
11
12 private Map<String, Object> parseMemory(final MemoryMXBean proxy) {
13     final Map<String, Object> memoryDataMap = Maps.newLinkedHashMap();
```

```
14      memoryDataMap.put(MEMORY_OPFC_KEY, ↩
            proxy.getObjectPendingFinalizationCount());
15      return memoryDataMap;
16  }
17
18  private MemoryMXBean getMemoryMXBean(final String objectName) {
19      try {
20          return ↩
              ManagementFactory.newPlatformMXBeanProxy(mBeanServerConnection(), ↩
              objectName, MemoryMXBean.class);
21      } catch (IOException ex) {
22          throw ex;
23      }
24  }
```

Code listing 5.2: MemorySampleCollector collectSample()

As one of the non-functional requirements in **??**, the client implementation may not cause a negative impact on source systems regarding system resources like cpu or disk usage. To make the code as "non-blocking" as possible, the implementation is based on the usage of the class `CompletableFuture`. It models an asynchronous computation and provides a reference to its results that will be available when the computation itself is completed. These computations like the JMX access in the example above, are potentially time-consuming [**Java8**]. The `CompletableFuture` allows the caller thread to return immediately to perform other operations instead of waiting for the result of the computation of JMX memory data, by delegating to a separate thread performing the operations defined in the `CompletableFuture`.

The code above wraps the computation of JMX memory data, which contains the data access via the MBeanServerConnection, introduced in **??**, and the preparation of result data in a separate thread performing the operations defined in the futures method body.

The implementations for the other `SampleCollectors` are almost the same, it only differs in the JMX management bean the data will be queried from.

Every `Collector` implementation is based on an internal registry of `SampleCollectors`, every implementation must provide its own registry required for the main `Collector`

process, that delegates to its registered SampleCollectors, aggregates their results and creates the overall result for the Collector. To split the collection into individual sample groups has the advantage, that it makes the implementation very flexible regarding the data to collect. To add further data sources, it will be enough to provide an implementation of the SampleCollector interface and to add an entry to the registry.

```
1  private static Map<String, SampleCollector> jvmSampleRegistry(final ↩
       MBeanServerConnection mBeanServerConnection) {
2      final Map<String, SampleCollector> registry = Maps.newHashMap();
3      registry.put(MemorySampleCollector.SAMPLE_KEY, new ↩
           MemorySampleCollector(mBeanServerConnection));
4      registry.put(ThreadSampleCollector.SAMPLE_KEY, new ↩
           ThreadSampleCollector(mBeanServerConnection));
5      ...
6      return registry;
7  }
8
9  @Override
10 public CollectorType getCollectorType() {
11     return CollectorType.JVM_JMX;
12 }
13
14 @Override
15 protected void checkRegistry() {
16     if (getSampleRegistry().isEmpty()) {
17         throw new JmxCollectorException();
18     }
19 }
```

Code listing 5.3: Sample registry for "JvmCollector"

The JvmCollector implementation only provides the sample registry, the type of collector as well as a simple check if SampleCollectors are registered at all. These methods are defined in the AbstractCollector class, discussed in the next section.

5.1.3 AbstractCollector

The abstract base class for all `Collector` implementations that contains the registry of `SampleCollectors` to be used in the main collection process shown below.

```
1 private final Map<String, SampleCollector> sampleRegistry;
```

Code listing 5.4: "AbstractCollector" sample registry

The collection process, defined by the `collect()` method in the `Collector` interface is implemented in this abstract class and therefore the same for all implementations. The only requirement for implementing classes is to provide realizations for the abstract methods in this class. The following steps are performed:

```
1 final List<CompletableFuture<Map<String, Object>>> sampleResultCPList = ↩
      getSampleRegistry().values()
2     .stream()
3     .map(SampleCollector::collectSample)
4     .collect(Collectors.toList());
```

Code listing 5.5: "AbstractCollector" Fetch sample futures

This demonstrates the Stream features available since Java 8, that enable the processing of collections in a more functional manner more focussed on the data transformations rather than the data itself. The implementation creates a Stream of registered `SampleCollectors`, collect the data for each sample and create a list of containing futures, regarding to to `SampleCollector` interface.

To merge the computations from multiple `SampleCollectors` and extract the the data from `CompletableFutures`:

```
1 final List<Map<String,Object>> sampleResults =
2     sampleResultCPList
3         .stream()
4         .map(CompletableFuture::join)
5         .collect(Collectors.toList()))
```

Code listing 5.6: "AbstractCollector" Merge and extract data

```
1 final Map<String, Object> dataMap = Maps.newLinkedHashMap();
2 sampleResults.forEach(dataMap::putAll);
3 final CollectorResult collectorResult = new ↩
      CollectorResult(getCollectorType().name().toLowerCase(),dataMap);
4 return collectorResult;
```

Code listing 5.7: "AbstractCollector" Create CollectorResult

At the end, the `CollectorResult` containing the type and the data will be generated.

The implementations discussed in coming sections are all based on the concept, to divide the collection into separate units and to create an overall result by aggregating individual sample data. As a result of this approach, the implementations keeps flexible what makes it easy to add or remove sample data just by providing implementations of the `SampleCollector` interface or removing an item from the internal registry. In addition, as shown above in the `JvmCollector` class in **??**, the collection of sample data and the preparation of the `CollectorResult` can be realized with just a few lines of code, which makes the integration of further data sources quite easy. This will be demonstrated in the coming section.

5.1.4 FlinkRestCollector

This collector implementation fetches data from the HTTP monitoring API that comes with Apache Flink and uses the endpoints introduced in **??**.

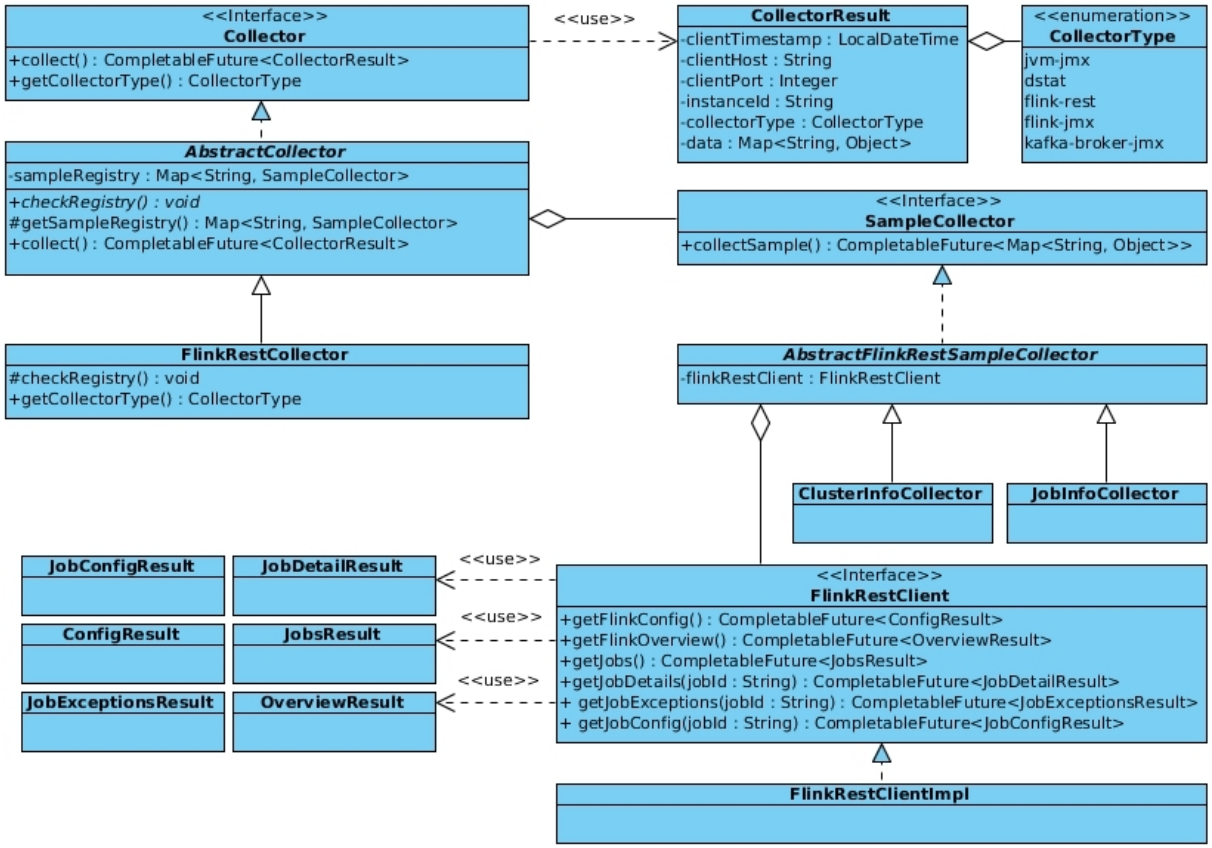

Figure 5.2: Class diagram 'FlinkRestCollector'

Following the sample approach described above, this `Collector` implementation provides its internal sample registry, containing the `SampleCollector` implementations based on the data that is desired and shall be collected. Because the monitoring API provides data concerning general server and cluster information as well as detailed information about jobs and their state, the registry contains two appropriate `SampleCollectors`:

```
1 private static Map<String, SampleCollector> flinkRestSampleRegistry(final ↩
      FlinkRestClient flinkRestClient) {
2     final Map<String, SampleCollector> registry = Maps.newHashMap();
3     registry.put(ClusterInfoCollector.SAMPLE_KEY, new ↩
          ClusterInfoCollector(flinkRestClient));
4     registry.put(JobInfoCollector.SAMPLE_KEY, new ↩
          JobInfoCollector(flinkRestClient));
5     return registry;
```

```
6 }
```

Code listing 5.8: "FlinkRestCollector" Sample registry

The implementation of `SampleCollector` just differs in the way of data access. While the `JvmCollector` uses an `MBeanServerConnection` to query data using the JMX management interface, `FlinkRestCollector` is based on a REST client internally to request data, which builds on the `RestTemplate` class, provided by the Spring Framework:

```
1  final CompletableFuture<OverviewResult> flinkOverviewFuture = ↩
       restClient().getFlinkOverview();
2  ...
3  final Map<String, Object> dataMap = Maps.newLinkedHashMap();
4  dataMap.put(FLINK_JOBS_RUNNING_KEY, flinkOverview.getJobsRunning());
5  dataMap.put(FLINK_FINISHED_KEY, flinkOverview.getJobsFinished());
6  dataMap.put(FLINK_CANCELLED_KEY, flinkOverview.getJobsCancelled());
7  dataMap.put(FLINK_FAILED_KEY, flinkOverview.getJobsFailed());
8  final Map<String, Object> resultMap = Maps.newLinkedHashMap();
9  resultMap.put(SAMPLE_KEY, dataMap);
10 return resultMap;
```

Code listing 5.9: "FlinkRestCollector" Rest client

By providing this sample data, this `Collector` creates a `CollectorResult` containing the data fetched from Apache Flinks monitoring API without the need to change the main collector process.

5.1.5 DStatCollector

To collect system related data defined in **??**, this implementation uses the Dstat system tool, that provides multiple parameters to specify the data to be displayed. According to the data analysis, following parameters will be used:

```
1 private static final String[] DSTAT_COMMAND = {"dstat", "-t",
2     "--cpu", "--top-cpu-adv", "--top-cputime", "--top-cputime-avg",
3     "--disk", "--disk-tps", "--disk-util",
4     "--net", "--socket", "--tcp", "--udp",
```

```
5     "--io", "--top-io-adv", "--lock", "--fs",
6     "--mem", "--top-mem", "--page", "--swap", "--vm",
7     "--sys", "--load", "--ipc", "--unix",
8     "--proc", "--proc-count", "--top-latency", "--top-latency-avg",
9     "--full",
10    "--float", "1", "0"};
```

Code listing 5.10: "DstatCollector" program parameters in

According to the given parameters, the `DstatCollector` class is based on the implementations for `CpuSampleCollector`, `DiskSampleColletor`, etc., which will have to be registrered using a sample registry, as discussed in the sections above.

Starting the Dstat process with the given parameters results in string containing three lines, where only the third line ist required to to gather data of. To start the external process and to retrieve the resulting output, a `ProcessBuilder` is used, which is part of the java.lang package.

```
1  final ProcessBuilder processBuilder = new ProcessBuilder(DSTAT_COMMAND);
2  processBuilder.redirectErrorStream(true);
3  final Process process = processBuilder.start();
4  try (BufferedReader processOutputReader =
5      new BufferedReader(new InputStreamReader(process.getInputStream()))) {
6          final String dstatResult = processOutputReader.lines()
7              .map(String::toString)
8              .collect(Collectors.joining(System.lineSeparator()));
9          final int exitCode = process.waitFor();
10 }
```

Code listing 5.11: ProcessBuilder 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:

```
1  final Pattern CPU_USAGE_PATTERN = Pattern.compile("" +
2      "(\\d+(\\.\\d+)?)(\\s*)" +
3      "(\\d+(\\.\\d+)?)(\\s*)" +
4      "(\\d+(\\.\\d+)?)(\\s*)" +
```

```
5       "(\\d+(\\.\\d+)?)(\\s*)" +
6       "(\\d+(\\.\\d+)?)(\\s*)" +
7       "(\\d+(\\.\\d+)?)")
8   ...
9   final Matcher matcher = CPU_USAGE_PATTERN.matcher(raw.trim());
10  final Map<String, Object> cpuUsageMap = Maps.newLinkedHashMap();
11  if (!matcher.matches()) {
12      LOG.warn("Unable to parse 'CpuUsage'");
13  } else {
14      try {
15          cpuUsageMap.put(CPU_NAME_KEY, cpuName);
16          cpuUsageMap.put(CPU_USAGE_USER_KEY, Float.valueOf(matcher.group(1)));
17          cpuUsageMap.put(CPU_USAGE_SYSTEM_KEY, ↩
                Float.valueOf(matcher.group(4)));
18          ...
19          } catch (NumberFormatException ex) {
20              LOG.warn("Unable to parse 'CpuUsage'");
21          }
22  }
```

Code listing 5.12: "CpuSampleCollector", Extract sample data

5.1.6 FlinkJmxCollector and KafkaBrokerJmxCollector

The collection of application data using JMX for Apache Flink and Apache Kafka is working in the same way as discussed in the previous sections. The implementations both provide `SampleCollector`s for collecting data from the managed resources which are listed for Apache Kafka in Appendix A by using a `MBeanServerConnection` and querying the resources by their JMX ObjectName, see **??**. Since Apache Flink is just providing a very basic set of application data in its current version, containing rudimentary JVM data like cpu load and basic information about current jobs and their state, this data will be collected nevertheless. It will be assumed to the metric system will be improved in upcoming versions of Apache Flink. Furthermore the data is redundant, because the

implementation of the `JvmCollector` and `FlinkRestCollector` provides a more detailed set of JVM and job data.

5.2 CollectorClient

After discussing the `Collector` implementations in the previous section, this section introduces the `CollectorClient`, the core software component representing the entry point for bringing data into the system. This is the component that needs be installed on Apache Flink and Apache Kafka source systems and has the following responsibilities:

1. Register itself on application start with client-discovery
2. Provide metadata for the `CollectorManager` component available via REST resource
3. Provide an interface to trigger data collection "on demand" from the `CollectorManager`s web UI
4. The main collection process, based on registered `collector` implementations
5. Transport data to the message broker

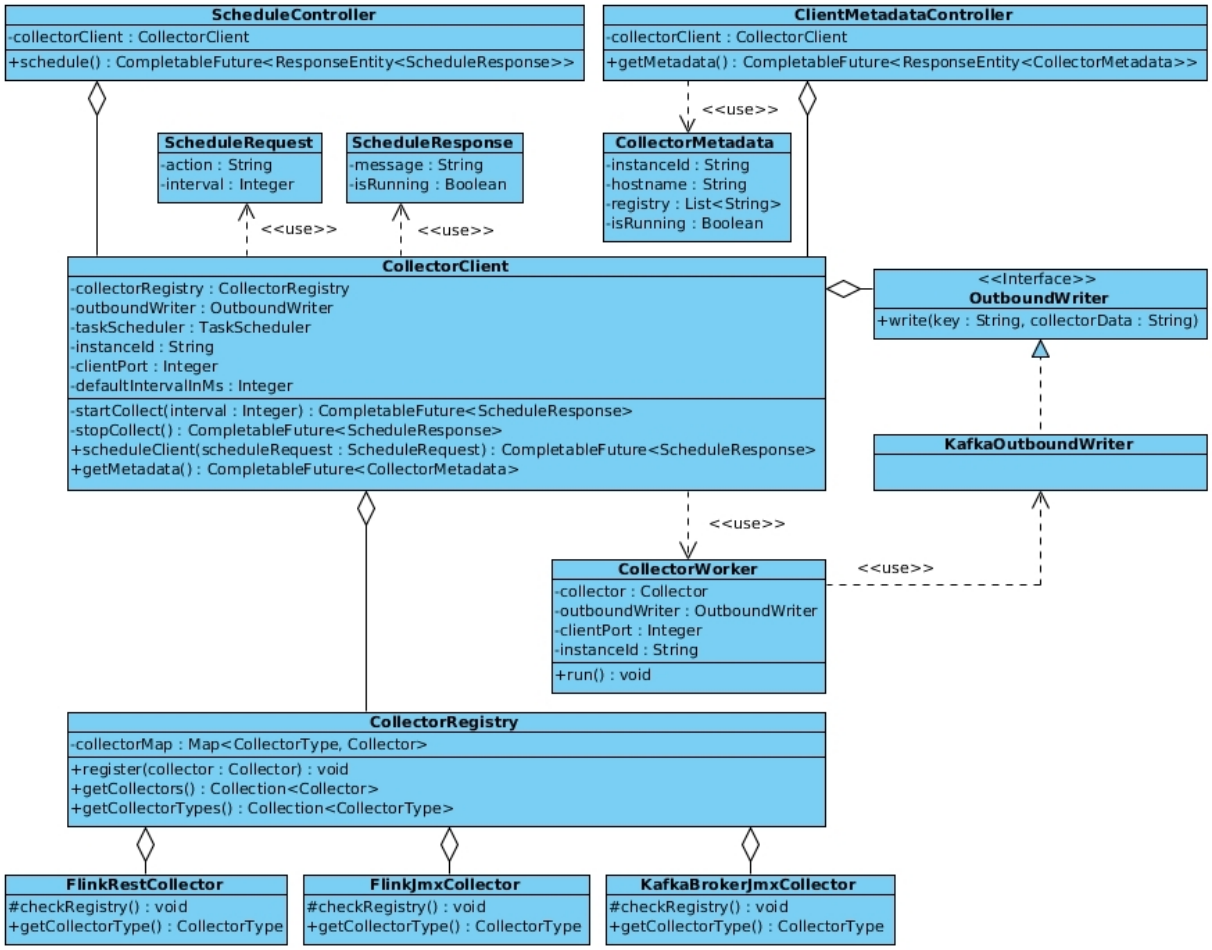

Figure 5.3: Class diagram 'CollectorClient'

The `CollectorClient` is a small web service, a self-containing jar-file with an embedded Jetty servlet container realized using Spring Boot. To register the client with the *Consul Client-Registry*, Spring provides an annotation-based approach to accomplish this functionality within spring-cloud sub-project, a set of tools for the development of cloud-based applications following common patters of distributed systems like configuration management, cluster state, service-discovery, etc. .

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class CollectorClientApp {
4
5     public static void main(String[] args) {
```

```
6         SpringApplication.run(CollectorClientApp.class, args);
7     }
8 }
```

Code listing 5.13: "CollectorClientApp", Client registration

The `@EnableDiscoveryClient` annotation seen above requires a Maven dependency to the `spring-cloud-starter-consul-all` artefarct, which provides and implementation to register the annotated Spring Boot application within the Consul service discovery. Based on usefull default configurations, this application registers itself the default Consul host `localhost` and port `8500` and is available to be discovered by the `CollectorManager` component.

The `CollectorClient` provides a Restful web service for fetching metadata for the respective client. This resource is required by the `CollectorManager` component for displaying detailed client data in the user interface.

```
1  @Async
2  @RequestMapping(value="/client/metadata", method = RequestMethod.GET, ↩
       produces = APPLICATION_JSON_VALUE)
3  public CompletableFuture<ResponseEntity<CollectorMetadata>> getMetadata() {
4      LOG.debug("Entering getMetadata()");
5      final CompletableFuture<ResponseEntity<CollectorMetadata>> responseCP ↩
           = collectorClient.getMetadata()
6          .thenApply(metadataMap -> {
7              LOG.debug("Fetched client metadata: {}", metadataMap);
8              return new ResponseEntity<>(metadataMap, HttpStatus.OK);
9          });
10         LOG.debug("Immediately return from getMetadata()");
11         return responseCP;
12     }
```

Code listing 5.14: "ClientMetadataController", Metadata REST endpoint

Both code snippets are a good examples for the "minimum of fuss" required to enable small web based services that registers itself with a service-discovery server. In addition, the last snippet enables a REST resource, localized by the API path `/client/metadata`, listening

for GET requests and delivering metadata of the client in JSON format. Furthermore, the capabilities of asynchronous computations is demonstrated, the @Async annotation in combination with the future implementation as the return value forces the framework to return immediately and not to block the caller thread, see ??. Internally, Spring Boot uses an ExecutorPool for the execution of asyncronous computations in a different thread.

In analogy to the metadata enpoint seen above, the client realizes a uniform REST interface that enables the scheduling of the data collection process. In this case, the servive expects POST requests, that contain a ScheduleRequest in their request body provided by the clients that use this service. This body must be JSON formatted and contains the action to trigger(start/stop data collection) and the time intervall between each collection process, five seconds for example. As result, a JSON response will be returned, containg a status of the schedule request and the status of the CollectorClient (running/stopped).

```
1  @RequestMapping(value = "/client/schedule", method = RequestMethod.POST, ↩
       consumes = APPLICATION_JSON_VALUE)
2  public CompletableFuture<ResponseEntity<ScheduleResponse>> ↩
       schedule(@RequestBody @Valid final ScheduleRequest request,
3      final BindingResult errors) {
4  LOG.debug("Received schedule request, action={}, interval={}", ↩
       request.getAction(), request.getInterval());
5      if (errors.hasErrors()) {
6          throw new CollectorClientException("Invalid schedule request");
7      }
8      return collectorClient.scheduleClient(request)
9          .thenApply(collectorScheduleResponse ->
10             new ResponseEntity<>(collectorScheduleResponse, HttpStatus.OK));
11 }
```

Code listing 5.15: "ScheduleController", REST endpoint

Whereas the REST services seen above just take HTTP requests, the CollectorClient contains the implementation for collecting the data based on the concept of a registry of Collectors. In contrast to the sample registry explained in the Collector implementations above, the registry holds the Collectorr implementations, and not any SampleCollector which are part of the Collectors. The decoupling of Collector im-

plementations from the `CollectorClient` by using a dynamic registry allows to add further `Collectors` and to remove existing ones without changing the `CollectorClient` implementation.

The scheduling of the client uses Spring's `TaskScheduler` interface that abstracts the scheduling of tasks based on different kinds of triggers, where a task is an implementation of the `Runnable` interface, defined in the Java core package `java.lang.thread`. Based on a Stream of registred `Collector` implementations, the client creates instances of the `CollectorWorker` class, that encapsulates the collection process for execution in a separate thread which will be provided to the task `TaskScheduler`.

```
1 private List<ScheduledFuture<?>> collectorTaskFutures = null;
2 ...
3 collectorTaskFutures = collectorRegistry.getCollectors()
4     .stream()
5     .map(collector -> new CollectorWorker(collector, clientPort, ↩
          outboundWriter, instanceId))
6     .map(collectorWorker -> ↩
          taskScheduler.scheduleWithFixedDelay(collectorWorker, interval))
7     .collect(Collectors.toList());
```

Code listing 5.16: "CollectorClient", collector registry

The `CollectorWorker` implements `Runnable` interface, hence it must provide an implementation of the parameter-less `run()`-method. Within its implementation, it fetches the `CompletableFuture` containing collected data, creates a `CollectorResult` and enriches it with client information, that makes the data set uniquely identifiable.

```
1 private final Collector collector;
2 private final OutboundWriter outboundWriter;
3 ...
4 @Override
5 public void run() {
6     LOG.debug("Collector worker starts...");
7     collector.collect()
8         .thenAccept(collectorResult -> {
9             try {
```

```
10                final CollectorResult result = new ↩
                      CollectorResult(collectorResult.getCollectorType(),
11                    collectorResult.getData(), now(), ↩
                          InetAddress.getLocalHost().getHostAddress(),
12                                clientPort, instanceId);
13                final String jsonData = JsonUtils.toJson(result);
14                outboundWriter.write(collector.getCollectorType().name().toLowerCase(), ↩
                      jsonData);
15            } catch (UnknownHostException ex) {
16                throw new CollectorClientException(ex.getMessage());
17            }
18    });
19    LOG.debug("Collector worker finished");
20 }
```

Code listing 5.17: "CollectorWorker", collector registry

After the result is created, it will be transfered into its JSON representation and written into a Apache Kafka topic, by using the `OutboundWriter` interface, whose implementation `KafkaOutboundWriter` uses the class `KafkaTemplate`, provided by the Spring sub-project spring-kafka.

```
1  private final KafkaTemplate<String, String> kafkaTemplate;
2  private final String kafkaOutboundTopic;
3  ...
4  @Override
5  public void write(final String key, final String jsonData) {
6      LOG.debug("Trying to send data to Kafka");
7      final ListenableFuture<SendResult<String, String>> sendResultFuture =
8          kafkaTemplate.send(kafkaOutboundTopic, key, jsonData);
9      sendResultFuture.addCallback(new ↩
          ListenableFutureCallback<SendResult<String, String>>() {
10         @Override
11         public void onSuccess(final SendResult<String, String> response) {
12                LOG.debug("Successfully send data to Kafka, topic={}, ↩
                      key={}", kafkaOutboundTopic, key);
13         }
```

```
14
15          @Override
16          public void onFailure(final Throwable ex) {
17              final OutboundWriterException exception = new ↩
                    OutboundWriterException(format("Error sending data to ↩
                    Kafka: %s", ex.getMessage()));
18              LOG.warn(exception.getMessage());
19              throw exception;
20          }
21      });
22  }
```

Code listing 5.18: "KafkaOutboundWriter", Send data

At this point, the collection process is finished. After the collection interval elapsed, which is configured to be five seconds as default value, everything will be triggerd again by the `TaskScheduler` described above.

5.3 CollectorManager

The `CollectorManager` is a small web appliction representing the management component in the *"Collector-Platform"*, it realizes a basic HTML based user interface and provides an overview of all clients registred in the platform as well as a detailed view of individual clients. In addition, the data collection process can be triggered by using this interface.

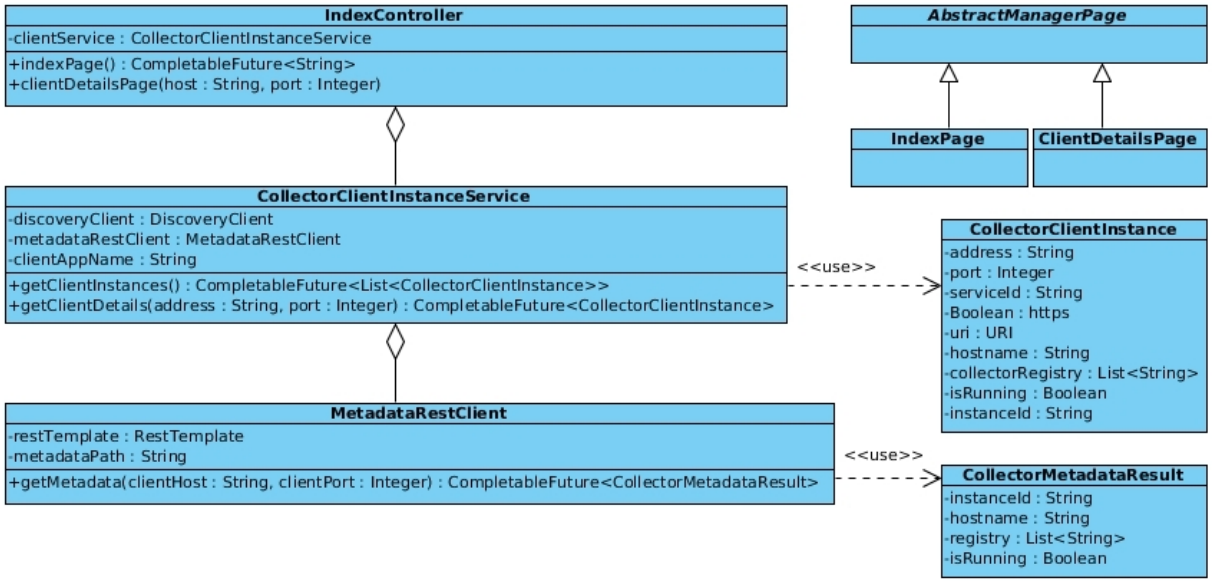

Figure 5.4: Class diagram 'CollectorManager'

To retrieve the required information of which clients exist and are registered in the system platform, the `CollectorManager` uses the `DiscoveryClient` implementation provided by spring-cloud.

```
1  private final DiscoveryClient discoveryClient;
2  private MetadataRestClient metadataRestClient;
3  ...
4  final List<CollectorClientInstance> clients = ↩
       discoveryClient.getInstances(clientAppName)
5      .stream()
6      .map(serviceInstance ->
7          CollectorClientInstance.of(serviceInstance.getHost(), ↩
               serviceInstance.getPort(), serviceInstance.getServiceId(),
8          serviceInstance.isSecure(), ↩
               serviceInstance.getUri())).collect(Collectors.toList());
9          LOG.debug("Fetched all client instances: {}", clients);
10         return clients;
```

Code listing 5.19: "CollectorClientInstanceService", Get client instances

The `DiscoveryClient` provides access to the Consul Client-Registry and makes a list of

all registered `CollectorClient`s available that will be displayed in the user interface. In addition, it sends a REST request to the `CollectorClient` itself to enrich existing Consul data with metadata provided by client as seen in the `CollectorClient` implementation above.

5.4 Summary

The last chapter discussed implementation details for existing data collectors available for Apache Flink and and Apache Kafka. It introduced the concept of a "sample registry", that divides the main data collection into separate units of work, represented by implementations of the `SampleCollector` interface. The "main" collectors, means classes that implement the `Collector` interface, aggregate an overall result based on individual data from multiple `SampleCollector`s.

The `CollectorClient` is based on implementations of the `Collector` interface, which are also organized by an internal registry. It represents a small REST-based web service that manages the collection process based on registered `Collector`s and provides a REST interface to enable the collection of data "on-demand".

The `CollectorManager` acts a client for the web services provided by the `CollectorClient`. It uses the REST resources discussed for listing registered clients, showing detailed client information and starting and stopping the collection process in a basic HTML-based user interface.

Wheras the `Collector` implementations do not depend on Spring Boot, the framework is used to realize the `CollectorClient` and `CollectorManager` applications. It enables the development of standalone, self-contained applications with less efford and makes the integration of 3rd party applications or frameworks, in this example Apache Kafka and Consul, very easy.

The next chapter introduces the local test environment and explains the setup of the prototype application to enable the evaluation of the software solution according to the requirements defined in **??**.

6 Test and Evaluation

The last chapter discussed implementation details for the *"Collector-Platform"* introduced in **??**. For its realization, Java 8 as programming language head been chosen in combination with the Spring Boot framework which allowed the rapid implementation of self-containing, distributed applications.

In this section, the focus will be on testing and evaluating the proposed system solution to ensure compliance with the criteria defined in **??** and **??** and explains the setup of a *"Collector-Platform" Cluster* for the manual testing of the system architecture define in **??**.

6.1 Automated Tests

A significant approach for testing software products are automated tests that will be executed in the building process. In Java, unit tests are used for this purpose. The *Collector-Platform* uses Maven as Build-Managemement solution, means the provided test implementations in form of JUnit test classes will be triggered automatically and the sucessfull passing all existing test cases is a requirement for a successfull building process.

The test classes are divided into pure unit tests for testing the functionality of a separate class without its dependencies. Unit tests describe the testing of program components in isolation from other related contributing program components whereas integration tests refer to the overall system and often require infrastructure components like databases to be started before the functionality can be tested.

Based on the requirement collect data from different data sources, it was required to provide running instances of Apache Flink and Apache Kafka for testing the implementations of the respective collectors, it means most of the test in the *"Collector-Platform"* are represented by integration tests.

The Maven Failsafe-Plugin procvides a mechanism to distinguish unit and integration tests by a naming convention that unit tests have the ending "Test" and "IT" for integration test what allows the integration tests to be excluded from the Maven `test` phase. The integration tests can by triggered by using the `mvn verify` command, after the required infrastructure components had been provided, what will be explained in the next section.

The build process of the *"Collector-Platform"* was supported by the public build server provided by `https://travis-ci.org`. It allows the automated software-build in form of build jobs and provides a basic build pipeline. Described by pushing to the public code repository on GitHub and triggering the build job automatically via webhook, the pipeline provided direct feedback of the code that has been implemented just now.

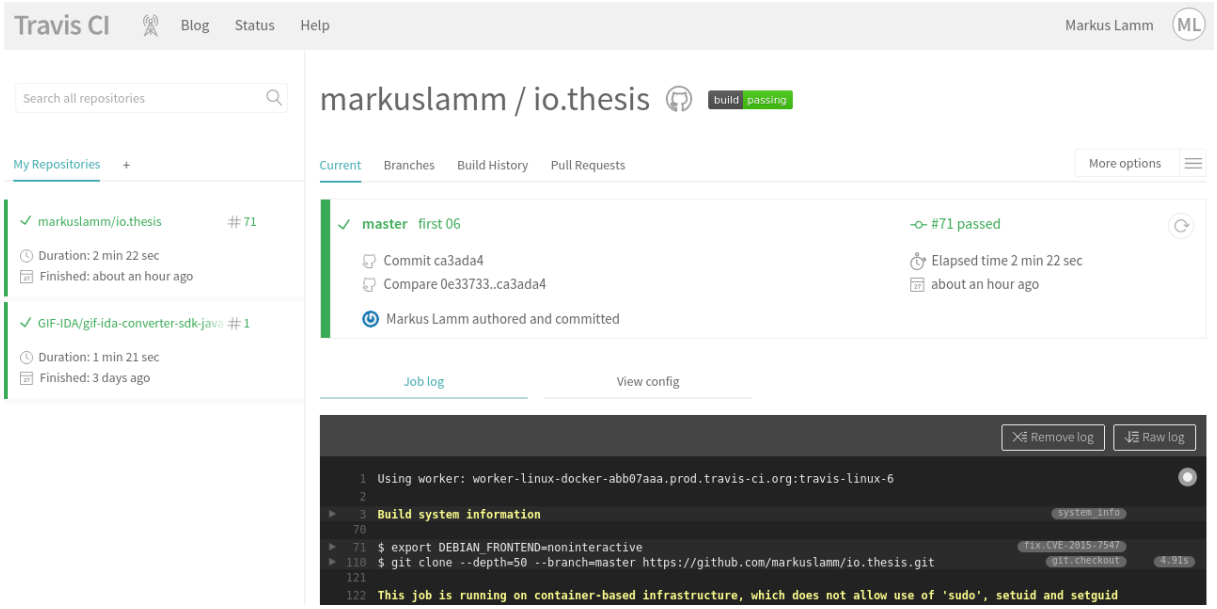

Figure 6.1: Travis CI

6.2 "Collector-Platform" Cluster

According to the proposed architecture in **??**, *"Collector-Platform" Cluster* consists of the mandatory infrastructure components *Kafka Message-Broker*, *Consul Client-Registry*, *Logstash-Processor* and *Elasticserach* to fulfill the requirements discussed in **??**. In addition one instance of Apache Flink's Job- and TaskManager each will be required, to create a basic Flink cluster acting as data source for the collecting client, as well as the *Collector-Manager* componenent for managing the distributed *CollectorClient* instances.

One main challenge of the technical realization of the system solution was to create a cluster of application nodes operating in an own network distinguishable by their IP addresses. The *"Collector-Platform"* uses Docker to achieve this goal. Docker provides a "software containerization platform" that allows to wrap applications in a complete filesystem that contains everything needed to run in form of a container. This container provides an isolated platform for applications to run on Docker host systems and contains an unix operation system and all required software components reuired by the application. Docker uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces and allow independent containers to run within a single host instance, avoiding the overhead of starting and maintaining virtual machines.

The applications to be "containerized" will be defined in by a Dockerfile that defines the intructions to build the application container. The Docker file contains the the behavior of the application when it is started by the `docker-engine`. The following example show the definition for the *CollectorManager* component:

```
1 FROM anapsix/alpine-java
2 MAINTAINER markus.lamm@googlemail.com
3 ADD collector-manager-app/target/collector-manager-app.jar ↩
     /apps/collector-manager-app.jar
4 ENTRYPOINT ["java","-jar","/apps/collector-manager-app.jar"]
```

Code listing 6.1: Dockerfile "CollectorManager"

The resulting container is based on the Docker image `anapsix/alpine-java` which provides an unix operating system and a Java installation as a base system, the *Collector-Manager* will be installed on. In line 3, the `collector-manager-app.jar` will be copied

from the local file system to a virtual path inside the Docker container. The last command in line 4 defines the entry point for the `docker-engine`, this commmand will be executed on container startup, means the *CollectorManager* jar file will be executed by the JVM provided by the `anapsix/alpine-java` base image.

Based on the required components discussed above the application needs the following Docker containers:

- One Apache Kafka container as message broker and source system for data collection. The Kafka container depends on another container for Apache Zookeper as a centralized service for configuration and distributed synchronization and is mandatory for the Kafka container to work.
- One Consul container as client discovery service
- Two Apache Flink containers, one Job- and TaskManager as source systems
- One container conatining Logstash, Elasticserach and Kibana
- One container for the *CollectorManager* component

For the orchestration of the container infrastructure, Docker provides a file format that allows the definion of required containers and their dependencies in a custom `docker-compose.yml` file. The next section covers main configuration details for the containers of the *"Collector-Platform"* cluster.

6.2.1 Configuration

The Apache Kafka is a core component of the platform and operates as a buffer between the data producing *CollectorClient*s. The following excerpt from the `docker-compose.yml` file defines the container for Kafka:

```
1  kafka:
2    container_name: kafka
3    image: wurstmeister/kafka
4    ports:
5      - "9092:9092"
```

```
6       - "9997:9999" # jmx
7       - "9097:9091" # collector-client
8     environment:
9       KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
10      KAFKA_ADVERTISED_HOST_NAME: 192.168.2.100
11      KAFKA_CREATE_TOPICS: ↩
            "collector-outbound-topic:1:1,flink-outbound-topic:1:1"
12      JMX_PORT: 9999
13      SPRING_PROFILES_ACTIVE: kafka-broker-jmx
14      SPRING_CLOUD_CONSUL_HOST: consul
15      SPRING_CLOUD_CONSUL_PORT: 8500
16      CLIENT_PORT: 9091
17      KAFKA_BROKER_ADDRESS: kafka:9092
18    volumes:
19      - /var/run/docker.sock:/var/run/docker.sock
20    depends_on:
21      - zookeeper
22      - consul
```

Code listing 6.2: Container definition "kafka"

6.3 Docker Deployment

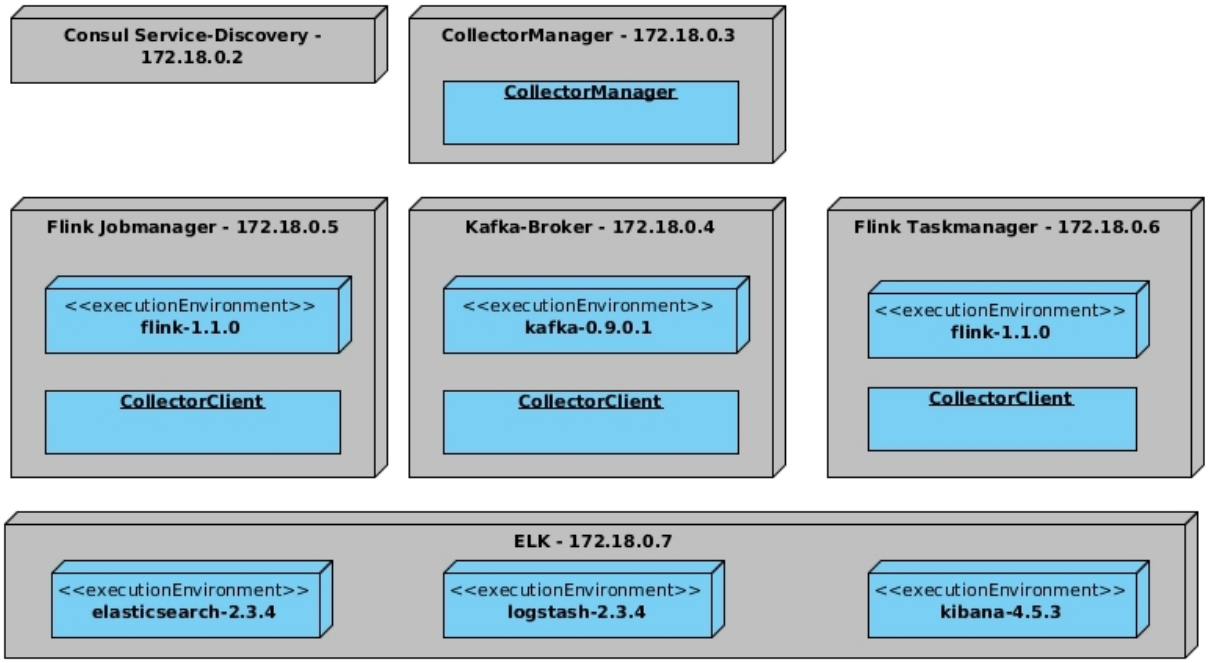

Figure 6.2: Docker Deployment diagram

Build images

6.4 Discussion

6.5 Summary

TODO

7 Conclusion

TODO

7.1 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 and dstat process

More performance with more "system" languages, go? c?

dynamic sample collectors

List of Figures

List of Tables

List of Source Codes

A

A.1 Collectors JSON results

A.1.1 JvmCollector

A.1.2 DstatCollector

A.1.3 FlinkRestCollector

A.1.4 FlinkJmxCollector

A.1.5 KafkaBrokerJmxCollector

A.2 Apache Flink 1.1.0 API JSON responses

```
GET /jobs
```

A.3 Apache Kafka 0.9.0.1 MBeans

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