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

Bachelor's Thesis

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

at

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

1. Examiner and Supervisor:	Prof. Dr. Stefan Edlich
2. Examiner:	Prof. Dr. Elmar Böhler

Submitted by:	Markus Lamm
Matriculation number:	s786694
Date of submission:	06.09.2016

Ackknowledgements

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

Abstract

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

Contents

1 Introduction

1.1 Motivation

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

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

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

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

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

1.2 Objective

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

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

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

1.3 Structure of thesis

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

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

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

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

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

2 Basic Concepts

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

2.1 Big Data

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

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

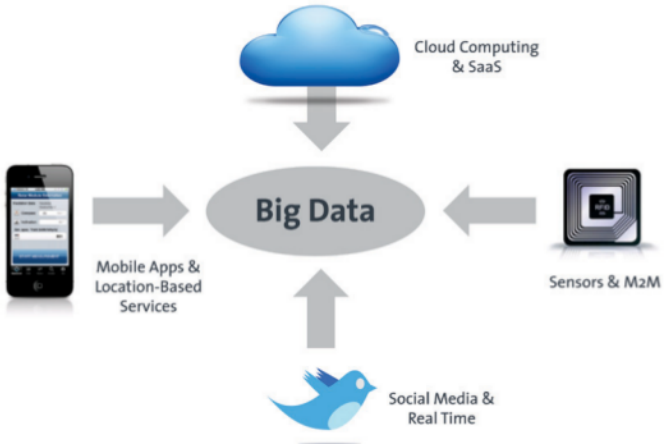

Figure 2.1: Sources of Big Data[Jör14b]

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

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

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

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

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

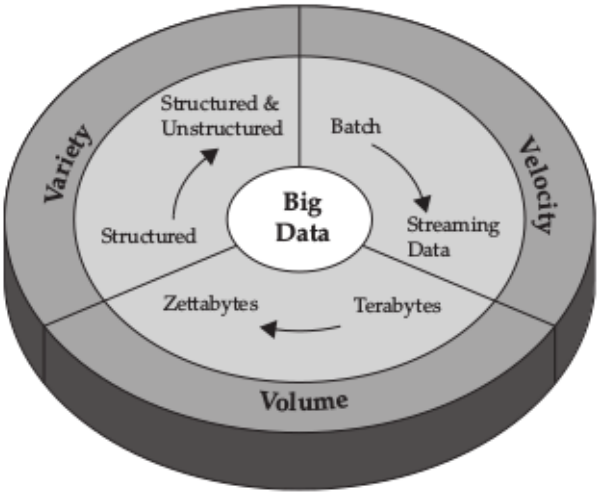

Figure 2.2: The three 'V's of Big Data[Pau12]

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

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

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

2.2 Big Data Analytics Applications

Big Data Analytics describes the process of collecting, organizing and analyzing large volumes of data with the aim to discover patterns, relationships and other useful information extracted from incoming data streams [Nat15]. The process of analytics is typically

performed using specialized software tools and applications for predictive analytics, data mining, text mining, forecasting and data optimization.

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

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

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

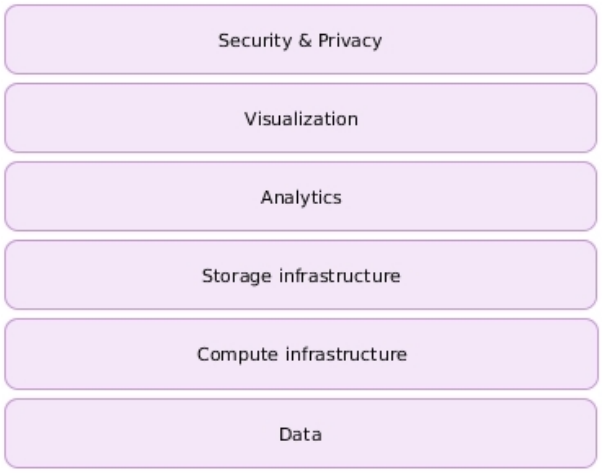

Figure 2.3: Taxonomy of Big Data Analytics Applications [Gro14]

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

2.3 Stream Processing

Computing infrastructures on big data currently differ on whether the processing on streaming data will be computed in batch mode, or in real-time/near real-time. This

section is focussed on processing continous data streams in real-time/near real-time and introduces Apache Flink and Apache Kafka as representants of streaming frameworks.

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

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

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

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

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

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

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

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

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

2.3.1 Apache Flink

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

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

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

```
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 snippet 2.1: Basic Apache Flink streaming application

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

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

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

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

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

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

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

2.3.2 Apache Kafka

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

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

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

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

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

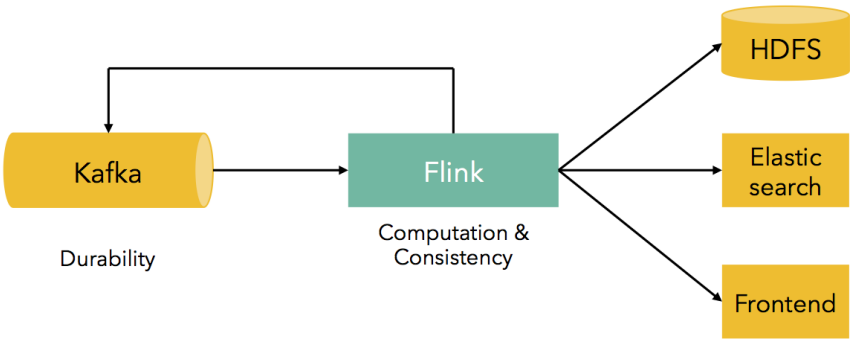

Figure 2.4: A typical Kafka-Flink pipeline[Rob]

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

2.4 Representational State Transfer (REST)

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

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

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

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

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

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

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

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

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

2.5 Java Management Extensions (JMX)

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

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

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

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

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

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

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

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

2.6 Summary

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

3 Requirements Analysis and Specification

After a short introduction to the basic terms and Apache Flink and Apache Kafka in context of stream processing, this chapter examines what different kind of data are available for both of the systems. According to the results of the data analysis, the functional and non-functional requirements of the software system which is forming the core of the present thesis will be defined.

3.1 Data Analysis

In preparation of this thesis my supervisor Prof. Dr. Stefan Edlich once said *"Sie sammeln alles, was nicht bei drei auf dem Baum ist"*. According to this statement, 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 that is availbale for Apache Flink and Apache Kafka.

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

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

3.1.1 System data

Observation of cpu-, disk- and memory-utilization, why [Höb00]. Dstat system util introduction

To monitor the performance, such as processor, memory, network, and system utilization. The servers should always in its best performance and should be available when needed. To monitor all of those components in Linux system, we already know iostat (monitor system input output), vmstat (monitor memory utilization) and ifstat (monitor network usage). What if you can have 1 tool which has those 3 functions above and even more? We can do it using dstat tool.

3.1.2 Application data

Apache Flink provides application data via Monitoring REST API, describe REST Since version 1.1.0 new Metrics data via JMX Analyze Flinks REST data

KORRELATION sytem and application data

3.2 Data Quality

Define DQ, evaluate quality for data above

3.3 Functional Requirements

The main objective of this bachelor thesis is to implement a software solution to collect .... "COLLECT EVERYTHING! live and historical sources

Describe "big picture" functionality see [Les14], follows distributed character of Big Data Analytics Applications, provide "on demand" data collection, as much data as possible, realtime?, three main components, break down for:

3.3.1 Collection

collect data in clustered environments

3.3.2 Transport

Scalability with message broker

3.3.3 Persistence

Accessibility for AI, UI applications

3.4 Non-Functional Requirements

Performance, scalability,

3.5 Summary

4 Architecture and Implementation

Distributed system -> distributed collection, cloud environments, microservice architecture, service-discovery, communication via REST, Publish(client) -> Topic <- Subscribe Logstash,

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

4.1 Architecture

TODO

4.1.1 Technologies

4.1.2 Infrastructure Components

make refs to taxonomy

Service-Discovery

Registraction for CollectorClients

Message-Broker

Queueing, see Marz15

compute layer

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

Indexer

Receive messages from Kafka, route data, create ES index, why, describe context BDAA

compute layer

Persistence

ES as search index for time-series based data, easy vizualization with Kibana, why?

storage layer

4.1.3 Software Components

CollectorClient

The CollectorClient tier is our entry point for bringing data into the system... A module to gather the event streams from data sources.

data layer

CollectorManager

Gives overview, uses Consul as service-discovery

CollectorDataProcessor

module to analyze the streams creating derived streams and persist flat data -> data transformation

analytics layer

4.2 Implementation

Introduce software stack, why used?

4.3 The "collect"-algorithm

Java8, CPs, non-blocking streams

4.4 Summary

Maybe Spring alternatives, Lagom, VertX, Play? Maybe collector as agent, Instrumentation instead of microservice, alternatives REST, maybe (Web-)Sockets possible secururity risk because remote JMX, firewalls

5 Evaluation

5.1 Local test environment

5.2 Docker environment

5.3 Observations

5.4 Discussion

5.5 Summary

6 Conclusion

TODO

6.1 Summary

6.2 Outlook

List of Figures

List of Tables

List of Source Codes

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Image resources

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

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

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

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

A

A.1 Diagrams

A.1.1 Use Case diagram

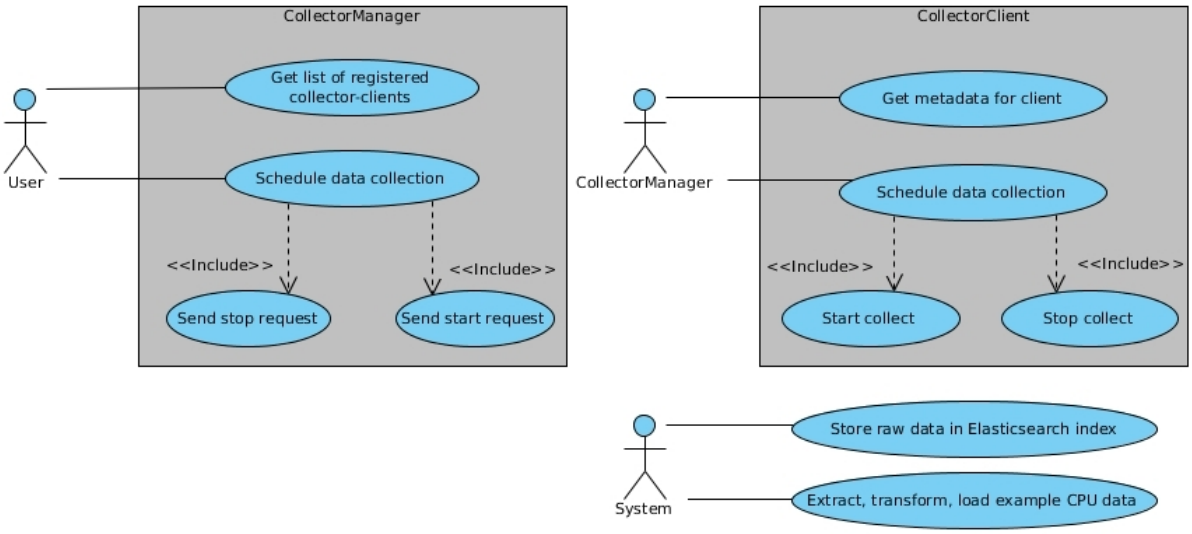

Figure A.1: Use Case Diagramm

A.1.2 Class diagrams

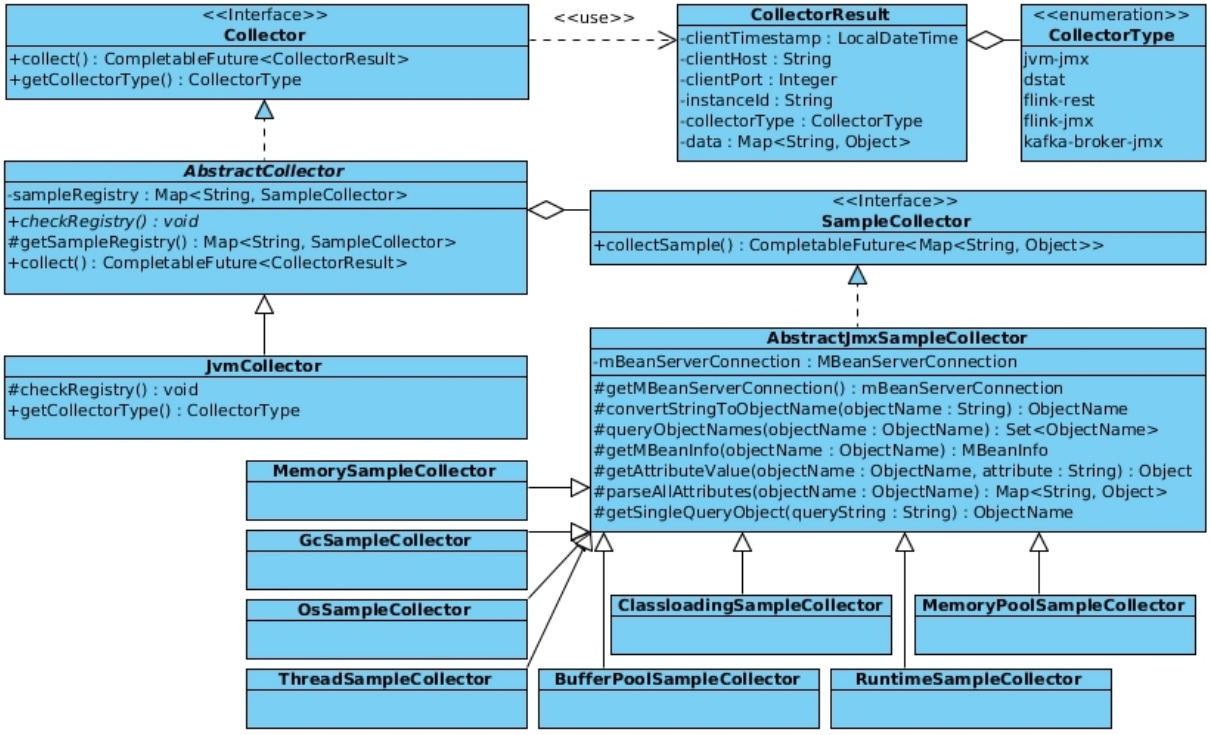

Figure A.2: Class diagram 'JvmCollector'

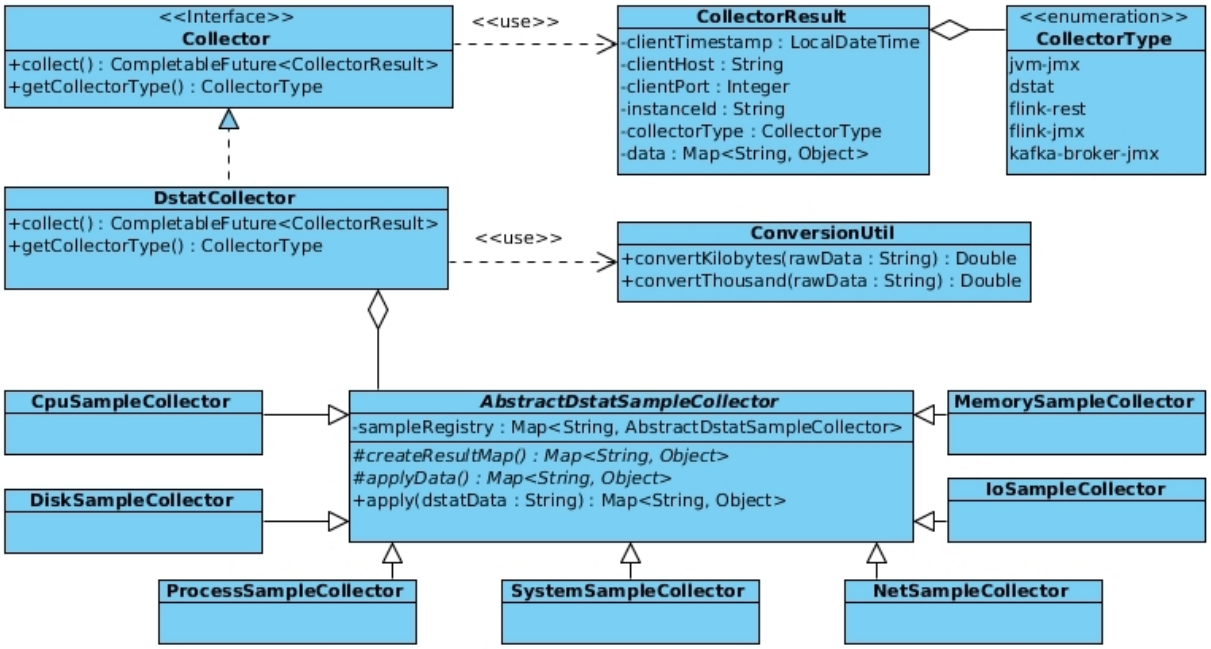

Figure A.3: Class diagram 'DStatCollector'

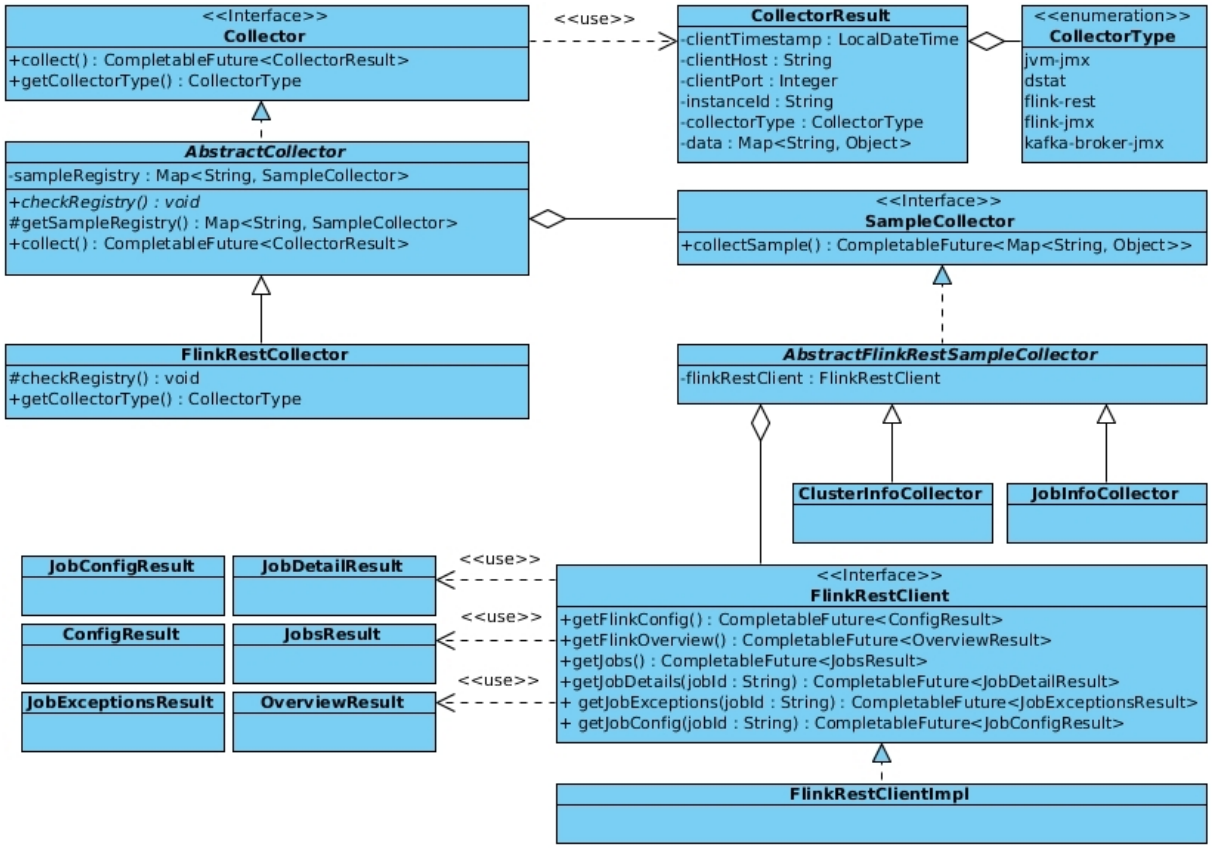

Figure A.4: Class diagram 'FlinkRestCollector'

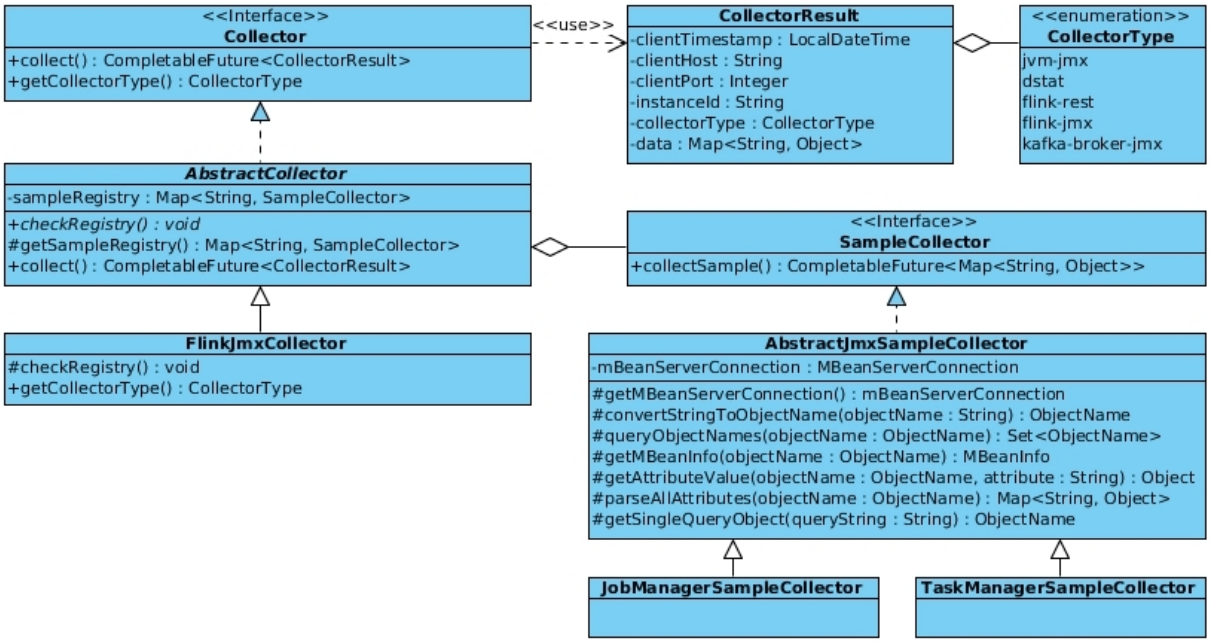

Figure A.5: Class diagram 'FlinkJmxCollector'

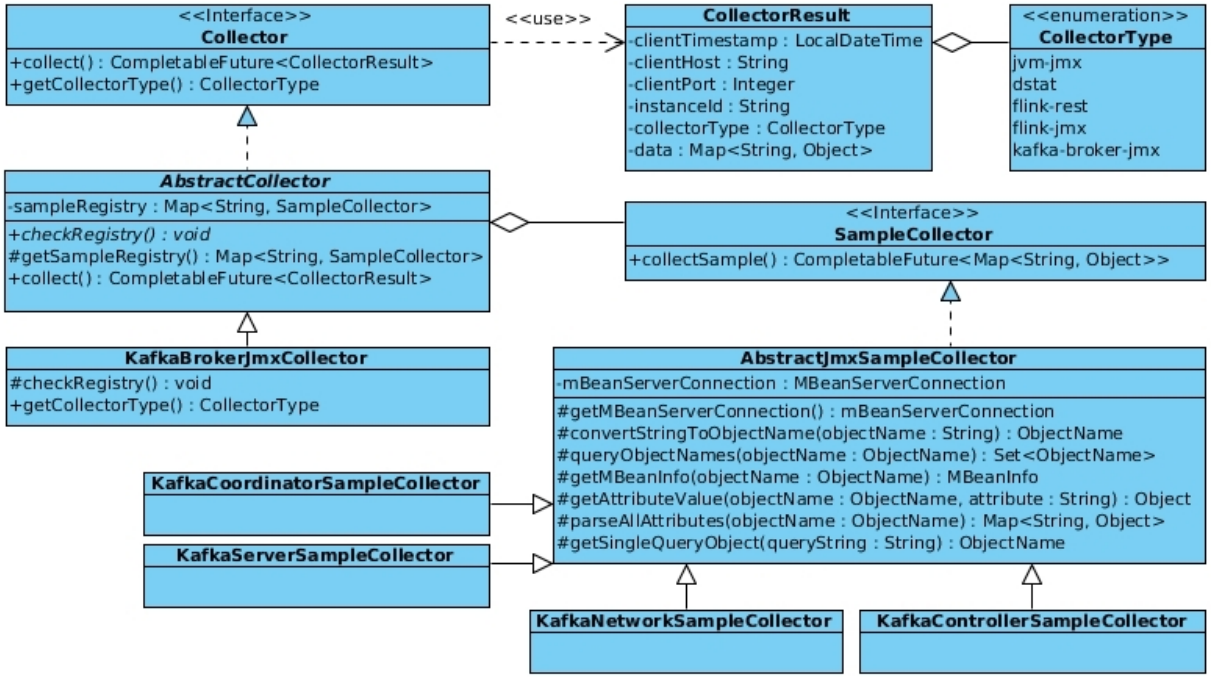

Figure A.6: Class diagram 'KafkaBrokerJmxCollector'

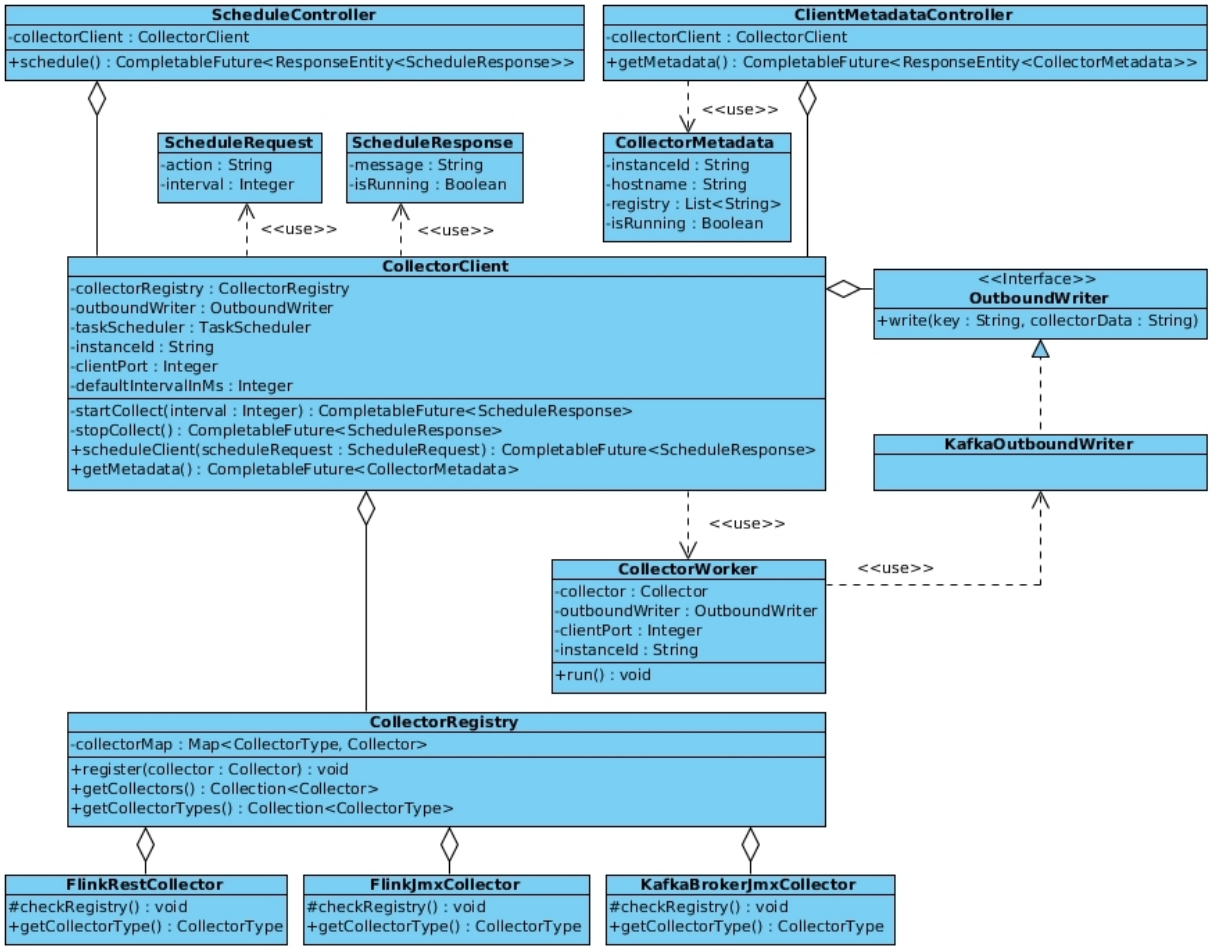

Figure A.7: Class diagram 'CollectorClient'

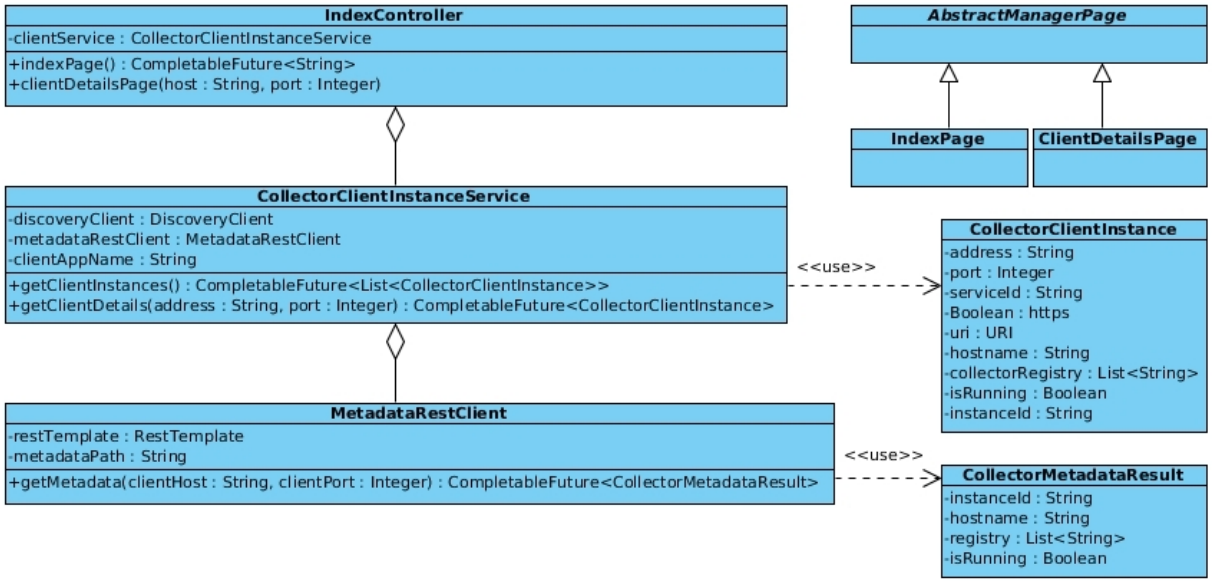

Figure A.8: Class diagram 'CollectorManager'

A.1.3 Sequence diagrams

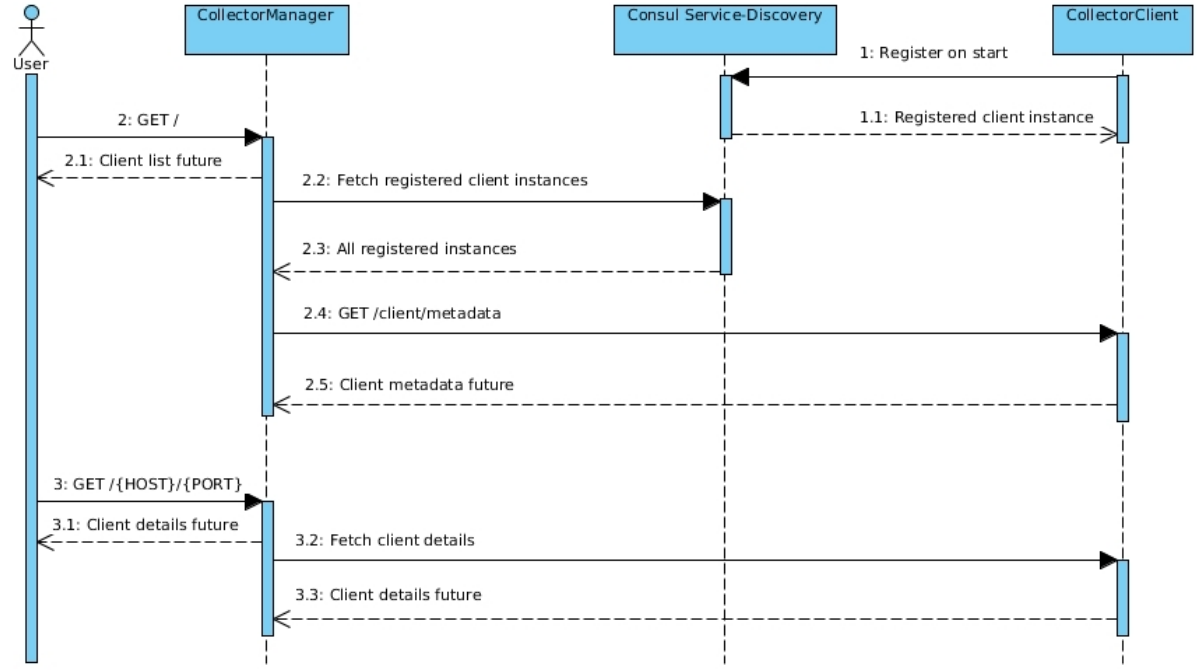

Figure A.9: Sequence diagram 'Client discovery'

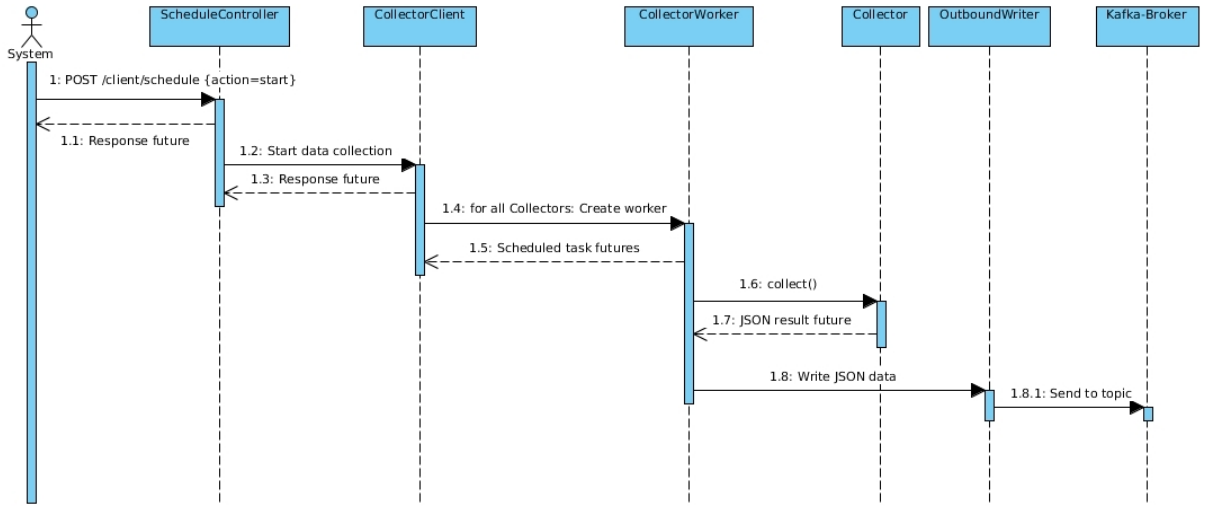

Figure A.10: Sequence diagram 'Client scheduling'

A.1.4 Component diagram

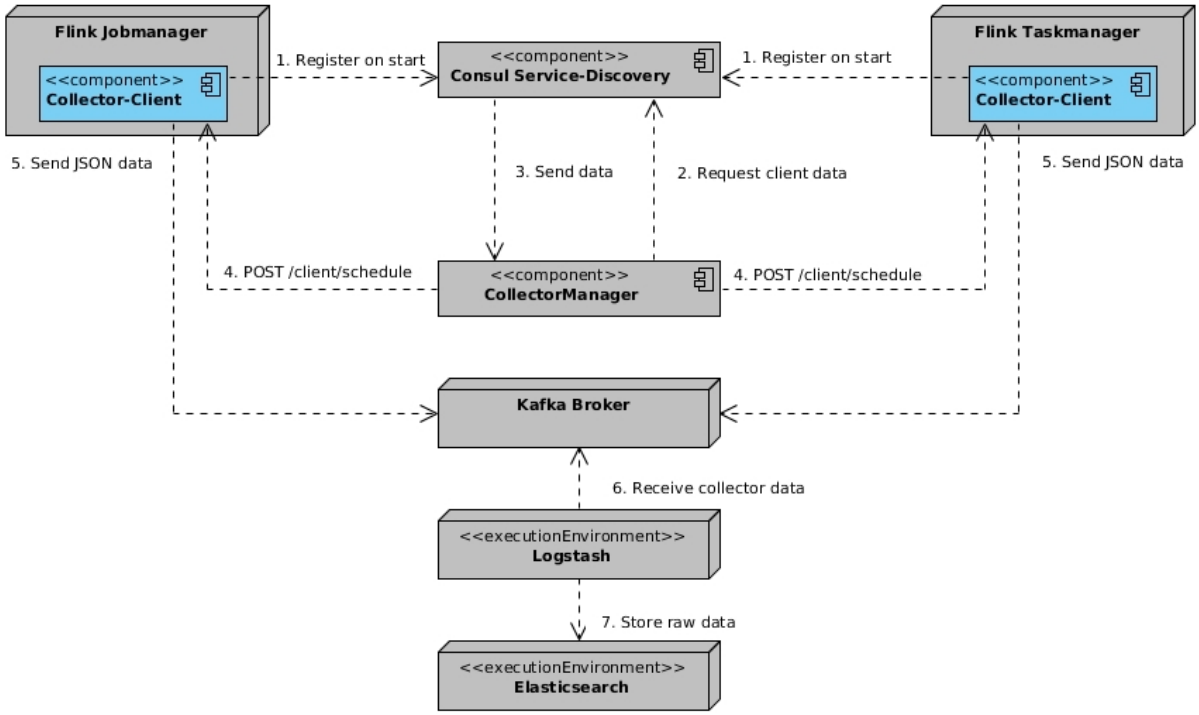

Figure A.11: Component diagram

A.1.5 Deployment diagram

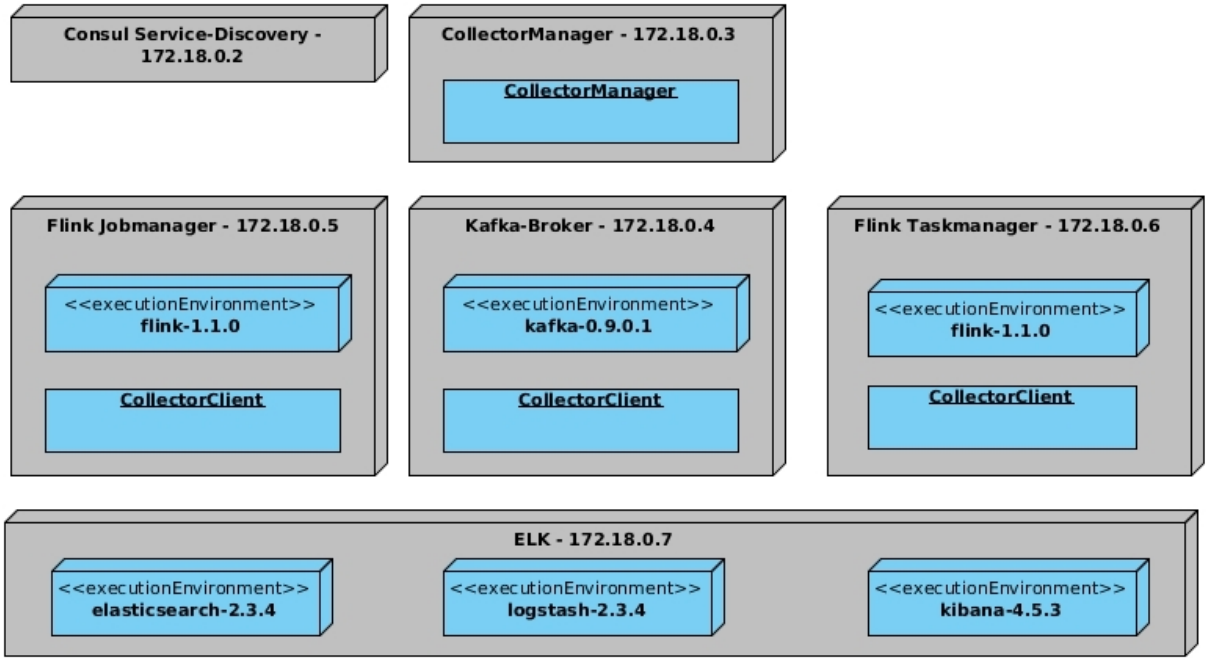

Figure A.12: Deployment diagram

A.2 Collector JSON payloads

A.2.1 DStatCollector

```
{
    "clientTimestamp": "2016-08-12T20:59:30.608",
    "clientHost": "127.0.1.1",
    "clientPort": 9091,
    "instanceId": "collector-client:25ba074a0d81dd5c8e2c7b4c73152dd2",
    "collectorType": "dstat",
    "data": {
        "disk": {
            "read": 112.0,
            "write": 436.0,
            "utilization": 6.49,
```

```
            "transactions": {
                "read": 5.72,
                "write": 8.9
            }
        },
        "process": {
            "runnable": 0.0,
            "uninterruptible": 0.0,
            "new": 1.4,
            "count": 271,
            "latency-highest-total": {
                "name": "compiz",
                "value": 8475.0
            },
            "latency-highest-avg": {
                "name": "cat",
                "value": 101.0
            }
        },
        "memory": {
            "usage": {
                "used": 5548032.0,
                "buffer": 217088.0,
                "cache": 1626112.0,
                "free": 695296.0
            },
            "process-most-expensive": {
                "name": "java",
                "value": 1510400.0
            },
            "paging": {
                "in": 1.22265625,
                "out": 10.8
            },
            "swap": {
                "used": 312320.0,
```

```
            "free": 7987200.0
        },
        "vm": {
            "hardPageFaults": 0.5,
            "softPageFaults": 990.0,
            "allocated": 1476.0,
            "free": 1500.0
        }
    },
    "system": {
        "load-avg": {
            "1m": 1.16,
            "5m": 1.28,
            "15m": 1.17
        },
        "interrupts": 1225.0,
        "contextSwitches": 3178.0,
        "ipc": {
            "messageQueue": 0.0,
            "semaphores": 0.0,
            "sharedMemory": 29.0
        },
        "unix-sockets": {
            "datagram": 38,
            "stream": 799,
            "listen": 38,
            "active": 761
        }
    },
    "io": {
        "read": 5.72,
        "write": 8.9,
        "process-most-expensive": {
            "name": "upstart",
            "pid": 3210,
            "read": 502.0,
```

```
            "write": 210.0,
            "cpuPercentage": 0.0
        },
        "filelocks": {
            "posix": 33.0,
            "flock": 4.0,
            "read": 13.0,
            "write": 25.0
        },
        "filesystem": {
            "openFiles": 13344,
            "inodes": 48582.0
        }
    },
    "cpu": {
        "usage": [
            {
                "name": "cpu0",
                "user": 17.0,
                "system": 3.3,
                "idle": 77.0,
                "wait": 2.6,
                "hwInterrupt": 0.0,
                "swInterrupt": 0.0
            },
            {
                "name": "cpu1",
                "user": 17.0,
                "system": 3.4,
                "idle": 77.0,
                "wait": 2.7,
                "hwInterrupt": 0.0,
                "swInterrupt": 0.4
            }
        ],
        "process-most-expensive": {
```

```
            "name": "java",
            "pid": 32528,
            "cpuPercentage": 5.8,
            "read": 102.0,
            "write": 272.0
        },
        "process-cpu-time-highest-total": {
            "name": "Xorg",
            "time": 40.1
        },
        "process-cpu-time-highest-avg": {
            "name": "docker-compos",
            "time": 225.0
        }
    },
    "net": {
        "traffic": [
            {
                "name": "docker0",
                "send": 0.0,
                "received": 0.0
            },
            {
                "name": "wlp2s0",
                "send": 0.0,
                "received": 0.0
            }
        ],
        "sockets": {
            "total": 1.0,
            "tcp": 19.0,
            "udp": 8.0,
            "raw": 0.0,
            "ipFragments": 0.0
        },
        "tcp-sockets": {
```

```
156                 "listen": 20.0,
157                 "established": 43.0,
158                 "syn": 0.0,
159                 "timeWait": 5.0,
160                 "close": 0.0
161             },
162             "udp": {
163                 "listen": 17.0,
164                 "active": 0.0
165             }
166         }
167     }
168 }
```

Code snippet A.1: DStatCollector JSON payload

A.3 Screenshot

A.4 Graph

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