FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Enhanced Performance Testing and Monitoring of JVM-based Distributed Data-Processing Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Bernhard Nickel

Matrikelnummer 0925384

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Dipl.-Ing. Michael Vögler, BSc

Wien, 21.11.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Enhanced Performance Testing and Monitoring of JVM-based Distributed Data-Processing Applications

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Bernhard Nickel
Registration Number 0925384

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Mag. Dr. Schahram Dustdar
Assistance: Dipl.-Ing. Michael Vögler, BSc

Vienna, 21.11.2015

(Signature of Author)

(Signature of Advisor)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Bernhard Nickel
Rueppgasse 24/26, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I am thankful to my advisor Prof. Schahram Dustdar for giving me the opportunity to write this thesis at the Distributed Systems Group. My co-advisor Michael Vögler, who guided me throughout the thesis, deserves a particular thank-you for always taking the time to provide me with valuable input and feedback.
When thinking about the past years at university, I want to thank my fellow students Dominik Strasser and Gregor Schauer for mutual support and for making the experience of group work and peer learning a pleasant and mostly entertaining one.
Besides my thesis advisors and fellow students, I want to thank Andreas Abart, a former manager and mentor, and now good friend, for inspiring me at the right time to work towards an academic degree.
Last but not least, I want to thank my family and friends for their support throughout the past years.

Abstract

In the age of big data with ever-growing data volumes, data-processing applications face considerable performance challenges. If they do not fulfill their performance requirements, they do not deliver their intended benefit to their organization. Therefore, performance testing and monitoring is crucial for organizations as it enables them to test, analyze and assess the performance of their data-processing applications. Since single machines have not kept up with the growing data volumes, data-processing applications have to scale across clusters, grids or other distributed infrastructures. Whereas distribution allows such applications to meet their performance requirements, it comes at a cost. Besides the design and manageability challenges that emerge, performance testing and monitoring become more difficult to conduct. This especially applies to data-processing applications, where monitoring has not been considered at design time. There are existing testing and monitoring solutions for distributed systems. Unfortunately these tools are often limited in their scope: Either they are focused on certain metrics, such as a server's resource metrics, or bound to a particular environment or data-processing engine.
The goal of this work is to investigate how the performance of distributed JVM-based data-processing applications can be tested and monitored independently from a particular environment or data-processing engine. The different challenges when monitoring a JVM-based distributed data-processing application are analyzed step by step, from defining proper metrics, dealing with data acquisition and publication, to measurement data analysis. Based on the result of the analysis, a design for a framework that allows to monitor and test any JVM-based distributed data-processing application is proposed. To demonstrate the feasibility of our design, a proof-of-concept implementation of the framework is developed. Finally, in order to evaluate the framework and to show that it serves its purpose, it is applied to a demonstration scenario implemented based on both, Apache Spark Streaming and Apache Storm, where the resulting measurement data is analyzed and the results are discussed.

Kurzfassung

In Zeiten von Big Data, mit ständig-wachsenden Datenmengen, sind Datenverarbeitungsprogramme mit beachtlichen Performance-Herausforderungen konfrontiert. Da Datenverarbeitungsprogramme, welche ihren Performance-Anforderungen nicht gerecht werden, nicht ihren eigentlich angedachten Mehrwert liefern, sind Performance-Überwachung und Tests daher wichtige und nicht zu vernachlässigende Teile beim Einsatz solcher Programme, da sie es erlauben, Performance zu analysieren, zu beurteilen und Grenzen auszuloten. Weil Datenverarbeitungsprogramme, die nur auf einzelnen Computern laufen, in der Verarbeitung nicht mehr mit den entstehenden Datenmengen zurecht kommen, werden häufig verteilte Systeme eingesetzt. Unglücklicherweise hat der Einsatz solcher verteilter Systeme seine Nachteile: Abgesehen davon, dass es viel schwieriger ist solche Systeme zu entwerfen und zu verwalten, ist das Überwachen und Testen wesentlich schwieriger als bei einfachen Programmen. Das trifft insbesondere auf Datenverarbeitungsprogramme zu, bei denen die Überwachung nicht zum Zeitpunkt des Entwurfs berücksichtigt wurde, da es sehr viel schwieriger ist die Überwachungslogik im Nachhinein einzubauen. Es gibt zwar fertige Lösungen und Produkte zur Überwachung und zum Testen verteilter Systeme, diese haben aber häufig einen eingeschränkten Fokus: Entweder, sie sind limitiert auf bestimmte Metriken, wie zum Beispiel Ressource-Metriken eines Servers, oder erfordern auf den Einsatz bestimmter Software-Umgebungen oder Software-Engines.
Das Ziel dieser Arbeit ist es, eine Lösung zu finden, wie man JVM-basierte verteilte Datenverarbeitungsprogramme überwachen und testen kann, ohne dabei an eine bestimmte Software-Umgebungen oder Software-Engine gebunden zu sein. Die dafür notwendigen Anforderungen werden Schritt für Schritt, von der Definition geeigneter Metriken, über die Erfassung und Verteilung von Messdaten bis hin zur Analyse, diskutiert. Auf Basis dieser Anforderungen, wird ein Entwurf für ein Framework zur Überwachung und Durchführung von Tests JVM-basierter verteilter Datenverarbeitungsprogramme vorgestellt. Um die Plausibilität des Entwurfs zu überprüfen, wird eine Proof-of-Concept Implementierung entwickelt. Abschließend wird das Framework auf zwei Implementierungen eines Demonstrationszenarios, eine basierend auf Apache Spark Streaming und eine basierend Apache Storm, angewandt und die resultierenden Messdaten analysiert, um die Anwendbarkeit und den Mehrwert dieser Arbeit zu im praktischen Anwendungsfall zu zeigen.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

Applications that do not fulfill their performance requirements generally do not deliver their intended benefit to their organization [50]. In the age of big-data with ever-growing data volumes, this is especially demanding for data-processing applications, which have to perform well in order to be able to process accruing amounts of data. Hence, performance testing and monitoring is crucial as it enables organizations to analyze, assess and test the performance of their data-processing applications. As the processing capabilities of single machines have not kept up with these growing data volumes, data-processing applications have to scale their computations across clusters, grids or other forms of distributed infrastructures [75]. Besides the design and manageability challenges of such distributed applications, both, performance testing and monitoring are becoming more complex and difficult to conduct as performance measurements of multiple computers (we call them workers in this thesis) have to be considered. Additionally, proper metrics are more difficult to identify and to analyze once recorded. Acquiring the measurement data, especially of applications where monitoring has not been considered from the beginning, is not always straightforward. Furthermore, certain requirements for centralizing monitoring data should be satisfied in order to maximize the benefit of monitoring [69].
There are existing solutions [22] [47] [70] [42] for both, measuring purely technical metrics, such as CPU utilization, memory usage, etc. of single workers, and collecting measurement data of distributed applications centrally. However, these solutions are lacking certain features, which would make performance monitoring even more targeted and effective. Monitoring solutions that simply collect technical metrics often do not provide enough insights on how an application performs, since runtime performance of an application's processes is not measured and can not be analyzed. Also, solutions that can collect measurement data of a distributed application centrally usually do not provide any functionality to acquire measurements.
Hybrid solutions, such as Ganglia [22], which can acquire technical metrics of a distributed application's workers out-of-the-box and centralize them, can be extended to collect arbitrary measurements, but do not provide any functionality to acquire performance measurements of

existing applications [43]. Acquiring measurements, if not considered in advance at design time, requires additional code and often changes in existing applications. Even distributed data-processing engines such as Apache Storm [64] or Apache Spark [62], that provide monitoring functionality for their workers, have limitations as their monitoring capabilities are strictly bound to their engines' processing model.
The goal of this thesis is to investigate how the performance of distributed JVM-based data-processing applications can be tested and monitored independent from a particular processing model or data-processing engine, and to propose a framework design that meets the resulting requirements. The main focus of this thesis work is how measurement functionality can be added to existing applications without making changes to their code and having to rebuild them.

1.2 Aim of the Work

The goal of this thesis is to analyze and discuss the challenges of monitoring and testing JVM-based distributed data-processing applications. Furthermore, solutions and approaches that address these challenges are examined and a design for a monitoring and testing framework is proposed. To demonstrate the feasibility and applicability of the proposed framework, a proof-of-concept implementation will be developed. To leverage the frameworks applicability, modules for a simple integration with the data-processing engines Apache Storm and Apache Spark are provided. To demonstrate the framework's usage, a problem that suits the requirements for a distributed data-processing application will be selected and implementations for both, Apache Storm and Apache Spark will be developed. The developed framework will then be applied to the implementations and tests be executed in order to create measurement data. Finally, acquired measurement data will be analyzed and the results of the analysis are discussed.

1.3 Methodological Approach

First, the main issues when monitoring and testing distributed applications are analyzed. Metrics and methods to measure them will be defined, existing practices for collecting and centralizing data will be put in context for performance monitoring. Non-functional requirements that must be satisfied for monitoring performance of distributed applications will be identified. Additionally, in order to meet the specific demands of the particular scope of this thesis, methods for extending existing applications will be investigated and analyzed. Second, technologies and frameworks that help to meet the defined requirements are evaluated. Third, a design for a performance testing and monitoring framework will be proposed. Next, to prove the feasibility of the proposed design, a proof-of-concept framework will be developed and it will be shown how existing data-processing engines can be integrated. In order to demonstrate the applicability of the framework and to generate data that can be analyzed, a test scenario will be defined and executed. Finally, an analysis of the test scenario's data will be conducted and the analysis' results will be discussed.

1.4 Organization

The remainder of this thesis is structured as follows:

- Chapter 2 defines what JVM-based distributed data-processing application are, what performance testing and monitoring is and how that can be done.
- Chapter 3 discusses available testing and monitoring solutions for distributed systems, and how this thesis work differentiates from them. Furthermore, this chapter discusses relevant research that has been conducted in that particular field.
- Chapter 4 selects features and requirements for monitoring and testing JVM-based distributed data-processing applications, and proposes a design that meets these requirements.
- Chapter 5 describes implementation details of a proof-of-concept framework developed in the course of this thesis work.
- Chapter 6 defines requirements for a proper demonstration use case, proposes a use case that meets these requirements, presents implementations of the selected use case and discusses the demonstration results.
- Chapter 7 recapitulates the findings of this thesis and gives an outlook on future research.

CHAPTER 2

Background & Analysis

In this chapter, we discuss the requirements and challenges when monitoring and testing the performance of distributed data-processing applications, which are relevant to the research presented in this thesis. First, proper metrics for measuring an application's performance are identified. Second, different approaches of how to acquire measurement data based on the identified metrics are explored. As this thesis focuses on JVM-based distributed data-processing applications, Apache Spark and Apache Storm, two data-processing / streaming engines that run on a JVM, will be taken into closer consideration. Third, methods to modify and/or extend existing applications in order to acquire measurement data without having to make code changes or rebuilding are discussed. Fourth, to enable analysis of the acquired measurement data, concepts for centralizing the measurement data are debated. Finally, we discuss data analytics approaches, where a differentiation between post-test or post-execution analysis, and real-time analysis will be made.

2.1 JVM-based Distributed Data-Processing Applications

Definition & Goals

Unfortunately, there exists no unique definition of what a distributed data-processing application is. Even worse, there aren't any unique definitions of the terms *distribution* (in relation to computing), *application* or *system* (in relation to software) either.
When it comes to distributed computing, a term that is quite commonly used is *distributed systems*. However, there are various definitions for that common term too [66]. Tanenbaum et al. define a distributed system as "A distributed system is a collection of independent computers that appears to its users as a single coherent system" [66]. The fact that this collection of independent computers (we also call them workers in this thesis) appears to its users as a single system means, that these computers need to communicate. Thus, the computers need to be interconnected, and a computer network must be established. Since virtualization has become common over the past years, a computer may not actually be a physical device, it could also

be a virtual machine connected to other virtual or physical computers through both, virtual and physical networks.

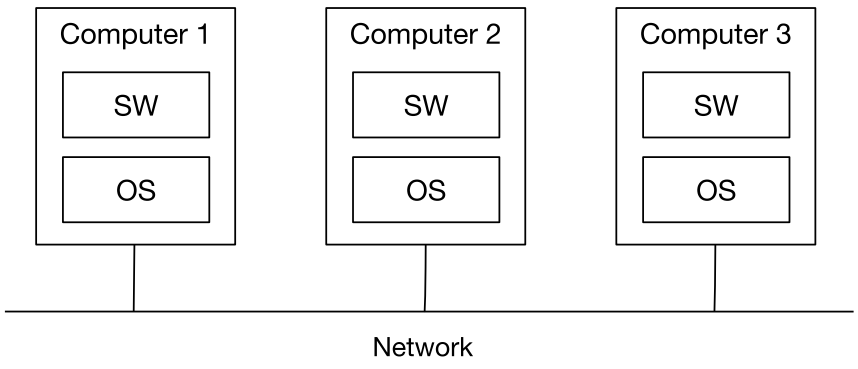


Figure 2.1: Computer network example

Figure 2.1 shows an example of computers connected by a network. The computers shown in this figures are a simplified model where a computer consists of an operating system (OS) and additional software (SW).

Now that we have defined what a distributed system is, we will differentiate between *system* and *application*. In this thesis, we define an application as "a program designed to perform a specific function directly for the user or, in some cases, for another application program" [4] and a system as "operating systems and any program that supports application software" [61].

Data-processing is the work that is done on data to extract and/or create information [58].

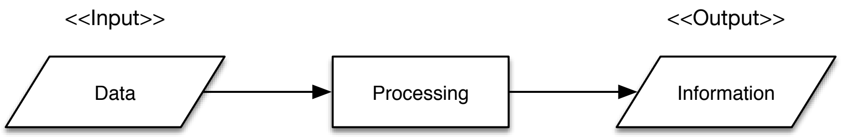


Figure 2.2: Data processing

Figure 2.2 shows an overview of what data-processing is. The central element, the *Processing* step, often consists of various activities such as:

- Validation
- Transformation
- Aggregation
- Analysis

In summary, a data-processing application is a software that performs a specific function directly for the user or for another application, where the specific function is to process data in order to extract and/or create information. A distributed data-processing application is a data-processing application, where the processing work is done by multiple, through a network interconnected, computers.
The goal of a data-processing application, therefore, is to create value from data. Subsequently, the goal of a distributed data-processing application is to create value from ever growing amounts of data, which can not be processed by single computers anymore, by scaling out to distributed computers.

Java Virtual Machine (JVM)

“The Java virtual machine is an abstract computer“ [72]. Its specification [63] defines features and functionalities each JVM implementation must have, and the class file format that must be processable [72]. Every Java virtual machine implementation must be able to load class files using a class loader and execute the byte code it contains [72] [63].

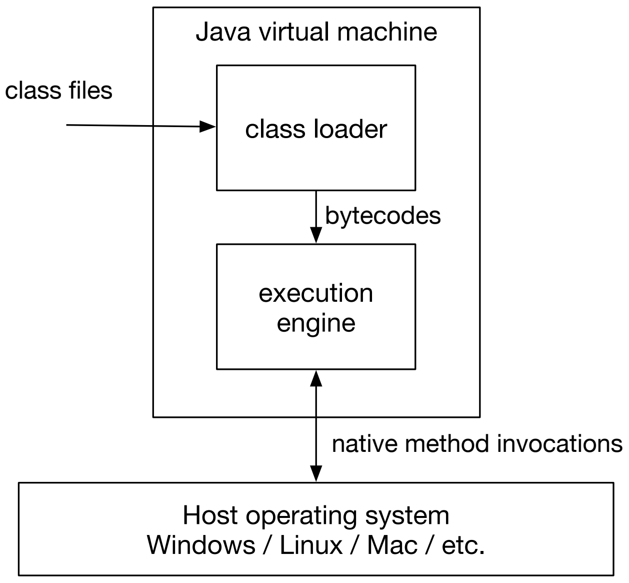


Figure 2.3: Java virtual machine

Figure 2.3 shows an overview the Java virtual machine architecture on top of a host operating system.
Initially developed for running Java applications platform-independently, thus for running a compiled class file on any operating system, the Java virtual machine has evolved to a platform for many other programming languages that can be translated to class files. Three prominent other examples are Scala [60] (a functional programming language), Groovy [26] (a dynamic

object-oriented programming language that is often used for scripting) and Clojure [12] (a Lisp dialect).

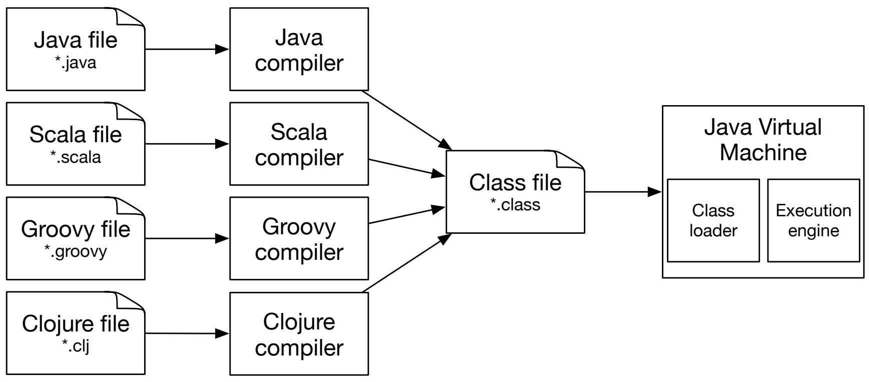


Figure 2.4: Different programming languages for the JVM

Figure 2.4 illustrates how different programming languages can be used for developing JVM-based software.

Given that, we consider a JVM-based application as software, that consists of class files containing byte code, which can be executed on any Java virtual machine implementation that follows the specification defined in [63].

Distributed Data-Processing Frameworks & Systems

When building a distributed data-processing application, there are, as for all kind of distributed systems / applications, certain non-functional domain-independent requirements and goals that must be taken into account.

- **Scalability**: It must be easy to add or remove resources when required. Besides that, a system must still be manageable, even if it has grown considerably [66].
- **Fault-Tolerance**: As hardware components in a distributed system can fail, distributed systems must be designed to be fault-tolerant and continue processing with minimal impact in such a case [70].
- **Resource Sharing**: It should be possible to dynamically share resources in a distributed system.
- **Efficiency**: Since distributed systems are often used to scale out and increase performance, they must have good performance characteristics. Distributing load must be efficient in order do maximize performance increases.

- **Extensibility**: As functional requirements can change over time, a distributed system must be extensible to meet newly emerging challenges.
- **Management and Administration**: As distributed systems are often used for business critical processes, operation and maintenance teams need to be warned early and must be able to trace errors. Thus, administration is a critical requirement.

Building a distributed data-processing application, where all these requirements are considered, from scratch is rather impractical. First, it is expensive, since the efforts that emerge with the fulfillment of these requirements must not be underestimated. Developing such features takes in many cases at least as much effort as developing the functional features. Besides development efforts, testing complexity increases considerably. Second, due to extra efforts that emerge with fulfilling these requirements, it takes more time to develop an application. As many software projects are time-critical nowadays, building everything from scratch is therefore not a viable option. Third, it requires special skills, knowledge, and experience. Engineers that posses these skills and knowledge and are experienced in this field are hard to find and in constant demand, thus hiring can take months.
As a result, frameworks and systems that provide features to fulfill these requirements are often used. Since this thesis approach for monitoring and testing distributed data-processing applications is designed for arbitrary JVM-based applications, we will take two prominent JVM-based frameworks/engines, Apache Spark and Apache Storm, into closer consideration.

Apache Spark

In general, Apache Spark [62] is a cluster computing platform, which is designed for large-scale distributed data-processing [39]. Its programming abstraction, called resilient distributed datasets (RDDs), is based on the MapReduce model [16], but allows efficient data sharing across parallel computation stages [75]. A resilient distributed dataset is a read-only set of records that can be partitioned, and therefore manipulated in parallel. RDDs can be created out of data in storage or be derived from other RDDs. Since all of these operations must be deterministic, Spark has enough information to derive RDDs from other datasets at all times and does not need to have RDDs permanently materialized [75]. The Spark API provides functions for working with RDDs. These functions can be categorized in transformations, such as map, flat map, reduce or filter functions, and actions to actually do something with a data set (e.g., storing to a database). Additionally, Spark provides features for stream processing, an SQL-like model for processing data, a machine learning and a graph computing component.

Figure 2.5 shows Spark's components. The Spark Core is basically a computational engine that is responsible for scheduling tasks, distributing load, managing memory and monitoring applications across workers in a cluster [39]. It also contains the API for working with RDDs programmatically.
The Spark SQL package provides functionality to work with structured data by allowing developers to express queries in SQL and HiveQL in combination with programmatic data manipulation functionality provided by Spark's RDD API [39].
MLib is a library that contains common machine learning functionalities such as classification,

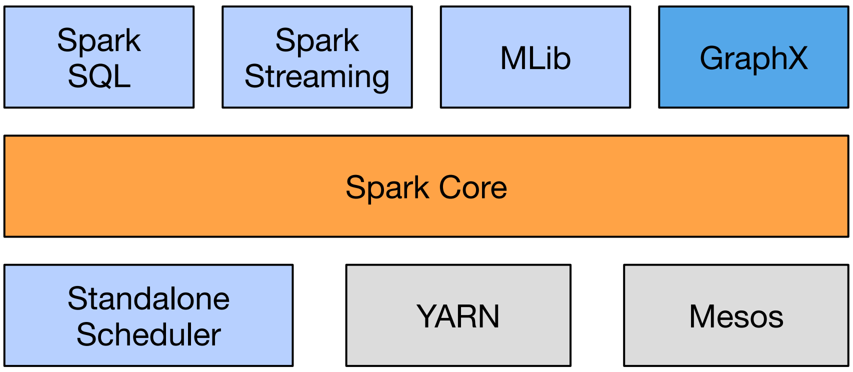


Figure 2.5: Apache Spark components [39]

clustering and collaboration filter algorithms [39].
GraphX is a library that extends Spark RDD API with functionality for working with graphs. It allows to create a directed graph with arbitrary properties attached to vertices and edges, which can be processed using common graph algorithms (e.g., PageRank) [39].
Since Spark is designed to scale up to many workers, it must provide functionality to manage resources in a cluster or to be integrable in a cluster manager. Spark provides a simple standalone scheduler for cluster resource management, but can also be integrated into an Apache YARN [38] or Apache Mesos [38] cluster.

Spark Streaming Spark Streaming is a component of Spark that allows developing streaming applications by using an API similar to Spark's RDD API. It is based on an abstraction named discretized streams. A discretized stream is a sequence of data arriving over time, which is is represented as a sequence of RDDs, where data is combined in RDDs in a certain interval within Spark [75]. In other words, Spark forms micro-batches out of a sequence of arriving data, as illustrated in Figure 2.6.

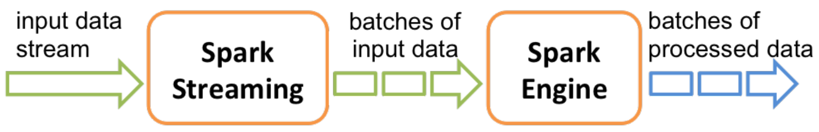


Figure 2.6: Apache Spark data flow [18]

Spark Streaming Programming Model The Spark Streaming programming model is similar to the normal Spark programming model. It can be separated into three different parts:

- **Input Sources**: Receiver functions that receive data from a data source (e.g., a database or socket and emit data to a discretized stream).
- **Transformations**: As Spark, Spark Streaming allows to transform RDDs of a discretized stream. However, the main difference to Spark is that each transformation results in a new discretized stream, thus a sequence of RDDs. An example of this process is shown in Figure 2.7, where each block represents an RDD. There are two different types of transformations that Spark Streaming supports:

  Stateless Transformations: Simple, RDD to RDD transformations such as map, flat map, reduce and filter.

  Stateful Transformations: Transformations on a stream that track data across time.
- **Output Operations**: Operations for doing something with the processed data (e.g., storing it in a database).

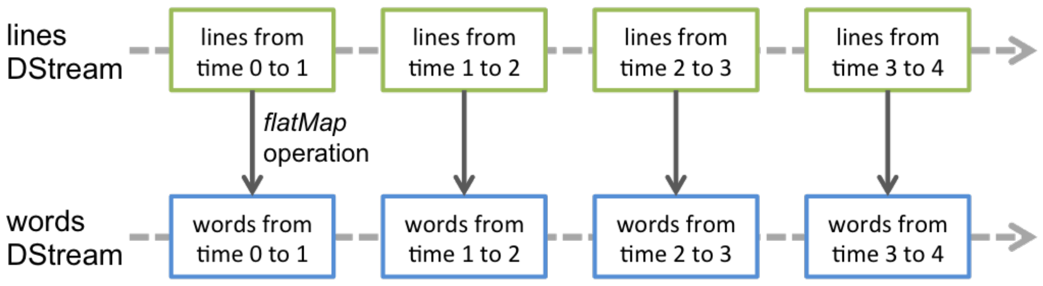


Figure 2.7: Spark Streaming example [62]

Apache Storm

Apache Storm is a real-time distributed data-processing engine, which has been developed at Twitter [70]. It is designed to be scalable, resilient, extensible and efficient. Whereas its underlying concept is completely different, Apache Storm is an alternative to Apache Spark Streaming from a functional point of view. At its core, Apache Storm is based on streams of tuples flowing through topologies. A topology is a directed graph where vertices can be considered as computation nodes and edges as the data flow between the nodes. There are two different types of nodes:

- **Spouts** are sources of tuples.
- **Bolts** process tuples. A bolt can be both, a consumer and a producer of tuples, thus one bolt can be linked with another bolt within a topology.

An exemplary Twitter topology is shown in Figure 2.8. The *TweetSpout* emits tuples (tweets) into the topology. The *ParseTweetBolt* processes tweets and separates them into words, and the *WordCountBolt* counts the number of words in a tweet [70].

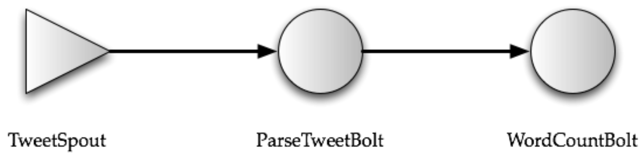


Figure 2.8: Exemplary Apache Storm topology [70]

Since Apache Storm is designed to be scalable, it runs on a distributed cluster, and/or on clusters managed by a cluster manager such as Apache Mesos [70]. Hence, there might be several instances of a bolt running on different workers in a cluster. In such cases, Storm automatically distributes tuples from producers to consumers using different strategies [70]:

- **Shuffle Grouping**: Randomly distributes tuples to bolt instances.
- **Field Grouping**: Distributes tuples by field values.
- **All**: All bolts receive all tuples.
- **Global**: Sends all tuples of a stream to a single bolt.
- **Local**: All tuples are sent to the bolt running on the same worker.

Figure 2.9 shows an example of two different distribution strategies.

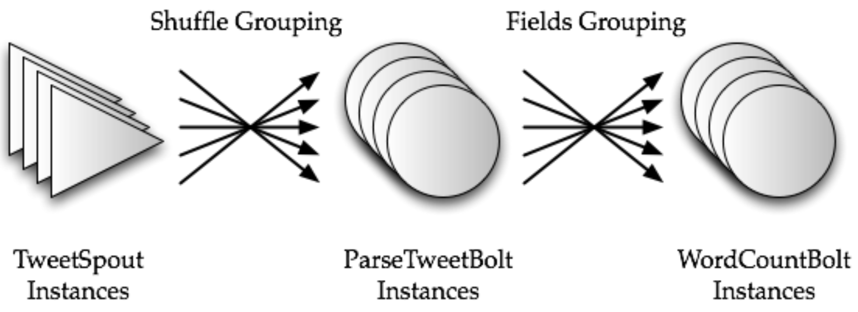


Figure 2.9: Distribution strategies [70]

In a cluster, each worker receives tasks related to a spout or bold that can be executed from a master worker. This means that cluster workers are not directly linked to the Storm topology and, therefore, easier to manage and more scalable [70].

2.2 Performance Testing & Monitoring

In this section, we will discuss what performance is, the goals and benefits of performance testing and monitoring, and how testing and monitoring are related to each other.

Performance Definition

Molyneaux [50] argues that the matter of a well-performing application is ultimately a question of perception: “A well-performing application is one that lets the end user carry out a given task without undue perceived delay or irritation. Performance really is in the eye of the beholder“ [50]. This definition certainly applies in simple cases, where it is obvious who or what the user is (e.g., when a person browses a website). However, since data-processing applications often run in background, it is not obvious who the end user is and if there are any. We argue that there are end users and that this definition applies. Given the fact that any data-processing application serves a purpose since it wouldn’t exist otherwise, there has to be an end user, either an active or at least a passive one. For data-processing applications, where the results are used actively by somebody, as it is the case for data analytics applications, that person is the end user and perceives how the application performs. For data-processing applications where the result data is not actively used (e.g., data archives), we argue that the end user is the owner of the data-processing application’s results. This can be organizations, cooperations, governments as well as humans. Since organizations, cooperations or governments do not have any perceptions themselves, an application’s performance is perceived by their agents, hence mostly employees. For instance, when an archiving application is not capable of dealing with the amounts of accruing data, a system administrator will notice that and perceive the application as not performing well enough.
Unfortunately perception can not be measured accurately. However, there are indicators that can be considered as proxy attributes. Proxy attributes are attributes that measure impacts indirectly (e.g., if a user perceives an application as too slow, one can analyze runtime behavior). If a system administrator reckons that an application is not capable of dealing with the accruing amounts of data, one can check throughput and capacity. These indicators are discussed in Section 2.3.

Performance Testing

In this thesis, we consider three different approaches of performance testing:

1. **Testing against requirements:** In this approach, performance requirements are well-defined and quantifiable. The load that has to be simulated during a test run can be derived from the requirements. After a test run, the performance measurement data is evaluated and analyzed in order to conclude whether the application has passed or failed the test. For example, if a requirement specifies that 100000 incoming messages must be processable per second, a test where at least 100000 messages are sent per second has to be conducted.

2. **Testing to evaluate limits:** In this approach, there are no specified requirements. During a test run, the load has to be increased over time. An analysis of the performance mea-

surement data allows to draw a conclusion on an application's performance and where the limits are.

3. **Testing for comparison:** This approach is used when there are multiple implementations of the same functionality (e.g., by different software vendors or when different frameworks for the same application have been used). The load and input data is defined upfront a test and does not necessarily have to be aligned with performance requirements. A test run includes execution of all to be tested implementations using the given input data and load configuration. The performance measurement data allows to compare the performance of the different implementations.

All three approaches require that performance measurement data for analysis is collected. This can only be done if an application's performance is monitored.

Performance Monitoring

There is no unique definition of what software performance monitoring is. In this thesis, we define performance monitoring for distributed data-processing applications as the steps required to analyze an application's performance behavior.

These steps consist of:

1. **Monitoring Data Acquisition:** The process of generating performance measurement data.

2. **Monitoring Data Publication:** The process of publishing data to either a monitoring client application for real-time analysis, or a central data repository for post-test / post-execution analysis.

3. **Monitoring Data Management:** Managing and processing monitoring data if stored in a central repository.

4. **Monitoring Data Analysis:** The process of extracting information and knowledge from monitoring data.

In Section 2.1 we defined some non-functional requirements and goals that must be taken into account when building a distributed data-processing application. When thinking about a monitoring solution that is supposed to be capable of monitoring such an application, these requirements must be fulfilled by the monitoring solution as well. Additionally, there are two requirements that must be considered :

- **Performance Impact**: Assuming that a monitored application's performance is a critical issue, the impact by the monitoring process should be as low as possible.

- **Overhead**: Monitoring-caused additional resource usage per worker should be kept low since performance critical applications often have high resource demands [43].

2.3 Metrics

In this section, we discuss metrics that we use for measuring an application's performance. In general, we distinguish between two types of metrics:

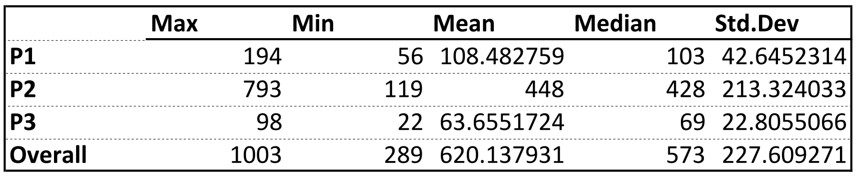
- **Runtime-Related Metrics**: These metrics provide information how an application performs regarding runtime behavior. They are useful as one can tell if a system has to be scaled up if more data has to be processed.
- **Resource-Related Metrics**: These metrics provide information on how much resources are used by an application. Using these metrics, it is possible to analyze if the resources are sufficient or further resources are needed if the load increases.

Runtime, Throughput and Capacity

Runtime

When it comes to performance measurement, the first metric one will look at is response-time or runtime per process or process step [50]. In other words, the amount of time it took a system to run a process or just a single process step.
As there will be a lot of single runtime measurements for each process or process step when running a performance test or monitoring a running system, looking at each runtime measurement individually would be rather impractical. Therefore, aggregation of measurement data using statistical methods is crucial when it comes to performance metrics. Some basic statistical aggregations one would apply are:

- Maximum: The largest value of a set of values [44].
- Minimum: The smallest value of a set of values [49].
- Mean, Average: By mean or average we refer to the arithmetic mean as defined in [45].
- Median: The middle value of an ordered set as defined in [46].
- Standard Deviation: A measure of how spread values of a set are [17].
- N^{th}- Percentile: The value, which below N percent of values in a set fall [53].



	Max	Min	Mean	Median	Std.Dev
P1	194	56	108.482759	103	42.6452314
P2	793	119	448	428	213.324033
P3	98	22	63.6551724	69	22.8055066
Overall	1003	289	620.137931	573	227.609271

Figure 2.10: Runtime data example

Figure 2.10 provides a tabular view of an example analysis of runtime data, where runtime measurement data of single process steps (P1, P2 and P3) has been aggregated per process and for the overall runtime.

Throughput & Capacity

Throughput describes the number of transactions, operations or objects, or a data volume that can be processed in a certain amount of time [50]. Common examples are bytes per seconds, packets per second, tuples per second, or received messages per second.

Capacity describes how many transactions, operations or objects can be processed in a particular time period, thus simultaneously [50]. For instance, if the processing runtime for a transaction is 500ms, and no transactions can be processed in parallel, capacity would be two transactions per second. If two transaction can be processed simultaneously instead, capacity would double to 4 transactions per second.

Server & JVM Metrics

There are many different server metrics, operating system dependent and independent, that can be observed. However, CPU load and time, and memory usage are particular important as they give a good picture of how a server (worker) deals with increasing load [50]. When considering memory usage, the heap space is of particularly importance for JVM applications as the heap space is used for object allocation. If heap usage exceeds the heap maximum, the JVM throws an out of memory error and stops processing. Therefore, heap usage has to be considered when testing and monitoring a JVM-based application. In this thesis, we will take the following server and JVM metrics into consideration:

System CPU Load

System CPU load describes the CPU usage of the entire system [15]. As system CPU load is a snapshot value at a particular point in time, polling CPU load in a defined interval and aggregating polled measurements (as for runtime measurements described on page 15) gives a better overview of CPU usage while running an application. In this thesis, we will focus on maximum, minimum and mean aggregations for System CPU Load.

Process CPU Load

Process CPU load describes the CPU usage for a certain process [15]. Same as system CPU load, Process CPU Load is a snapshot value at a particular point in time, therefore applying aggregations as described for system CPU load is required for process CPU load likewise.

Process CPU Time

Process CPU time describes the CPU time, the amount of time in nanoseconds or clock ticks the CPU was used for processing, used by a certain process [15]. When combining CPU time with runtime of a certain process, effectiveness of CPU usage can be analyzed (e.g., of different data processing engines).

Heap Memory Usage

Heap Memory Usage describes the memory usage of the heap space of a certain JVM-process [36]. As CPU load, heap memory usage is a snapshot value at a particular point in time. Applying aggregations as for CPU load or profiling over defined time periods, are two common ways to observe an application's heap memory usage behavior.

2.4 Data Acquisition

In this section we will discuss how the measurement data for the metrics discussed in Section 2.3 can be acquired or, if they can not be acquired directly, derived.

General

Runtime

Considering the JVM architecture [72], the simplest approach to obtain runtime measurements of a process, is to measure a classes method invocation time, as illustrated in Listing 2.1 .

```
//Process to be measured starts here ...
someClassInstance.doSomething(..);
//and ends here.
```

Listing 2.1: Code sample of a single method call that could be measured

However, this approach only works if the following preconditions are met:

A classes method reflects the to be measured process This precondition should be met if an application has a good object-oriented design. One criterion for good object-oriented design is that each module (or class) should be logically coherent and that modules are loosely coupled [73]. If this criterion is met, a facade or service classes, or any other classes method should then reflect a certain functionality. Unfortunately, there might be cases where this is not the case.

```
while(true) {
    Object o = fetchSomeObject();
    if(o != null) {
        //Process to be measured starts here ...
        b.doSomething(o);
        this.doSomething(o);
        SomeUtilClass.utilFunction(b);
        c.doSomething(o);
        //and ends here.
    }
}
```

Listing 2.2: Code sample where measuring a single method call is not sufficient

Listing 2.2 shows an example code block where a process is not reflected by a single method. There are two possibilities to obtain the process runtime for such cases:

- Insert measurement logic at process start and end.
- Sum up runtimes of methods called within the process.

Synchronous Processing When all methods, which are called by the observed method, are processed synchronously, the invocation time is equal to the process runtime. Nevertheless, as the JVM supports concurrency, this might not always be the case.

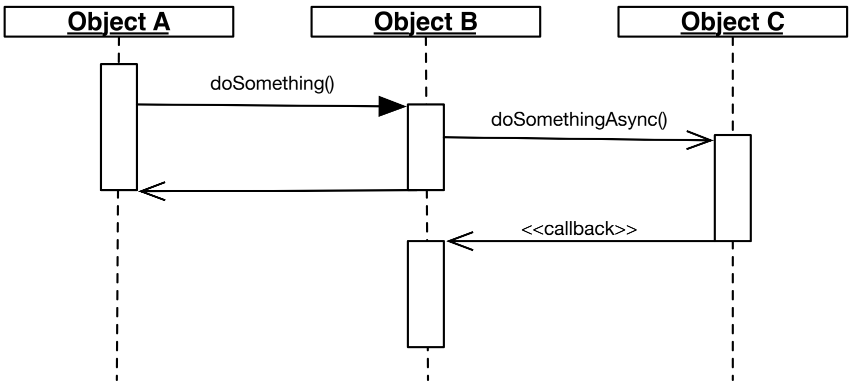


Figure 2.11: Asynchronous processing example

Figure 2.11 shows an example where the method *doSomething()* of *Object B* calls a method of *Object C* asynchronously. In that case, the process runtime would end, as to be seen from the sequence diagram's timelines, after the invocation of *Object B* method *doSomething()* has ended. Thus the invocation time of *Object B.doSomething()* is not a valid proxy for the runtime of the process. In this case the callback method of *Object B* must be taken into account.

Throughput

In comparison to runtime, throughput can not be obtained directly. Instead, throughput is a metric that can be derived from runtime measurement data. In this thesis we define throughput as in Equation 2.1.

$$throughput = \frac{count(R)}{\sum_{r \in R} duration(r)} \tag{2.1}$$

- *R* is the set of runtime measurement records as defined in Equation 2.2

- *count(..)* returns the number of elements in a set
- *duration(..)* returns the duration of a runtime measurement record by subtracting the start time from the end time

$$runtime\ measurement\ record = (starttime, endtime) \tag{2.2}$$

- *runtime measurement record* is a tuple
- *starttime* is a unix timestamp
- *endtime* is a unix timestamp

Capacity

Capacity, as throughput, can not be obtained directly. As for throughput, it is a metric that can be derived from runtime measurement data and using throughput. In this thesis we define capacity, as it is defined in queueing theory [2], in Equation 2.3.

$$capacity = arrivalRate/throughput \tag{2.3}$$

- *throughput* is defined in Equation 2.1
- *arrivalRate* is defined in Equation 2.4

$$arrivalRate = \frac{count(R)}{maxStarttime - mod(maxStarttime, interval) + interval} \tag{2.4}$$

- *R* is the set of runtime measurement records as defined in Equation 2.2
- *count(..)* returns the number of elements in a set
- *maxStarttime* is defined in Equation 2.5
- *mod(..)* is the modulo operator
- *interval* is the interval in milliseconds

$$maxStarttime = \max_{r \in R}(r.starttime) \tag{2.5}$$

- *R* is the set of runtime measurement records as defined in Equation 2.2
- *max(..)* maximum function

Note: The interval must correspond to the throughput's interval.

Server & JVM Metrics

Server metrics can be acquired using native operating system specific interfaces or interfaces provided by the Java virtual machine. In this thesis, we will only consider JVM-provided interfaces.

java.lang.management Package The *java.lang.management* package, "provides the management interfaces for monitoring and management of the Java virtual machine and other components in the Java runtime" [35]. The classes defined in this package are independent from the virtual machine implementation. For this thesis work, we use the class *MemoryMXBean* [36] provided in this package in order to get the heap memory usage of the JVM.

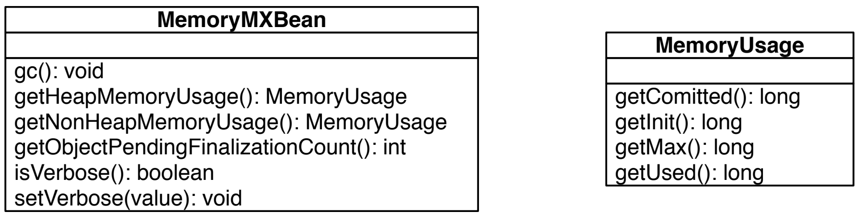


Figure 2.12: java.lang.management.MemoryMXBean

Figure 2.12 shows that the *MemoryMXBean* class provides a method to get the heap memory usage. The *getHeapMemoryUsage()* method returns a *MemoryUsage* object, which contains the maximum amount of heap memory in bytes that can be used and the amount of heap memory in bytes, that is currently used.

com.sun.management Package The *com.sun.management* package "contains Oracle Corporation's platform extension to the implementation of the java.lang.management API and also defines the management interface for some other components for the platform" [14]. This extension contains classes that provide additional server and JVM metrics. Since we are interested in system and process CPU load and process time for this thesis work, and these metrics are not provided by classes in the *java.lang.management* package, we use this package in order to get these metrics.

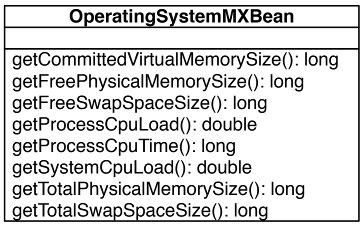

OperatingSystemMXBean
getCommittedVirtualMemorySize(): long
getFreePhysicalMemorySize(): long
getFreeSwapSpaceSize(): long
getProcessCpuLoad(): double
getProcessCpuTime(): long
getSystemCpuLoad(): double
getTotalPhysicalMemorySize(): long
getTotalSwapSpaceSize(): long

Figure 2.13: com.sun.management.OperatingSystemMXBean

Figure 2.13 shows the measures provided by the *OperatingSystemMXBean* [15].

Aggregation As discussed in Section 2.3, system CPU load, process CPU load and heap memory usage are snapshot values and should be aggregated for practical reasons. This requires two steps:

- Polling: Snapshot values must be polled in a defined interval. The polled value must be stored in memory.
- Aggregation: In a longer interval the polled values must be aggregated. The polled values must be cleared to start a new aggregation period.

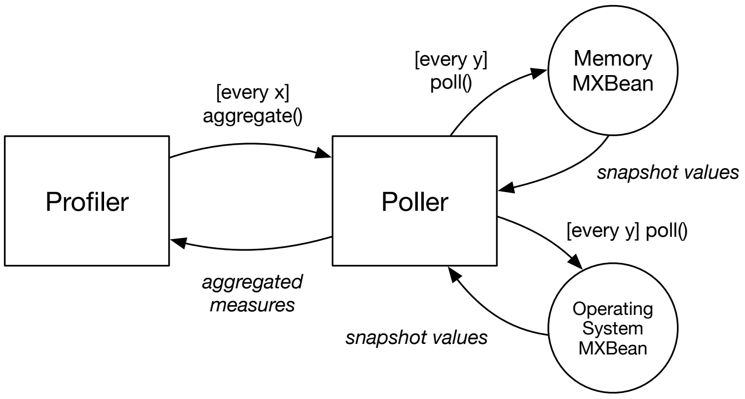


Figure 2.14: Aggregation

Figure 2.14 illustrates this aggregation process. The first interval *x* defines the aggregation interval. In this period, snapshot values are stored in memory and aggregated when the period is over. The second interval *y* defines how often both, the *OperatingSystemMXBean* and *MemoryMXBean* are polled. The amount of memory used by *Poller* is defined by the difference between the aggregation and polling interval, and how many metrics are observed.

Apache Spark Streaming

In Section 2.1 we discussed Apache Spark Streaming's programming model, which consists of transformations, input sources and output operations. The Spark API provides Java interfaces for these elements that contain methods, which can be monitored.

Input Sources To receive data from an input source, Apache Spark Streaming provides an abstract *Receiver* class. A definition with the basic methods for sending data to Spark is shown in Figure 2.15. To measure the time, how long it takes Spark to store the data to Spark Streaming, it would be sufficient to measure the invocation time of the *store* method. Since this does not reflect how long it actually takes to obtain the data that is transferred to Spark Streaming, it might make more sense to measure time at a different point in the control flow. This has to be analyzed for each *Receiver* implementation individually.

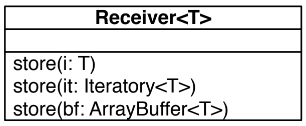


Figure 2.15: Apache Spark Receiver interface

Transformations

Stateless Transformations: Stateless transformations transform RDDs of a discretized stream one-by-one resulting in a new discretized stream of transformed RDDs. An overview of this process is shown in Figure 2.7. The four base stateless transformations are map, flat map, reduce and filter. The main API element for working with discretized streams in Java, the *JavaDStream* class provides methods for these transformations, which expect the transformation functions as arguments.

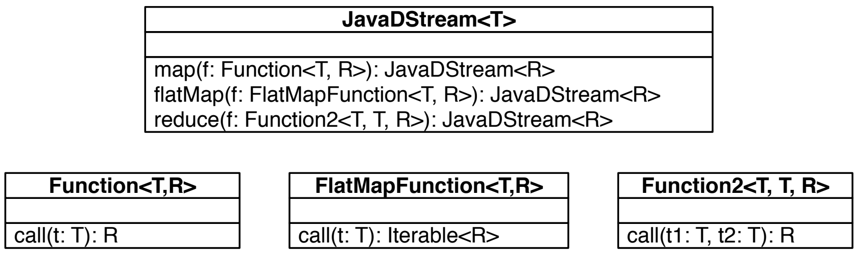

JavaDStream<T>
map(f: Function<T, R>): JavaDStream<R>
flatMap(f: FlatMapFunction<T, R>): JavaDStream<R>
reduce(f: Function2<T, T, R>): JavaDStream<R>
Function<T,R>
call(t: T): R
FlatMapFunction<T,R>
call(t: T): Iterable<R>
Function2<T, T, R>
call(t1: T, t2: T): R

Figure 2.16: Apache Spark API for stateless transformations

To measure the runtime of these transformations, it is sufficient to measure the method invocation time of the transformation function implementations passed to Spark Streaming.

Stateful Transformations: As mentioned earlier in this chapter, stateful transformations track data in a discretized stream across time in order to combine data of RDDs that have been received/processed earlier. Spark's API provides a *window* method for computing new discretized streams based on RDDs of the source stream and given time window. However, since data is simply aggregated within Spark Streaming, we do not consider these operations as relevant for a data-processing application's performance.

Output Operations Output operations serve the purpose of actually doing something with the transformed data in discretized streams. The *JavaDStream* class of Spark Streaming's API provides a *foreachRDD* method, which takes a function, as shown in Figure 2.16, as argument. In order to measure the output operation's runtime it is, as for stateless transformations, sufficient to measure the function's method invocation time.

Additionally, since start and stop timestamps, not just invocation times, are obtained, Spark's latency, the time between computation start and end of an input source or transformation and a transformation, or between a transformation and an output operation, can be calculated.

Apache Storm

Considering Apache Storm's architecture, as described in Section 2.1, runtime performance of a Storm-based application can be measured by observing spouts and bolts, where each spout or bolt can be considered as a single process step. Since the Storm API provides Java interfaces for both, spouts and bolts, the runtime of a spout or bolt can be measured by measuring the method invocation time of their main methods.

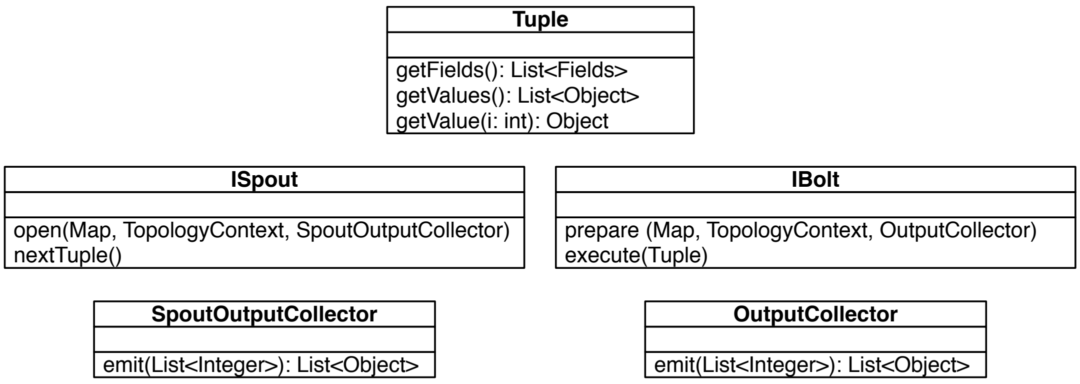

Tuple
getFields(): List<Fields>
getValues(): List<Object>
getValue(i: int): Object
ISpout
open(Map, TopologyContext, SpoutOutputCollector)
nextTuple()
IBolt
prepare (Map, TopologyContext, OutputCollector)
execute(Tuple)
SpoutOutputCollector
emit(List<Integer>): List<Object>
OutputCollector
emit(List<Integer>): List<Object>

Figure 2.17: Apache Storm API

Figure 2.17 shows the interfaces defined by the Storm API. A spout emits tuples via its *nextTuple* method, which is called by Storm's engine, and is responsible for creating/obtaining as well as emitting processing data using the *SpoutOutputCollector*. The *SpoutOutputCollector* is set by Storm via a spout's *open* method. When a bolt receives a tuple, Storm invokes its *execute(Tuple)* method. If a bolt has to emit tuples, it can do that via an *OutputCollector*, which set by Storm using a bolt's *prepare* method. Measuring invocation time of a spout's *nextTuple* and a bolt's *execute* method is a suitable approach for measuring the application's runtime performance. Additionally, since start and stop timestamps, not just invocation times, are obtained, Storm's topology latency, the time between computation start and end of two bolts or a receiver and a bolt, can be calculated.

2.5 Data Publication

In this section, we will discuss how acquired measurement data can be published. By publishing, we mean to make measurement data available to users and/or other applications. In general, we distinguish three different forms of publication: Logging, persistence and data distribution.

Logging

Many applications, especially server applications that run in the background and do not have a graphical user interface, use logging for providing information about their state in a human-readable fashion [27]. Logging is used for various purposes, such as problem diagnosis, quick debugging or monitoring applications [27]. Since performance measurements can be used for problem diagnosis if there are performance-related problems, and logging is used for monitoring applications in general, we have identified logging as one measurement data publication form that is important for performance measurement data.

Persistence

In general, persistence means that data created by a process outlives the processes lifetime. For this thesis work, it means that acquired performance measurement data, once published and persisted, can still be analyzed long after a test has been conducted or data-processing has been finished. Persistence can be achieved by storing created data on a persistent storage medium such as hard disk drives. There are two major concepts used for persisting data: Files and databases.

Files

As robust as the notion of a *file* has been over the past decades, the term is ambiguous and can, even within the domain of computing, be understood differently [30]. We define a *file* as an objected identified by a name, located in a file system where the location can be identified by a path, that contains data and is stored on any persistent storage medium, either locally or remotely. The content's structure, or format, depends on the file's content type and originator.

For this thesis work, where we use files for persisting performance measurement data, we will take two text-based file formats into consideration: CSV and XML.

CSV Comma-separated values (CSV) is a file format where each record is written to a single line and attributes are separated by a symbol, in many cases a comma or semicolon.

XML The Extensible Markup Language (XML) [10] was designed to store and transport data, and to be both, human- and machine-readable.

Databases

A database is a collection of data that is managed by a database management system. A database management system is a software that is able to manage large amounts of persistent data in an efficient and reliable way [6]. Depending on the database type, data can be stored in different structures. In this thesis, we focus on relational databases, thus databases where data is organized based on a relational model that consists of tables with rows and columns.

Data Distribution

In this thesis, we consider data distribution as mechanisms for transferring data from an application running on a worker in a distributed system, where performance measurement data is acquired, to other applications or different computers where performance measurement data is processed.

JMX

Java Management Extensions (JMX) is one part of the Java technology stack, which was designed for managing and monitoring Java applications [65]. It provides an API where resources can be presented as objects, called managed beans (MBeans) in JMX. These managed beans can either be accessed by clients or push notifications to a client. JMX supports remoting by default [65], which makes it suitable for monitoring workers in a distributed system. Since JMX is part of the Java technology stack and supported by many tools and various monitoring systems, we consider it as an important distribution mechanism for performance measurement data.

JMS

Java Messaging Service (JMS) is an API for accessing enterprise messaging systems [29]. Enterprise messaging systems, are systems where messages, asynchronous requests, reports or events, are sent from a producer to a consuming enterprise application [29]. JMS supports two different communication models [29]:

- **Point-to-Point Model:** In this model, messages are exchanged from one point to just one other point using queues.
- **Publish/Subscribe Model:** In this model messages are consumed by many consumers using so-called topics.

As JMS implementations are often used as reliable and scalable data exchange mechanisms in organizations, and messages are exchanged asynchronously, we consider it as an important technology for distributing performance measurement data.

Synchronous versus Asynchronous Data Publication

When publishing performance measurement data of a distributed data-processing application, one must consider whether to publish data synchronously or asynchronously. Publishing data synchronously might have a considerable impact on the application's runtime for a variety of reasons:

- Persisting data to files on hard disk drives consumes time.

- Persisting data to a database can, depending on the database management system, its data structures and transaction isolation, be very slow.
- When publishing data to a remotely located resource, network latency can cause significant slow downs.

Publishing data asynchronously can reduce the impact. However, when measurement data is produced faster than it can be published, depending on the pattern used for asynchronous processing, can cause severe issues such as memory overflows.

Asynchronous Processing Patterns

Thread-per-Task The thread-per-task pattern is the simplest way for performing tasks asynchronously. For each task, a new thread that performs the task will be created. For publishing performance measurement data, this would mean that a new thread is created for each measurement data record published. This approach has some disadvantages [24]:

- Thread Lifecycle Overhead: Thread creation and teardown consume significant computing resources.
- Resource Consumption: Threads consume resources, especially memory. If there are more created threads than processors, the threads sit idle.
- Stability: The number of threads that can be created is not infinite. If the data that has to be published is created faster than it can be processed, at some point the system will crash.

Thread Pools The goal of thread pools is to reuse and manage threads efficiently. Usually, a thread is bound to a work queue where tasks to be executed are held. Whenever a thread in the thread pool has completed a task, it queries the work queue for another task and waits for the next task [24]. Thread pools have the following advantages over the Thread-per-task pattern. First, the thread lifecycle overhead is reduced, which provides better responsiveness as threads do not have to be created first. Second, since thread pool implementations are often configurable [24], resources are better managed, which ensures better stability.

Producer-Consumer Pattern One could also overcome the stated disadvantages of the *Thread-per-task* by manually implementing a producer-consumer behavior using queue data structures. Whereas this approach is similar to *Thread pools* in behavior, the main difference is that messages, in the case of this thesis work performance measurements, are queued instead of tasks.

2.6 Integration

The monitoring features described in the previous two sections (Section 2.4 and Section 2.5) must somehow be integrated with the application to be monitored. There are several different ways how monitoring logic can be included, which will be discussed in this section.

Code Tangling

The most straight-forward approach for integrating monitoring logic is simply to include the required measuring code in the functional code directly. A practice that is called code tangling [20].

```
class SomeClass {
        public void someMethod() {
                // Monitoring start
                long t = System.currentTimeMillis();

                // the actual method functionality is implemented
                   here
                // ...
                someOtherClassObject.doSomething();
                // ...

                // Monitoring end
                long d = System.currentTimeMillis() - t;
                // save method runtime duration using some monitoring
                    classes
        }
}
```

Listing 2.3: Code sample

Listing 2.3 illustrates a code block where monitoring logic is added directly. However, code tangling has some drawbacks. First, if monitoring is not considered from the beginning, the functional code blocks must be modified afterwards. If not done carefully, the original code could be damaged, thus the functionality has to be tested again. Second, adding non-functional code directly to the functional code, weakens cohesion. The code blocks and classes become more confusing, which ultimately reduces maintainability and reusability and increases testing complexity [73] [20]. Third, since the monitoring logic has to be added for each monitored target separately, there is a lot of repetition of the same functionality, which violates the DRY (don't repeat yourself) principle [28]. Fourth, code tangling requires that the source code of the to be monitored application can be modified. Finally, the application has to be compiled and built again.

Object-Oriented Decomposition

When developing an application, there are many things that have to be considered. There are functional requirements, non-functional requirements, such as monitoring, and specified implementation details. All together, these are called concerns [20]. The design principle that should be followed to untangle these concerns and improve application design is called *separation of concerns*. A typical approach in object-oriented design to separate concerns and increase class cohesion is decomposition, to break down classes into multiple classes, where each class has a certain functional or non-functional responsibility. For adding functionality to a class, one might create a subclass that inherits a classes methods and attributes in order to extend it, as shown in

Listing 2.4, or wrap another class around it. Two defined design patterns [21] are commonly used for wrapping classes, the Decorator and Proxy pattern.

```
class A {
        public void doSomething() {
                //do something
        }
}

class B extends A {
        @Override
        public void doSomething() {
                //do something before A.doSomething()
                super.dosSomething();
                //do something after A.doSomething()
        }
}
```

Listing 2.4: Subtyping

Decorator & Proxy Pattern

Both, the decorator and proxy pattern allow to add functionally to a class. As shown in Figure 2.18, the decorator and the concrete class share a super type, typically an interface. The decorator instance has a reference to a delegate object, an instance of the concrete class.

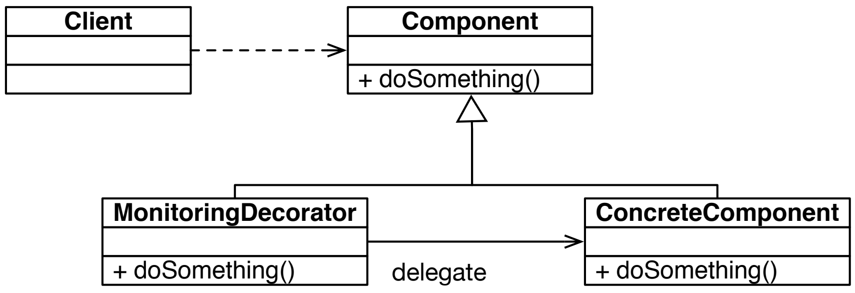


Figure 2.18: Proxy pattern

When a client calls the method of the decorator, the additional functionality is wrapped around the delegate object's method call, as shown in Figure 2.19

A proxy is similar to a decorator from an implementation point of view. However, there is a difference in their purpose and when they should be used. A proxy is supposed to control access to an object, whereas a decorator adds functionality [21].

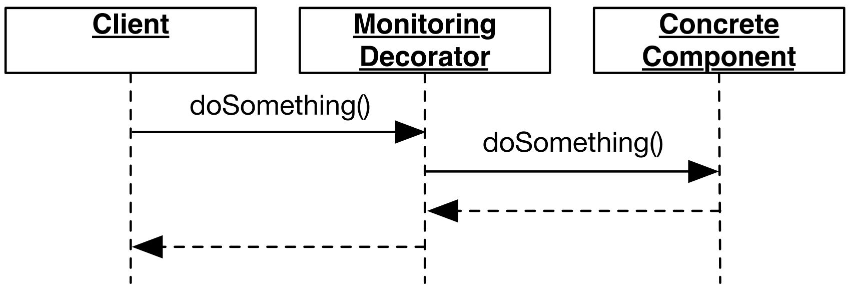


Figure 2.19: Proxy sequence

Object-oriented decomposition has advantages over code tangling. As it increases cohesion of single classes, the code is clearer, better maintainable and reusable. When adding additional functionality using one of these patterns, the original code does not have to be modified and can not be damaged accidentally. The disadvantage of this approach is less flexibility. Only method invocations can be extended for monitoring, which, as discussed in Section 2.4, might not be sufficient for monitoring certain processes. What object-oriented decomposition and code tangling have in common is, that it is required that the application source code can be modified, that the application has to be compiled and rebuilt, and that the DRY principle is violated to some extent.

Aspect-Oriented Programming

In traditional software engineering, software is usually decomposed into modules or objects by functionality. Other concerns, such as non-functional requirements or more detailed implementations issues, are recognized, but it is often left to developers to address these concerns in a program wherever appropriate [20]. In the worst case, this results in code tangling. More skilled programmers would follow the design principle of *separation of concerns*, where concerns are separated into separate modules or objects. However, object-oriented design techniques for addressing this issue, such as the decorator pattern, as discussed in the previous section, have their limits. Many concerns (e.g., measuring method invocation time), even if implemented only once, must be added at different places throughout an entire application. Implementing these so-called crosscutting concerns [20] purely based on object-oriented design techniques (e.g., by creating decorators for all affected components) results in code scattering and repetition, which ultimately reduces maintainability and evolvability.
In aspect-oriented programming (AOP), the issue of separating crosscutting concerns is addressed by introducing aspects.

Aspects

An aspect is “a modular unit designed to implement a concern“ [20]. It consists of advices, join points and pointcuts.
An advice is a piece of code that is executed at a join point (e.g., the code required to measure a method’s invocation runtime would be an advice). Depending on the platform used for aspect-oriented programming, an advice can be executed before, after or around join points.
A join point is a defined place in the structure or control flow of a program, where an advice can be added [20]. The join point model depends on the used aspect-oriented programming platform. However, common join points are method calls, method executions or exceptions. Some platforms, such as AspectJ [5] additionally support more sophisticated join points [41].
A pointcut, or as often called pointcut designator [20], is a selecting mechanism for join points. Since there might be many places (join points) in a program where the same advice should be applied, and explicitly stating each join point is rather impractical, pointcuts are required to apply advices. Additionally, depending on the aspect-oriented programming platform, pointcuts might add a context, such as method execution or method call, to join points [41].

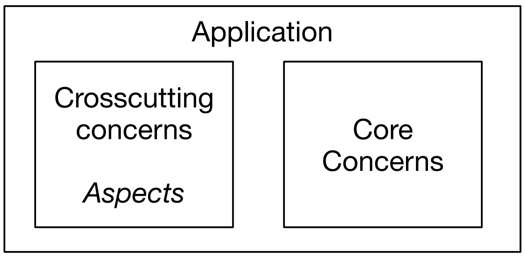


Figure 2.20: Application structure at development time

As shown in Figure 2.20, at development time an application’s structure can be split up into two major blocks. One block that deals with an application’s core concerns, which address its main functionalities, and one block that deals with crosscutting concerns, which are defined as aspects. As mentioned, this view only applies at development time. After compilation or at runtime, depending on how aspects are woven into the core concerns, aspects might not exist as structural blocks on their own anymore.

Weaving

Weaving is the process of composing core concerns and crosscutting concerns, which have been implemented as advices. Depending on the aspect-oriented programming platform, weaving can be done either statically, at compile-time where advices are compiled with the core code, or dynamically at load-time.

Figure 2.21 illustrates the process of compile-time weaving.

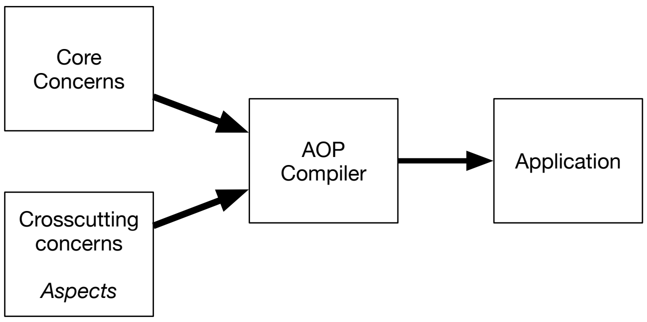


Figure 2.21: Compile-time weaving

AspectJ

AspectJ [5] is one of the most complete aspect-oriented programming implementations [41]. As all other AOP implementations, it consists of two parts:

- The language definition, where the grammar and semantics of the AspectJ language is defined.
- The actual implementation that consists of the compiler and weavers.

The AspectJ compiler produces Java byte code, which means that AspectJ can be used with any programming language that can be compiled to Java byte code and runs on any valid Java virtual machine implementation. In the beginning, AspectJ was an extension to the Java programming language, which introduced new keywords to write aspects. However, by now there also exists an alternative Java annotation-based syntax, which allows to write aspects in pure Java code.

```
public aspect LoggingAspect {
        pointcut loggedMethodCall() :
                call (public * *.*(..));

        before(): loggedMethodCall() {
                System.out.println("Calling␣some␣method");
        }
}
```

Listing 2.5: Aspect sample

Listing 2.5 shows a basic aspect written in the AspectJ language. An aspect is declared similarly to a class in Java. The *pointcut* keyword indicates a pointcut declaration, which is followed by the pointcut name. The pointcut name is followed by braces and then a colon. After the colon, a context (such as *call* for method calls or *execution* for method invocations) is followed by the join point selector. A full description of valid pointcut expressions can be found on the AspectJ website [5]. The advice is declared by the advice type followed by a colon and the pointcut name. AspectJ supports several different kinds of advice types. However, the most common types are: before, after, and around [41]. Listing 2.6 shows the same aspect declared in a Java annotation-based syntax.

```
@Aspect
public class LoggingAspect {

        @Pointcut("call␣(public␣*␣*.*(..))")
        public void loggedMethodCall() {}


        @Before("loggedMethodCall()")
        public void logMethodCall() {
                System.out.println("Calling␣some␣method");
        }
}
```

Listing 2.6: Annotation-based aspect sample

The aspect shown in Listing 2.5 and Listing 2.6 is a dynamic crosscutting construct. Dynamic crosscutting means to add additional behavior to a program. AspectJ also supports static crosscutting, which means to alter the static structure of classes or interfaces by adding methods or fields.

AspectJ Weaving We already stated that weaving can be done statically or dynamically. AspectJ supports both weaving approaches [41]. Static weaving in AspectJ can be separated into two basic weaving types:

- **Source Weaving:** In this case, the AspectJ compiler compiles the original Java program source code and the added AspectJ source code together. The result is Java byte code, where advices have been woven into the original source code.
- **Binary Weaving:** Binary weaving requires both, the original program code and aspects to be compiled to Java byte code in advance. The binary weaver then takes class files or JAR archives as input and weaves the advices into the original program code.

In AspectJ, load-time weaving is a special form of binary weaving. The class files of a program are instrumented at load-time by a Java agent [9] that is provided by the AspectJ framework. The Java agent, which is responsible for weaving at load-time, is configured via an XML file named *META-INF/aop.xml* located in the Java class path.

```
<aspectj>
        <aspects>
                <aspect name="sample.SampleAspect"/>
        </aspects>
</aspectj>
```

Listing 2.7: aop.xml sample

Listing 2.7 shows a basic example *META-INF/aop.xml*, where a single aspect is declared. Considering these weaving options, it is obvious that AspectJ itself causes little overhead and performance losses. Once weaved, an advice is no different than a usual Java method call [41]. When it comes to load-times weaving, the only thing that has to be considered is that startup time increases. However, since this thesis is focussed on data-processing applications, which are, as defined, mostly applications that run in background once started, the increased startup time is not a critical issue.

2.7 Analysis

Depending on the data publication method, as described in Section 2.5, performance measurement data can, in general, be analyzed in two different ways. If the data has been persisted, either directly or indirectly via a publication interface client, the data can be analyzed after a test has been conducted or a program / process has been finished. If a publication method has been chosen, where data can be consumed in real-time, it can also be analyzed in real-time. In this section, we discuss the different approaches.

Real-Time Analysis

Real-time analysis is basically waiting and watching for something to happen [50]. One can observe collected measurement data as it is acquired. Using a visualization tool, it can be made easy to detect changes and keep track of recent history in order to observe trends and performance behavior. Considering the metrics described in Section 2.3, real-time analysis is useful for monitoring server (worker) resource usage while an application is running. Measurements received could be reduced to states (e.g., in order to warn system administrators). Summarized, real-time analysis can be used to detect and forecast potential failures in order to take appropriate actions.

Post-Test & Post-Execution Analysis

Post-test or post-execution analysis is more powerful than real-time analysis. Once data is persisted, it can be analyzed differently by applying different analysis methods. Often, statistical methods and visualization tools are used to extract information from the collected data. In the case of applications that are running over a long period of time, one could use forecasting methods to check if resources are sufficient in the future or create stochastic models to estimate failure probability. Post-test analysis supports the different types of performance tests we described in Section 2.2, since it allows to:

1. Analyze if an application has satisfied defined performance requirements.

2. Analyze when an application reached its limits.

3. Compare results of different test runs.

2.8 Performance Test Execution

We identified two major challenges for executing performance tests of distributed data-processing applications:

1. **Running Scenarios:** In order to test a data-processing application, data that can be processed has to be created. For non-real-time data processing applications, this is simple since data that can be used for testing has to be created only once. The application reads test data and processes it; this process can be repeated many times. In contrast, real-time data-processing applications usually expect data to be pushed or emitted from an external source, where the behavior or creation pattern is often unknown. In order to test real-time data-processing applications, a testing framework must be capable of running scenarios, where test data is created and pushed to the tested application.
2. **Data Volume:** Distributed systems are designed to be scalable. In order to test whether a system scales or not, the load that can be created by a single machine might not be sufficient. Hence, a performance testing framework or application for testing distributed systems must allow to create load on different machines. Since managing and coordinating this process is hard to do manually, a performance testing solution must allow to coordinate machines that create test data automatically.

CHAPTER 3

State of the Art & Related Work

In this chapter we present the state of the art for the functional scope of our work, and approaches that are related to our work. In the first section, we discuss available monitoring and testing solutions for distributed systems and how our work differentiates from them, as well as relevant research that has been conducted in that particular field. In the second section, we describe work that is related to our work.

3.1 State of the Art

The Ganglia distributed monitoring system: design, implementation and experience

Ganglia [43] is an open-source distributed monitoring system for distributed systems, such as grids and clusters. It centrally collects certain metrics, such as CPU usage, memory and process information of workers in a distributed system and allows visualizing collected data. Ganglia is based on a hierarchical design and relies on a multicast-based listen/announce protocol. Figure 3.1 shows Ganglia's basic architecture, which consists of three main components:

- *gmetad* is responsible for federating single monitored workers as well as collecting and aggregating data via a multicast channel.
- *gmond* is responsible for monitoring single workers. It pushes pre-defined monitoring data to a multicast channel observed by *gmetad* instances.
- *gmetric* is an additional command-line tool for pushing application-specific monitoring data to the multicast channel observed by *gmetad* instances.

Additionally, there is a client-side library that provides an API for accessing some of Ganglia's features.

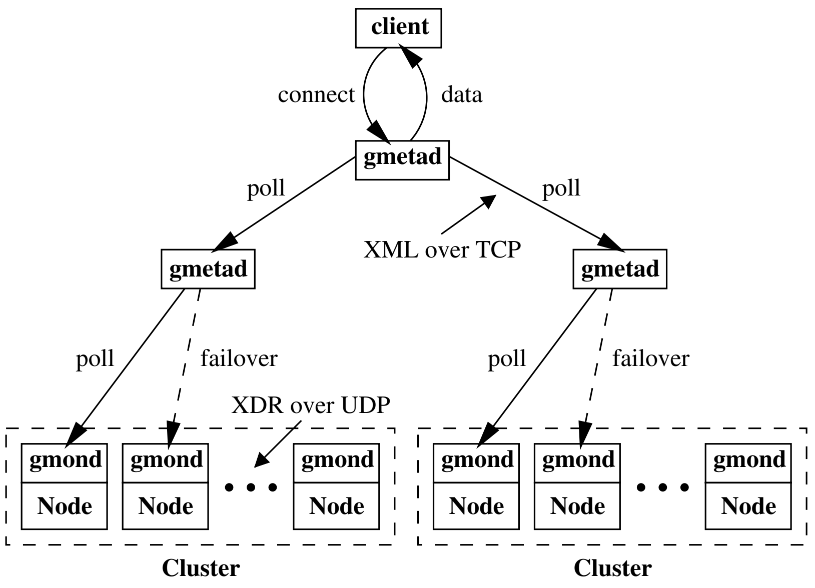


Figure 3.1: Ganglia Architecture [43]

Even though originally designed for collecting operating system related metrics, Ganglia is capable of collecting and processing JVM metrics using JMX via an extension that has been developed for the system [37]. However, Ganglia has its limitations as it is strictly bound to certain metrics, which means that additional metrics, such as the runtime of single process steps in a distributed data-processing application, can not be monitored. Furthermore, Ganglia does not provide the functionality to extend existing applications in order to acquire measurements.

Nagios

Nagios is an open source monitoring software solution, which is mainly built for monitoring network services with the purpose of failure detection [32]. A service can be a host, a network or a service metric, such as process runtime. Nagios consists of the Nagios server and sensors. The Nagios server itself consists of the Nagios core logic component, where all of Nagios main functionality is located, and plugins. A plugin is basically the interface to a sensor, where the data collected by a sensor, is processed to evaluate a state (*OK*, *WARNING*, *CRITICAL* or *UNKNOWN*), which serves Nagios main purpose of failure detection [8].

Figure 3.2 illustrates the Nagios architecture. Given that, since the goal of performance monitoring is to evaluate if a system's or application's performance is appropriate for achieving its functional goals, Nagios can be used for performance monitoring as states derived from performance measurements can indicate whether an application's goals can be achieved or not. However, there is a main difference in how performance monitoring can be achieved using Na-

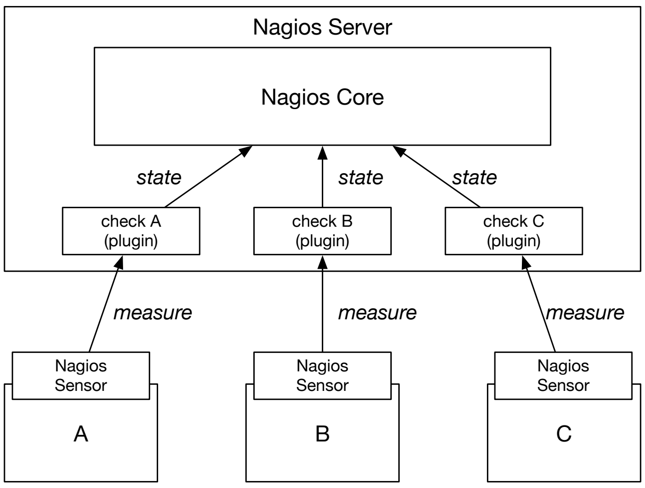


Figure 3.2: Nagios architecture overview [32]

gios. Whereas our work is focussed on collecting and analyzing performance measurements, Nagios is focussed on states, which means that any performance measurement acquired, must be reduced to a state by defining thresholds for metrics. Additionally, even though it is possible to develop custom sensors and additional plugins, a main difference to our work is, that Nagios does not provide any mechanism for extending applications that do not measure any performance metrics by default. However, theoretically it is possible to use measurements obtained by the framework that we will propose later in this thesis, for Nagios by developing a sensor based on the framework, and adding a plugin to Nagios that translated measurements into states.

Monitoring Capabilities of Distributed Data-Processing Frameworks & Systems

Apache Storm

Data-processing engines often provide monitoring functionalities. Apache Storm [64] uses metrics bolts to collect logs and publish metrics [70]. Certain pre-defined built-in metrics can be categorized into system metrics, which consist of measurements such as CPU utilization, memory usage, etc., and topology metrics, which provide some topology statistics such as tuples emitted per minute, tuple acknowledgments, etc. Even though Storm provides an API for adding custom metrics [48], the functionality for acquiring custom metrics would require changes in the application code or alternative ways for adding the measurement functionality as discussed in this thesis. Furthermore, Storm does only log measurements and does not provide any additional distribution or publishing feature for the acquired measurement data.

Apache Spark

Apache Spark [62] uses Metrics [47] for providing performance data of its components. Compared to Apache Storm, Metrics provides features [51] for distributing measured data to various storage mediums such as files, databases or even Ganglia. However, Spark's monitoring capabilities are bound to its engine's components and there is no way of adding additional monitoring functionality to existing applications.

Monitoring Grid Resources: JMX In Action

Balos [7] presents a monitoring solution named JIMS, where JMX resources are integrated in a monitoring architecture based on a service-oriented architecture (SOA) [56]. A three layered architecture as shown in Figure 3.3 is presented.

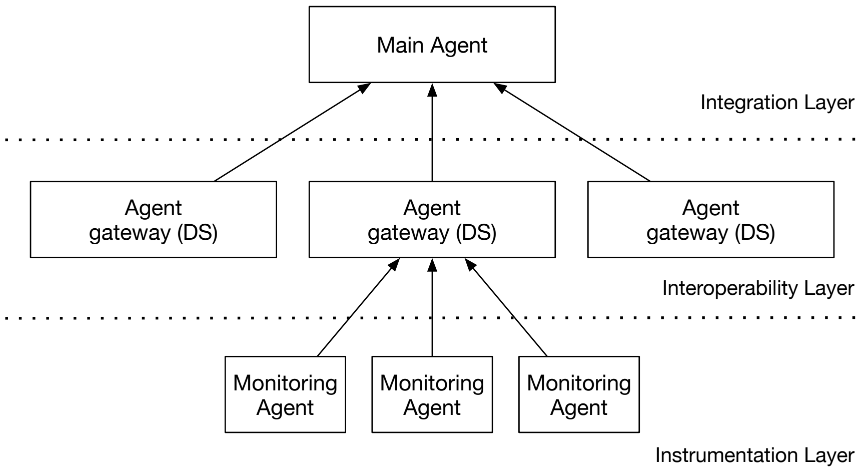


Figure 3.3: JIMS architecture

The instrumentation layer's main responsibility is to acquire monitoring information and manage resources (e.g., by reading and writing attributes of a resource's JMX *MBean* or obtaining information from a locally running SNMP agent). *Monitoring Agents* expose their resources via JMX-RMI to *Agent Gateways* in the interoperability layer.
The interoperability layer consists of *Agent Gateways* that have two major functions. First, an *Agent Gateway* provides access to a *Monitoring Agent's* resources to clients (illustrated as *Main Agent*) via web services. This serves the solution's main goal of integrating JMX resources in a SOA-based environment. The second responsibility of an *Agent Gateway* is auto-discovery of *Monitoring Agents*. In order to achieve that, an *Agent Gateway* sends multicast messages. If a *Monitoring Agent* receives such a message, it will respond with its RMI addresses of JMX connectors. Since the author states that auto-discovery is a key focus, *Agent Gateways* themselves are registered in a UDDI registry for discovery by clients. Additionally, a data warehouse concept with a generic and extensible data model for monitoring data is presented.
Besides the fact, that the focus of the presented solution is the integration of JMX-based monitoring in a SOA environment, there are other major differences to our framework that we want to mention:

1. Transportation Technologies: Depending on the layer, the work presented is strictly bound to JMX or web serves respectively. In our work, we focus on transportation technology independence and show that different transportation technologies can be used.
2. Data Model: The presented data model is generic in order to be extensible. In our work, we define a data model specific to the metrics defined in Section 2.3. Both approaches have their advantages and disadvantages. A generic data model is dynamically extensible, but its attributes do have less semantics. In contrast, a specific data model has to be extended to add new types of measurements, but is easier to analyze since each attribute has a unique semantic.
3. Data Acquisition: The presented work is limited to resource data with pre-defined data sources. Acquiring runtime measurements or adding monitoring functionality to existing applications is not covered by that solution.

A Grid Monitoring Architecture

Tierney et al. [69] describe an exemplary architecture for performance monitoring of computer grids. The architecture consists of three components: A producer, a consumer and a directory service.

- The discovery service is responsible for discovery.
- A producer makes performance data available.
- A consumer receives performance data from a producer.

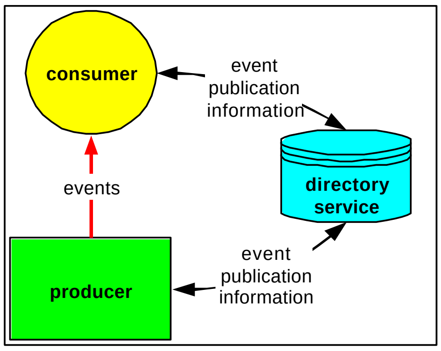


Figure 3.4: Grid Monitoring Architecture [69]

The basic architecture is illustrated in Figure 3.4. The paper focuses on the following requirements a performance monitoring system must satisfy:

- High Data Rate: Performance measurement data might be created at a high rate. The monitoring system must be capable of processing performance measurement data created in high volume and velocity.

- Minimal Measurement Overhead: Measurement-caused impact on monitored applications/systems should be kept minimal.
- Scalable: Since resources in a distributed systems can scale up, the monitoring solution itself must be scalable in order to be able to deal with emerging loads.

The requirements we defined in Section 2.2 coincide with these requirements. Nevertheless, the work presented in this thesis differs to the work discussed in this section. First, the main scope of this thesis work is on the producer. We focus on how to acquire measurement data and integrate measurement functionality in arbitrary existing JVM-based applications. Second, in this thesis we analyze different concepts for satisfying the stated requirements.

MODAClouds Monitoring Platform

Casale [11] presents a platform for monitoring, which goal is to automatically improve quality of service attributes of cloud-based services. The overall solution presented consists of a monitoring platform, a self-adaption platform and an execution platform. A solution overview is shown in Figure 3.5. The monitoring platform is the solution's basis, as it collects and analyzes data required for taking actions to improve quality of service attributes. Since data analytics is not the major scope of this thesis, we will focus on the *Data Collector* component presented in the *MODAClouds monitoring platform*.

A *Data Collector* (DC) is responsible for collecting monitoring data. In contrast to what we define as data acquisition, a *Data Collector*, as described, does usually not create measurement data, it collects data that has already been created by some other component, application or system. Therefore the solution provides collectors for consuming JMX data, reading databases or reading log files. Considering that, a *Data Collector* could theoretically be an interface to a *Publication* component of this thesis framework. However, the author states that there could also be *embedded DCs* for obtaining metrics, which are injected at application level using code injection or aspect-oriented programming. Albeit this approach follows the same principle as this thesis work, the author does not discuss any details on how measurements can be obtained or how the impact of distributing measurement data can be minimized.

A Remote Tracing Facility for Distributed Systems

Ehm et al. [19] describe an architecture for a remote tracing solution. The centerpiece of the solution, named *CMW*, is a log server that processes logs of observed clients (control servers). The log server forwards and archives log messages from an ActiveMQ [1] JMS message queue. The data sources for the described solution could be either C++ or Java applications running on control servers. *Converters* collect logs and publish them to the ActiveMQ JMS queue. The architecture is illustrated in Figure 3.6.

The authors argue that monitoring-caused operation overhead on clients (control servers) should be kept as low as possible and therefore decided use ActiveMQ, a JMS broker implementation, for distributing data collected by *Converters* asynchronously. As we named JMS as a distribution technology in Section 2.5 and include a JMS publication component in our framework's design, as described in Section 4.3, we follow a similar approach. However, besides the

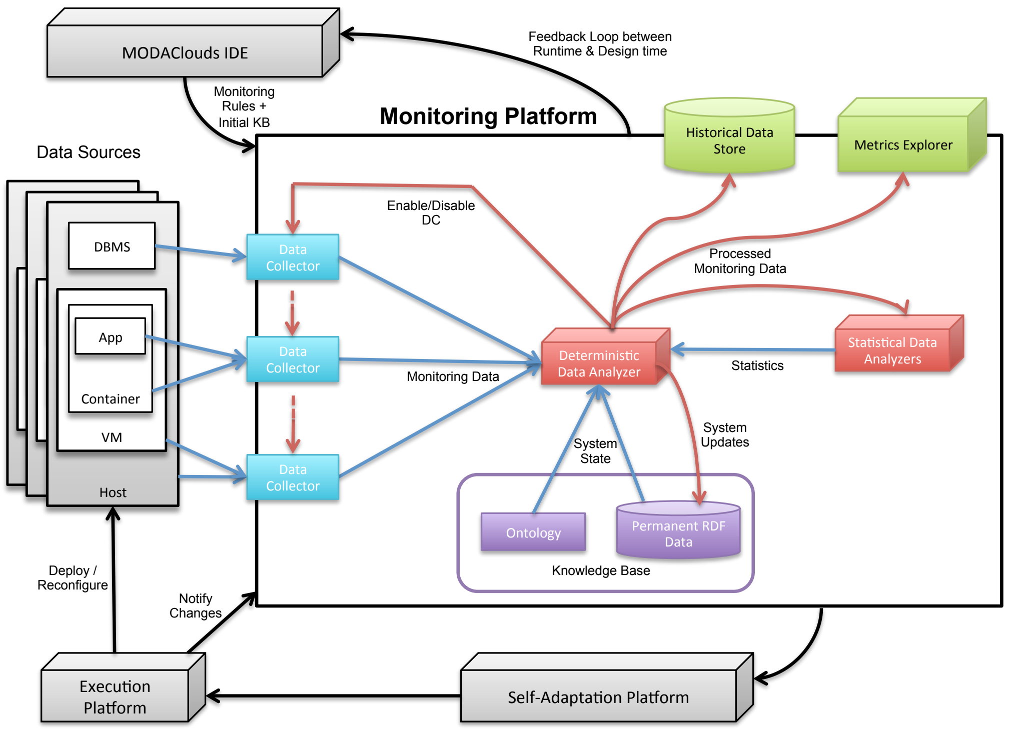


Figure 3.5: MODAClouds Monitoring Architecture [11]

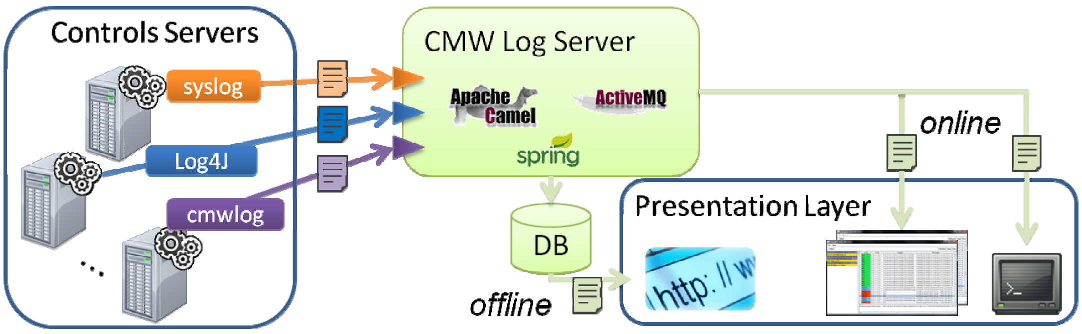


Figure 3.6: CMW Architecture [19]

fact that this thesis focuses on performance measurement data and not log data in general, there are some differences to our work:

1. We propose a universally applicable framework for JVM-based applications that provides interfaces for several persistence and distribution technologies, but does not manage measurement data or provide a user interface. The paper discussed in this section proposes a

fully fledged solution focussed on a certain environment.

2. The framework proposed in this thesis is designed to be integrated in any JVM-based application in order to create monitoring data. *CMW Collectors* only collect data and can only connect with Java applications that use Log4J [27].

Test Architectures for Testing Distributed Systems

Ulrich at el. [71] focus on issues arising when testing distributed systems. The tool that is described in this paper follows a similar approach as the framework proposed in this thesis. However, this paper's tool is constrained at certain points. A library to acquire measurement data for the tool is provided, but it does not provide any functionality to extend systems or applications that are tested. Second, this tool logs monitoring data to local files and does not support additional ways of recording data. Data publication is neither supported by the tool nor discussed in the paper.

3.2 Related Work

A Comparison of AOP based Monitoring Tools

Cojocar [13] compared different available aspect-oriented programming based monitoring tools. He selected InfraRED [33], Glassbox [23], Perf4J [54] and SpringSource AMS [3] and compared them based on several subjects. Some of the selected subjects are directly related to our research:

- **Language Dependency:** As the work presented in this thesis, all discussed monitoring tools are developed for JVM-based applications and use AspectJ.

- **Weaving:** Due to the fact, that all tools use AspectJ, they support both, compile-time and load-time weaving. The author states that load-time weaving is the preferred option for monitoring tools, as it does not require to rebuild monitored applications.

- **Source Code Modification:** As we do in this thesis, the author considers source code modification as an important criteria. The only compared tool that requires source code modifications is Perf4J, as it uses annotations for configuration.

- **Performance Metrics:** It is stated, that all tools support method invocation count and average execution time for method invocations. Some tools support additional performance statistics.

We decided not to use any of the named tools for two reasons. First, all tools focus on aggregated performance measurement data. They apply different aggregation methods within the monitoring components. For our work, we consider aggregations as a step conducted in the analysis phase and do not require such functionality within the monitoring component. Second, more important, none of these tools supports to add sequence identifier and evaluation of sequences, a critical requirement we define in Section 4.1.

JMangler – A Framework for Load-Time Transformation of Java Class Files

In Section 2.6 we concluded that aspect-oriented programming is the best-suited approach for integrating monitoring functionality for two reasons:

1. Aspects, that implement crosscutting concerns, form cohesive modules or classes, and monitoring logic needs not to be repetitive.

2. Depending on the weaving mechanism, monitoring logic can be added at load-time without having to modify the monitored applications sources.

Whereas the first advantage is a unique feature of aspect-oriented programming, the second is more related to the Java platform and could be achieved without using an AOP framework.

Kniesel et al. [40] present JMangler, a Java framework for class loader and JVM independent load-time transformation of Java classes. It supports various transformations, such as adding methods or fields to classes or changing *throws* clauses. In contrast to AspectJ, JMangler does not use Java instrumentation agents [34] [9], but provides a modified class loader, which replaces the default system class loader and performs class file transformations [40].

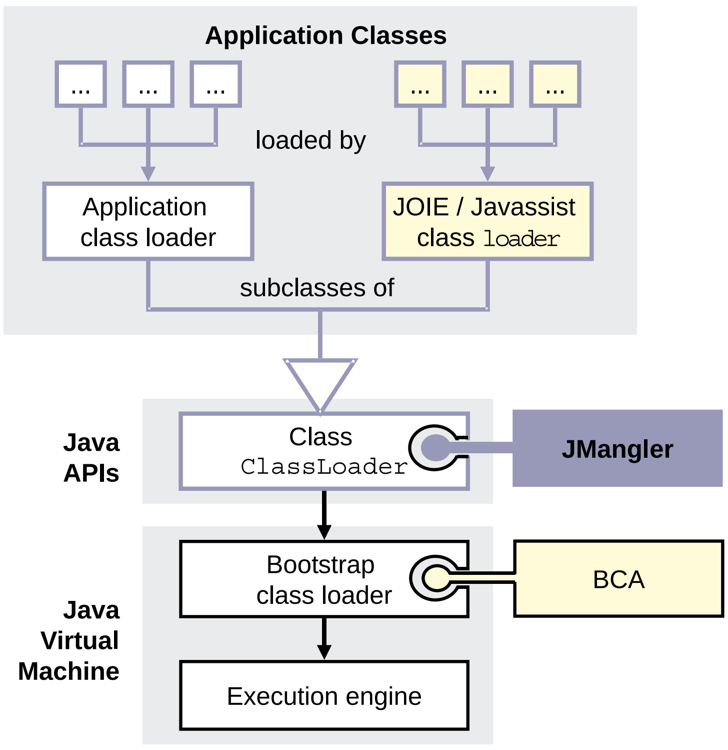


Figure 3.7: JMangler in the Java class loader architecture [40]

Figure 3.7 shows where JMangler is located in the Java class loading architecture.
Given that, most requirements stated in Section 4.1 could theoretically also be implemented

using decorators and JMangler. However, since we consider cohesion as an important criteria for our framework's software design, we concluded that implementing crosscutting concerns using AspectJ serves our purpose better than a decorator/JMangler implementation.

CHAPTER 4

Design

In Chapter 2 we discussed what performance is, how it can be measured, how monitoring logic can be integrated in applications and how monitoring data of a distributed system can be persisted or distributed. In this chapter we will define requirements for a monitoring and testing framework, and propose a design using the information discussed in Chapter 2. First, we define the features and requirements the framework has to fulfill and provide. Second, a domain model that describes entities and their attributes for measuring an application's performance is described. Third, the architecture that describes the required components and their interactions will be discussed. Fourth, to meet a requirement this thesis deals with, a design proposal for integrating or plugging measurement logic into an existing application is made. Finally, an approach for running easily definable scenarios based on an introduced domain-specific language will be discussed.

4.1 Features & Requirements

The features and requirements described in this section are the drivers for the design. The metrics defined in Section 2.3 are the basis for the functional features, whereas the concepts for data acquisition, data publication and integration are incorporated in the proposed design corresponding to the requirements defined in this section.

Runtime Performance Measurement

The first feature is to measure the runtime of a process or a process step. We want to know when (timestamp) processing has been started and finished. The runtime (duration) can be calculated by subtracting the start time from the end time, capacity and throughput can be be calculated if the number of records, start and end date are known (see Section 2.3). Additionally, by acquiring and storing the processing start and end time, it is possible to check network and framework latency between two processing steps or workers, and to compute the overall processing duration by checking the start time of the first step and the end time of the last step. The process step must be identifiable (e.g., by assigning a process step description). Since we are monitoring distributed systems, the worker on which a process is running on, must also be identifiable. As there might be multiple workers for the same process step, we want to be able to identify sequences, in other words the relations between runtime measurements of process steps and nodes.

JVM Profiling

The second feature is to measure a worker's resource utilization. We want to observe the server and JVM metrics discussed in Section 2.3 over time, which means that we want to record these measurements in a configurable interval. Since snapshot values (as system cpu load) are not expressive, we want to collect maximum, minimum and average aggregations for snapshot values in the given interval. Thus, there must be a configurable polling interval, in which snapshot values are obtained that get, at the measurement interval, aggregated.

Process Execution Profiling

The third feature is to profile a worker's resource utilization during the runtime of a particular process. We want to observe the server and JVM metrics discussed in Section 2.3 during the runtime of a process. The resource measurements should be obtained and recorded in a configurable interval.

Data Acquisition & Integration Requirements

In Section 2.4 and Section 2.6 we have discussed how monitoring data can be acquired, how monitoring functionality can be integrated and what the challenges are. However, we have not yet defined any requirements for the framework proposed in this thesis work.
We want that monitoring functionality can be integrated as flexible as possible. Simply measuring method invocation time is not sufficient for the reasons discussed in Section 2.4.
As discussed in Chapter 3, existing monitoring solutions are lacking the capability of adding monitoring functionality post-build time, and in most cases monitoring is not independent from the monitored application or used framework. The monitoring functionality should be entirely independent and it should be possible to add it to any application, without having to modify the application itself.

Data Publication Requirements

In Section 2.2 we discussed that monitoring solutions must be scalable and that the performance impact of the added functionality should affect the origin application as least as possible. Since persisting or distributing performance measurement data might consume a considerable amount of time, it must be possible to store or send data asynchronously. Since we propose a framework, not a fully-fledged monitoring solution, it should provide different persistence and distribution technologies and approaches so that it can be integrated in any environment. There should be interfaces that allow both, post-test and post-execution analysis, and real-time analysis. Given these requirements, we want to support the technologies discussed in Section 2.5 for transferring or storing data:

- **File**: It should be possible to export performance measurement data to CSV files.
- **JDBC**: It should be possible to store performance measurement data to a database directly using JDBC.
- **JMS**: It should be possible to publish performance measurement data to JMS queues or topics.
- **JMX**: It should be possible to monitor performance measurement data via JMX.
- **Log4J**: It should be possible to log performance measurement data using Log4J.

Framework Integration

As discussed in Section 2.1, there are good reasons for building distributed data-processing applications by using frameworks such as Apache Spark Streaming and Apache Storm. This thesis monitoring functionality should be integrable with such frameworks. For the two discussed frameworks, pre-defined integration concepts should be provided.

Performance Test Execution

In Section 2.2 we discussed different approaches for performance testing and certain requirements that must be met when executing performance tests. Considering that, we want that the framework provides an efficient way of running such scenarios. Furthermore, a mechanism for simulating random behavior in order to test unexpected load fluctuations should be available. Since we propose a framework design, we want the scenario mechanism to be flexible and that it can be integrated easily in any environment.

4.2 Domain Model

The domain model shown in Figure 4.1 is the starting point for the design we propose in this thesis.

Node

The central data element is a *Node*. A *Node* is a representation of a worker performing a specific task in a distributed data-processing application. It is identified by the *nodeId* attribute, which is a string. The *nodePurpose* attribute relates to what the node is doing. This allows to group nodes by functionality, but distinguish single nodes. For example, if there is an aggregation operation, which is intense in computation, there might be multiple instances for that operation. In such a case, there will be two or more nodes, which share a common purpose, but have different identifiers.

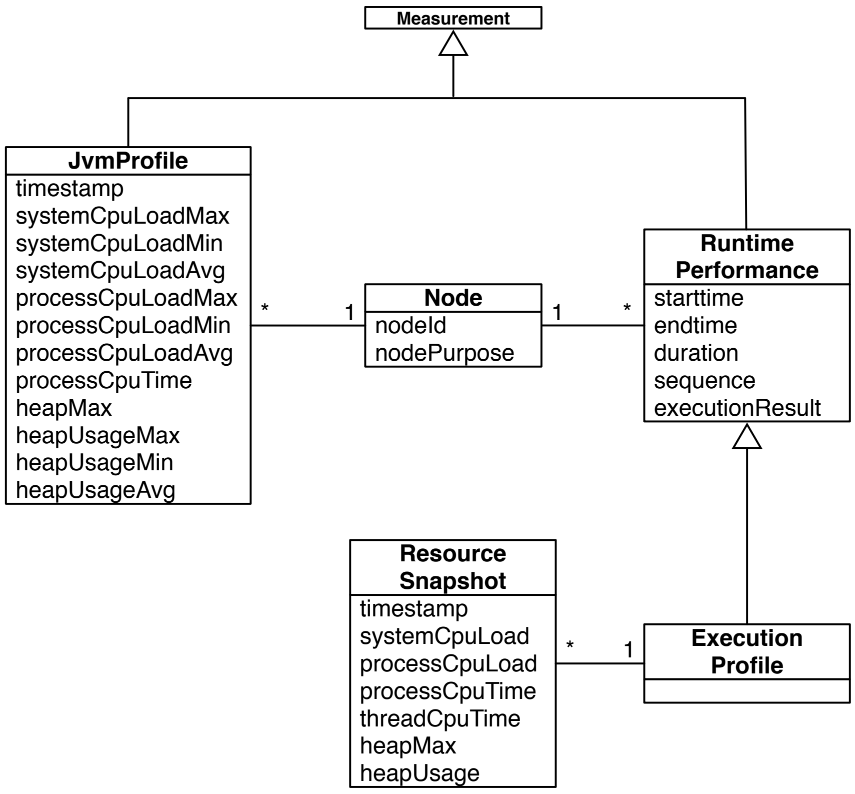


Figure 4.1: Domain model

Runtime Performance

Runtime measurements are modeled as *RuntimePerformance* entities. This entity consists of five attributes:

- *starttime:* The time when execution of the monitored code block started. Stored as unix timestamp.
- *endtime:* The time when execution of the monitored code block finished. Stored as unix timestamp.
- *duration:* Execution duration in nanoseconds.
- *sequence:* If a process consists of multiple steps, a set of *RuntimePerformance* records (linked to different nodes) represents the process. The *sequence* attribute is used to link these single records.
- *executionResult:* For analysis it might be interesting to filter runtime measurements by result (e.g., success or failed).

Jvm Profile

The *JvmProfile* element is used to monitor Java virtual machine resource statistics. Despite the fact, that the element has a single timestamp attribute, it provides aggregated information of the JVMs resources usage over a configured time period.

- *timestamp*: Time at the end of the observed period.
- *systemCpuLoadMax*, *systemCpuLoadMin*, *systemCpuLoadAvg*: Aggregated system CPU load.
- *processCpuLoadMax*, *processCpuLoadMin*, *processCpuLoadAvg*: Aggregated process CPU load.
- *processCpuTime*: CPU time consumed by the process in the observed time period.
- *heapMax*: Maximum heap space available.
- *heapUsageMax*, *heapUsageMin*, *heapUsageAvg*: Aggregated heap space usage.

Execution Profile

An *ExecutionProfile* is an extended *RuntimePerformance* entity, which is related to *ResourceSnapshot* elements. It is used to observe JVM resource usage for the runtime time of a particular monitored code block (the block, which runtime is indicated by the *RuntimePerformance* entity's attributes). *ResourceSnapshot*s are captured in a configurable interval and represent a snapshot at observation time.

Resource Snapshot

A *ResourceSnapshot* provides information of the JVM's resource usage at a point in time:

- *timestamp*: Time when the snapshot has been taken.
- *systemCpuLoad*: System CPU load at snapshot time.
- *processCpuLoad*: Process CPU load at snapshot time.
- *processCpuTime*: CPU time consumed by the process since the last snapshot has been taken.
- *threadCpuTime*: CPU time consumed by the thread since the last snapshot has been taken.
- *heapMax*: Maximum heap space available.
- *heapUsage*: Heap usage at snapshot time.

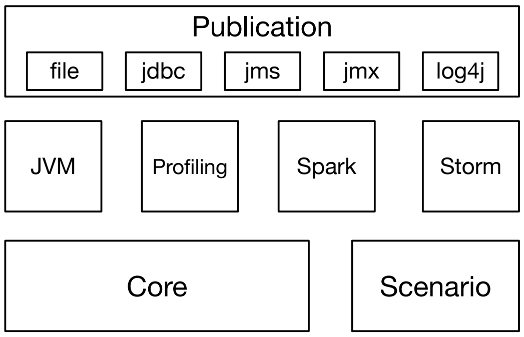


Figure 4.2: Component overview

4.3 Architecture

Component View

Figure 4.2 shows the framework's components. The *Core* component is the centerpiece of the framework's monitoring functionality and is referenced by any other monitoring related component. It contains the domain model's classes, a factory to instantiate *Node* objects, the basic runtime performance measurement functionality, the interface definition for the *PublicationService*, an abstraction for transferring or storing *Measurements*, a default implementation of the *PublicationService* that writes measurements to *System.out*, and provides implementations of different parallelization patterns.
The *JVM* component contains all classes for monitoring the JVM's resources. If JVM monitoring is not required, the framework can be deployed without the JVM component.
The *Profiling* component's responsibility is to provide the functionality required for profiling JVM resources for a particular code block, thus to create *ExecutionProfiles*.
The *Spark* and *Storm* components provide an out-of-the-box integration with Apache Spark and Apache Storm respectively, that we decided to develop for this thesis. However, additional blocks for other frameworks, processing engines, platforms or applications can be added easily by using the provided features of the *Core* component, as we did for the *Spark* and *Storm* components.
There exists no *Publication* component as shown in Figure 4.2. The block simply groups all publication components by way of illustration. The single components within that group provide functionality to log, persist or distribute *Measurements* according to their names (File, JDBC, JMS, JMX, Log4J). The *Scenario* component provides testing functionality. It contains interface definitions to run testing scenarios. In order to define scenarios in a simple and effective way, this component also provides an interpreter for a domain-specific language designed for running scenarios.

Target Application Interaction

Earlier in this chapter, we defined the requirements for data acquisition and integration. There is one requirement that must be considered when thinking about how the framework should interact with the monitored target application. It is stated that integration must be as flexible as possible and that simply measuring method invocation is not sufficient. Considering the analysis of how data can be acquired and options for integration discussed in Section 2.6, only the last option, aspect-oriented programming, is a viable option for meeting this requirement.
In general, aspects that acquire performance measurements are woven into the target application. These aspects use functionality provided by the framework's core package and publish the measurement data using one or multiple publication components.

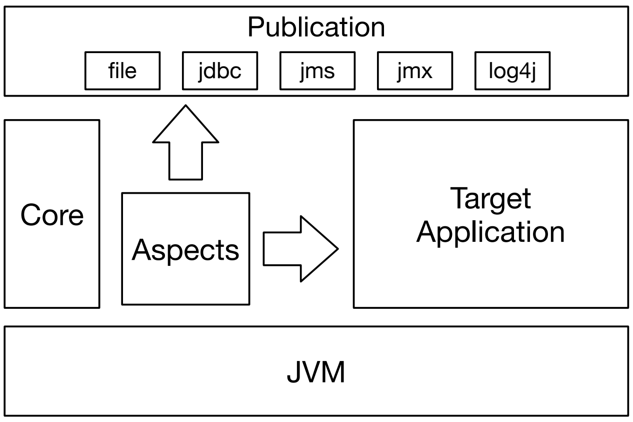


Figure 4.3: Architecture overview

Figure 4.3 illustrates how the framework interacts with the target application. Once an aspect has been woven into the target application, the advised code then interacts with the framework's provided functionality when executed. In Figure 4.4 the interaction is illustrated in detail. A class called *RuntimeAspect* is woven around a target's method. When a caller supposedly calls the target's method, the call is intercepted by the *RuntimeAspect* and the advised runtime measurement code can be executed before and after the target code is executed. The obtained measurement is then published by using the *PublicationService* interface provided by the framework's core component.

Integration

In Section 4.1 we defined the requirements that monitoring functionality should be independent from the application itself, and that it should be possible to add the functionality without having to modify the application.
The first requirement is fulfilled by the decision to use aspect-oriented programming techniques, as the advices that acquire the monitoring data are independent code blocks that can be applied to any pointcut. The second requirement is also fulfilled by using aspect-oriented programming

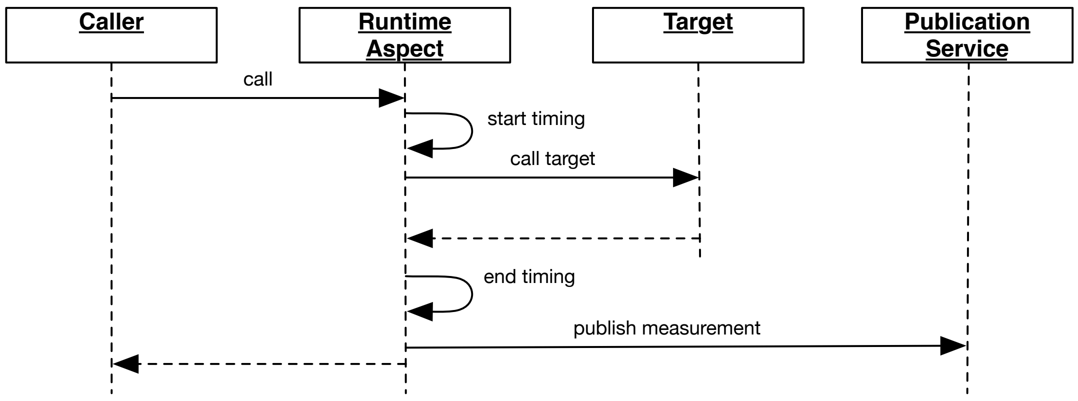


Figure 4.4: Aspect and Publication service overview

techniques, but there are additional issues that must be considered. Using compile-time source code weaving does not require to modify an application's source code, however, that requires to posses the source code and being able to build the application. The fact, that this is often not the case, leaves two options for integrating the monitoring functionality:

- Compile-Time Binary Weaving
- Load-Time Weaving.

Compile-time binary weaving is done only once and does not slow down application starting time. Load-time weaving, on the other hand, is more flexible as it is simple to remove single or all aspects by restarting the application.

Measurement Acquisition

Measurement data is acquired by aspects, or to be more precise, by the advised code. Depending on the measurement to be acquired, data acquisition is executed differently.

Runtime Measurements

For *RuntimeMeasurement* records, the basic process to obtain measurement data consists of three steps: Start timing before the monitored code block, stop timing after the monitored code block and publish the data. Considering aspect-oriented programming design principles, there are two different approaches of how this can be done:

1. Using *Around* advices: In this case an entire method is wrapped around a pointcut. This process is shown in Figure 4.4. This approach only works if the measured process is reflected by a single method.

2. Using *Before* and *After* advices: In the second case there are two advised methods that are invoked. Figure 4.5 illustrates this approach. The starting time taken in the Before advised code, must be made available for *After* advised code. Since a join point can be reached by multiple threads simultaneously, concurrency must be considered for this approach.

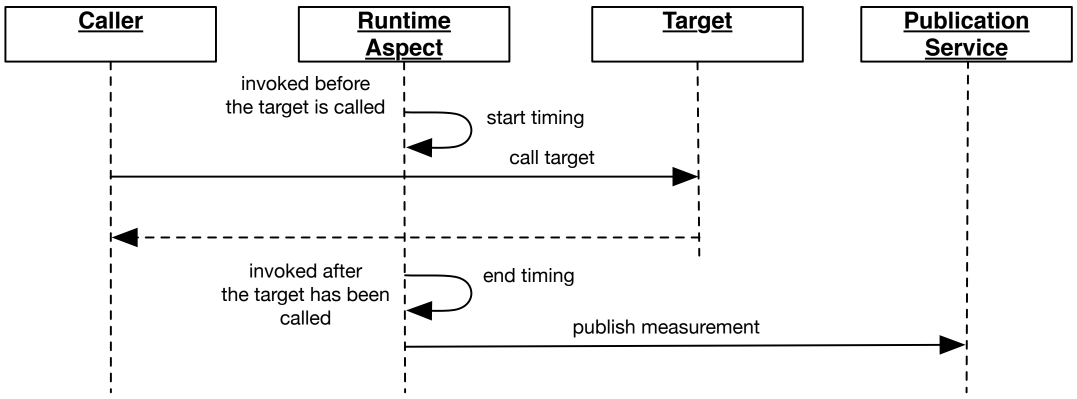


Figure 4.5: Before/After sequence

Execution Profiles

ExecutionProfile records are taken in background while the monitored code block is executed. The advised code must start a separate profiling thread for obtaining resource snapshots and stop the thread after the code block's execution has finished. Finally, the advised code has to publish the measurement containing the resource snapshots. The process is illustrated in Figure 4.6

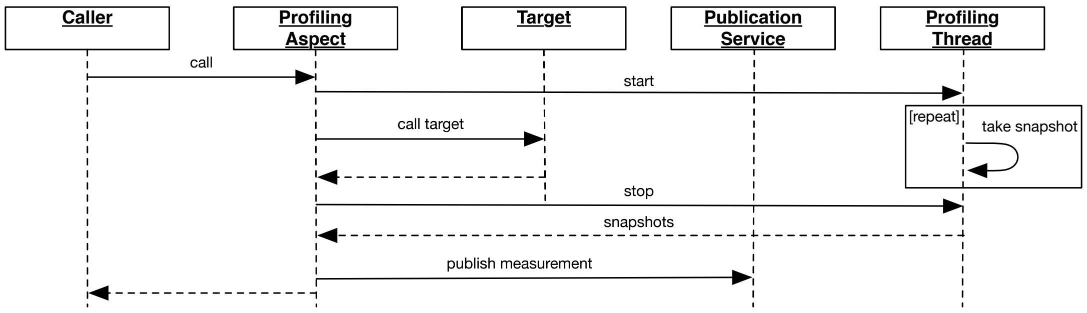


Figure 4.6: Process Profiling

JVM Profiles

JvmProfile records are taken in a configurable interval in background once JVM profiling has been started. Hence, the aspect's responsibility is to start profiling. A join point that is only triggered once at application start should be selected.

Node Assignment

The domain model described in Section 4.2 shows that each taken measurement is associated with a *Node*. In order to create a *Node* instance for an aspect that acquires measurements, the framework provides a node factory. This *Node* factory contains three different methods to create nodes:

- *getNode()*: This method returns a *Node* configured for the running application.
- *getNode(id)*: This method returns a *Node* for a certain identifier. The identifier can be any string (e.g., the name of an aspect).
- *getNode(object)*: This methods returns a *Node* using the given object's canonical class name as identifier. This method allows that measurement aspects, which are used for different join points, can still be related to different *Nodes* (e.g., by using the aspect's target object as argument for *getNode(object)*.

Measurement Publication

In Section 2.5 we discussed different data publication forms. Earlier in this chapter, we stated that the framework proposed in this thesis, must support these different forms since it is not a closed fully-fledged monitoring solution since topics as data visualization and analysis features are not covered by the framework. An interface *PublicationService* that defines a *publish* method, which accepts any *Measurement* is part of the framework's core component. A publication component must have an implementation of this interface so that it can be used by the framework. For the required publication forms, components that contain a *PublicationService* implementation are provided. An overview of the publication components and their *PublicationService* implementation is shown in Figure 4.7.

In order to use a publication component, the binaries must be added to the class path and the framework configured to integrate the *PublicationService* implementation. Multiple implementations can be combined through a provided *PublicationService* implementation that forwards measurements to all configured concrete *PublicationService* instances.

By default, publication is done synchronously. As a result of our analysis discussed in Section 2.5, we know that synchronous publication can have a considerable impact on an application's performance. Since we stated that the performance impact should be kept minimal in Section 4.1, the framework supports different asynchronous publication modes.

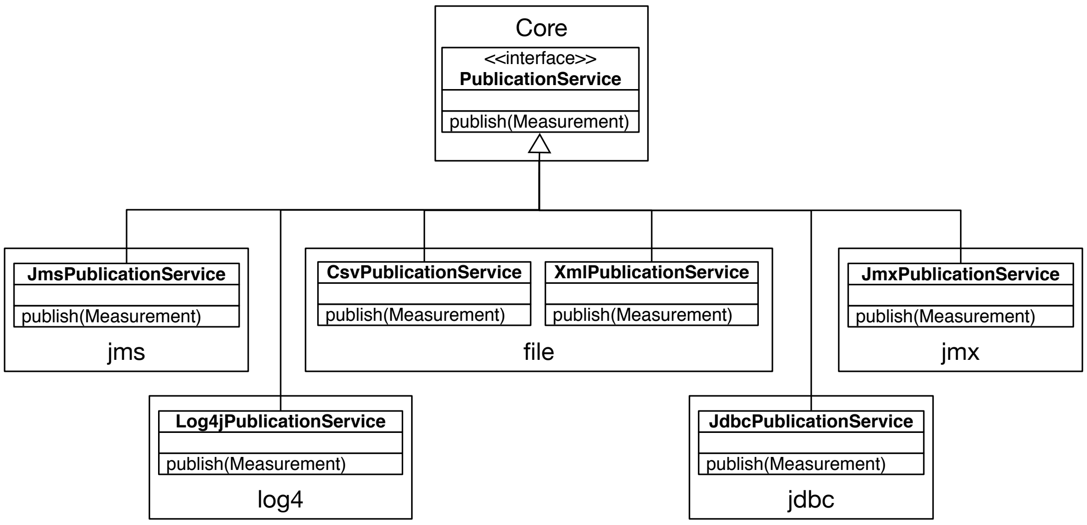


Figure 4.7: Publication Services

Asynchronous Publication Modes

Publication modes are implemented as proxies, as described Section 2.6. There is a *Publication-Service* implementation for each mode, which has its own implementation of the *publish* method where the mode's behavior is added before delegating the origin *publish* method call.

Thread-per-Task A thread-per-task approach, as described in Section 2.5 is supported. Figure 4.8 shows that when the *PublicationService* is called, the method call is intercepted by a proxy, which creates a new thread that calls the delegate object's method.

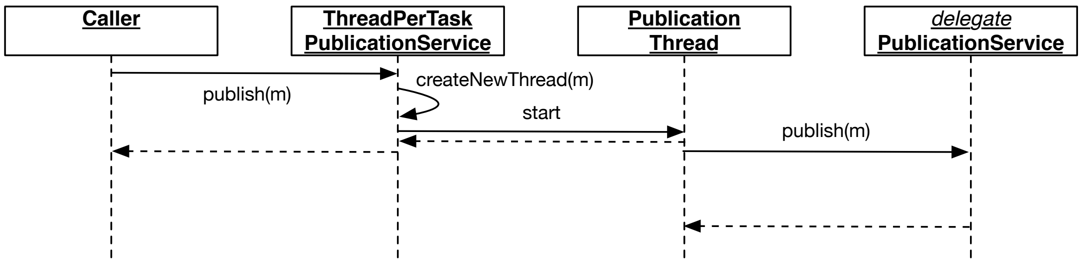


Figure 4.8: Thread-per-task publication

Thread Pool A *PublicationService* implementation for thread pools, which uses the Java *ThreadPoolExecutor* [68] implementation is illustrated in Figure 4.9. The *publish* method call is intercepted by a proxy *PublicationService*, which creates a task and queues the task using the *ThreadPoolExecutor*. The *ThreadPoolExecutor's* threads poll the task queue and execute the tasks.

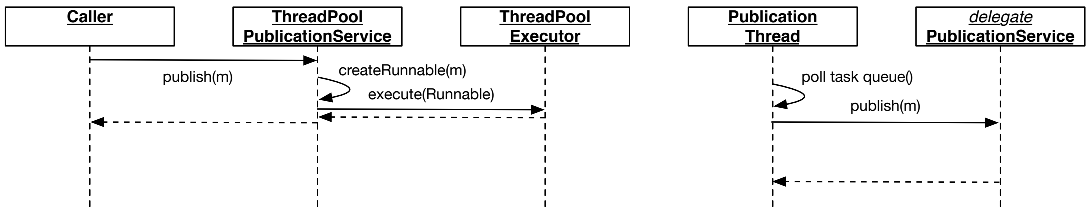


Figure 4.9: Thread pool publication

Producer/Consumer A *PublicationService* for publishing measurements asynchronously following a producer/consumer pattern is shown in Figure 4.10. The main difference to the thread pool implementation is, that measurements instead of tasks are enqueued.

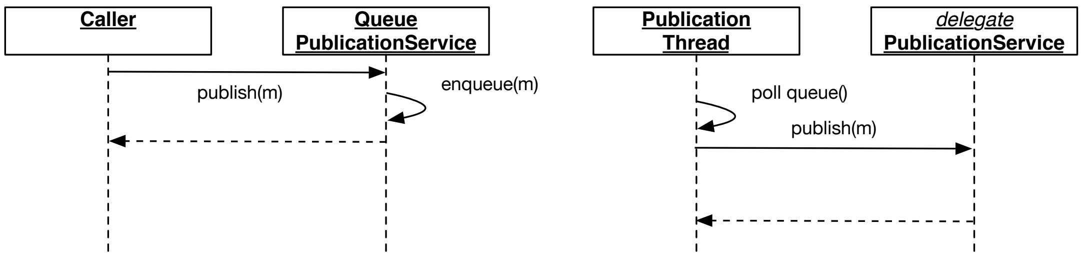


Figure 4.10: Producer/Consumer publication

4.4 Framework Configuration

Since the framework's functionality has to be independent from the target application, it must also be independently configurable. Therefore the framework provides a *Configuration* class, which has a static *getProperty(String propertyName)* method. When invoked for the first time, a lookup for a properties file in the Java class path is performed. If a properties file is found, the properties will be loaded and kept in memory. After loading the properties from the properties file, a check if a property with the given name has been loaded will be conducted. If found, the property value will be returned. If no property has been found with the given name, a check if a Java system property with the given name exists, will be performed. If a system property exists, the value of the system property will be returned, if there is no such system property, the method will return *null*. The methods general process steps are illustrated in Figure 4.11.

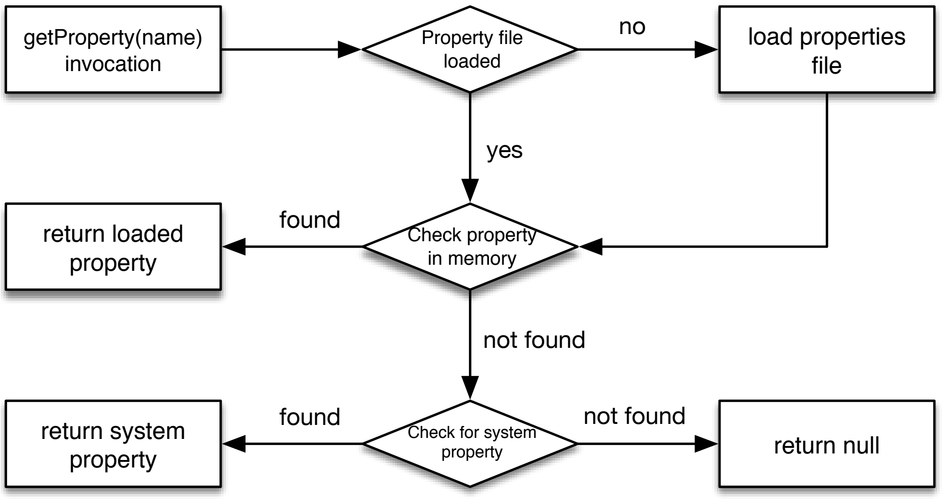


Figure 4.11: Configuration loading process

4.5 Scenarios & DSL

In Section 4.1 we stated that the framework should provide a mechanism for running performance tests in an efficient and flexible way. In this section, we present such a mechanism that allows to create and run testing scenarios. We separate it into two different blocks:

1. The functionality required for running and integrating scenarios.
2. A domain-specific language to define scenarios efficiently.

The centerpiece of the mechanism, as shown in Figure 4.12, is the *Scenario* interface, which defines a *run*, method where two arguments are expected: a *Factory* and a *PopulationService*. The *Factory* is the source for generating test data objects. There are no limitations of what a factory might do (e.g., querying test data from a database or simply generate random data). The *PopulationService* acts as the interface to the data-processing application to be tested. A *Scenario* implementation uses the factory to create test data and the population service to populate the created test data to the application. This process is illustrated in Figure 4.13

The framework provides four basic *Scenario* implementations out of the box:

- **RunOnceScenario:** A run once scenario is the simplest case. The sequence illustrated in Figure 4.13 is executed exactly once.
- **ListScenario:** A list scenario takes a list of scenario instances as constructor argument. When executed, all scenarios in the given list will be executed in the list order.
- **LoopScenario:** A loop scenario can be initialized in four different ways:

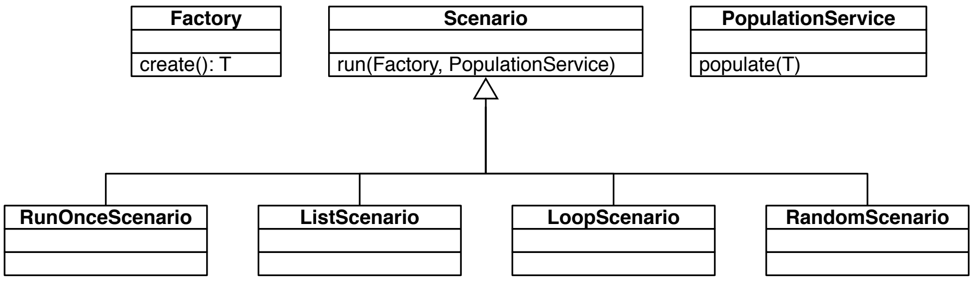


Figure 4.12: Scenario classes

1. Only with a *loop count*: The scenario's run method will be invoked *loop count* times.
2. With a *loop count* and a pause time: The scenario's run method will be invoked *loop count* times, where execution is paused for the given time after each invocation.
3. With a *loop count* and a scenario: The given scenario's *run* method will be invoked *loop count* times.
4. With a *loop count*, a pause time and a scenario: The given scenario's *run* method will be invoked *loop count* times, where execution is paused for the given time after each invocation.

- **RandomScenario:** A random scenario takes a list of scenarios instances as constructor argument and when its *run* method is called, a random scenario instance of the given list will be chosen and executed.

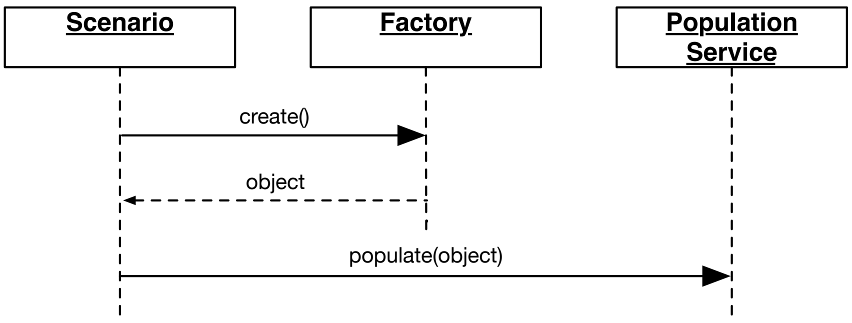


Figure 4.13: Scenario sequence

Domain-Specific Language

For initializing scenarios easily, we specified a domain-specific language for creating scenarios. A parser that accepts the DSL's expression is provided. It receives expressions as input and evaluates them. The parser's evaluation result is a *Scenario* instance that can then be used to run scenarios. This process is illustrated in Figure 4.14.

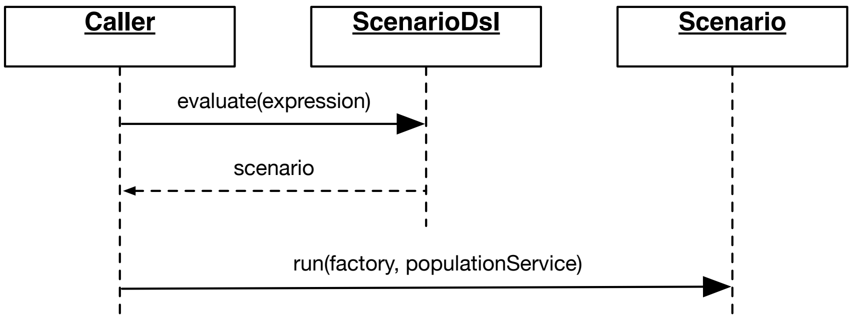


Figure 4.14: Scenario DSL sequence

Listing 4.1 shows an EBNF [74] definition of the expressions that are accepted by the provided parser.

```
loop statement = "loop" ;
once statement = "once" ;
list statement = "list" ;
random statement = "random" ;

braceleft =  "(" ;
braceright= ")" ;
comma = "," ;

number = digit, { digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

once = once statement, braceleft, braceright ;

list = list statement, braceleft, expression, {comma, expression},
   braceright ;

random = random statement, braceleft, expression, {comma, expression
   }, braceright ;

loop =  ( loop statement, braceleft, number, braceright ) |
                ( loop statement, braceleft, number, comma, number,
                   braceright ) |
                ( loop statement, braceleft, expression, comma,
                   number, braceright ) |
                ( loop statement, braceleft, expression, comma,
                   number, comma, number, braceright ) ;
```

```
expression = once | list | random | loop ;
```

Listing 4.1: DSL Definition in EBNF

There are four basic statements that can form an expression: *once, list, random* and *loop*, which return a *Scenario* implementation as described earlier correspondingly. *loop, list* and *random* take arguments corresponding to their classes constructors for initialization. The *once* statement is followed by an opening and closing brace, thus does not take any arguments.

CHAPTER 5

Implementation

In this chapter we will discuss the implementation specifics of our proof-of-concept framework that shows the feasibility of the design proposed in Chapter 4. First, some general implementation specifics are defined. Second and third, the details of the proof-of-concept framework integration with Apache Spark and Storm respectively are discussed. Finally, the implementation specifics of the proposed publication services are presented.

5.1 General

Aspect-Oriented Programming

The first thing we had to decide for the implementation was which aspect-oriented programming framework to use. In Section 2.6 we discussed AspectJ, the most advanced aspect-oriented programming implementation. Related to AspectJ, we made the following decisions for the proof-of-concept implementation:

1. **Weaving:** We decided to use load-time weaving using the AspectJ Java agent. This means, that applications start scripts have to add the Java agent as argument and that an *aop.xml* file has to be added to *META-INF* directory of the class path.

2. **Programming Style:** For our implementation, we used the annotation based AspectJ programming style. In combination with load-time weaving, this is the more appealing programming style since the AspectJ compiler is not required, and compiling the framework is more simple as it is only a one step process.

Pre-Defined Aspects

The framework implementation not only contains modules that support measurement acquisition as described in Section 4.3, but also abstract aspects that can be used by defining concrete pointcuts.

Runtime Performance

We implemented two different abstract aspects for measuring runtime performance. The first implementation is using *around* advices, where one pointcut has to be defined. The invocation time of the join point's target is measured. A stub of this aspect is shown in Listing 5.1. An abstract pointcut, which has to be defined when using this abstract aspect, is used for advising the monitoring code. The advice continues the execution at the given join point and measures invocation time. The advice *around* method's second argument, the object where the advice is weaved into, is used for creating a *Node* instance as described in Section 4.3.

```
@Aspect
public abstract class AbstractRuntimePerformanceAspect {
    @Pointcut
    public abstract void scope();

    @Around("scope()␣&&␣this(jpo)")
    public Object around(ProceedingJoinPoint pjp, Object jpo) throws
        Throwable {
        ...
    }
}
```

Listing 5.1: Runtime Performance Aspect

The abstract aspect can be used in an *aop.xml* AspectJ load-time weaving configuration file, where the pointcut for the aspect is defined. Listing 5.2 shows an *aop.xml* example where this abstract aspect is used.

```
<aspectj>
    <aspects>
        <concrete-aspect name="at.ac.tuwien.infosys.rosebery.storm.
            aspect.BoltRuntimePerformanceAspect" extends="at.ac.tuwien
            .infosys.rosebery.common.aspect.
            AbstractRuntimePerformanceAspect">
             <pointcut name="scope" expression="execution(*␣backtype.
                storm.topology.IRichBolt.execute(..))" />
        </concrete-aspect>
    </aspects>
</aspectj>
```

Listing 5.2: *aop.xml* example

A *concrete-aspect* element defines a new aspect. A new name must be given to the aspect and the *extends* attribute defines the abstract aspect. The abstract aspect's expected pointcuts can be set using the *pointcut* element.
The second implementation of the *Runtime Performance* aspect is using *before* and *after* advices. This aspect expects two pointcuts, one for defining the start of a code block to be monitored, and one that defines the end of that code block. Runtime measurement starts before the first and ends after the second pointcut.

```
@Aspect
```

```
public abstract class AbstractStartStopRuntimePerformanceAspect {

    @Pointcut
    public abstract void startScope(Object jpo);

    @Pointcut
    public abstract void stopScope();

    @Before("startScope(jpo)")
    public void start(Object jpo) {
        ...
    }


    @AfterReturning(pointcut = "stopScope()", returning = "result")
    public void stop(Object result) {
        ...
    }

    @AfterThrowing(pointcut = "stopScope()", throwing = "throwable")
    public void stop(Throwable throwable) {
        ...
    }
}
```

Listing 5.3: Before/After Runtime Performance Aspect

Listing 5.3 shows the abstract aspect implemented for measuring runtime using *before* and *after* advices. For using this implementation, we have to consider how the *before* and *after* advices are joined. By default, aspects are instantiated as singletons [41]. This means, that simply storing the start timestamp in a local member variable of the aspect class could result in wrong measurement data if there are multiple threads. We decided to join the advices using variables, which are bound to threads. This results in the precondition for the usage of this aspect, that the monitored code block has to start and end within the same thread.

Sequencing In Section 4.1 we defined the requirement that it should be possible to relate runtime measurements of single process steps. In other words, to establish a sequence of measurements. To enable that, there must be an identifier for a sequence, which must be passed on from process step to process step. Since we can not assume that data passed on within a data-processing application does carry such a sequence identifier, and if, that there is no generally applicable approach for extracting it, we developed a feature for adding and passing on sequence identifiers within an application. This feature consists of two parts:

- A static crosscutting advice to add a sequence identifiers to data objects used by the data-processing applications.
- Two dynamic crosscutting aspects that create and pass on sequences.

Static Crosscutting Listing 5.4 shows the interface definition for sequenced objects. This interface declares a getter and setter method for a sequence attribute. Using AspectJ's functionality for static crosscutting, to change an object's structure, we can add this interface to any object. Since this also requires an implementation of the interface, we provided a default implementation as shown in Listing 5.5.

```
public interface SequencedObject extends Serializable {
    public String getSequence();
    public void setSequence(String sequence);
}
```

Listing 5.4: SequencedObject interface

```
public class SequencedObjectImpl implements SequencedObject {
    private String sequence;

    @Override
    public String getSequence() {
        return sequence;
    }

    @Override
    public void setSequence(String sequence) {
        this.sequence = sequence;
    }
}
```

Listing 5.5: SequencedObject implementation

Using AspectJ's *@DeclareMixin* annotation we can add this interface to any Java object [41]. Listing 5.6 shows an example of how the interface and the default implementation can be added to an object using *@DeclareMixin*.

```
@Aspect
public class SequencedTupleAspect{
    @DeclareMixin("at.ac.tuwien.thesis.Tuple")
    public static SequencedObject createSequencedTuple() {
        return new SequencedObjectImpl();
    }
}
```

Listing 5.6: Aspect for adding a sequence to an object

Dynamic Crosscutting Adding a sequence attribute to objects passed on within an application is only the prerequisite for the actual sequencing functionality. Sequences must be created and passed on automatically. We implemented two abstract aspects, one for creating sequences, one for passing on sequences. Listing 5.7 shows the abstract aspect for creating a sequence. It should be given a pointcut that reflects the beginning of a data-processing application's processing steps before the first runtime measurement is taken.

```
@Aspect
public abstract class CreateSequenceAspect {
    @Pointcut
    public abstract void created(Object o);

    public void createSequence(Object o) {
        if (o instanceof SequencedObject) {
            ((SequencedObject)o).setSequence(UUID.randomUUID().
               toString());
        }
    }
}
```

Listing 5.7: Aspect for creating sequences

Once a sequence is created, it must be passed on by each processing step. Since the monitored application is not aware of sequences, this has to be done by an aspect. Listing 5.8 shows an abstract aspect for passing sequences on. Since we can not tell upfront how the code for passing sequences on has to be advised, we have not declared the *assignSequence* method as an advice, because this has to be done case by case.

```
@Aspect
public abstract class SequencePassOnAspect {

    @Pointcut
    public abstract void finished(Object in, Object out);

    public void assignSequence(Object in, Object out) {
        if (in instanceof SequencedObject) {
            if (out instanceof SequencedObject) {
                ((SequencedObject)out).setSequence(((SequencedObject)
                   in).getSequence());
            }
        }
    }
}
```

Listing 5.8: Aspect for passing sequences on

However, we implemented another abstract aspect for the common case, where an object given as argument is the input carrying the sequence and the object returned by a method is the result, where the sequence has to be set. This abstract aspect is shown in Listing 5.9.

```
@Aspect
public abstract class AfterReturningSequencePassOnAspect extends
   SequencePassOnAspect {

    @AfterReturning(value = "finished(in,␣tmp)", returning = "out")
    public void assignSequence(Object in, Object tmp, Object out) {
        super.assignSequence(in, out);
    }
}
```

Listing 5.9: Aspect for passing sequences on after a result has been returned

This approach with static and dynamic crosscutting has one major limitation. Where as it works well for non-standard Java objects, it can not be used for adding sequences to standard Java data types (e.g., String, Double, List, etc.) used within an application, since these classes can not be modified by AspectJ.

JVM Profiling

For JVM profiling, we implemented an abstract aspect, which requires a pointcut for profiling start. As described in Section 4.3, the pointcut should ideally refer to a join point that is invoked only once. Listing 5.10 shows the abstract aspect.

```
@Aspect
public abstract class JvmProfilingAspect {
    @Pointcut
    public abstract void scope();

    @Before("scope()")
    public void beforeScope() {
        try {
            Class.forName("at.ac.tuwien.infosys.rosebery.jvm.
                profiling.ProfilingThread");
        } catch(ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Listing 5.10: Aspect JVM profiling

The advised code only loads a class that is responsible for acquiring JVM measurements. The loaded class is a *Runnable*, which holds a static reference to an instance of itself, which is initialized when the class is loaded. In its constructor, a new thread for polling resource data and a thread for data aggregation, as described in Section 2.4, are started. This ensures that, even if the advised code is executed more than once, only one profiling thread is started. Listing 5.11 shows a class stub with the major parts as described in this paragraph.

```
public class ProfilingThread implements Runnable {

    private static final ProfilingThread instance = new
       ProfilingThread();

    ...

    public ProfilingThread() {
        ...
        pollingRunnable = new PollingRunnable(pollingInterval);
        Thread pollingThread = new Thread(pollingRunnable);
        pollingThread.setDaemon(true);
        pollingThread.start();


        ...

        Thread thread = new Thread(this);
        thread.setDaemon(true);
        thread.start();
    }

    ...
}
```

Listing 5.11: JVM profiling thread

Execution Profiling

Execution profiling works similarly to runtime performance measurement. The only difference is that a profiling thread, which acquires resource snapshots for the observed time has to be started and stopped.

```
@Aspect
public abstract class AbstractProfilingAspect {

        ...

    @Pointcut
    public abstract void scope();


    @Around("scope()␣&&␣this(jpo)")
    public Object around(ProceedingJoinPoint pjp, Object jpo) throws
       Throwable {
        ProfilingRunnable profilingThread = startProfiling(Thread.
           currentThread().getId(), interval);

        ExecutionProfile ep = new ExecutionProfile();

                ...

        ep.setSnapshots(stopProfiling(profilingThread));

                ...
    }
    private ProfilingRunnable startProfiling(long threadId, long
        interval) {
        ProfilingRunnable runnable = new ProfilingRunnable();
        runnable.setThreadId(threadId);
        runnable.setInterval(interval);
        executor.execute(runnable);
        return runnable;
    }

    private Set<ResourceSnapshot> stopProfiling(ProfilingRunnable
        runnable) {
        runnable.interrupt();
        return runnable.getResult();
    }
}
```

Listing 5.12: Execution Profiling Aspect

Listing 5.12 shows the aspect for profiling resources during the execution of a process. Compared to runtime measurement, as shown in Listing 5.1, there is additional functionality, which starts and stops a profiling thread.

5.2 Apache Spark

In Section 2.4 we discussed the API of Apache Spark Streaming's programming model and how performance measurement data can be acquired in Spark Streaming in general. In this section, we present aspects for creating and passing on sequences and for measuring runtime in Spark Streaming's programming model.

Sequencing

Sequence Creation

To enable sequencing, sequence identifiers have to be created. In Section 2.4 we described that Spark Streaming provides a *Receiver* class, which contains methods for sending data to Spark Streaming's engine. To create a sequence, we apply the *CreateSequenceAspect* as described in Section 5.1 to *Receivers*.

```
@Aspect
public class ReceiverCreateSequenceAspect extends
    CreateSequenceAspect {

    @Override
    @Pointcut("call(*␣org.apache.spark.streaming.receiver.Receiver.
        store(java.lang.Object))␣&&␣args(o)")
    public void created(Object o) {}

    @Before("created(o)")
    public void createSequence(Object o)  {
        super.createSequence(o);
    }
}
```

Listing 5.13: CreateSequenceAspect for Spark Streaming *Receivers*

Listing 5.13 shows the implementation of a *CreateSequenceAspect* for Spark Streaming. A pointcut for a *Receiver's store* method is defined, and the sequence identifier is assigned before the *store* method is called.
What has to be considered is, as described in Section 5.1, that the object sent to Spark by the *store* method, must be an implementation of the described *SequencedObject* interface. Hence, the static crosscutting aspect using the *@DeclareMixin* annotation, as for the tuple example in Section 5.1, must be applied.

Passing Sequences On

For passing sequence identifiers on, we must distinguish between *Functions*, used for map functions and output operations, *FlatMapFunctions* used for flat map functions, and *Function2* implementations, used for reduce functions.

Functions *Functions* are used for map functions and output operations. Sequence identifiers can only passed on when *Functions* are used for mapping, where one object is transformed into another. When a *Function* is used as output operation, the return type is typically *java.lang.Void*, and thus no sequence is passed on, since *java.lang.Void* objects are not instances of *SequencedObject*. As a *Function's call* method takes one object as argument and simply returns a result object, no specifics have to be considered and we simply implemented the abstract *AfterReturningSequencePassOnAspect* for *Functions*.

```
@Aspect
public class FunctionSequencePassOnAspect extends
    AfterReturningSequencePassOnAspect {

    @Override
    @Pointcut("execution(*␣org.apache.spark.api.java.function.
        Function.call(java.lang.Object))␣&&␣args(in)␣&&␣this(out)")
    public void finished(Object in, Object out) {}

    @Override
    @AfterReturning(value = "finished(in,␣tmp)", returning = "out")
    public void assignSequence(Object in, Object tmp, Object out) {
        super.assignSequence(in, out);
    }
}
```

Listing 5.14: AfterReturningSequencePassOnAspect for Spark Streaming *Functions*

Listing 5.14 shows an *AfterReturningSequencePassOnAspect* implementation used for *Functions*.

FlatMapFunctions *FlatMapFunctions* must be treated different to normal *Functions*, since the result of their *call* method is not a single object, but an iterable collection of objects, where the sequence identifier has to be assigned to each object in the collection, which can be seen in Listing 5.15.

```
@Aspect
public class FlatMapFunctionSequencePassOnAspect extends
    SequencePassOnAspect {

     @Override
     @Pointcut("execution(*␣org.apache.spark.api.java.function.
         Function2.call(java.lang.Object,java.lang.Object))␣&&␣args(in)
         ␣&&␣this(out)")
     public void finished(Object in, Object out) {}

     @AfterReturning(value = "finished(in,␣tmp)", returning = "out")
     public void assignSequence(Object in, Object tmp, Object out) {
          Iterable it = (Iterable)out;

          for(Object o : it) {
               super.assignSequence(in, o);
          }
     }
}
```

Listing 5.15: FlatMapFunctionSequencePassOnAspect for Spark Streaming *FlatMapFunctions*

Function2 *Function2* implementations are used for reduce functions. A reduce function takes two objects as input and evaluates them to a single result object. Whether to check if both input objects carry sequence identifiers and which one to choose depends on the application. We implemented an example, where the sequence identifiers of both input objects have to be equal so that the sequence identifier is passed on.

```
@Aspect
public class Function2SequencePassOnAspect {

     @Pointcut("execution(*␣org.apache.spark.api.java.function.
         Function2.call(java.lang.Object,java.lang.Object))␣&&␣args(in1
         ,␣in2)")
     public void finished(Object in1, Object in2) {}

     @AfterReturning(value = "finished(in1,␣in2)", returning = "out")
     public void assignSequence(Object in1, Object in2, Object out) {
          if (in1 instanceof SequencedObject && in2 instanceof
              SequencedObject) {
               if (((SequencedObject)in1).getSequence().equals(((
                   SequencedObject)in2).getSequence())) {
                    if (out instanceof SequencedObject) {
                     ((SequencedObject) out).setSequence(((
                        SequencedObject) in1).getSequence());
                  }
               }
            }
         }
}
```

Listing 5.16: Function2SequencePassOnAspect for Spark Streaming *Function2* implementations

Listing 5.16 shows an aspect for *Function2* implementations, where both input objects must carry the same sequence identifier.

Runtime Performance Measurement

For measuring the runtime of a Spark Streaming application we implemented three aspects. One for *Receivers*, one for *Functions* and *FlatMapFunctions*, and one for *Function2* implementations. There is only one aspect for *Functions* and *FlatMapFunctions*, since there is no difference in measuring their runtime and obtaining their sequence identifiers.

Receivers

As stated in Section 2.4, a *Receiver's store* method's invocation time only reflects the time it took Spark to store the data to its cluster, but not how long it actually took to obtain the data. Since it is not possible to define a generally applicable pointcut for measuring the time it took to obtain the data, *Receivers* must be treated individually. However, we implemented an aspect that measures the invocation time of a *Receiver's store* method.

```
@Aspect
public class ReceiverRuntimePerformanceAspect extends
   SequencedRuntimePerformanceAspect {
    @Override
    @Pointcut("call(*␣org.apache.spark.streaming.receiver.Receiver.
       store(java.lang.Object))␣&&␣args(o)")
    public void scope(Object o) {}


    @Override
    @Around("scope(o)␣&&␣target(jpo)")
    public Object around(ProceedingJoinPoint pjp, Object jpo, Object
       o) throws Throwable {
        return super.around(pjp, jpo, o);
    }
}
```

Listing 5.17: *ReceiverRuntimePerformanceAspect* for Spark Streaming

Listing 5.17 shows an aspect for measuring a *Receiver's store* method's invocation time. A *SequencedRuntimePerformanceAspect* is similar to a *AbstractRuntimePerformanceAspect* described in Section 5.1, which takes sequence identifiers into consideration.

Functions and FlatMapFunctions

Functions and *FlatMapFunctions* can be measured using only one aspect, since in both cases a method that takes one argument, which carries a sequence identifier, as input and returns one object, must be measured.

```
@Aspect
public class FunctionSequencedRuntimePerformanceAspect extends
    SequencedRuntimePerformanceAspect {

    @Override
    @Pointcut("(execution(*␣org.apache.spark.api.java.function.
        Function.call(java.lang.Object))␣||" +
            "␣execution(*␣org.apache.spark.api.java.function.
                FlatMapFunction.call(java.lang.Object)))␣&&␣args(o)")
    public void scope(Object o) { }

    @Around("scope(o)␣&&␣this(jpo)")
    public Object around(ProceedingJoinPoint pjp, Object jpo, Object
        o) throws Throwable {
        return super.around(pjp, jpo, o);
    }
}
```

Listing 5.18: *FunctionRuntimePerformanceAspect* for Spark Streaming

Listing 5.18 shows an aspect for measuring a *Function's* or *FlatMapFunction's call* method's invocation time.

Function2

A *Function2* is slightly different to a *Function* as it takes two arguments as input. It must be treated differently since it can not be generally determined, which sequence identifier for a measurement has to be chosen. We implemented an aspect for measuring a *Function2* implementation's runtime performance, where a sequence identifier is only assigned if both input objects share the same sequence identifier.

```
@Aspect
public class Function2SequencedRuntimePerformanceAspect extends
    SequencedRuntimePerformanceAspect {
    public void scope(Object o) { }

    @Pointcut("execution(*␣org.apache.spark.api.java.function.
        Function2.call(java.lang.Object,␣java.lang.Object))␣&&␣args(o1
        ,␣o2)")
    public void scope(Object o1, Object o2) { }

    @Around("scope(o1,␣o2)␣&&␣this(jpo)")
    public Object around(ProceedingJoinPoint pjp, Object jpo, Object
        o1, Object o2) throws Throwable {
        String sequence = null;
        if (o1 instanceof SequencedObject && o2 instanceof
            SequencedObject) {
            if (((SequencedObject) o1).getSequence().equals(((
                SequencedObject) o1).getSequence())) {
                sequence = ((SequencedObject) o1).getSequence();
            }
        }

        return super.around(pjp, jpo, sequence);
    }
}
```

Listing 5.19: *Function2RuntimePerformanceAspect* for Spark Streaming

Listing 5.19 shows an aspect for measuring a *Function2's call* method's invocation time as described.

5.3 Apache Storm

In Section 2.4 we discussed the API of Apache Spark Storm's programming model and how performance measurement data can be acquired in Storm in general. In this section we present aspects for creating and passing on sequences and for measuring runtime in Storm's programming model.

Sequencing

Adding and passing on sequences is less straight forward for Storm as for Spark. Considering Storm's API, as described in Section 2.4, *OutputCollectors* don't take tuples as arguments. However, Storm takes any object, or a list of objects, as argument, automatically wraps *Tuple* instances around the object and then sends tuples to the next element in the topology. This means that the target for sequence identifiers must be tuples, which are created in Storm internally. To add sequence identifiers we decided to create aspects using pointcuts that check entire control flows of an application. Whenever a *Tuple* instance is created when an *OutputCollector's* emit method called by a spout or bolt, we add or pass a sequence identifier on respectively.

Sequence Creation

Listing 5.20 shows an aspect for creating sequences for Storm spouts by defining four pointcuts. The first one targets the *TupleImpl* classes constructor, a second one a spout's *nextTuple* method, a third one an *OutputCollector's emit* method. The last pointcut combines the first three pointcuts, where the application's control flow is checked. This means, that the advised code is invoked after a new tuple instance has been created when an *OutpoutCollector's emit* method is called by a spout's *nextTuple* method.

```
@Aspect
public class SpoutCreateSequenceAspect extends CreateSequenceAspect {
    @Pointcut("execution(*backtype.storm.tuple.TupleImpl.new(..))␣&&␣
       this(o)")
    public void created(Object o) {}

    @Pointcut("execution(*␣backtype.storm.spout.ISpout.nextTuple(..))
       ")
    public void spoutNextTuple() {}

    @Pointcut("execution(*␣backtype.storm.spout.ISpoutOutputCollector
       .emit(..))")
    public void spoutCollectorEmit() {}

    @Pointcut("cflow(spoutNextTuple())␣&&␣cflow(spoutCollectorEmit())
       ␣&&␣created(o)")
    public void createdInFlow(Object o) {}

    @After("createdInFlow(o)")
    public void createSequence(Object o) {
        super.createSequence(o);
    }
}
```

Listing 5.20: *CreateSequenceAspect* for Storm

Passing Sequences On

Listing 5.21 shows an aspect for passing sequences on for Storm bolts. As for creating sequences, the aspect consists of four pointcuts. The first pointcut targets the *TupleImpl* classes constructor, a second one a bolt's execute method, where the incoming tuple is taken into consideration, and a third one, which targets the *OutputCollector's emit* method. The last pointcut combines the first three pointcuts, where the application's control flow is checked. The advised code is called when a new tuple instance has been created and an *OutpoutCollector's emit* method is called by a bolt's *execute* method. The sequence identifier is passed on from the bolt's input tuple to the tuple emitted via the *OutpoutCollector*.

```
@Aspect
public class BoltSequencePassOnAspect extends SequencePassOnAspect {
    @Pointcut("execution(*backtype.storm.tuple.TupleImpl.new(..))")
    public void newTuple() {}

    @Pointcut("execution(*␣backtype.storm.topology.IBasicBolt.execute
       (backtype.storm.tuple.Tuple,␣..))␣&&␣args(in,␣*)")
    public void boltExecute(Tuple in) {}
    @Pointcut("execution(*␣backtype.storm.topology.
        IBasicOutputCollector.emit(..))")
    public void outCollectorEmit() {}

    @Override
    @Pointcut("cflow(boltExecute(in))␣&&␣cflow(outCollectorEmit())␣&&
        ␣newTuple()␣&&␣this(out)")
    public void finished(Object in, Object out) {}

    @Override
    @After("finished(in,␣out)")
    public void assignSequence(Object in, Object out) {
        super.assignSequence(in, out);
    }
}
```

Listing 5.21: *SequencePassOnAspect* for Storm

Runtime Performance Measurement

Measuring runtime performance for Storm's spouts and bolts is done by measuring their main method's invocation time.

Spouts

Listing 5.22 shows an aspect implementation of a *RuntimePerformanceAspect* for spouts, where the invocation time of a spout's *nextTuple* method is measured.

```
@Aspect
public class SpoutRuntimePerformanceAspect extends
    AbstractRuntimePerformanceAspect {

    @Override
    @Pointcut("execution(*␣backtype.storm.spout.ISpout.nextTuple(..))
        ")
    public void scope() {}

    @Around("scope()␣&&␣this(jpo)")
    public Object around(ProceedingJoinPoint pjp, Object jpo) throws
        Throwable {
        return super.around(pjp, jpo);
    }
}
```

Listing 5.22: *RuntimePerformanceAspect* for Storm spouts

Bolts

Listing 5.23 shows an aspect implementation of a *RuntimePerformanceAspect* for bolts, where the invocation time of a bolt's *execute* method is measured.

```
@Aspect
public class BoltSequencedRuntimePerformanceAspect extends
    SequencedRuntimePerformanceAspect {

    @Override
    @Pointcut("execution(*␣backtype.storm.topology.IBasicBolt.execute
        (backtype.storm.tuple.Tuple,␣..))␣&&␣args(o,␣*)")
    public void scope(Object o) {}

    @Around("scope(o)␣&&␣this(jpo)")
    public Object around(ProceedingJoinPoint pjp, Object jpo, Object
        o) throws Throwable {
        return super.around(pjp, jpo, o);
    }
}
```

Listing 5.23: *RuntimePerformanceAspect* for Storm bolts

5.4 Publication

In this section we will discuss the implementation specifics of the different publication technologies described in Section 4.3.

File

The file publication service implementation allows to export measurement data to a comma-separated file format. Different types of measurement data are exported into separate files per measurement type. The format within a measurement type follows the same principle, where each objects attributes are separated by a semicolon.

RuntimePerformance

The file for runtime performance measurement records consists of seven columns:
Node.nodeId;*Node.nodePurpose*;*sequence*;*starttime*;*endtime*;*duration*;*executionResult*

ExecutionProfile

The file for execution profile measurements can contain two different types of rows: A row for an execution profile measurement record, which is followed by its associated resource snapshot records. A row that contains an execution profile measurement consists of seven columns: *Node.nodeId*;*Node.nodePurpose*;*sequence*;*starttime*;*endtime*;*duration*;*executionResult*

If there are associated resource snapshot records for that execution profile, the row is followed by rows for resource snapshots, which consist of seven columns:
timestamp;*systemCpuLoad*;*processCpuLoad*;*processCpuTime*;*threadCpuTime*;*heapMax*;*heapUsage*

JvmProfiles

The rows in a file for JVM profile measurement records consist of 14 columns:
Node.nodeId;*Node.nodePurpose*;*timestamp*;*systemCpuLoadMax*;*systemCpuLoadMin*; *systemCpuLoadAvg*;*processCpuTime*;*processCpuLoadMax*;*processCpuLoadMin*;*processCpuLoadAvg*; *heapMax*;*heapUsageMax*;*heapUsageMin*;*heapUsageAvg*

JDBC

We implemented a JDBC publication service that inserts measurement data in a relational structure that suits the domain model defined in Section 4.2. Figure 5.1 illustrates the structure with its tables and references.

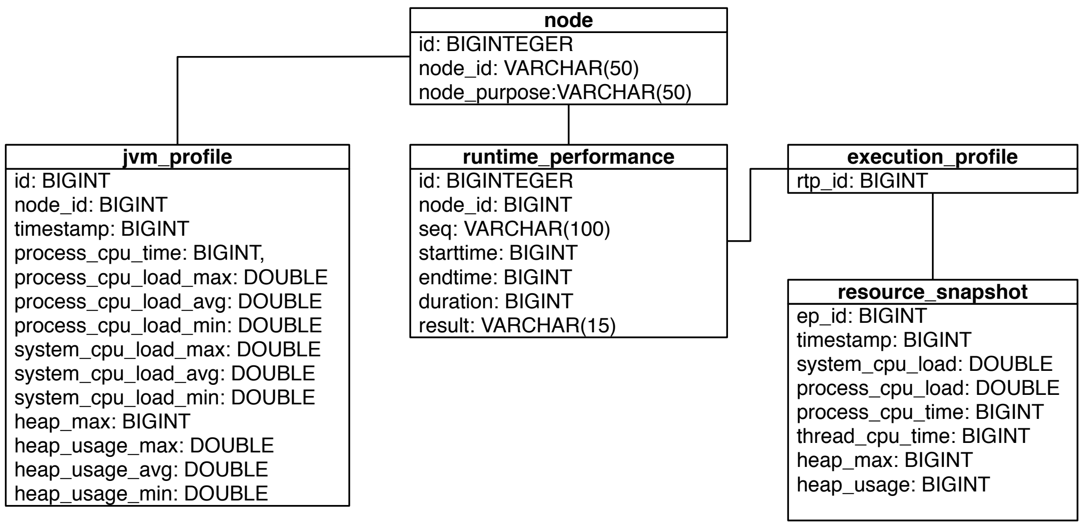


Figure 5.1: Table structure for the implemented JDBC publication service

SQL scripts for creating the defined table structure differ from database system to database system. Listing 5.24 shows a script for creating the table structure for a PostgreSQL database [57].

```
CREATE TABLE node (
    id BIGSERIAL PRIMARY KEY,
    node_id VARCHAR(50),
    node_purpose VARCHAR(50),
    CONSTRAINT node_unique_ct UNIQUE (node_id, node_purpose)
);
```

```
CREATE TABLE runtime_performance (
    id BIGSERIAL PRIMARY KEY,
    node_id BIGINT,
    seq VARCHAR(100),
    starttime BIGINT,
    endtime BIGINT,
    duration BIGINT,
    result VARCHAR(15),
    CONSTRAINT rtp_unique UNIQUE (node_id, starttime, endtime),
    CONSTRAINT rtp_node_fk FOREIGN KEY (node_id) REFERENCES node(id)
);

CREATE TABLE execution_profile (
    rtp_id BIGINT PRIMARY KEY,
    CONSTRAINT rtp_ep_fk FOREIGN KEY (rtp_id) REFERENCES
       runtime_performance(id)
);

CREATE TABLE resource_snapshot (
    ep_id BIGINT,
    timestamp BIGINT,
    system_cpu_load DOUBLE PRECISION,
    process_cpu_load DOUBLE PRECISION,
    process_cpu_time BIGINT,
    thread_cpu_time BIGINT,
    heap_max BIGINT,
    heap_usage BIGINT,
    CONSTRAINT rs_pk PRIMARY KEY (ep_id, timestamp),
    CONSTRAINT rs_ep_fk FOREIGN KEY (ep_id) REFERENCES
       execution_profile(rtp_id)
);

CREATE TABLE jvm_profile (
    id BIGSERIAL PRIMARY KEY,
    node_id BIGINT,
    timestamp BIGINT,
    process_cpu_time BIGINT,
    process_cpu_load_max DOUBLE PRECISION,
    process_cpu_load_avg DOUBLE PRECISION,
    process_cpu_load_min DOUBLE PRECISION,
    system_cpu_load_max DOUBLE PRECISION,
    system_cpu_load_avg DOUBLE PRECISION,
    system_cpu_load_min DOUBLE PRECISION,
    heap_max BIGINT,
    heap_usage_max DOUBLE PRECISION,
    heap_usage_avg DOUBLE PRECISION,
    heap_usage_min DOUBLE PRECISION,
    CONSTRAINT jvmp_unique UNIQUE (node_id, timestamp)
);
```

Listing 5.24: Script for table creating for a PostgreSQL database

JMS

We implemented a JMS publication service that publishes JMS object messages to any configured JMS destination (e.g., a queue or topic). Since we did not want to bind the JMS publication service to a specific JMS implementation, the connection factory used for creating JMS connections can also be configured. The implemented JMS publication service looks up both, the destination and the connection factory, in the JVM's local JNDI [52] directory using defined resource names.

```
public class JmsPublicationService implements PublicationService {

        ...

    public JmsPublicationService() {
        try {
            Context context = new InitialContext();

            ConnectionFactory connectionFactory = (ConnectionFactory)
                 context.lookup(Configuration.getProperty(
                CONNECTION_FACTORY_RESOURCE_SYSTEM_PROPERTY));
            connection  = connectionFactory.createConnection();
            destination = (Destination) context.lookup(Configuration.
                getProperty(DESTINATION_RESOURCE_SYSTEM_PROPERTY));
        } catch(NamingException | JMSException e) {
            throw new RuntimeException(e);
        }
    }


    private <T extends Measurement> void sendMessage(T t) throws
       JMSException{
        Session session = connection.createSession(true, 0);
        MessageProducer producer = session.createProducer(destination
            );
        producer.send(destination, session.createObjectMessage(t));
        session.commit();
        producer.close();
        session.close();
    }
}
```

Listing 5.25: JMS publication service implementation

Listing 5.25 shows a snippet of the JMS publication service implementation, which shows how the JMS connection is initialized and how messages are sent.

JMX

We implemented a publication service that sends JMX notifications [65] for measurement data to registered receivers. For each measurement type, an MBean [65] with an associated object name will be created. A notification's notification type is set to "Measurement Notification". The notification user data attribute will be set to the measurement data object. This means, that a receiver, which consumes these notifications, must have the domain model implementation classes in its class path. Listing 5.26 shows how MBeans are initialized and messages are sent.

```
public class MeasurementNotificationSender extends
    NotificationBroadcasterSupport implements
    MeasurementNotificationSenderMBean {
     public MeasurementNotificationSender(Class<? extends Measurement>
          clazz) {
          MBeanServer server = ManagementFactory.getPlatformMBeanServer
              ();
          ObjectName objectName = null;
          message = MESSAGE_PREFIX + clazz.getName();

          try {
              objectName = new ObjectName(OBJECT_NAME_PREFIX +  clazz.
                  getSimpleName());
              server.registerMBean(this, objectName);
          } catch (Exception e) {
              throw new RuntimeException(e);
          }
     }


     public <T extends Measurement> void sendNotification(T t) {
          Notification n = new Notification(NOTIFICATION_TYPE,
              MeasurementNotificationSender.class.getName(),
              sequenceNumber++, message);
          n.setUserData(t);
          sendNotification(n);
     }
}
```

Listing 5.26: JMX MBean implementation

Log4J

The Log4J publication service is a simple implementation that logs measurement data in Log4J's *INFO* level. Each log entry has the prefix "Published measurement " followed by a string, which results from the measurement object's *toString* method.

CHAPTER 6

Demonstration

In this chapter, the evaluation of the proof-of-concept framework presented in Chapter 5 is described. First, the requirements for a proper demonstration use case are discussed. Second, a use case that meets these requirements is described. Third, the implementation details for the use case are discussed. Fourth, scenarios to run the use case are defined. Finally, once the scenarios have been executed, to demonstrate the value of this thesis framework, data analysis of the measured data will be conducted and the results will be discussed.

6.1 Scenario

Requirements

To create data that can be analyzed, we have identified four requirements that a demonstration scenario has to meet:

- **High Complexity:** In order to test the performance and scalability of a distributed application, it should be easy to increase computation load of the scenario. High complexity ensures that minor changes of the input data will increase the computation load significantly.
- **Horizontally Divisible:** The scenario should be horizontally divisible. By that, we mean that it should be able to split the scenario up into multiple process steps, where each step is responsible for a certain functionality. Only such a scenario, with multiple computation steps that can be executed on different workers in a distributed system, where data has to be transferred, is suitable for testing a distributed data-processing application for two reasons. First, the steps will be different in complexity and thus have different execution runtimes. Parallelization of more complex steps should decrease the overall execution time of a scenario if the load is distributed properly, which is one major subject of investigation. Second, as data has to be transferred, using different serialization or communication technologies as well as network performance will affect the results. A scenario, where

these factors are not considered, does not properly reflect real-world usage of distributed systems.

- **Vertically Divisible:** By vertically divisible we mean that it should be able to divide processing data and execute computation steps in parallel. The fact that serial processing on single machines is not able to cope with ever increasing amounts of data is a major reason for using distributed systems, where data is processed in parallel to reduce processing time. Thus, a test scenario should fulfill this requirement so that parallelization and its effects can be tested and evaluated.

- **High Data Volume:** The data volume that has to be processed and transferred should be high since we want to investigate heap memory usage behavior of our different applications. If no significant amounts of data are transferred or processed and there are no major changes in the JVM's heap memory usage, we will not be able to investigate how our applications perform regarding memory usage, or if framework internals do affect memory usage.

Problem Description

The scenario we have defined for our demonstration use case is a slightly modified version of the well-known Traveling salesman problem. Since there are many formulations and variations of the Traveling salesman problem [59] we use the following formulation:

A TSP graph G is a complete weighted undirected graph specified by a pair (N, d) where N is a set of nodes and d is a function that translates the distance between two nodes to numerical values. d satisfies two conditions:

1. Symmetry: $d(i, j) = d(j, i)$ for all i and j in N.
2. $d(i, j) >= 0$ for all i and j in N.

A $path$ of the TSP graph G is a set of edges that describes a path containing each node exactly once (i.e., a Hamiltonian graph [25]). The $path\ distance$ is the sum of all edges distances. The solution for our modified traveling salesman problem is a $path$ with the minimal possible $path\ distance$.
The nature of the Traveling salesman problem fulfills two of the requirements described in Section 6.1:

- **High Complexity:** The problem is known to be NP-hard [31].

- **Horizontally Divisible:** A program for the path can be divided into several steps. One example would be:

    1. Find paths
    2. Calculate path distances
    3. Find minimal solution

Whether a program is vertically divisible is implementation specific, the data volume depends on the implementation (e.g., how the program is divided) and actual input size.

Data Source & Domain

The Traveling salesman problem itself is domain independent. Since we focus on performance measurements in our work and do not consider the actual problem solution as relevant, we have not applied the problem to a particular domain and used randomly generated input data. Graph vertices are created by generating two random numbers, where the two generated numbers describe the coordinates in a two-dimensional space.

6.2 Implementation

A solution for the described problem has been implemented for both, Apache Spark and Apache Storm. However, the implementations are designed so that they share as much features as possible in order to make the results comparable.

Architecture

We have decided to split the problem up into five process steps.

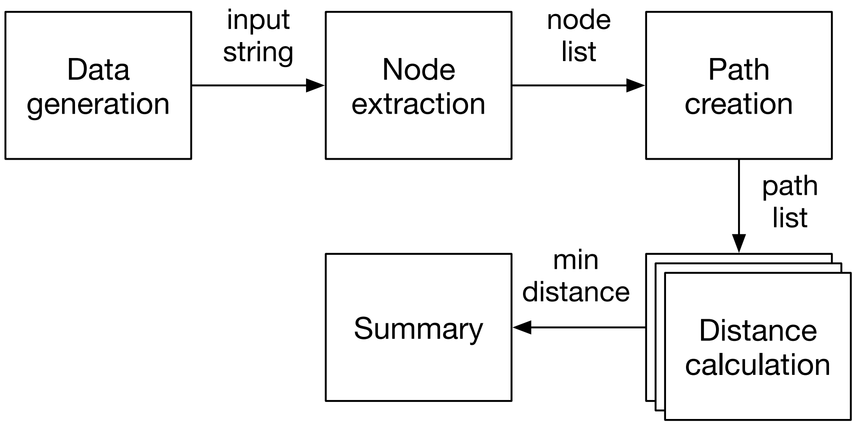


Figure 6.1: Architecture overview

Figure 6.1 shows how we split up the problem. It also shows that one step, *Distance calculation*, can be executed in parallel.

1. **Data Generation:** In this first step input data is created. We've defined that processable input data is a random string containing vertices in the following format: “$a[$“$+number+$ “, “$+number+$ “$]$“ where $number$ is a positive integer. There can be random characters

before and after a vertex. In our implementation we used UUID strings. The first number is the x-coordinate and the second the y-coordinate in a two-dimensional space.

```
fbbca05f-caf3-410b-b831-c48baa8e7bff40836da0-df5d-4f73-8649-
    a4fb0fe2818b5b58bcc7-a67c-45f6-b161-b445a258cf85a[29,70]
    c386a926-f95a-4e30-bdb6-c4723675bdc24c5cb53c-7350-45d4-8f84-
    d03dfe9fdcf8056a600c-3af6-4e0c-b5e7-0a6c365ddf0ba[3,52]10
    f6c44b-33af-4f24-9c70-11e84d605f9b7e5a72f2-a501-4e94-8021-
    f92d343515e2a[88,27]67c61aef-4407-422d-b3c8-550084
    b05669d7adfb84
```

Listing 6.1: Input example

Listing 6.1 shows a generated input data example that contains three vertices: a[29,70], a[3,52] and a[88,27].

2. **Extraction:** The second step is to extract vertices from the string. The result of this step is a list of *Node* objects. A *Node* has an *id*, *x* and *y* attribute. The *id* is a number to identify the node, *x* and *y* are the node's coordinates. Figure 6.2 shows the *Node* class.

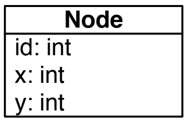


Figure 6.2: Node class

3. **Path Creation:** In this step all paths are created. Since a path is a permutation [55] of nodes, the number of all paths is defined by the number of node permutations ($n!$ where n is the number of nodes [55]). As stated, any implementation based on our proposed architecture must allow that distance calculations can be executed in parallel. It must be possible to split up paths into subsets and pass them on to workers responsible for distance calculation asynchronously. That means that a path calculation worker does not wait for a distance calculation worker's subset result, before passing the next subset on to another distance calculation worker.

4. **Distance Calculation:** The distance between two nodes in a two-dimensional space can be calculated by using the Pythagorean theorem [67]. The length of the two required edges is the difference between the greater and lesser x and y coordinates. The distance between the nodes is then defined by the Pythagorean theorem.

   Figure 6.3 illustrates the distance and equation 6.1 shows how the distance is calculated.

$$distance = \sqrt{\begin{array}{r}((max(A.x, B.x) - min(A.x, B.x))^2 + \\ ((max(A.y, B.y) - min(A.y, B.y))^2\end{array}} \tag{6.1}$$

   In the last step, the minimal distance of the received set of paths is calculated.

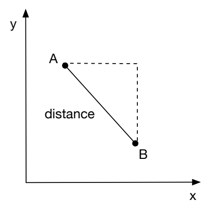


Figure 6.3: Distance calculation

5. **Summary:** The summary step is responsible for summarizing the partial results of the distance step. Here, the final result, thus the minimal distance of all paths, is detected.

Both implementations share the basic architecture from a component and sequence point of view.

Test Data Generation

We already specified how the structure of an input string has to look like and that we use UUIDs in between vertices. It is not specified how many UUIDs there are in between two vertices and how many vertices an input string has to contain. For our evaluation, we define that each input string has to contain exactly ten vertices. In order to simulate random behavior, both, the coordinates of vertices and the number of UUIDs in between vertices are uniformly distributed random integers from zero to a defined upper bound. The upper bound for a vertices coordinates is 99 and the upper bound for the number of UUIDs is 99999. Given that, and the fact that each character of a string on the described platform has a size of one byte, the expected input string size is as follows: ((
0.9 *(probability of a 2-character coordinate)* * 2 *(size of a 2-character coordinate)* +
0.1 *(probability of a 1-character coordinate)* * 1 *(size of a 1-character coordinate)*
) * 2 *(2 coordinates per vertex)* + 4 *(remaining vertex characters)* +
36 *(length of a UUID string)* * 50000 *(expected value of the number of UUIDs)*
) * 10 *(number of vertices)* = 18000078 *bytes* = 17.166 *mega bytes*

Scenario

In order to create a sufficient amount of measurement data, we defined two scenarios that we ran for both implementations. We executed 25 test runs two times, once with a break of 1 second in between each run and a second time with a break of only 10 milliseconds between each run. Considering the domain-specific language described in Section 4.5, we declared our scenarios as follows: *loop(25,1000)* or *loop(25,10)* respectively. For each of the four scenario executions (two times with the Spark implementation with both breaks, two times the Storm implementation with both breaks), we selected different node identifiers for each step, so that we can distinguish the recorded measurement data.

Apache Spark Streaming Implementation

We implemented the described problem solution according to the application architecture description using Apache Spark Streaming. First, we implemented a *Receiver*, which creates data. For all other steps we implemented *Functions*, which are used by map or flat map transformations and one output operation.

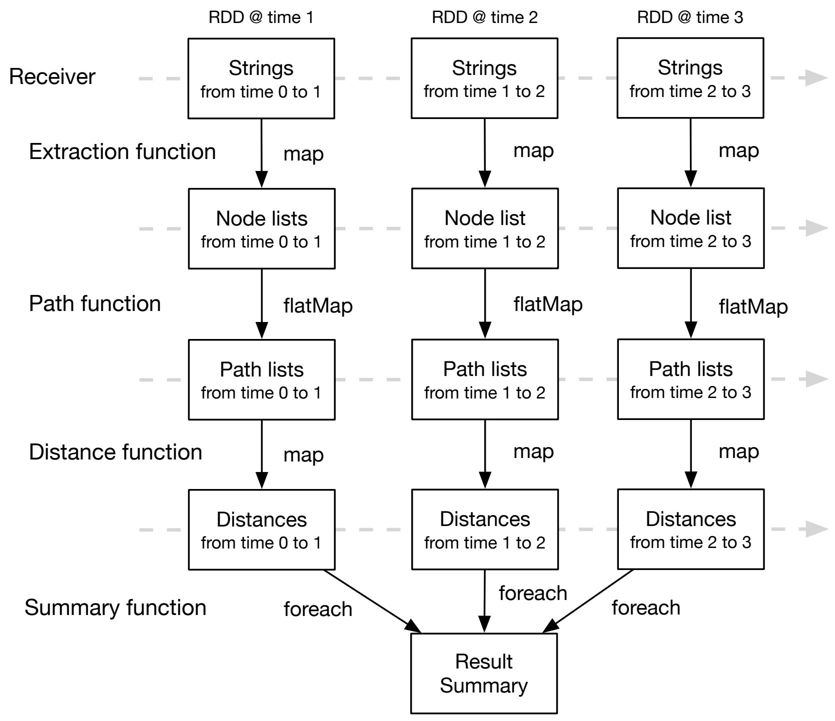

RDD @ time 1
RDD @ time 2
RDD @ time 3
Receiver
Strings from time 0 to 1
Strings from time 1 to 2
Strings from time 2 to 3
Extraction function
map
Node lists from time 0 to 1
Node list from time 1 to 2
Node list from time 2 to 3
Path function
flatMap
Path lists from time 0 to 1
Path lists from time 1 to 2
Path lists from time 2 to 3
Distance function
map
Distances from time 0 to 1
Distances from time 1 to 2
Distances from time 2 to 3
Summary function
foreach
Result Summary

Figure 6.4: Spark Implementation

Figure 6.4 shows an overview of our Apache Spark Streaming application. The *Receiver* creates data, which are stored in Spark RDDs over time. We decided to use a time window of seven seconds because the amount of data that will accrue within seven seconds will be considerably different for the two scenarios we decided to run, and thus lead to different performance results. This means every seven seconds an RDD is created, which contains all data records stored to Spark Streaming within the past seven seconds. The second step, extraction is done via a map function, where lists of nodes are extracted from the input strings in an RDD. Paths are created in the course of a flat map transformation. A path list contains no more than 200000 elements. If there are more paths for a node list, there will be multiple path lists created for that node list. There are two reasons why we decided to use a limit of 200000 paths per list. First, the size of the message that has to be transferred. A single path contains ten *Node* elements, that contain two four-byte integers, which means that the size of a single path is around 80 bytes. Since there are ten factorial (around 3.6 million) paths, one single message containing all paths would be huge in size (around 290 megabytes), where as a list containing 200000 paths has a size of around 16 megabytes, which we consider as appropriate for our evaluation. Second, since we want to evaluate the implementation's parallelization and task scheduling behavior, splitting a path list up by 200000 leads to 19 distance calculation tasks per input message, which we consider as an amount big enough for creating a sufficient amount of analysis data. Using an identifier, each path list can be associated to its source node list. The path creation step is followed by a map transformation, which calculates distances for each path list and determines the minimal distance. The summary function determines the absolute minimum distance for an input string by aggregating received minimum distances using the set identifier.

Apache Storm Implementation

As for Spark Streaming, we implemented the described problem solution according to the architecture description by using Apache Storm. For each step described, we implemented a Storm component. The first step, data creation is implemented as a spout that creates and emits data to the topology. All other steps are implemented as bolts, linked together using Storm's *shuffle grouping* method as described in Section 2.1. Path bolts are different to other bolts since they might emit multiple tuples, where each tuple contains a set of paths with a maximum size of 200000 (for the same reasons as for the Spark implementation). If there are less than 200000 paths for the given input data, only one tuple containing all paths is emitted, if there are multiple tuples for an input node list, path lists can be associated with their source, and thus with each other, using an identifier.

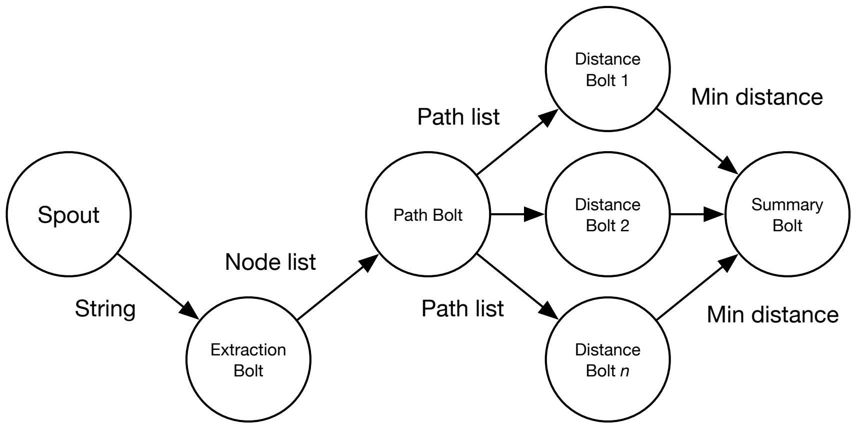


Figure 6.5: Storm implementation

Figure 6.5 shows an overview of our Storm implementation and its data flow. Since calculating and comparing the distance of all paths can take a considerable amount of time when done time-serially, we decided to split up the set of paths and parallelize distance calculation.

6.3 Execution

To create analyzable data, we executed both described problem solutions. Since the main focus of this thesis is the engineering perspective of how performance can be tested and monitored, and not a performance analysis itself, we executed the implementations just on a single machine instead on a cluster. However, this still demonstrates the applicability of the described concepts and the proof-of-concept implementation.

Test Machine Specification

- **Model:** MacBook Pro (Retina, 13-inch, Late 2013)
- **Operating System:** OS X 10.10.5
- **CPU:** Intel Core i5 2.4 GHz Dual Core, 3 MB L2 Cache
- **Memory:** 8 GB 1600 MHz DDR3
- **Java:** Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
- **JVM:** Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)

6.4 Data Analysis

In this section, we analyze the measurement data created by the test runs. By comparing the results of both implementations, we will show a huge benefit of the framework proposed in this thesis: A common data model for measurement data of different data-processing applications, implemented using different frameworks, that allows a direct comparison by visualizing data.

Figure 6.6 shows an overview of runtime measurements. The durations (end time - start time) are aggregated by node purpose and node identifier. The node identifier allows to associate a record with the particular run or implementation (10ms or 1 second break, Spark or Storm). The different columns show different node purposes (which reflect process steps), whereas the rows are different aggregations (maximum, minimum, average, sum). There are a few findings in this overview we want to discuss:

- Spark/Storm difference at creation: In Section 2.4 we stated that measuring the invocation time of a Spark *Receiver's store* method will not reflect the time consumed for actually creating, reading or receiving the data. Only the time used for transferring the data to Spark is measured. This explains the huge gap when comparing Spark and Storm results of the first step (creation).
- Spark/Storm difference at summary: There is a huge gap between Spark and Storm for the last process step. The explanation for this gap is as follows: Spark immediately starts the invocation of output operations for each time slot, even when no data is received within a time slots period. However, when no data is received, Spark blocks the output operation's function invocation and waits for a considerable amount of time. This behavior distorts

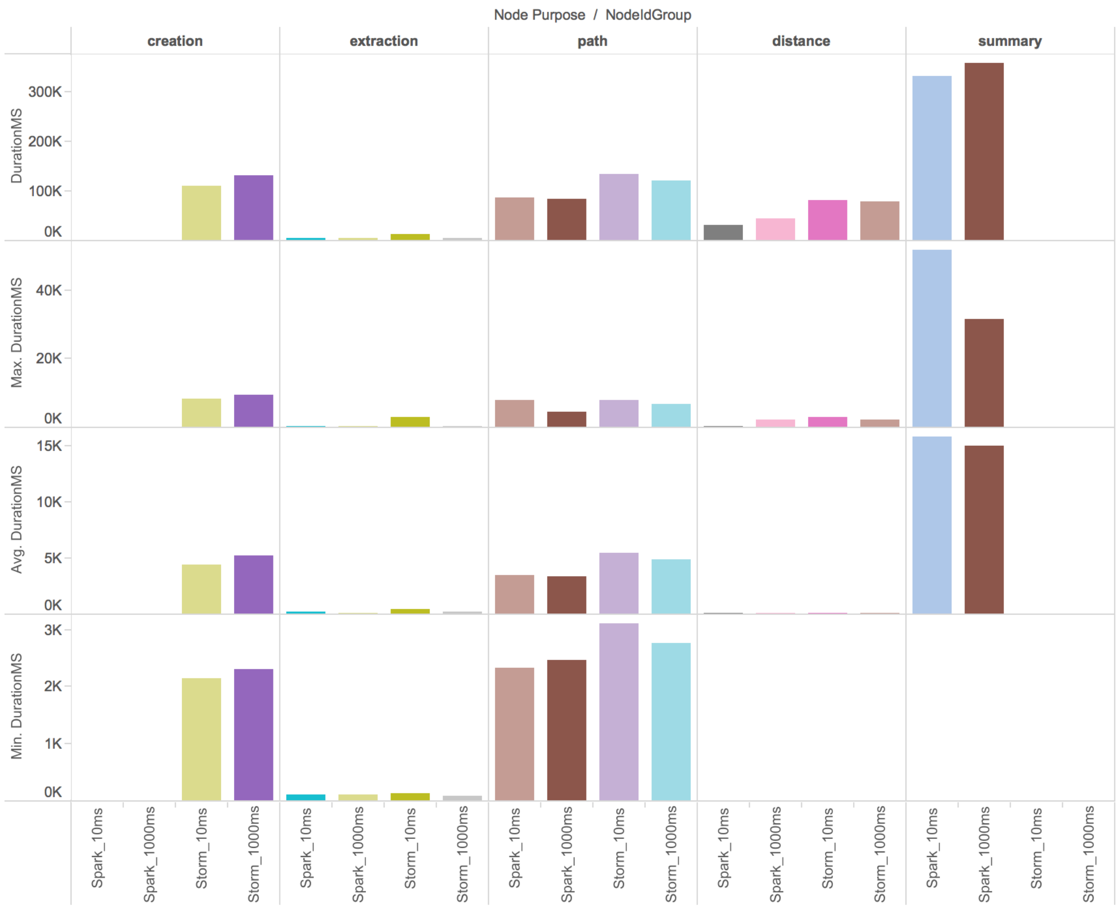


Figure 6.6: Comparison of durations. The y-axes are scaled differently for each aggregation to highlight the differences between the steps, load configurations, and implementations.

the results for the last process step, since for the first few time slots no data has been passed to the output operation as the calculations have not been finished. These records have a considerable impact on the aggregated data.

- A third finding is that the Spark implementation performs better regarding runtime measurements at path and distance. Figure 6.7 shows a more detailed chart of durations sums that makes this finding more visible. The color indicates the number of records (count) that have been summed up. Light green means that there have been few records that have been aggregated (at minimum 25 since we created 25 input strings for each implementation and load configuration), dark green mean that there have been more records (e.g, for the distance step, because the lists of all paths have been split up).

When summing up the sums of all records, except *creation* and *summary*, where framework differences distort the results, for each run and implementation, the gap is even more visible. Figure 6.8 shows these running sums of all records per implementation and scenario.

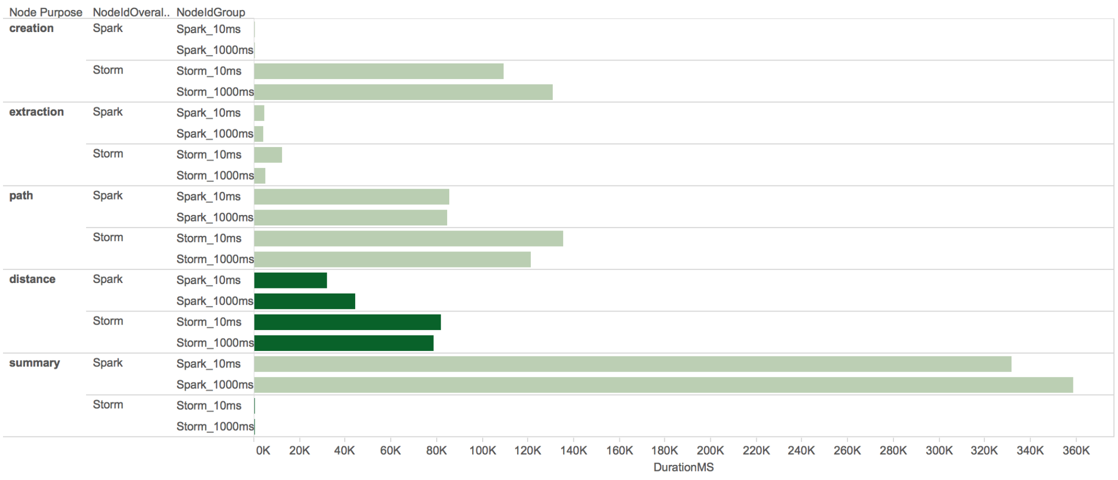


Figure 6.7: Detailed comparison of duration sums

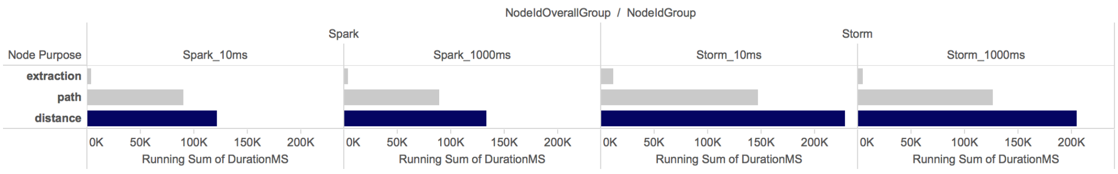


Figure 6.8: Running sum of durations

Considering these results, it might appear that Spark performs significantly better. However, simply looking at runtimes of single process steps does not include the time that has been consumed by the framework or for transferring data from step to step. Using sequences, runtime measurement records of different process steps can be related to each other and the time between a step's end time and the followed step's start time be determined. Figure 6.9 shows the latency between the single steps.

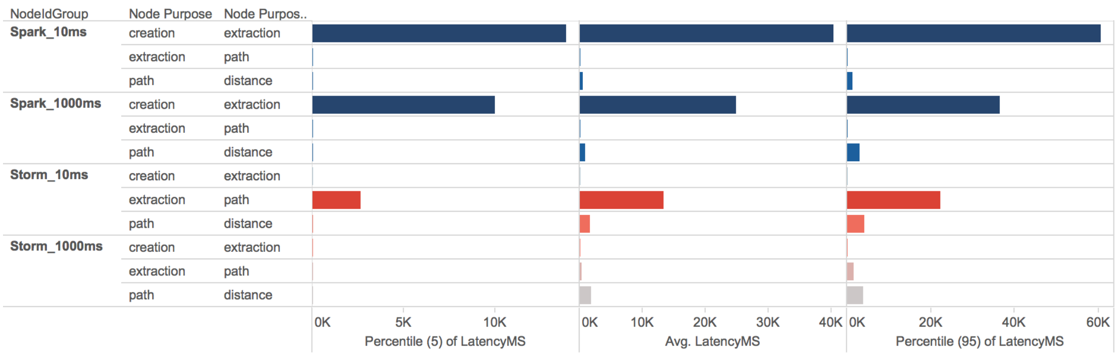


Figure 6.9: Latency

These results suggest that Spark performs poorly regarding latency, but the huge gaps can be explained by the differences between Spark and Storms processing model. First, in Section 2.1 we mentioned that Spark creates micro batches over time and in Section 6.2 we decided to choose a time window of seven seconds. This means that after the *store* method of our *Receiver* has been invoked, it can take up to seven seconds until Spark passed the created data record to the extraction function. Second, since Spark combines data records in RDDs within a seven second period, the amount of data that has to be transferred from the receiver to the first mapping function can be considerable in size, this especially applies to the scenario with a data creation delay of only 10 milliseconds. Storm, in comparison to Spark, emits a tuple at arrival time to its topology. And third, Figure 6.9 reflects the differences between Spark's and Storm's task scheduling model. Both, Spark and Storm have a fixed amount of task executors when running in local mode. However, the difference is that Spark reserves task executors for *Receivers*, which means that a *Receiver* is running at all times, whereas functions are scheduled and executed when a task executor becomes available. In Storm, *Spout* tasks are treated equally to *Bolt* tasks, which means that their executions are also paused when there are no task executors available. For Spark this means that *extraction* tasks are only executed when a task executor is available, thus tasks may be paused for a while, whereas a *Receiver* is running all the time, which, additionally to the time consumed by the batch window, adds a delay between these two processing steps. In Storm's processing model, where *Spout* tasks are scheduled as well, the bottleneck is the longest running processing step, which is the *path* step in our application. After an *extraction* task has been executed, it might take some time until a task executor becomes available for running the *path* task of the preceded *extraction* task's result, as the task executors might be busy with running other queued *path* tasks. When all task executors are busy with executing *path* tasks, no more tuples are emitted by *Spouts*, thus there is no huge latency between the *Spout* and the *extraction-Bolt*.
Given that differences in runtime of single process steps and latency, we need to take a look at total runtime in order to draw a conclusion on whether Spark or Storm performed better.

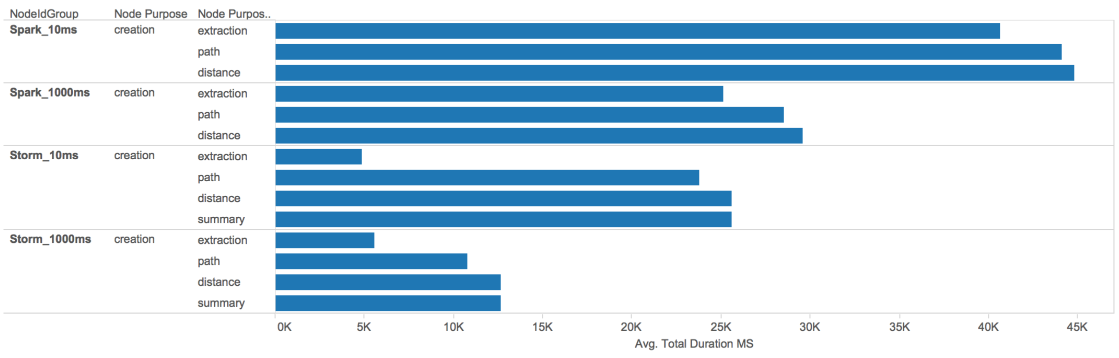


Figure 6.10: Average total runtime (*endtime* - *starttime*)

Figure 6.10 shows the average cumulated (by process step) total runtime of all created data records. By average cumulated total runtime we mean the time between the first measurement record (the start time of the *creation* step) for a created data record and a followed step's end time. As for latency, single measurement records are related to each other by sequence identifiers. The first column shows the implementation and scenario (combined), the second column the process step of which the start time is used (creation, since it is the first step), the third column the process step of which the end time is used for calculation and the last column the actual total runtime. This figure reflects the long latency and process step runtimes we discussed earlier. Given that, one could argue that, overall, the Storm implementation performed significantly better than the Spark implementation. However, what this chart does not show, is the processing timeline. If many records have huge latency between processing steps, since they are combined in a batch and not transferred immediately, but processed quickly after the batch (RDD) that contains all records has been transferred, it might be the case, that, even if total processing runtime time per record has been higher on average, an implementation might have been faster to process all records.

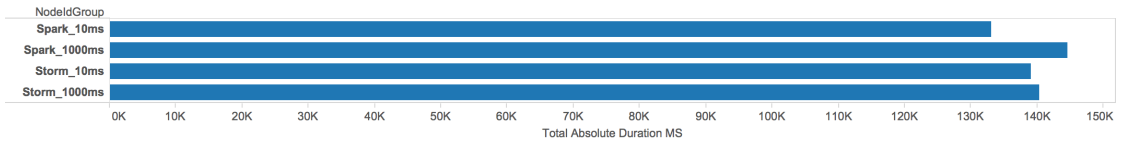


Figure 6.11: Absolute total duration (*max(endtime) - min(starttime)*)

Figure 6.11 shows the time between the absolute minimum start time that has been recorded for the *creation* processing step and the absolute maximum end time of the *summary* processing step that has been recorded. Considering this chart, the Spark implementation performed better for the test run with a 10 millisecond delay between data creation, and the Storm implementation better for the test run with a 1 second delay between data creation.
Given that, we argue that the performance measurement data created by our framework reflect the differences between Storm's continuous operator processing model and Spark's discretized stream processing model discussed in [75]. Less overhead for fault-tolerance, task scheduling, etc. when combining single records in RDDs reduces processing complexity, thus requires less CPU time in general, which explains faster processing times of single process steps. On the other side, time windows to combine data records prevent immediate results for single records and lead to high latency between data reception and the first processing steps. Whereas the advantages and disadvantages of these processing models are not subject of this thesis, the analysis of our framework's data proves the applicability, purpose and benefits of the work presented in this thesis.

CHAPTER 7

Conclusion & Future Work

7.1 Conclusion

Application performance is crucial as applications, which do not fulfill their performance requirements, do not serve their intended purpose. In the age of big data with ever-growing amounts of data, this is particularly true for data processing applications. Due to this growing amounts of data, applications must be scaled since single machines can not keep up with the amounts of data to be processed. In order to scale appropriately and provide sufficient resources for such distributed data-processing applications, their performance must be monitored and tested. As monitoring and testing distributed applications is more sophisticated than monitoring non-distributed applications, we analyzed and discussed a series of arising challenges, proposed a design for a testing and monitoring framework, developed a proof-of-concept framework, demonstrated its applicability by monitoring and testing two different solution implementations for a defined problem, and analyzed the results. We started by defining what performance actually is and which metrics can be used for determining it. After that, we analyzed how these measurements can be acquired from a JVM-based application. In the course of that, we particularly focussed on how performance data acquisition can be integrated in applications that already exist, where monitoring has not be considered from the beginning. Since the main focus of this thesis are distributed applications, we discussed how performance measurement data can be published (e.g., persisted) so that it can be analyzed easily. Based on the results of these discussions, we proposed a design for a framework that enables monitoring of any distributed JVM-based application. To show the design's feasibility, we developed a proof-of-concept framework and applied to a demonstration scenario implemented based on both, Apache Spark Streaming and Apache Storm. Finally, we analyzed and discussed the data created by our framework for these applications and showed that it serves its purpose and provides beneficial insights on an application's performance behavior.

7.2 Future Work

Further Analysis

The data analysis discussed in Section 6.4 for the demonstration use-case is very simple. More sophisticated statistical methods and visualization techniques could provide deeper insights and help to draw better conclusions. Furthermore, forecasts can be made for resource planning and stochastic models could be derived to evaluate a system's behavior under uncertain load characteristics.

Deeper Integration of Spark & Storm

The integration of this thesis framework with Apache Spark and Apache Storm can be extended. The current integration is focussed on Spark and Storm's main programming models. Integrating the framework deeper into Spark and Storm could provide additional, engine internal and/or more detailed processing, data for deeper and more accurate analysis.

Virtual Machine Independence

As described in Section 2.4, this thesis work relies on the *com.sun.management* package provided by the Oracle implementation of the Java virtual machine. This means that other JVM implementations, such as the OpenJDK implementation, are not supported by the framework described in this thesis. Virtual machine independence could be achieved by introducing an additional abstraction layer, which checks the JVM implementation, for acquiring the measurements currently acquired using this package.

Out-of-the-box Integration with Monitoring Tools

Many organizations and cooperations use monitoring tools such as Ganglia, Nagios and Splunk for monitoring their applications and IT systems. These tools often provide interfaces for collecting data in their repositories. Additionally to the communication technologies already supported, interfaces for established systems could be implemented in order to reduce systems integration efforts when introducing our framework.

Automated Distributed Testing

The testing capabilities of this thesis work are limited in its current state. The framework, as described, does not support automated distributed testing, which means that test load can only be created by a single machine, which might not be sufficient for testing a large-scale distributed data-processing application, or, when using multiple machines, must be coordinated manually. There exist concepts for distributed testing (e.g., where a master coordinates multiple test-agents in order to generate load), which could be adapted and integrated in this thesis work.

Result Validation

The testing capabilities of this thesis framework are simple and are only suited for generating performance measurement data. Typically, when testing an application's functionality, the application results are validated to check if it works properly. Since increased load can cause misbehavior, validating application results additionally would be a useful extension to the current work.

Network Monitoring

A distributed data-processing application performance depends on the network performance. If the network is not able to cope with the load, latency will increase and slow down the entire data-processing process. To get a holistic view of the data-processing performance, measuring network performance is required. Network measurement data could be integrated in the data model proposed in this thesis. This would allow deeper performance analysis in general and support root-cause analysis in the case of performance issues.

Bibliography

[1] ActiveMQ. http://activemq.apache.org/. Accessed: 10/2015.

[2] Arnold O Allen. *Probability, Statistics, and Queueing Theory*. Academic Press, 1990.

[3] SpringSource AMS. http://static.springsource.com/projects/tc-server/6.0/ams/doc/springsource Accessed: 10/2015.

[4] TechTarget Application. http://searchsoftwarequality.techtarget.com/definition/application. Accessed: 09/2015.

[5] AspectJ. https://eclipse.org/aspectj/. Accessed: 09/2015.

[6] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Database Systems: Concepts, languages & architectures*. McGraw-Hill, 1999.

[7] Kazimmierz Balos, Dominik Radziszowski, Pawel Rzepa, Krzysztof Zielinski, and Slawomir Zielinski. Monitoring grid resources: Jmx in action. *Department of Computer Science, AGH-University of Science and Technology*, 2004.

[8] Wolfgang Barth. *Nagios: System and network monitoring*. No Starch Press, 2008.

[9] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 135–144, New York, NY, USA, 2007. ACM.

[10] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16, 1998.

[11] Giuliano Casale. Monitoring platform initial release. 2013.

[12] Clojure. http://clojure.org/. Accessed: 09/2015.

[13] Grigoreta S Cojocar and Dan Cojocar. A comparison of aop based monitoring tools. *UNIVERSITATIS BABEŞ-BOLYAI INFORMATICA*, page 65, 2011.

[14] com.sun.management package.
https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/package-summary.html. Accessed: 09/2015.

[15] com.sun.management.OperatingSystemMXBean.
https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/operatingsystemmxbean.html. Accessed: 08/2015.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] Standard Deviation. http://mathworld.wolfram.com/standarddeviation.html. Accessed: 09/2015.

[18] Apache Spark Streaming Documentation.
http://spark.apache.org/docs/latest/streaming-programming-guide.html. Accessed: 09/2015.

[19] Felix Ehm and Andrzej Dworak. A remote tracing facility for distributed systems. In *Conf. Proc.*, volume 111010, page WEMAU001, 2011.

[20] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Ak?it. *Aspect-oriented Software Development*. Addison-Wesley Professional, first edition, 2004.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison Wesley, 1994.

[22] Ganglia. http://ganglia.sourceforge.net/. Accessed: 08/2015.

[23] Glassbox. http://glassbox.sourceforge.net/glassbox/home.html. Accessed: 10/2015.

[24] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[25] Hamiltonian Graph. http://mathworld.wolfram.com/hamiltoniangraph.html. Accessed: 09/2015.

[26] Groovy. http://www.groovy-lang.org/. Accessed: 09/2015.

[27] Samudra Gupta. *Pro Apache Log4j. Second Edition*. Apress, 2005.

[28] Wang Haoyu and Zhou Haili. Basic design principles in software engineering. In *Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on*, pages 1251–1254, Aug 2012.

[29] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 2002.

[30] Richard Harper, Eno Thereska, Siân Lindley, Richard Banks, Phil Gosset, William Odom, Gavin Smyth, and Eryn Whitworth. What is a file? *Technical Report MSR-TR-2011-109*.

[31] Karla L Hoffman, Manfred Padberg, and Giovanni Rinaldi. Traveling salesman problem. In *Encyclopedia of Operations Research and Management Science*, pages 1573–1578. Springer, 2013.

[32] Emir Imamagic and Dobrisa Dobrenic. Grid infrastructure monitoring system based on nagios. In *Proceedings of the 2007 workshop on Grid monitoring*, pages 23–28. ACM, 2007.

[33] InfraRED. http://infrared.sourceforge.net/versions/latest/. Accessed: 10/2015.

[34] java.lang.instrument Package. http://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html. Accessed: 10/2015.

[35] java.lang.management package. https://docs.oracle.com/javase/8/docs/api/java/lang/management/package-summary.html. Accessed: 09/2015.

[36] java.lang.management.MemoryMXBean. https://docs.oracle.com/javase/8/docs/api/java/lang/management/memorymxbean.html. Accessed: 08/2015.

[37] jmxetric. https://github.com/ganglia/jmxetric. Accessed: 08/2015.

[38] Supun Kamburugamuve, Geoffrey Fox, David Leake, and Judy Qiu. *Survey of Apache Big Data Stack*. PhD thesis, Ph. D. Qualifying Exam, Dept. Inf. Comput., Indiana Univ., Bloomington, IN, 2013.

[39] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark*. O'Reilly, 2015.

[40] Günter Kniesel, Pascal Costanza, and Michael Austermann. Jmangler-a framework for load-time transformation of java class files. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 98–108. IEEE, 2001.

[41] Ramnivas Laddad. *AspectJ in Action. Second Edition*. Manning, second edition, 2010.

[42] Monitoring Local, Remote Applications Using JMX 1.2, and JConsole. http://www.onjava.com/pub/a/onjava/2004/09/29/tigerjmx.html. Accessed: 08/2015.

[43] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[44] Maxiumum. http://mathworld.wolfram.com/maximum.html. Accessed: 09/2015.

[45] Arithmetic Mean. http://mathworld.wolfram.com/arithmeticmean.html. Accessed: 09/2015.

[46] Statistical Median. http://mathworld.wolfram.com/statisticalmedian.html. Accessed: 09/2015.

[47] Metrics. http://metrics.dropwizard.io/. Accessed: 08/2015.

[48] Apache Storm Metrics. https://storm.apache.org/documentation/metrics.html. Accessed: 08/2015.

[49] Minimum. http://mathworld.wolfram.com/minimum.html. Accessed: 09/2015.

[50] Ian Molyneaux. *The Art of Application Performance Testing*. O'Reilly, 2009.

[51] Apache Spark Monitoring. http://spark.apache.org/docs/latest/monitoring.html. Accessed: 08/2015.

[52] Java Naming and Directory Interface. http://docs.oracle.com/javase/8/docs/technotes/guides/jndi/index.html. Accessed: 09/2015.

[53] Percentile. http://mathworld.wolfram.com/percentile.html. Accessed: 09/2015.

[54] Perf4J. https://github.com/perf4j/perf4j. Accessed: 10/2015.

[55] Permutation. http://mathworld.wolfram.com/permutation.html. Accessed: 09/2015.

[56] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[57] PostgreSQL. http://www.postgresql.org/. Accessed: 09/2015.

[58] R. Jayaprakash Reddy. *Business Data Processing and Computer Applications*. A P H Publishing Corporation, 2004.

[59] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977.

[60] Scala. http://www.scala-lang.org/. Accessed: 09/2015.

[61] TechTarget Software. http://searchsoa.techtarget.com/definition/software. Accessed: 09/2015.

[62] Apache Spark. http://spark.apache.org. Accessed: 08/2015.

[63] The Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se8/html/. Accessed: 09/2015.

[64] Apache Storm. http://storm.apache.org/. Accessed: 08/2015.

[65] Benjamin G Sullins and Mark Whipple. *JMX in Action*. Manning Publications Co., 2002.

[66] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, 2007.

[67] Pythagorean Theorem. http://mathworld.wolfram.com/pythagoreantheorem.html. Accessed: 09/2015.

[68] Java ThreadPoolExecutor. http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/threadpoolexecutor.html. Accessed: 09/2015.

[69] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A grid monitoring architecture, 2002.

[70] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[71] Andreas W. Ulrich, Peter Zimmerer, and Gunther Chrobok-Diening. Test architectures for testing distributed systems, 1999.

[72] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 2007.

[73] Mario Winter. *Methodische objektorientierte Softwareentwicklung*. dpunkt.verlag, 2005.

[74] Niklaus Wirth. Extended backus-naur form (ebnf). *ISO/IEC*, 14977:2996, 1996.

[75] Matei Zaharia. An architecture for fast and general data processing on large clusters. *Electrical Engineering and Computer Sciences University of California at Berkeley*, 2014.