Master Thesis
Computer Science

Scaling UIMA

Simon Gehring

Am Jesuitenhof 3
53117 Bonn
gehring@uni-bonn.de
Matriculation Number 2553262

At the
Rheinische Friedrich-Wilhelms-Universität Bonn

supervised by
Prof. Dr. Heiko Röglin and Dr. Timm Heuss

September 16, 2018

Contents

Contents **i**

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Implementation Requirements . . . 2
- 1.3 Outline . . . 3

2 Basics **5**
- 2.1 UIMA-Family . . . 5
  - 2.1.1 Apache UIMA . . . 6
  - 2.1.2 UIMAfit . . . 8
  - 2.1.3 UIMA-CPE . . . 9
  - 2.1.4 UIMA-AS . . . 10
- 2.2 Distributed Computation . . . 12
  - 2.2.1 MapReduce . . . 13
  - 2.2.2 Resilient Distributed Datasets . . . 14
- 2.3 Docker . . . 15
- 2.4 Related Work . . . 15
  - 2.4.1 Leo . . . 15
  - 2.4.2 v3NLP . . . 17

3 Scaling UIMA with Spark **18**
- 3.1 Technologies . . . 18
- 3.2 Implementation . . . 19
  - 3.2.1 Initialization . . . 19
  - 3.2.2 Transport . . . 20
  - 3.2.3 Process . . . 22
  - 3.2.4 Result . . . 22
- 3.3 Data Distribution . . . 23
  - 3.3.1 Serialization . . . 23
  - 3.3.2 Compression . . . 24

4 Evaluation **28**
4.1 Setup 28
4.1.1 Apache Spark 29
4.1.2 UIMA-AS 31
4.2 Results 32
4.2.1 Extensibility 33
4.2.2 Maintainability 33
4.2.3 Scalability 33

5 Summary **34**
5.1 Limitations 34
5.2 Conclusion 34
5.3 Availability 34
5.4 Outlook 35

Glossary **I**

Bibliography **IV**

Abstract

In this thesis, we will evaluate different means of scaling UIMA (Unstructured Information Management Architecture), using modern technologies like Docker, a container virtualization solution, and Spark (Apache Spark), a cluster computing framework. We will compare said implementations with the native UIMA-AS (UIMA Asynchronous Scaleout) approach in terms of processor and memory efficiency, ease of implementation and maintainability. The evaluation will be based on a specific scenario, however it will be easily configurable by exchanging very few lines of code.

Update, was wir eigentlich evaluaten, sobald das Evaluation chapter fertig ist.

Chapter 1

Introduction

Natural language is most commonly used to transmit information human-to-human. While most of this interaction takes place orally or written on paper, the digital revolution and the rise of social media increased the amount of digitally stored natural language tremendously. Gantz and Reinsel predicted 2012 that the amount of digital data stored globally will double about every two years until at least the year 2020 [GR12].

Many opportunities arise from this amount of digital data, specifically in the field of machine learning. In 2011, IBM's QA (Question Answering) system "Watson" famously outmatched professional players in the quiz show "Jeopardy!" [Fer12, ESI+12]. Kudesia et al. proposed 2012 an algorithm to detect so called CAUTIs[1], common hospital-acquired infections, by utilizing a NLP (Natural Language Processing) analysis with precomputed language models on the medical records of patients [KSDG12].

Current projections suggest a growth of revenues from the natural language processing market in North America, Western Europe, and the Asia-Pacific region of at least a factor of three until 2024 [Stad, Stab, Stac].

1.1 Motivation

Natural language is inherently unstructured and hardly machine readable. Even a seemingly easy task, like separating a sentence into words is still an ongoing research topic [PT18]. Since many NLP frameworks like Stanford's Core NLPSuite [MSB+14], The Natural Language Toolkit [BL04] and Apache OpenNLP [Apaa] exist, there was a need for generalizing the NLP approach. In 1995, the University of Sheffield began developing GATE (General Architecture for Text Engineering), a graphical development application for generic NLP problems [CMBT02]. Aside of algorithms for typical NLP tasks like tokenization, sentence splitting or part of speech tagging, GATE provides a graphical interface for developing custom algorithms in form of plugins, which can use the results of previous NLP analyses. However, the possibilities for scaling GATE applications are sparse. In 2012, GATECloud.net launched, a proprietary, cloud based GATE

[1] Catheter-associated Urinary Tract Infections

computation interface [TRCB13].

In [FL04], Ferrucci and Lally introduced UIMA, another general purpose NLP framework. Initially developed by IBM, UIMA has been open-source and is maintained by Apache since 2006. UIMA itself provides no built-in NLP analyses, but offers a common analysis structure among its plugins, which is used to combine different NLP approaches into one single framework. A popular implementation example of UIMA is IBM's QA system "Watson", which famously outmatched professional players in the quiz show "Jeopardy!" in 2011 [Fer12, ESI+12]. For this matter, UIMA was configured to run on thousands of processor cores to achieve a feasible reaction time [ESI+12]. Although this example seems to demonstrate that Apache UIMA is indeed very scalable, it has its drawbacks. The native UIMA scaling framework UIMA-AS is configured by XML (Extensible Markup Language) files, which are difficult to maintain. Furthermore it does not work out-of-the-box with other UIMA derivatives like UIMAfit.

In this thesis, we will implement and evaluate a scale-out framework for UIMA, which utilizes modern technology to ensure maintainability and scalability.

1.2 Implementation Requirements

The implementation should meet a number of requirements. First and foremost, the underlying framework, Apache UIMA, must not be limited in functionality. Since one of UIMAs strengths is the modularity and ease of plugin development, the scale-out framework to implement is required to work with native UIMA classes without any restriction. While this requirement sounds easy to meet, it is actually quite limiting since UIMA plugins can be arbitrary Java-Code. Without special care, such code is not necessarily thread-safe and thus can not be safely executed multiple times in parallel within one JVM (Java Virtual Machine).

The metrics, the scale-out framework should optimize, include:

- CPU Usage
- RAM Usage
- Document Throughput
- Byte Throughput
- Maintainability
- Implementability
- Code Quality

With CPU and RAM Usage being the percentage of actually used (virtual or real) resources, higher values being more preferable. Even if many of those resources go into administrative tasks, a scaling framework should use as many of the available resources as possible when faced with a large list of parallel jobs. Thus, a value near one should

Ich sollte hier mal nach Quellen gucken. Ist zwar meist selbsterklärend, aber ein paar Schaden nicht.

be aimed for. Obviously, at the same time, resource allocation when idling should be as low as possible.

Kam mir gerade die Idee. UIMA-AS deployed Services, die immer da sind (ie deren Modelle immer im RAM sind). Spark schießt alles sofort wieder ab, sobald es nicht wieder benötigt wird.

A little more obvious metric is the achieved throughput of data. Since collections of input documents can be shaped very differently, both, a high document and byte throughput are aimed for. When handling small documents, maybe even single words or sentences, the byte throughput is very limited to the actual input size and the large number of documents is responsible for the size of the input data collection. In such a situation, a high document throughput would be preferred to a high byte throughput. On the other hand, on larger documents, a high byte throughput is the more accurate metric since it is independent from the individual size of the input documents.

The framework is aimed to work in large academic but also industry compliant environments. Therefore the maintainability is important. It describes the amount of effort to maintain any current usage of the framework, for example changing the underlying NLP algorithm, modifying the hardware setup or making configuration changes. This is inherently more complicated by the genericness of UIMA, which allows for sophisticated plug-in initialization and configuration logic, making on-the-fly changes more difficult.

Furthermore the framework should be easy to utilize. Code that already has been written for single threaded execution should be easily reusable. This is especially important for UIMA since large repositories of plug-ins like the DKPro Core (Darmstadt Knowledge Processing Software Repository) [DKP] already exist and are infeasible to be rewritten.

TODO: Alle plugins in plug-ins ändern

Equally important as ease of utilization is the longevity of the framework. Ensuring a maximum of Code Quality is necessary to avoid large refactorings and API (Application Programming Interface) changes in the near future. In a non-academic environment, applications often have to last for a long time before replacement. Thus, a robust underlying code is aimed for.

Source? Keine gefunden ;_;

Obviously the last three bullet points, Maintainability, Implementability and Code Quality, are not easily measured, since those are subject to individual perception. However both, Maintainability and Implementability can be measured in LoC (Lines of Code). There are many static code quality analyzers, which output a score, or at least a count of quality issues. Such a score can be used to measure the quality of the frameworks code.

Code Quality Analyzers finden und citen

1.3 Outline

First, in Chapter 2, the functionality of UIMA is detailed, especially with focus on distributed computing and scaling. For this matter, both native UIMA scaling frameworks, UIMA-AS and CPE (Collection Processing Engine) are subject in said chapter. Also Spark, a cluster-computing framework, will be briefly explained since it will form the foundation of the frameworks scaling capabilities. Lastly, approaches like Leo and v3NLP will be introduced. Those are also UIMA based scaling solutions, which both suffer from different problems.

Chapter 3 will explain the scaling framework in detail. It starts with the choice of technology and proceeds to the implementation. This includes macroscopic network

Refinen, sobald Implementation geschrieben ist.

views and microscopic code details.

Then, Chapter 4 deals with the results of the framework implementation. For this, the requirements given in Section 1.2 are evaluated against the framework, UIMA-AS and the single-threaded approach.

Lastly, Chapter 5 summarizes the results, including possible limitations of the implementation. It also gives an outlook on how to extend and publicize the framework to the general public.

Chapter 2

Basics

In this chapter, the two most important technologies for the framework are explained. This is necessary to get an understanding of the technical difficulties it challenges and how it works. First, in Section 2.1 UIMA is introduced. After an in-depth introduction into the original framework designed by IBM, UIMAfit (Factories, Injection, and Testing library for UIMA) will be explained. UIMAfit builds on top of UIMA, providing the developer with a native Java interface for creating and instantiating plug-ins. Strongly related to the framework introduced in Chapter 3 are the two native scaling frameworks UIMA-CPE (UIMA Collection Processing Architecture) and UIMA-AS.

The second section of this chapter will be about Spark. While no advanced knowledge is needed to comprehend the usage of Spark as a distributed computation framework, it will still be a substantial part of the UIMA scaling framework in Chapter 3. Thus, a rather superficial overview of its structure and distribution algorithm will be given.

Ich habe es zwar nicht benutzt, aber eine kleine Einführung in HDFS könnte nützlich sein. Ich werde häufiger (im Kontext von BigData) anmerken, dass Daten vermutlich von einem verteiltem Dateisystem, etwa einem HDFS kommen und dahin geschrieben werden.

Since numerous attempts have been made, scaling UIMA in different settings, with varying implementation requirements, some related work will be presented in Section 2.4, namely Leo, providing a native Java interface for UIMA-AS, and v3NLP, a framework especially designed for usage in a medical environment and with plug-ins of such sort.

Although most important aspects and concepts of UIMA are also defined in the specifications, some minor changes and additions were made in the implementations. Since the framework must handle the actual implementation, all the presented concepts will be taken from Apache UIMA instead of the UIMA specification of 2009.

2.1 UIMA-Family

Unstructured Information Management Architecture (UIMA) Version 1.0 itself is an OASIS (Organization for the Advancement of Structured Information Standards) standard from 2009[1] that defines an interface for software components, or plug-ins, which are called analytics. Those analytics are supposed to analyze unstructured information and assign machine readable semantics to it. The standard also defines ways to represent

[1] `http://docs.oasis-open.org/uima/v1.0/uima-v1.0.html`, last accessed on 2018-09-03.

and interchange this data between analytics in favor of interoperability and platform-independence.

Apache UIMA is the open-source implementation of said UIMA specification. A common problem with Apache UIMA is scaling [DCR+15, ESI+12, RBJB+10]. It provides two distinct interfaces to analyze larger collections of unstructured data itself, with one being UIMA-AS and the other being the more dated and less flexible CPE [FLVN09]. Apache UIMA is available for Java and C++, while its scaling solutions, UIMA-CPE and UIMA-AS are only available for Java, which is why this thesis focuses on the Java implementation. Since UIMA and Apache UIMA have very similar names, which may lead to confusion it is common practice to call the implementation simply UIMA and explicitly state when talking about the specification. This practice will be adopted for the rest of the thesis.

Unbedingt bilder hinzufügen und establishen was eine Pipeline ist!!1!

2.1.1 Apache UIMA

Apache UIMA is one of few general approaches to implement NLP solutions and the only commonly known implementation of the specification with the same name. With a very modular architecture, UIMA is a popular tool that can easily be applied to a majority of NLP problems. A large part of the popularity of UIMA stems from the large DKPro Core collection of components, containing hundreds of analysis modules and precomputed language models [EdCG14], which are easily imported into existing Java projects with the build automation tool Apache Maven [DKP].

UIMA is usually used to process not a single but whole corpora of documents. A document in this sense is text, although the UIMA specification permits other data types as well. However, UIMA can not handle other data types without serializing it first. The UIMA specification, as well as the implementation do not directly pose limitations to the document size but since documents are stored in native Java String variables, which itself are implemented as arrays of chars, the practical limit of documents sizes is dependent on the JVM version and lies around one to two gigabytes [Staa] per document. In the context of UIMA, such a document is called a SofA (Subject of Analysis).

An analytic in the UIMA specification is called an *Analysis Engine* in the implementation. For the most part, an AE (Analysis Engine) is code, that gets an input CAS (Common Analysis System) and produces a number of analysis results on the SofA. Common examples for AEs are Segmentation, Tokenization, and Part-of-Speech finding algorithms [DKP]. However, since an AE contains arbitrary Java code any form of analysis can be instrumentalized by UIMA. It is defined by an XML Analysis Engine descriptor. Such an AE can either be a so-called *primitive* or an *aggregate* engine. An aggregate engine simply contains one or more other AEs, that are aggregated into one single engine.

Analysis results are stored as annotations. An annotation has at least two attributes `int begin` and `int end`, indicating the start and end index of the SofAs substring this annotation is associated with. This concept is theoretically extendable to any kind of SofA that contains any sort of subsets, for example images or audio and video streams.

However, this is impractical for reasons mentioned above. It is possible, but uncommon, to define other types of subsets on a string that – for example – permit multiple segments. Such subsets can be implemented in a custom implementation of the `AnnotationBase` class, which in turn may omit the concept of a `begin` and `end`. Since an important reason of the popularity of UIMA lies in the large AE repositories and the possibility to reuse already published code, custom annotation implementations are rarely used because it would most likely lead to incompatibilities. However, sub classing the `Annotation` class is often done to ensure type safety. Building such an annotation hierarchy leads to the creation of a Type System.

The Type System is a schema of all available types of annotations that may be associated with a current SofA, thus it provides the meta data for the annotations. It is defined by an XML Type System descriptor that is usually used by the *JCasGen*, a Java code generator for UIMA types. A XML Type System descriptor may define an super Type System from which to inherit all types. This can be used to subclass types that are not defined in the current context and encapsulate all in a single larger Type System.

The SofA, all analysis results in form of annotations that are compliant to an underlying Type System and the Type System itself are stored together in one large object, called a CAS. It is the sole input an AE gets, since it incorporates the complete context needed. The annotations are stored in a larger index to optimize for efficient access. Furthermore, a CAS object provides different Views, lightweight versions of a CAS, that store their own SofA and annotation index. These views are identified by a String, while the original data of the CAS is usually called the *Initial View*.

A *Collection Reader* implements an interface very similar to the well-known `Iterator`, namely it provides the functions `boolean hasNext()` and `CAS getNext()`. A Collection Reader usually takes the role of initializing the CAS with the SofA. Well known Collection Readers achieve this by reading from a file system, a database or `Collection` object, but any other collection may be read by implementing a custom Collection Reader. It is also configured by writing an XML descriptor file.

Multiple Analysis Engines that form a complete flow of analysis are commonly known as a *pipeline*. Since multiple AEs can be aggregated into one, a pipeline is usually an instance of a single aggregate analysis engine. Sometimes a pipeline is meant to also include a Collection Reader, however this will not be the case in this thesis. Because of the convention to call analysis results annotations, AEs are often called *Annotators*, which is not correct in general, since engines do not need to attach any annotations to the input CAS.

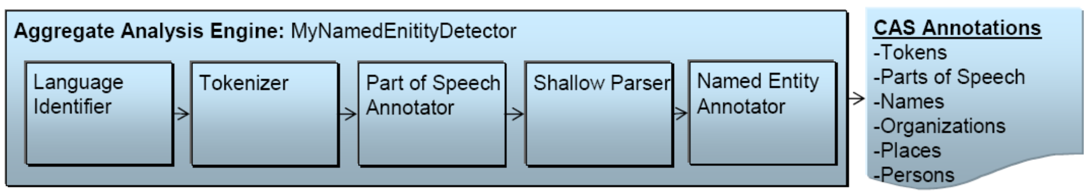

Figure 2.1: An example UIMA pipeline for named entity recognition [Apae].

Figure 2.1 shows a simplified view of an analysis pipeline for named entity recognition. Given a CAS by a Collection Reader (omitted here), the pipeline which is really just a aggregate Analysis Engine starts to identify the language of the text with an analysis engine specifically designed to do exactly this. This AE stores the results inside the CAS and forwards it to the next engine in line, which is a tokenizer.

A tokenizer annotates words, sentences and punctuation and is highly language dependent. It uses the analysis result given by the AE before to decide upon an algorithm or model to use according to the found language. After tokenization, a Part-Of-Speech Tagger annotates each words part of speech with a different annotation according to a tag set. There are a number of tag sets for most languages, as for example the Penn Treebank Project tag set[1] for English and the STTS (Stuttgart-Tübingen-TagSet) for German. The Part-Of-Speech Tagger is highly language dependent because of this. It also utilizes the results of the tokenizer, since it iterates over all annotations that are words.

Afterwards the CAS gets put into a Shallow Parser, which analyzes Part-Of-Speech tags and their semantic relation among other tags in the same sentence. In a sentence 'I like green apples.' a Part-Of-Speech Tagger would correctly decide that 'green' is an adjective and apples is a noun. However, a parser would combine those two to form 'green apples', a Noun Phrase, because 'green' is an adjectival modifier of 'apples'. A Parser may also be used to improve the results of a previous Part-Of-Speech tagging.

A Named Entity Recognizer then takes the CAS object and looks for fitting entities. This is commonly a Noun of a given list, but can be more sophisticated, depending on the wanted precision, the entity type and computation speed. After the last part of the pipeline returns, the analysis is done. The resulting CAS now includes a number of analysis results in form of annotations and can now be extracted or processed further.

2.1.2 UIMAfit

Since UIMA needs XML descriptor files to configure and describe most of its components, especially pipelines and type systems, developing for it is very XML heavy and leads to code that is hard to maintain. Apache UIMAfit is a framework that builds on UIMA, providing an interface to programmatically describe, instantiate and deploy UIMA components [OB09]. UIMAfit also provides an interface to dynamically write XML descriptor files for UIMA components. However, since it is able to instantiate and deploy said components without the need of XML files, those are mostly ignored. UIMA-AS, a native UIMA scaling framework described in Section 2.1.4, is known to be widely incompatible with UIMAfit which is what led to the creation of Leo, described in Section 2.4.1.

UIMAfit has been part of the Apache UIMA project since 2012 and is therefore officially supported [dC].

[1] `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`, last accessed on 2018-09-08.

2.1.3 UIMA-CPE

UIMA-CPE was the first method to add distributed computation capability to UIMA. Nowadays it has been replaced by UIMA-AS and is mostly obsolete. It made use of so called *CAS Consumers*, engines that do not analyze the CAS, but extract the needed analysis results from it and process the data as wanted. Common uses for CAS Consumers are writing analysis results into a database or serializing the whole CAS into a XMI (XML Metadata Interchange) file. CAS Consumers have been deprecated and replaced by AEs since 2006 UIMA-CPE because CAS Consumers do not provide any new functionality or are semantically different from Analysis Engines. Historically a CAS Consumer would not add anything to a CAS object. This convention of a reading-only Analysis Engine is often used to provide maximum modularity among UIMA engines.

Another concept exclusive to UIMA-CPE are CAS Initializers, which also have been deprecated for over a decade, but are still included in UIMA. A CAS Initializer was responsible to populate a CAS from an object given by the Collection Reader. It therefore implemented the function `initializeCas(Object document, CAS cas)`. This was used for more complex collection reading capabilities and CAS Initializers are generally seen as a plug-in to Collection Readers to extend their functionality. If – for example – only table of contents of larger documents are meant to be analyzed, then a Collection Reader would read the whole document and pass it to the CAS Initializer, which would search for a table of contents and fill the CAS with its findings and discard the rest of the document.

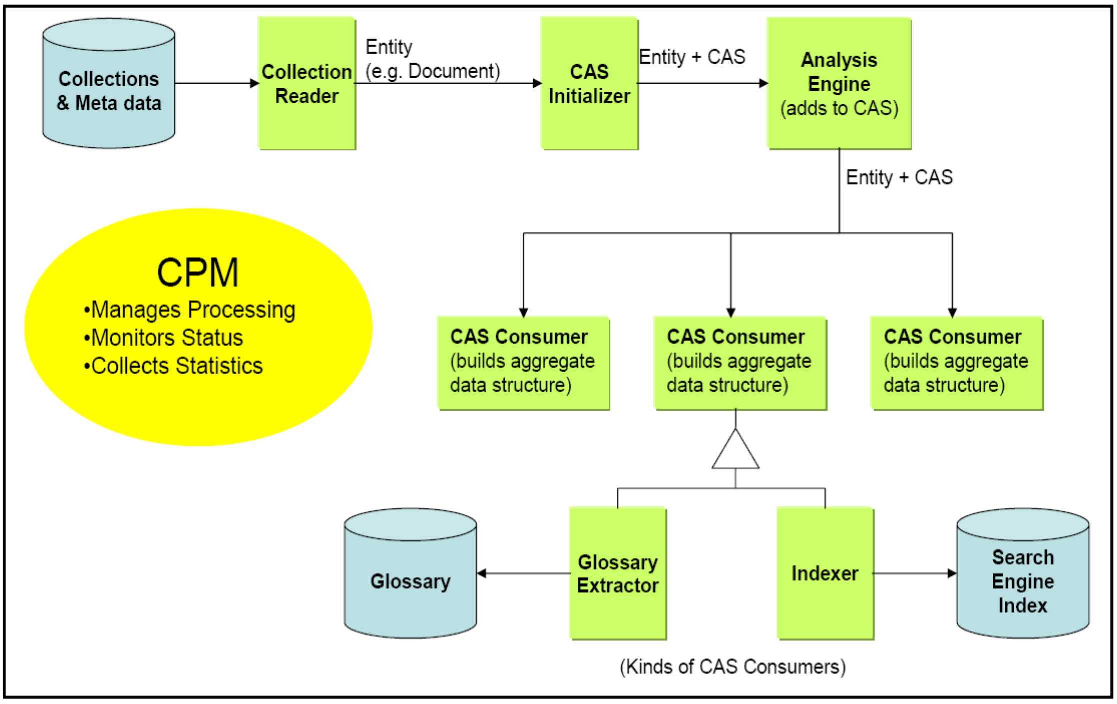

Figure 2.2: All UIMA-CPE components [Apaf].

Figure 2.2 shows a complete example pipeline. It starts with any kind of collections, maybe containing meta data. A common example would be a folder hierarchy with last modified timestamps. The Collection Reader is aware of this collection and implements an `Iterator` like interface, returning plain `Object`s. These are given to the CAS Initializer. Notice, that the CAS Initializer must be aware of what kind of entity the Collection Reader sends it. The CAS Initializer then fills a CAS object with some data from the given input `Object`. It might also create some first annotations to store meta data inside the CAS, such as the source document URL (Uniform Resource Locator) or the creation timestamp.

The CAS is then sent to the pipeline, containing one or more AEs, providing analysis results in form of annotations that are stored inside the CAS. Notice that the corresponding CAS object for a document always stays the same identical object. A CAS and its corresponding document are therefore closely associated to each other. After the analysis phase, the CAS is sent to the CAS Consumers. Those aggregate the analysis results and process it further. This process is commonly the indexing into a database or printing logs to a log file or console. Since CAS Consumers have read-only access to the CAS object they get, all of them might be processed in parallel, provided that the consumers do not interfere with each other.

All these components in combination with the UIMA CPM (Collection Processing Manager) forms the UIMA-CPE. The Collection Processing Manager provides configuration options for deployment, instantiation, and error recovery. It monitors the whole process and collects statistics. By configuration of the CPM scaling is possible either locally or on distributed machines.

For all three components introduced in this Section 2.1.3 XML descriptor files are needed for configuration. The concept of a UIMA-CPE is widely incompatible with UIMAfit, described in Section 2.1.2. UIMAfit is able to instantiate a CPE, but relies on some hardcoded default configuration, making complex multithreading applications impossible[1].

2.1.4 UIMA-AS

UIMA-AS is the successor of UIMA-CPE, providing more flexibility for scaling and deploying than its predecessor. It deploys AEs as services and registers them at a broker. UIMA-AS ships with a preconfigured instance of Apache ActiveMQ, which is an open source message broker that implements the Java Messaging Service. Other implementations can also be used though. If an UIMA-AS client now queries the broker, it submits a serialized CAS object to the input queue that is responsible for the wanted analysis. When any registered service finishes its current job, it pulls a new CAS from the broker and starts processing. This analysis process can also be multithreaded inside a single service. This is configurable by the deployment XML descriptor files of the AEs, but must be handled with care since multiple instances of Analysis Engines in the same JVM

[1] `https://uima.apache.org/d/uimafit-2.0.0/api/org/apache/uima/fit/cpe/CpeBuilder.html`, last accessed on 2018-09-08.

share static resources. After finishing the process, either successfully or by failing, the service returns the CAS object to the brokers corresponding output queue where it waits until the broker finds time to forward it to the waiting client. The described process can be seen in Figure 2.3. The user-defined AE get wrapped by a UIMA-AS controller, that handles communication with the input and output queue. These queues are provided by a broker, here ActiveMQ.

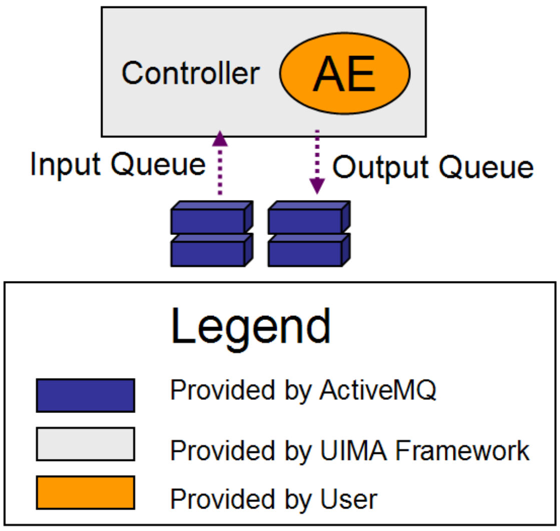

Figure 2.3: An Analysis Engine as a service in UIMA-AS [Apad].

To provide capabilities of a more complex analysis flow instead of the simple synchronous order, the user can implement what is called a *Flow Controller*. An aggregate Analysis Engine can have at most one Flow Controller, that handles what AE gets the CAS next. Usually UIMA defaults to the `FixedFlow` class, which executes AEs one after another, but more sophisticated flows can be implemented. If an aggregate Analysis Engine contains such a Flow Controller, further queues are installed. Figure 2.4 shows such an advanced pipeline containing a flow controller and two delegate Analysis Engines. Submitting a CAS to the aggregate Analysis Engine queues it into the queue of the Flow Controller (FC). When it finishes its current computation, the Controller is faced with the decision what delegate Analysis Engine should obtain the CAS for processing and appends it to the corresponding queue. Notice that said queue is also provided by ActiveMQ (or any other implementation). After analysis, the CAS is sent back to the Flow Controller, or more specifically its output queue. It may now decide to send the processed CAS to another delegate AE or stop processing and output it to the brokers output queue. The large amount of queue may seem excessive, but it is necessary to provide a synchronous execution of the Flow Controller and – if configured – the Analysis Engines.

Since the CAS object contains everything, the SofA, all analysis results, the type system, and maybe even different views, it can grow quite large over the span of a

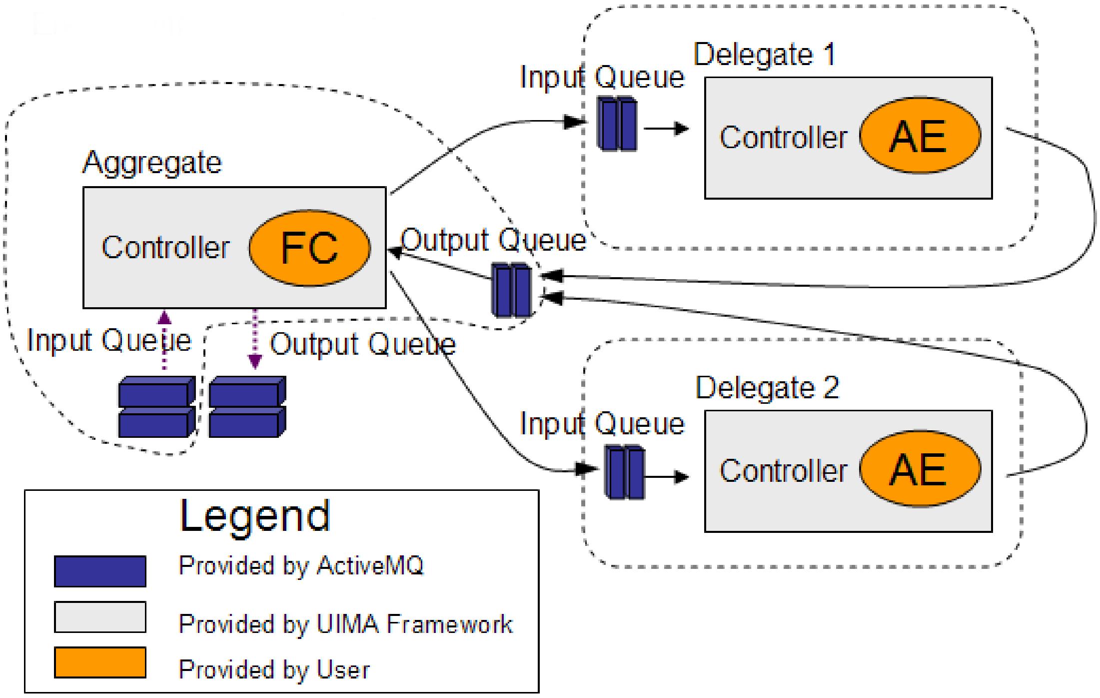

Figure 2.4: An aggregate Analysis Engine as a service in UIMA-AS [Apad].

complicated pipeline. This forms a problem in UIMA-AS, since the CAS has to be serialized for every transport inside the system, from client to broker, from broker to service, from Flow Controller to delegate Analysis Engines and the whole way back. Serialization however is a costly task and even if the underlying NLP analysis is very sophisticated, serialization might not be negligible. Epstein et al. handle this problem in [ESI+12] by avoiding serialization on local instances and introducing a sparse Delta-CAS serialization, containing only changes in respect to an original CAS.

As most parts of the native UIMA framework, UIMA-AS is configured by writing an XML descriptor, containing all the necessary data for deployment. A dynamic creation of said descriptor files is currently not possible with UIMAfit, but is provided by Leo, described in Section 2.4.1.

2.2 Distributed Computation

When handling large sets of data, a single machine may not be sufficient to solve the given task in a feasible time. In such a scenario, it would be desirable to just add more computation power to finish said task quicker. However, even if the given task permits parallelization, which is generally not obvious, distributing a problem among multiple machines, or similarly processing cores, is not trivial. For many problems a

Bisschen history? Eigentlich langweilig, ich erkläre hier schon extrem simple Dinge.

large administrative and communicative overhead aggravates the effort to parallelize.

In this section, some models for parallel and distributed computation are describes. Those concepts aim to generalize the problem of distributing a task while at the same time try not to be too restrictive in their interface.

2.2.1 MapReduce

MapReduce is a programming model for distributed computation of large sets of data on clusters of processing cores and usually multiple machines. Google introduced the MapReduce model in 2003 and used it a few years before announcing 2014 to switch to a less restrictive framework [DG08].

The MapReduce process consists of three phases, *Map*, *Shuffle*, and *Reduce*. The shuffle phase is usually provided by an implementing framework, both other phases are to be implemented by a user. Let the input data set be C, with identifiers I. More specifically this means that the following holds:

$$\forall c \in C : \exists! i \in I : i \text{ is associated with } c.$$

Furthermore let K be a set of keys and V a set of intermediate values. Then the map function maps the input data with the associated identifier to a list of key-value pairs:

$$\text{map} : I \times C \to (K \times V)^*$$

Then the `reduce` function reduces a key and a list of all associated intermediate values to a single intermediate value:

$$\text{reduce} : K \times V^* \to K \times V$$

The `reduce` function is often described with a range of just V, because it never changes its parameter of K and just passes it through. After all value lists have been reduced to contain only a single value $v \in V$, they form the final output $(K \times V)^*$.

The canonical example for this model is the problem of counting the number of occurrences for each word in large documents or even larger corpora of documents [DG08]. Recall that for given input documents C and corresponding identifiers I, which might be filenames or URLs, one expects a list of key-value pairs containing $k \in K$, an identifier for a single word (likely the word itself), and $v \in V$ an integer value describing the words occurrences. Figure 2.1 shows an example implementation of said behavior. Notice that the pseudocode class `Word` does refer to a substring containing a single word and not the unit of data.

First all documents C and their identifying information I are put into $|C|$ instances of the `map` function. The results are $|C|$ lists of word-integer pairs. Notice that these intermediate results are not yet distinct. This means that several entries of even the same list might be equal if the corresponding word occurs more than once in one document. Now follows the shuffle operation, which collects intermediate results with the same key on as few machines as possible. This is a costly procedure, since data must be sent over the network. In the third phase, the reduction algorithm gets a word and a number of corresponding counting integers which it just adds and returns. Notice that – in this example – the first execution of the `reduce` function will receive a word and a list of ones. This is because the `map` function initialized each word counter with exactly one.

All intermediate results per word can now be reduced further until only one value remains, which is the final output value. Since the MapReduce model does not define an ordering on the lists given to the `reduce` function, it must be associative and commutative to always yield the same result regardless of the inputs ordering. However, MapReduce implementations usually guarantee a fixed ordering to simplify programming the `reduce` function.

```
List<Pair<Word, Integer>> map(Id docIdentifier, Text docText) {
  List<Pair<Word, Integer>> result = new List<>();
  for(Word w in documentText) {
    result.append((w, 1));
  }
  return result;
}

Pair<Word, Integer> reduce(Word w, List<Integer> intermediate) {
  Integer result = 0;
  for(Integer i in intermediate) {
    result += i;
  }
  return (w, result);
}
```

Listing 2.1: Example pseudocode implementation of the MapReduce model to count word occurrences.

Dean and Ghemawat found in [DG08] that many real world applications are describable in the MapReduce model. However, it is still very restrictive and has been abandoned by Google for this very reason. A popular open-source implementation of MapReduce is Apache Hadoop, or more specifically Hadoop MapReduce. It is therefore part of Apache Hadoop, a collection of utilities to handle large amount of data in computation. Apache Hadoop is popular for the HDFS (Hadoop Distributed File System), a high performance distributed file system.

2.2.2 Resilient Distributed Datasets

RDDs (Resilient Distributed Datasets) provide an interface that are very similar to the native Java `Collection`. They were initially developed in 2012 by Zaharia et al. in [ZCD+12] as a response to iterative algorithms being inefficient in current computing frameworks such as MapReduce.

An RDD can be created by either of two ways. First, stable data collections such as native Java `Collection` instances or a number of files in a file system can be initialized as RDDs. The other way of creating is a deterministic operation on an already existing RDD. These operations are called *transformations* by Zaharia in [ZCD+12]. Since RDDs are immutable collections, calling such a transformation on an existing RDD is the only way of obtaining the wanted resulting RDD.

Being immutable, RDDs allow to be materialized lazily. This means that the issued transformations are executed just in time, when a materialized form of the RDD is necessary. This happens on actions like counting the number of objects in the RDD or serializing it to a file. Before executing these transformations, an acyclic graph is built to represent the necessary transformations of computing said RDD. Furthermore, RDDs are sliced into partitions, atomic pieces. These partitions can then be distributed among the clusters nodes and computed whenever necessary. For the distribution of said partitions, Spark utilizes the knowledge of the issued transformation to evaluate dependencies of resulting RDDs to their predecessor. This means for example, that a `count` operation that follows a large amount of transformations does not necessarily force Spark to actually compute all these transformations. For example a transformation `crossProduct` on data sets X and Y is guaranteed to result in a collection of size $|X| \cdot |Y|$. Materialization of $X \times Y$ is not necessary. This technique of lazy transformation provides a simple way of fault tolerance since only the RDD lineage and not the complete materialized RDD itself must be replicated among the different machines.

The only current implementation of RDDs is Apache Spark, introduced along with RDDs in 2012 [ZCD+12].

2.3 Docker

Urgh, reden über Docker

2.4 Related Work

Taking effort to scale UIMA has been done numerous times. The most prominent result was the implementation of the question answering system Watson [ESI+12]. This approach used native UIMA-AS, although the engineers changed the UIMA-AS source code themselves. Other approaches that are trying to be generic solutions while not posing too many restrictions or being too intrusive into the native UIMA concepts or even code, are Leo and v3NLP.

2.4.1 Leo

The Leo framework was developed by VINCI (VA Informatics and Computing Infrastructure) to allow for the easy deployment of annotators in an UIMA-AS environment [oVA]. Since it wraps around most concepts of UIMA and UIMA-AS, its architecture closely resembles UIMA-AS. This can be seen in Figure 2.5. Given a number of instances of `LeoAEDescriptor`, which are compatible with the native UIMA Analysis Engine, Leo is able to automatically write an UIMA-AS deployment descriptor file and use it to deploy a `Service` instance. This also is just a wrapper around UIMA-AS native service, which tries to register to a given broker implementation. Leo does not provide a broker implementation by its own, but depends on an existing UIMA-AS installation, which in turn provides ActiveMQ, as described further in Section 2.1.4. Given such an instance of a

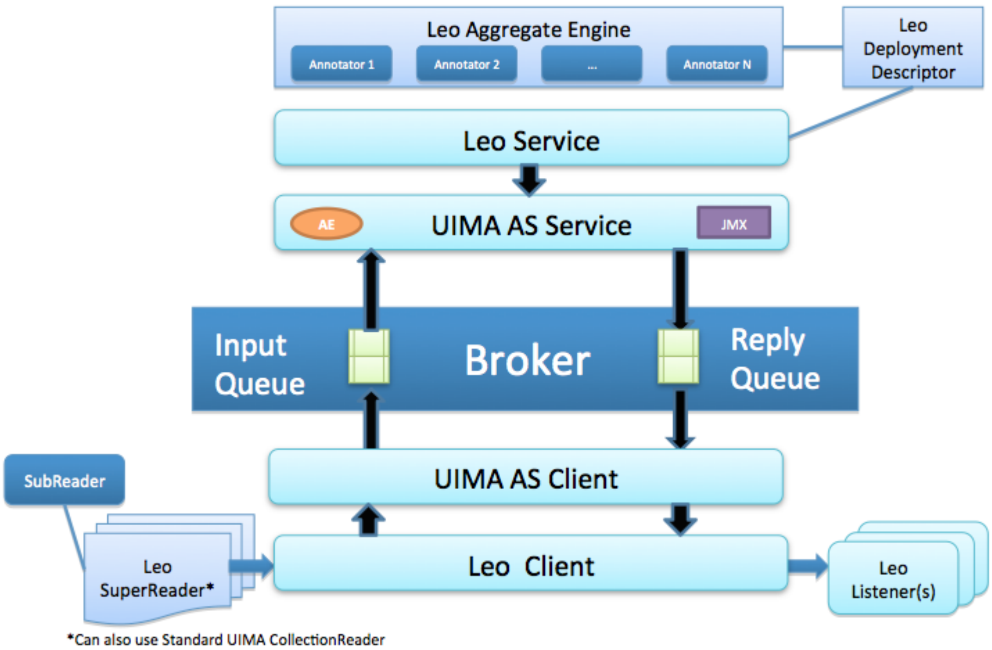

Figure 2.5: The Leo architecture wrapping around UIMA-AS [oVA]

broker-service architecture, Leo is now able to perform requests to said broker. The Leo `Client` class provides access to the native UIMA-AS capability of querying the services for analysis results. For this, Leo utilizes the class `LeoCollectionReader`, which also is just a wrapper around the native UIMA Collection Readers and can be easily converted to one and vice versa.

The Leo source code repository[1] shows that the last change was in January 2018, without providing any comments or history[2]. This, and the fact that the project only has four contributors and no pull request as of the time of writing shows that the Leo framework project is not well-maintained.

A static code analysis with FindBugs[3] on the current source code[4] shows 29 potential

[1]`https://github.com/department-of-veterans-affairs/Leo`, last accessed on 2018-09-14.
[2]`https://github.com/department-of-veterans-affairs/Leo/commit/038f7d5c542fa564c2997403769943ac47638692`, last accessed on 2018-09-14.
[3]`http://findbugs.sourceforge.net/`, last accessed on 2018-09-14.
[4]Commit 038f7d5c542fa564c2997403769943ac47638692 on 2018-09-14

bugs, of which seven are rated as 'scary' by FindBugs. Another code analysis tool, SonarLint[1] which also checks the coding style found a total of 626 bugs, vulnerabilities and code smells.

2.4.2 v3NLP

The framework v3NLP was first introduced in 2011 by Divita and Trietler in [DT11] as a successor to HITEx[2]. Both, HITEx and v3NLP were initially built on top of the Gate framework but switched to UIMA since. v3NLP provides capabilities to process NLP tasks especially in the medical field. It is therefore heavily influenced by the context of a medical environment. It provides 34 pre-composed pipelines, all related to clinical text analysis. However, it enforces the use of the CHIR[3] Common Model, which is an ontology of labels from already existing NLP systems. This model is encoded in a default type system the framework is using and it can be extended at will. This is supposed to ensure interoperability between different NLP components that were developed for v3NLP. Notice that this poses a restriction on the user of the framework, since they are forced to always include this type system [DCT+16]. The framework also provides scaling capabilities that internally utilize the native UIMA-AS scalability.

While the v3NLP framework provides many functionalities for NLP research in the medical context, it is a large project in respect to UIMA, UIMA-AS and Leo (described in Section 2.4.1). All the v3NLP git repositories combined span 23.3 GB of content and contain about 1.8 million lines of Java code in the latest commit alone. Thus it is a heavyweight tool, which is used best in a medical environment where as many capabilities as possible are actually used.

At the time of writing, the last commit to the framework repository was on December 2017, nine months in the past. According to the v3NLP website[4] this is expected to be the very last commit.

[1] `https://www.sonarlint.org/`, last accessed on 2018-09-14.
[2] Health Information Text Extraction
[3] Consortium for Healthcare Informatics Research
[4] `http://inlp.bmi.utah.edu/redmine/docs/v3nlp-framework/News.html`, last accessed on 2018-09-14.

Chapter 3

Scaling UIMA with Spark

In this chapter we will first discuss the choice of Spark as a distribution technology. Afterwards the framework implementation details will be documented with a special focus on data distribution, namely Serialization and Compression.

3.1 Technologies

In Section 2.2, two fundamentally different computation models were introduced, namely MapReduce and RDDs. Both are generic models how to parallelize and distribute workload among multiple processing cores. While MapReduce poses more restrictions on the underlying function, it is also more widely used than RDDs. Gopalani compared 2015 in [GA15] both methods against each other, choosing Spark as the RDD implementation and Hadoop Map Reduce as the implementation for the MapReduce model. For the example case of K-Means, he finds that Spark performs roughly 50 % better in terms of pure speed. While K-Means is a valid choice for an algorithm that can be used in many fields, it is not representative for all parallelizable algorithms. Since such a choice does not exist and UIMA poses no restrictions on what classes of algorithms can be run inside an AE, the choice of whether to use RDDs or a MapReduce implementation is still non-trivial.

Many algorithms in NLP work according to language models, which are usually not only language- but also domain-dependent. To get general purpose language models, the Wikipedia corpus is often input into a machine learning algorithm. However, a general purpose Wikipedia model does often not suffice in terms of domain specific vocabulary and a custom model must be trained. This is where Spark comes into play, because it claims to perform better on *iterative tasks* than a MapReduce approach [Wil17]. With language model training being a substantial part of many NLP problems, RDDs seem to suit the NLP needs better than a MapReduce approach. Being the only current implementation of RDDs, the choice of distribution framework falls to Spark.

3.2 Implementation

The framework presented here consists of several classes that implement different tasks. The frameworks main class `SharedUimaProcessor` delegates all work to the corresponding classes. One complete execution of the framework, say an analysis of one corpus of documents, contains several steps to make. First, the framework must be instantiated. This is done by the actual user. They then order the instance of `SharedUimaProcessor` to process a pipeline according to the output of a given collection reader. To accomplish this, the framework has to read the collection, wrap documents into CAS objects and send them along with the serialized pipeline description to its workers. After the analysis part is complete, the CAS objects get sent back where they are wrapped into a `AnalysisResult` object to get access.

Figure 3.1 shows the flow of documents in a UML activity diagram. After getting read its wrapped inside a CAS and distributed among worker nodes. There the CAS are processed and then sent back. The following sections will describe these steps in detail.

3.2.1 Initialization

The initialization of the framework consists of two parts. First, since it depends on a running Spark infrastructure, one of such must be installed. Estimating the performance of algorithms on Spark clusters is possible, but hard [WK15, GA15], especially because it heavily depends on the actual code being processed. Since both, UIMA and the framework presented here provide the capability to process documents with arbitrary Java code, no assessment can be given here. By the architecture of the framework the number of usable machines are capped by the number of documents. However, since the corpus to process is usually large, especially in a situation when utilizing a scaling framework is necessary, this poses no sensible limitation. Another trivial bound is a minimum number of machines, since a single machine would process all CAS faster on a native UIMA instance than a Spark cluster containing only one worker could. This is because a Spark cluster still has to administrate its only worker. The CAS has to be serialized and deserialized twice. The local UIMA instance skips this.

Given an Spark cluster, or more specific, the corresponding Java object `JavaSparkContext`, the framework itself must be instantiated. This is useful to process on multiple Spark clusters within the same JVM. The class `SharedUimaProcessor` provides a constructor

```
SharedUimaProcessor(JavaSparkContext, CompressionAlgorithm, CasSerialization, Logger)
```

While the first parameter `JavaSparkContext` was explained above as providing the necessary API to Spark for the framework to use, the others have not yet been described. The `CompressionAlgorithm` and `CasSerialization` parameters are optional and may be null. They are implementations of interfaces provided by the framework to specify how CAS should be serialized and compressed for network transport. This is explained further in Section 3.3. The last of the constructors arguments is an implementation of the popular logging framework interface `org.apache.log4j.Logger`[1].

[1] `https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Logger.html`, last ac-

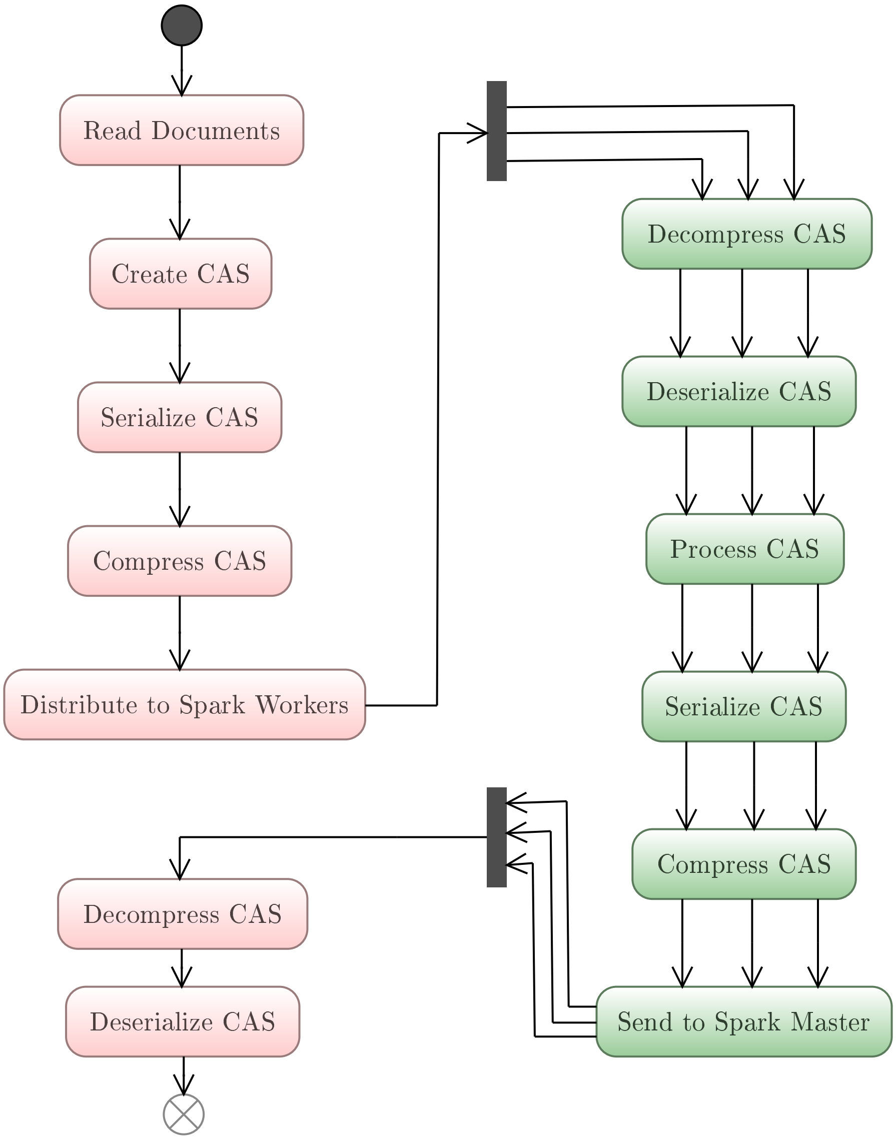

Figure 3.1: An UML (Unified Modeling Language) activity diagram of the CAS distribution process.

3.2.2 Transport

Depending on how Spark is configured, the user code either is executed directly on the master node (standalone) or on an unrelated machine that sends all necessary parameters to the master node (cluster mode). Usually the standalone mode is chosen only for development or trivial clusters of a single machine, because the underlying call to execute a function is synchronous in such a configuration, therefore the process is not monitorable

cessed on 2018-09-13.

until the call returns. Figure 3.2 shows the whole process for a cluster mode configuration. Given the initialization described in Section 3.2.1, a collection reader would read documents into a collection of CAS objects. These are then serialized and compressed by algorithms also provided by the user. This is described further in Section 3.3. However, after successfully compressing the CAS, it gets sent to the Spark master node. Notice that this transmission is not necessary in standalone mode, since the `SharedUimaProcessor` is then instantiated on the master node itself. Not only the CAS are needed to analyze

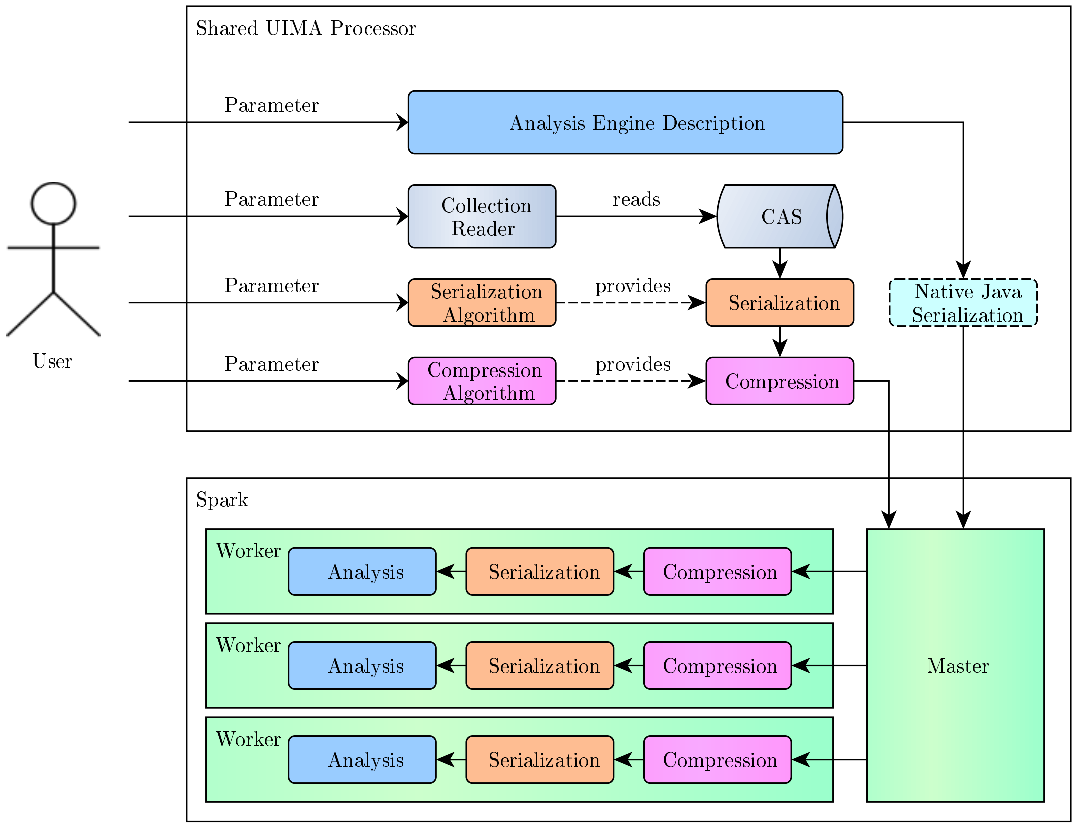

Figure 3.2: A schematic for the Shared UIMA Processor in cluster mode.

the documents but also the analysis algorithm itself. Analysis Engines, however, are not serializable by Java, since they do not implement the required interface. This is why the framework does not accept instantiated pipelines in form of an aggregate `AnalysisEngine`, but only non-instantiated pipelines as `AnalysisEngineDescription`, which implements the `Serializable` interface. The `AnalysisEngineDescription` itself can not be executed but can be used to instantiate the corresponding AE. This happens on all worker node simultaneously and is combined with a non-trivial amount of computation time, since authors of Analysis Engines are encouraged to load data on the actual instantiation. Many NLP related algorithms need trained models or dictionaries that are relatively large. Both, source?

UIMA and Spark provide broadcast read-only variables to load such larger models only once, possibly saving on network and computation resources.

3.2.3 Process

As shown in Figure 3.2, one can see that a single pipeline is deployed per worker node, or more specifically per JVM. This is important to avoid a limitation of UIMAs generic nature. Since AEs consist of arbitrary code, they are generically not thread-safe. To meet the condition not to restrict any UIMA capabilities, the framework must not pose any restrictions on the Analysis Engines, which includes a guarantee for thread-safety. Instantiating exactly one pipeline per JVM circumvents the problem for the most part, as even static variables accessed by one instance are invisible to other instances. It is to mention that threading issues can still be encountered when accessing external data. The other way such problems may occur is when deploying an aggregate Analysis Engine containing a delegate AE multiple times. In such a case a custom flow controller could be provided to execute both AEs simultaneously. However, this is also a problem in UIMAs original architecture and can be easily avoided by just using the default Flow Controller or by not adding the same Analysis Engine multiple times in one pipeline.

3.2.4 Result

The result type of the framework differs heavily from other frameworks like UIMA-AS. The resulting class, `AnalysisResult` is very similar to a `List<CAS>`, with a few but substantial differences. Figure 3.3 shows the UML class diagram of the `AnalysisResult` class. Internally it stores a `JavaRDD<SerializedCAS>`, which is a class of the Spark context. It delegates almost all commands to the underlying `JavaRDD` object, however, some functions that are sensible in the UIMA environment are also provided by this class, for example a `saveAsXmi` method, that saves all containing CAS objects into a folder. A `JavaRDD` behaves much like a `List` outside the Spark context. The `SerializedCAS` is an internal class that represents a CAS that was serialized and compressed with the corresponding algorithms. It simply contains the serialized `Byte` array and provides an interface for deserialization by delegating the calls to the corresponding user provided algorithms. It also exposes a `size()` function to get the number of bytes needed by the compressed and serialized CAS. This is useful for evaluating algorithms that implement the `CasSerialization` and `CompressionAlgorithm` interfaces. The `SerializedCAS` class itself implements the native Java `Serializeable` interface and is therefore serializable by the JVM.

While `JavaRDD<SerializedCAS>` behaves similarly to `List<SerializedCAS>`, it is yet fundamentally different in what it does exactly. The native Java `Collection` implementations all store data on the local JVM and access them whenever they are needed. However, a `JavaRDD` is still a distributed data set among all the worker nodes that provided at least one of the resulting `SerializedCAS`. It can now be collected by the `AnalysisResult` function `collect`.

Then all CAS objects get sent back to the master node. This is a fundamental difference to UIMA-AS. Notice that collecting all analysis results is usually *not* desired when talking about big data collections, because a single machine is likely not able to receive these large amounts of data or store it in a timely matter. Instead, a CAS Consumer should be provided at the end of the pipeline. Recall from Section 2.1.3 that a CAS Consumer is the same as an Analysis Engine in terms of implementation. However, such a CAS Consumer would extract the needed analysis results, which are most likely only a sparse subset of all given annotations, and use or store them. This storage is usually done in a database or a distributed file system like HDFS to obtain all results in one place without the need to wait for a single hard drive to write large amounts of data.

3.3 Data Distribution

Since all the input data, in form of documents, and output data, in form of analysis results, must be transmitted over a network, be it virtual or real, the serialization of larger Java objects play a role in performance. Since both, the input and the output, are stored inside a CAS object it suffices to find a suitable serialization algorithm for those. However, finding an optimal algorithm is not trivial and usually even depends on the input data. Larger documents produce larger CAS, which in turn need a longer time to be deployed to the corresponding Spark workers. However, small documents still are no guarantee for small CAS sizes, since analysis results can be of arbitrary size and number, depending on the UIMA pipeline.

Furthermore it can be useful to compress serialized data, depending on the network setup and the serialization algorithm. Most native UIMA serializations produce XML files, which are very verbose and well compressible. Compression algorithms specifically designed for XML files achieve packing ratios of up to 80 % [GS05, MPC03, Sak09]. However, such algorithms often come at the price of a relatively high runtime. This is especially undesirable if the transmitted data is small or the serialization sparse and the expected compression ratio is low.

Since an optimal choice for both, serialization and compression, is not possible for the general case the framework exposes two interfaces, namely `CasSerialization` and `CompressionAlgorithm`. Figure 3.4 shows the relationship between the framework main class `SharedUimaProcessor` and both interfaces. Additionally, two implementations that are already provided by the framework are shown in the model.

3.3.1 Serialization

In [ESI$^+$12] Epstein et al. explain how serialization of CAS was an important bottleneck and a problem to solve. They configured UIMA-AS in several ways to serialize only the parts of the CAS object that are needed for further analysis. Obviously this can not be done in the general case when the underlying analysis algorithms are unknown, which is why the framework takes an instance of `CasSerialization` as an optional parameter.

An instance of said interface implements two methods with the signatures shown in Listing 3.1.

```
public byte[] serialize(CAS cas);
public CAS deserialize(byte[] data, CAS cas);
```

Listing 3.1: CasSerialization method signatures

While the signature of the `serialize` method is intuitive, this does not immediately apply to the `deserialize` function. Here, a previously created CAS object is given as a parameter for two reasons. First, UIMA allows for the configuration of a custom `CasInitializer`, which can alter the CAS object immediately after creation. Although the usage of `CasInitializers` has been deprecated since at least 2006, it is still a feature of UIMA and must therefore be taken care of [Apaf]. By creating a new CAS on the target JVM, the framework first executes the `CasInitializers` and then passes the resulting CAS to the `deserialize` function. The second reason for this additional parameter is to pass the current UIMA type system. The serialized data might include annotations of types that are unknown to the native UIMA type system and therefore must be defined before deserialization. Although a parameter `TypeSystem` would have been sufficed, the first reason implies the requirement of a complete CAS parameter. Since the created CAS already includes the full type system description, available by `cas.getTypeSystem()`, the framework abstains from passing another parameter to the `deserialize` method. If `CasInitializers` get removed from UIMA, this might be a feasible change in the future.

The framework already ships with two implementations of the `CasSerialization` interface, namely `XmiCasSerialization` and `UimaCasSerialization`. The `XmiCasSerialization` creates complete XMI files, containing the SofA, all analysis results and even the used type system description. To accomplish this, it uses the UIMA `XmiCasSerializer` class. Thus, the `XmiCasSerialization` implementation of `CasSerialization` acts as a mere wrapper. The second serialization algorithm `UimaCasSerialization` also just wraps around the native UIMA class `Serializer`, which is the same serialization algorithm UIMA-AS uses to distribute and retrieve CAS objects. As shown in Figure 3.4, both `XmiCasSerialization` and `UimaSerialization` are also implementing a singleton pattern, because no instance dependent information must be stored for either of them. However, one could implement a `CasSerialization` that stores context dependent information, for example the underlying type system.

3.3.2 Compression

Since the compression results are very dependent on the use case, data size and serialization algorithm, the framework provides the user with a `CompressionAlgorithm` interface. An implementation of said interface exposes two methods with signatures as shown in Listing 3.2.

```
public byte[] compress(final byte[] input);
public byte[] decompress(final byte[] input);
```

Listing 3.2: CompressionAlgorithm method signatures

Completely abstracted from any UIMA concept, this interface simply expects two functions, `compress` and `uncompress` to behave such that for every input `byte[] X` holds that `X = decompress(compress(X))`. While this is the only technical requirement for this interface, it is usually desired to have `|X| > |compress(X)|`. Since both methods act UIMA unaware, reducing the object size by omitting parts of the CAS is not possible without deserializing the CAS first, a step that is defined in the `CasSerialization` interface and not accessible from this context.

The framework ships with two implementations of the `CompressionAlgorithm` interface. If defaults to the `NoCompression` class, simply implementing the identity with `X = compress(X)`, effectively disabling any kind of compression. This is useful if network delay is negligible, especially in virtual networks inside a single machine or on low latency environments. A compression algorithm would need computation time to process all transmitted CAS, while saving only a minimum of transfer time. Secondly, the class `ZLib` implements the DEFLATE compression, which is a general purpose lossless compression algorithm, commonly used in ZIP files. As seen in Figure 3.4 both classes implement the singleton pattern, because no instance data has to be stored for either compression algorithm. However, one could implement an algorithm that stored such information, for example a complete corpus spanning dictionary.

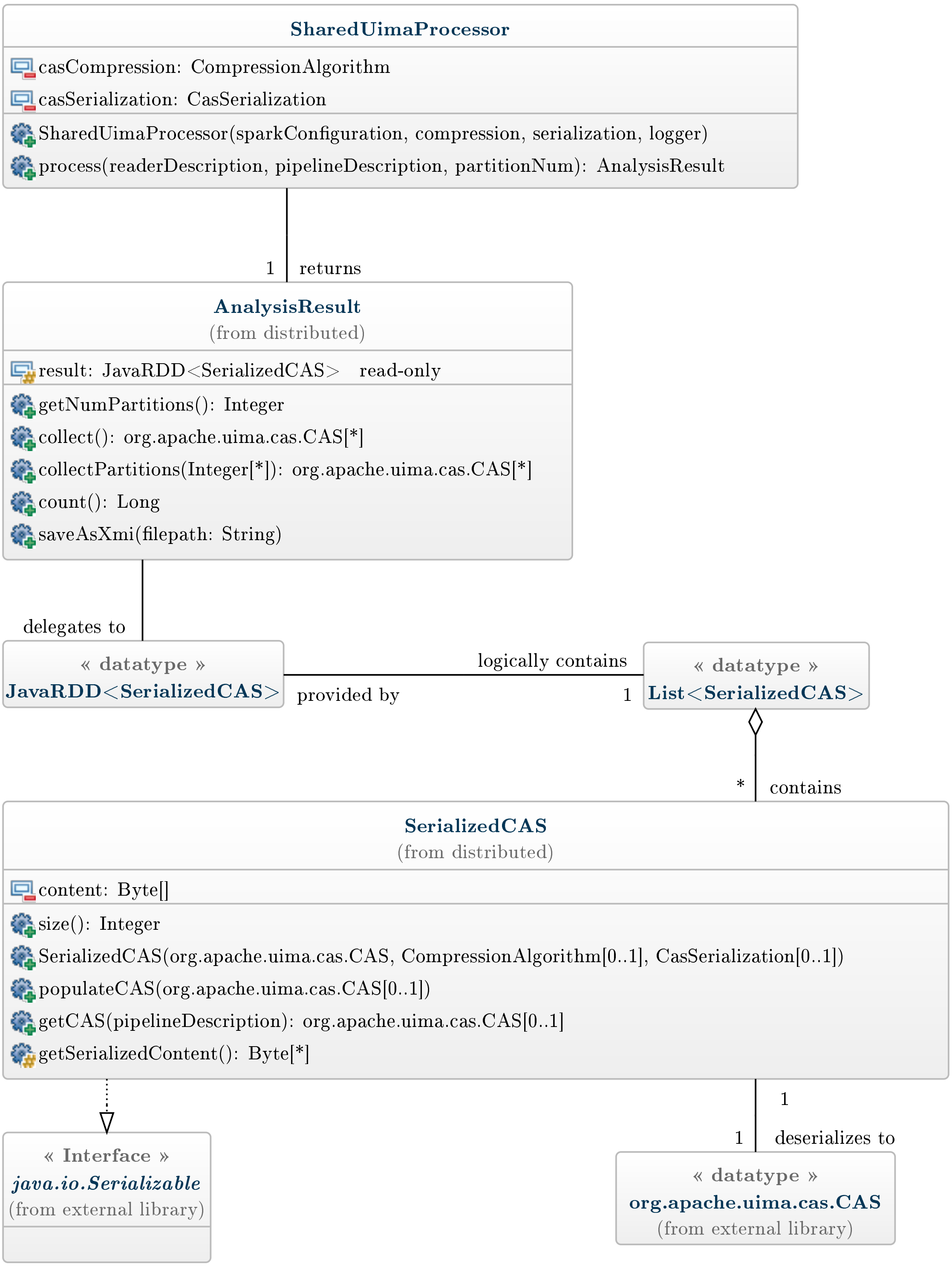

Figure 3.3: An UML class diagram of the frameworks result classes.

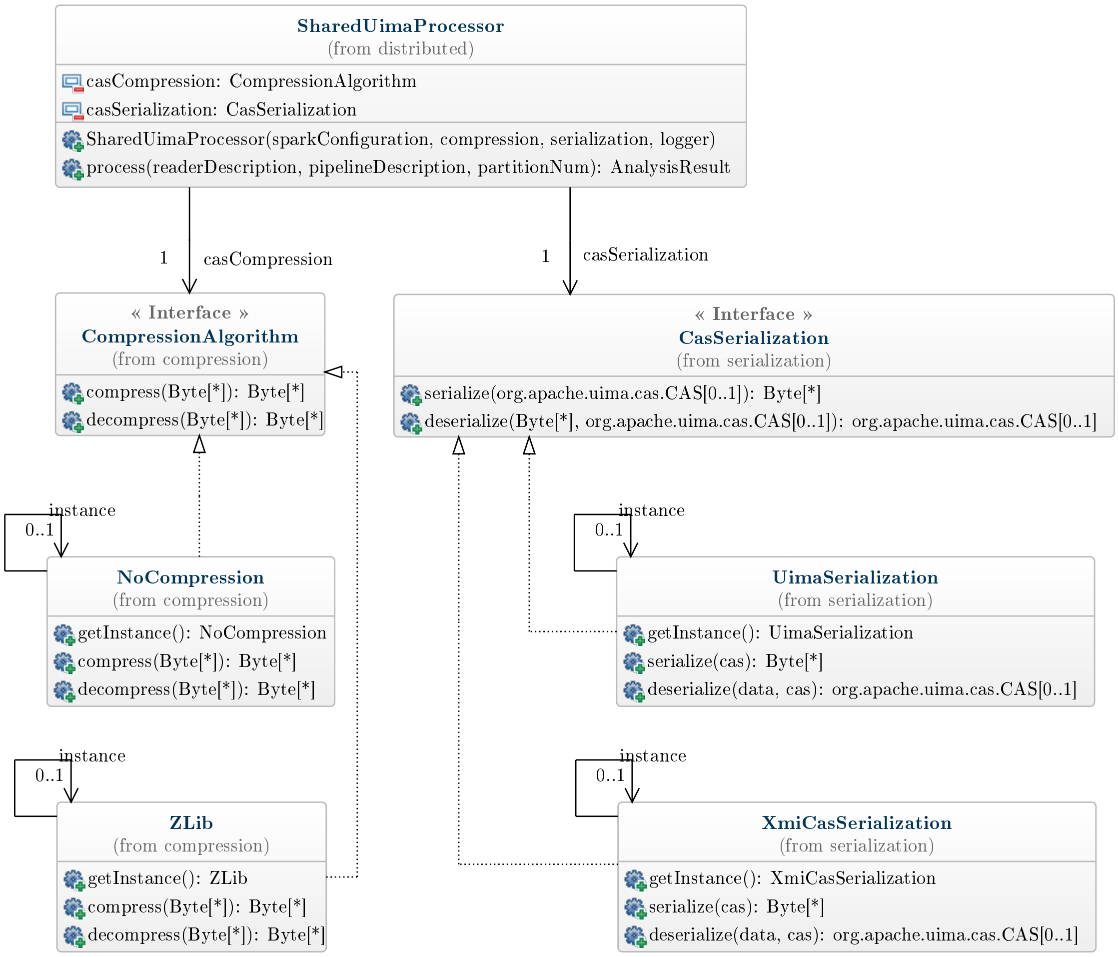

Figure 3.4: An UML class diagram of the serialization and compression interfaces.

Chapter 4

Evaluation

In the following sections, the framework introduced in Chapter 3 will be compared to existing UIMA distribution approaches. For this reason, a sophisticated evaluation architecture was deployed on a virtual network, which is described in Section 4.1. The comparison will follow along the metrics given in Section 1.2 'Implementation Requirements'.

Was wir definitiv vergleichen können ist Extensibility und Maintainability, also weiche Metriken. Memory Consumption hab ich nicht mit geloggt, das empfand ich als zuviel Aufwand für zuwenig return. Hintergrund ist, dass sowohl UIMA-AS, als auch Spark nur kleinen Overhead haben. Wir reden hier von <1GB. Jede halbwegs erwachsene Pipeline, mit mindestens einem oder zwei Modellen übertrifft das. Somit bin ich grundsätzlich einfach davon ausgegangen, dass Speicher "genug" da ist, also nichts geswappt wird (und offensichtlich nichts in den OOM-Killer läuft). Ich denke das ist sinnvoll so. Hat man einen Rechner, auf dessen RAM die Pipeline, inklusive aller Modelle, passt, dann passt da auch noch Spark/UIMA-AS drauf. Passt die Pipeline nicht, hat man sowieso Pech.

4.1 Setup

Daniel fragen was ich für Hardware hatte.

Setting up a testing and benchmarking environment for the framework given in Chapter 3 and UIMA-AS is not trivial when accounting for comparability. This is because the given framework and UIMA-AS function fundamentally different from each other. The following descriptions for Docker architectures suggest a highly similar setup, but even the initialization of both systems differ in terms of concept. While the actual analysis code inside the aggregate AE is provided at the UIMA-AS services inherently, this does not hold for a Spark cluster. Such a Spark cluster waits for tasks until one arrives and orders it to execute code according to a JAR file, whose URL is given in a parameter. However, the same cluster can work on other problems as well, without reconfiguration and reinitialization. Different tasks can even be processed in parallel, since the Spark is not just usable by the frameworks API.

This concept is fundamentally different from UIMA-AS, where a service is bound to exactly one aggregate Analysis Engine and therefore one pipeline. If another NLP algorithm has to be processed, another UIMA-AS service must be instantiated and registered to the broker, at another queue name. While the old services are still up, system resources for those can not be reallocated and idle until a CAS object for exactly this AE is sent to the broker.

These two concepts both have their advantages and disadvantages. The way Spark allocates resources for tasks makes it more flexible to use. Different pipelines can easily be processed without making changes to the computation cluster. A cluster of UIMA-AS services would need heavy reconfiguration for such a case. However, since UIMA-AS services do not discard their resources after finishing the current task, a pipeline does not need to be reinitialized if another CAS should be analyzed. Since NLP algorithms commonly depend on large dictionaries or language models, this saves on loading time and especially disk I/O. A Spark cluster would immediately forget the context of the current pipeline and would have to reinitialize it.

This also comes into play when evaluating the running time of both concepts. Given a more sophisticated pipeline and a ready-to-use Spark cluster on one hand and a number of UIMA-AS instances on the other hand, the pipelines in the UIMA-AS services would have already been initialized, while the pipelines in the Spark architecture are initialized on-the-fly. This holds not true for AEs that load their resources lazily. However, no such assumption can be made in general.

To evaluate both frameworks, and the single threaded approach to get a sense of the administrative workload, a cluster of multiple computers were needed. To simulate a network of machines, Docker was chosen as a virtualization concept for multiple reasons. First, Dockers resource footprint is way smaller than one of a virtual machine, therefore more resources can be allocated for the actual benchmark. Secondly, a docker image is completely reproducible. While this can also be achieved with configuration management tools like Chef[1], Puppet[2] or Ansible[3], defining a Dockerfile allows for easy reproduction and configuration.

Notice that network transport delay between the various components are trivial in a simulated network without any artificial delay. For this reason, compression was disabled for the framework presented in this thesis.

4.1.1 Apache Spark

In Figure 4.1, one can see the Docker architecture used in the evaluation. A custom created image for Spark was used for both, the master and the worker nodes. This image is based on OpenJDK and will be available as described in Section 5.3. The Spark worker nodes can be scaled at will by a simple Docker parameter. Both containers, *jar-provider* and *document-provider* are instances of nginx[4] HTTP servers. The document provider simply provides access to the complete corpus of documents via ordinary HTTP GET

[1] `https://www.chef.io/chef/`, last accessed on 2018-09-16.
[2] `https://puppet.com/de`, last accessed on 2018-09-16.
[3] `https://www.ansible.com/`, last accessed on 2018-09-16.
[4] `https://www.nginx.com/`, last accessed on 2018-09-16.

requests. These documents could have also been simply copied into the *submitter* image, however an HTTP provider server for the document corpus was chosen to equalize I/O delay both, UIMA-AS and Spark would have. This approach is also easily modifiable since the documents provided by the nginx server are a volume, pointing to a persistent folder inside the host file system. Similarly, the jar-provider provides the needed JAR files for

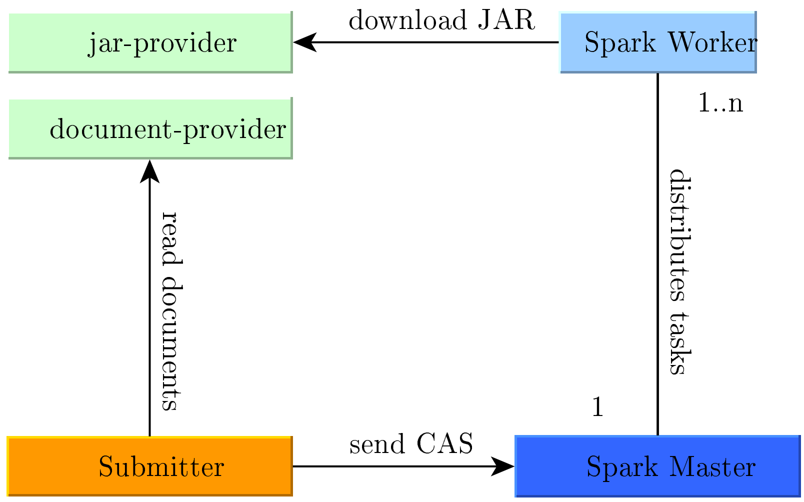

Figure 4.1: The evaluation architecture with Spark inside a Docker environment.

Spark worker nodes. As described above, a Spark cluster is a general computation cluster and not pipeline-specific. Thus, to execute code of an analysis engine, the corresponding JAR file must be provided to the whole cluster. While this can also be achieved by mounting a persistent folder inside each worker and the master node, it seemed a cleaner solution to expose the JAR file by another nginx HTTP server. This is closer to real world applications, because a JAR file will most likely not first be copied to each worker nodes file system before executing, especially since it is not trivial to predict which workers are actually processing the given tasks and at what time. In this architecture, any worker can access the JAR file at any point of its lifetime.

The submitter container is also an instance of the Spark image described above, but only issues the initial command to the cluster. Spark works in one of two modes. First, the standalone mode would let the submitter container also download the JAR file and execute the code until a Java command issues the cluster to work on some tasks in a distributed manner. In cluster mode, a worker node would be allocated by the clusters master to execute the Java code. When a Java command orders Spark to parallelize some work, more resources are allocated for said tasks. However, the initial worker node would be unavailable for this phase. Since the architecture is more comparable to the UIMA-AS architecture in the standalone mode, it is chosen for the evaluation.

4.1.2 UIMA-AS

The Docker architecture in the evaluation setup for UIMA-AS is very similar to the one for Spark. In Figure 4.2, the deployment composition is shown. For the UIMA-AS services, the underlying ApacheMQ broker and the submitter container, an UIMA-AS Docker image was composed. As with the Spark image described in Section 4.1.1, it is based on an OpenJDK image and will be available to the public according to Section 5.3. First, the UIMA-AS services register themselves at the given broker instance. At the

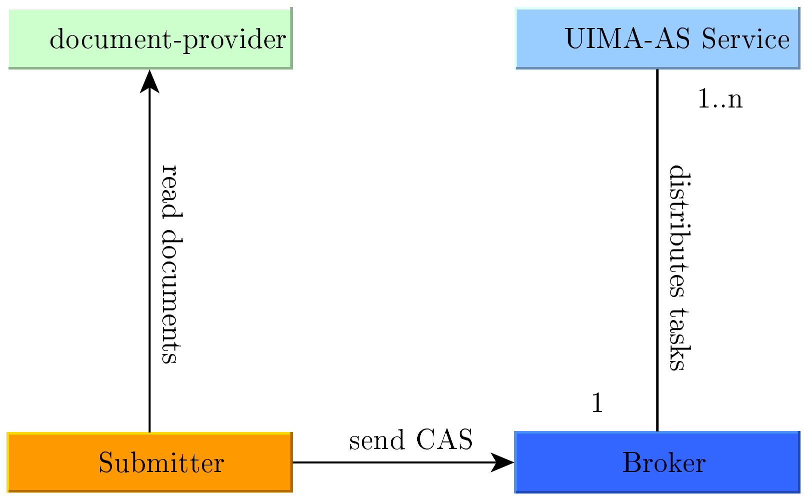

Figure 4.2: The evaluation architecture with UIMA-AS inside a Docker environment.

same time the services initialize their corresponding pipeline. This happens in contrast to Spark, where the given cluster is completely UIMA agnostic and initializes the pipelines when they are needed.

In the same way as in Section 4.1.1, the submitter reads the documents per HTTP from the document-provider, which is an instance of the identical image used in the Spark evaluation. Notice that even the logic of reading the corpus is identical, since it is wrapped inside a `CollectionReader`, which produces CAS object that can be processed further. The CAS are then sent to the broker and distributed among the services. An important distinction to make is, that in the UIMA-AS concept, the CAS object must return to the submitter. In contrast, the concept of the framework built on top of spark does allow the collection of the resulting CAS, but discourages it, since it may be a bottleneck on large corpora or large pipelines providing many analysis results.

4.2 Results

The evaluation setup given above was used to analyze a corpus of 3036 text files, taken from the larger dataset of the project Gutenberg[1] [Lah14]. This subset was cleaned by metadata, license information and transcribers notes and were therefore seen as more realistic input data than books containing such artifacts. Furthermore books were chosen as analysis items, because they contain large amount of natural language and may revolve around very different domains. This can be important for the performance of algorithms that solve specific NLP tasks, for example Named Entity Recognition.

Since the performance is most likely very dependent on the actual analysis to run, or more specifically the given pipeline, three different pipelines were used to compare UIMA-AS and Spark:

- Parsing
  1. *Language Setter*: Sets the CAS language to English.
  2. *Stanford Segmenter [MSB+14]*: Segments the text into tokens for further processing.
  3. *Stanford PoS Tagger [MSB+14]*: Finds the parts-of-speech for all identified tokens [TKMS03].
  4. *Malt Parser[NH05]*: Locates the grammatical components of sentences.
- Mixed Named Entity Recognition
  1. *Language Setter*: Sets the CAS language to English.
  2. *OpenNLP Segmenter [Apaa]*: Segments the text into tokens for further processing.
  3. *Mate PoS Tagger [Boh10]*: Finds the parts-of-speech for all identified tokens.
  4. *ClearNLP Lemmatizer [MSB+14]*: Generates the base forms for all identified tokens.
  5. *Berkeley Parser[PBTK06, PK07]*: Locates the grammatical components of sentences.
  6. *Stanford Named Entity Recognizer [MSB+14]*: Finds occurrences of the entities 'Person', 'Location', 'Organization' and 'Misc' and numerical entities.
- OpenNLP Named Entity Recognition
  1. *Language Setter*: Sets the CAS language to English.
  2. *OpenNLP Segmenter [Apaa]*: Segments the text into tokens for further processing.
  3. *Mate Lemmatizer [Boh10]*: Generates the base forms for all identified tokens.

[1] `http://www.gutenberg.org/`, last accessed on 2018-09-16.

4. *OpenNLP PoS Tagger [Apaa]*: Finds the parts-of-speech for all identified tokens.
5. *OpenNLP Named Entity Recognizer [Apaa]*: Finds occurrences of entities according to a pre-defined model.

All the given analysis engines (except for the Language Setter) are provided by the DKPro Core repository via the build tool maven.

4.2.1 Extensibility

Die extensibility ist hier zweiseitig zu betrachten, weil wir zum einen UIMA haben, was durch das Annotator-Plugin-System sehr extensible ist, was NLP-Funktionalität angeht. Das steht zumindest im Gegensatz zu v3NLP, was zwar auch Plugins zulässt, allerdings nicht die nativen UIMA-Dinger frisst (soweit ich weiß, muss ich noch bestätigen).

Zum anderen stellt sich die Frage inwiefern Spark-Konzepte weiter auf das Framework geworfen werden können. Es punktet zwar dadurch, dass es mit BigData umgehen kann, verliert allerdings durch das POJO, das der User zurückbekommt, an Spark-Funktionalität. Diese ist erweiterbar, allerdings nur wenn man den Quellcode selbst umschreibt (ie. das Projekt forked). Das ist zwar auch änderbar, ich will jetzt allerdings keien neuen features mehr zum FW hinzufügen, die nicht nur die Benchmarks invalidieren, sondern auch Fehler beinhalten können.

Interessant wäre vielleicht noch zu erwähnen, dass mein FW ein Serialization- und Compression- Interface anbietet, durch das der User diese beiden Aspekte quasi selbst einstellen kann. Beides macht einen großen Leistungsunterschied, besonders wenn man Network vs Localhost-Verkehr betrachtet. UIMA-AS bietet die Möglichkeit die Serialization selbst zu definieren, allerdings nicht die Compression. Der Serializer kann btw. natürlich auch dazu verwendet werden um Daten zu prunen. Das ist aber vom Anwendungsfall abhängig und hier nicht wirklich relevant, evtl sollte ich es allerdings trotzdem mal erwähnen.

4.2.2 Maintainability

Im Gegensatz zu UIMA-AS punktet hier natürlich auch mein FW. Ich sag nur XML-Dateien. Mein FW (ich hab dem noch gar keinen Namen gegeben) setzt auf die Spark-Infrastruktur. Damit ist es genauso Maintainable wie dieses, was auch immer das heißen mag. Ich gehe davon aus, dass services wie AWS sowas übernehmen.

4.2.3 Scalability

Ein bisschen seltsam, das als Metrik hinzuzunehmen, aber trotzdem sollte man sich Gedanken darum machen, was passiert wenn wir ZU bigData haben. Bei UIMA-AS würde als erstes vermutlich der Broker streiken, weil es keinen Broker-Broker gibt. Bei Spark kann es mehrere Master in einem Netzwerk geben. Wie das geregelt wird, muss ich noch herausfinden.

Chapter 5

Summary

The following chapter concludes the thesis by summarizing its results and the frameworks limitations. Furthermore, an outlook on the frameworks source code availability and possible future changes will be given.

5.1 Limitations

Eventuell sogar schon in die Implementierung?

5.2 Conclusion

Das hängt ein wenig von den Ergebnissen ab. Ich hoffe auf sowas wie "Ich bin der tollste, UIMA-AS ist Dreck". Das spiegeln die Daten zwar nicht ganz wieder, aber hey :D

5.3 Availability

From November 2018 on, the frameworks code will be publicized on GitHub[1]. Since the framework is wrapped inside a maven project, it will also be uploaded to the central maven repository. Furthermore, another git repository that contains a working Spark dockerfile and the evaluation setup architecture will be published[2]. Next, the maven project that defines the benchmarking Java code will be available[3]. A hybrid project that consists of the deployment of UIMA-AS used in the evaluation and defines a dockerfile containing a working UIMA-AS installation will also be published on GitHub[4]. At last, this Thesis and the corresponding LaTeX code will be available on GitHub as well [5].

All repositories will be published under the MIT license and are therefore free to use.

[1] On `https://github.com/s-gehring/master-thesis-program`
[2] On `https://github.com/s-gehring/master-thesis-spark`
[3] On `https://github.com/s-gehring/master-thesis-benchmark`
[4] On `https://github.com/s-gehring/master-thesis-uimaas`
[5] On `https://github.com/s-gehring/master-thesis`

5.4 Outlook

First, the framework and all evaluation related code and resources will be made public according to Section 5.3. Other than that further improvements to the presented framework can still be made. The wrapping class `AnalysisResult` only provides a subset of its underlying `JavaRDD` functionality. The other functions were not needed at the time of writing and have therefore been neglected. However, an unwrapping of said RDD may be desired, making further processing of the underlying CAS possible. In such a way, one could benefit substantially more from Sparks optimization features.

Furthermore, more compression algorithms may be implemented in the future, making compression for different serializations feasible. New serialization techniques, like delta CAS as used in [ESI+12] could also help improve the frameworks performance.

Glossary

Analysis Engine

An Analysis Engine is a component of an UIMA NLP pipeline. As such it analyses given documents and enriches it with inferred information. An Analysis Engine may contain other Analysis Engines [Apaf]. 6

Apache Spark

Apache Spark is an open-source cluster-computing framework. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. 1

Gestohlener Text, muss noch paraphrasiert werden.

Application Programming Interface

3

Collection Processing Engine

Collection Processing Engines (CPE) are the first generation of UIMA native scaling solutions. A CPE contains a collection reader, which knows how to read the underlying collection, and CAS Consumers for the final analysis result extraction [Apaf]. 3

Collection Processing Manager

10

Common Analysis System

The Common Analysis System is a type of object in the UIMA framework. It contains the subject of analysis, the analysis result and a corresponding type system [Apaf]. 6

Darmstadt Knowledge Processing Software Repository

A collection of UIMA components for natural language processing. This includes analysis engines, language models and custom type systems [DKP, EdCG14]. 3

Docker

Docker is a virtualization solution based on containers. By using containers instead of fully fledged virtual machines Docker tries to reduce the system overhead per running application [doc15]. 1, 28, 29, 30, 31

Extensible Markup Language

2

Factories, Injection, and Testing library for UIMA

5

General Architecture for Text Engineering

1

Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. 14

Gestohlener Text, muss noch paraphrasiert werden.

Java Virtual Machine

2

Lines of Code

3

Natural Language Processing

Natural-language processing (NLP) is the discipline of collecting and analysing natural language. This includes for example speech recognition, natural language understanding and generation [Lid01]. 1

Organization for the Advancement of Structured Information Standards

5

Question Answering

Being a subfield of NLP, Question Answering (QA) is about extracting and understanding questions from natural language and answering them accordingly [JM14]. 1

Resilient Distributed Dataset

14

Stuttgart-Tübingen-TagSet

8

Subject of Analysis

The Subject of Analysis is the document that gets analyzed by a given UIMA application. It is contained in its corresponding CAS [Apaf]. 6

UIMA Asynchronous Scaleout

UIMA-AS is the second generation of UIMA native scaling solutions. It is based on a shared queue based service architecture [Apac] 1

UIMA Collection Processing Architecture

5

Unified Modeling Language

20

Uniform Resource Locator

10

Unstructured Information Management Architecture

UIMA is a general purpose framework to extract information from unstructured data [Apab, FLVN09]. Although any data format is supported, natural language texts are the most common one. 1

VA Informatics and Computing Infrastructure

15

XML Metadata Interchange

9

Bibliography

[Apaa] The Apache Software Foundation. Apache opennlp. `http://opennlp.apache.org/faq.html`. Accessed: 2018-08-29.

[Apab] The Apache Software Foundation. Apache uima – apache uima. `https://uima.apache.org/`. Accessed: 2018-02-26.

[Apac] The Apache Software Foundation. Getting started: Apache uima asynchronous scaleout. `https://uima.apache.org/doc-uimaas-what.html`. Accessed: 2018-02-26.

[Apad] The Apache Software Foundation. Uima asynchronous scaleout. `https://uima.apache.org/d/uima-as-2.8.1/uima_async_scaleout.html#ugr.async.ov`. Accessed: 2018-09-09.

[Apae] The Apache Software Foundation. Uima overview and sdk setup. `https://uima.apache.org/d/uimaj-2.9.0/overview_and_setup.html`. Accessed: 2018-09-08.

[Apaf] The Apache Software Foundation. Uima tutorial and developers' guides. `https://uima.apache.org/d/uimaj-2.4.0/tutorials_and_users_guides.html`. Accessed: 2018-02-26.

[BL04] Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.

[Boh10] Bernd Bohnet. Very high accuracy and fast dependency parsing is not a contradiction. In *Proceedings of the 23rd international conference on computational linguistics*, pages 89–97. Association for Computational Linguistics, 2010.

[CMBT02] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. Gate: an architecture for development of robust hlt applications. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 168–175. Association for Computational Linguistics, 2002.

[dC] Richard Eckart de Castilho. uimafit github repository. `https://github.com/apache/uima-uimafit`. Accessed: 2018-09-03.

[DCR+15] Guy Divita, M Carter, A Redd, Q Zeng, K Gupta, B Trautner, M Samore, and A Gundlapalli. Scaling-up nlp pipelines to process large corpora of clinical notes. *Methods of information in medicine*, 54(06):548–552, 2015.

[DCT+16] Guy Divita, Marjorie E Carter, Le-Thuy Tran, Doug Redd, Qing T Zeng, Scott Duvall, Matthew H Samore, and Adi V Gundlapalli. v3nlp framework: Tools to build applications for extracting concepts from clinical text. *eGEMs*, 4(3), 2016.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DKP] The DKPro Core Team. Dkpro core™ user guide. `https://zoidberg.ukp.informatik.tu-darmstadt.de/jenkins/job/DKProCoreDocumentation(GitHub)/de.tudarmstadt.ukp.dkpro.core$de.tudarmstadt.ukp.dkpro.core.doc-asl/doclinks/6/user-guide.html`. Accessed: 2018-02-26.

[doc15] What is docker? `https://www.docker.com/what-docker`, 2015. Accessed: 2018-02-26.

[DT11] G Divita and QZ Trietler. Finding medically unexplained symptoms within va clinical documents using v3nlp. *Using administrative databases to identify cases of chronic kidney disease: a systematic review*, page 9, 2011.

[EdCG14] Richard Eckart de Castilho and Iryna Gurevych. A broad-coverage collection of portable nlp components for building shareable analysis pipelines. In *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*, pages 1–11, Dublin, Ireland, August 2014. Association for Computational Linguistics and Dublin City University.

[ESI+12] Edward A Epstein, Marshall I Schor, BS Iyer, Adam Lally, Eric W Brown, and Jaroslaw Cwiklik. Making watson fast. *IBM Journal of Research and Development*, 56(3.4):15–1, 2012.

[Fer12] David A Ferrucci. Introduction to “this is watson”. *IBM Journal of Research and Development*, 56(3.4):1–1, 2012.

[FL04] David Ferrucci and Adam Lally. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348, 2004.

[FLVN09] David Ferrucci, Adam Lally, Karin Verspoor, and Eric Nyberg. Unstructured information management architecture (UIMA) version 1.0. OASIS Standard, mar 2009.

[GA15] Satish Gopalani and Rohan Arora. Comparing apache spark and map reduce with performance analysis using k-means. *International journal of computer applications*, 113(1), 2015.

[GR12] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16, 2012.

[GS05] Marc Georges Girardot and Neelakantan Sundaresan. System and method for schema-driven compression of extensible mark-up language (xml) documents, April 19 2005. US Patent 6,883,137.

[JM14] Dan Jurafsky and James H Martin. *Speech and language processing*, volume 3. Pearson London:, 2014.

[KSDG12] Valmeek Kudesia, Judith Strymish, Leonard D'Avolio, and Kalpana Gupta. Natural language processing to identify foley catheter–days. *Infection control and hospital epidemiology*, 33(12):1270–1272, 2012.

[Lah14] Shibamouli Lahiri. Complexity of Word Collocation Networks: A Preliminary Structural Analysis. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 96–105, Gothenburg, Sweden, April 2014. Association for Computational Linguistics.

[Lid01] Elizabeth D Liddy. Natural language processing. 2001.

[MPC03] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. Xpress: A queriable compression for xml data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 122–133. ACM, 2003.

[MSB+14] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[NH05] Joakim Nivre and Johan Hall. Maltparser: A language-independent system for data-driven dependency parsing. In *In Proc. of the Fourth Workshop on Treebanks and Linguistic Theories*, pages 13–95, 2005.

[OB09] Philip Ogren and Steven Bethard. Building test suites for UIMA components. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)*, pages 1–4, Boulder, Colorado, June 2009. Association for Computational Linguistics.

[oVA] Departments of Veterans Affairs. Leo overview. http://department-of-veterans-affairs.github.io/Leo/userguide.html. Accessed: 2018-09-14.

[PBTK06] Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440. Association for Computational Linguistics, 2006.

[PK07] Slav Petrov and Dan Klein. Improved inference for unlexicalized parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 404–411, 2007.

[PT18] Irina Pak and Phoey Lee Teh. Text segmentation techniques: A critical review. In *Innovative Computing, Optimization and Its Applications*, pages 167–181. Springer, 2018.

[RBJB+10] Cartic Ramakrishnan, William A Baumgartner Jr, Judith A Blake, Gully APC Burns, K Bretonnel Cohen, Harold Drabkin, Janan Eppig, Eduard Hovy, Chun-Nan Hsu, Lawrence E Hunter, et al. Building the scientific knowledge mine (sciknowmine): a community-driven framework for text mining tools in direct service to biocuration. *Language Resources and Evaluation*, page 33, 2010.

[Sak09] Sherif Sakr. Xml compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, 75(5):303–322, 2009.

[Staa] Stackoverflow. How many characters can a java string have? https://stackoverflow.com/questions/1179983/how-many-characters-can-a-java-string-have. Accessed: 2018-09-03.

[Stab] Statista. Revenues from the natural language processing (nlp) market in north america, from 2015 to 2024 (in million u.s. dollars). https://www.statista.com/statistics/607909/north-america-natural-language-processing-market-revenues/. Accessed: 2018-09-05.

[Stac] Statista. Revenues from the natural language processing (nlp) market in the asia-pacific region, from 2015 to 2024 (in million u.s. dollars). https://www.statista.com/statistics/607928/asia-pacific-natural-language-processing-market-revenues/. Accessed: 2018-09-05.

[Stad] Statista. Revenues from the natural language processing (nlp) market in western europe, from 2015 to 2024 (in million u.s. dollars). `https://www.statista.com/statistics/607915/western-europe-natural-language-processing-market-revenues/`. Accessed: 2018-09-05.

[TKMS03] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003.

[TRCB13] Valentin Tablan, Ian Roberts, Hamish Cunningham, and Kalina Bontcheva. Gatecloud. net: a platform for large-scale, open-source text processing on the cloud. *Phil. Trans. R. Soc. A*, 371(1983):20120071, 2013.

[Wil17] Micah Williams. Apache spark vs. mapreduce. `https://dzone.com/articles/apache-spark-introduction-and-its-comparison-to-ma`, aug 2017.

[WK15] Kewen Wang and Mohammad Maifi Hasan Khan. Performance prediction for apache spark platform. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 166–173. IEEE, 2015.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

Eidesstattliche Erklärung

Hiermit versichere ich, Simon Gehring, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken und Quellen, einschließlich Quellen aus dem Internet, entnommen sind, habe ich in jedem Fall unter Angabe der Quelle deutlich als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen, Karten und Abbildungen.

Unterschrift: ______________________________

Simon Gehring, Student
Universität Bonn