System Design, Final

Deliverable D2.2

Version FINAL

Authors: Zuhaitz Beloki[1], German Rigau[1], Aitor Soroa[1], Antske Fokkens[2], Kees Verstoep[2], Piek Vossen[2], Marco Rospocher[3], Francesco Corcoglioniti[3], Roldano Cattoni[3], Stefan Verhoeven[4], Mathijs Kattenberg[5]

Affiliation: (1) EHU, (2) VUA, (3) FBK, (4) Netherlands eScience Centre, (5) SURFsara

Building structured event indexes of large volumes of financial and economic data for decision making

ICT 316404

Grant Agreement No.	316404
Project Acronym	NEWSREADER
Project Full Title	Building structured event indexes of large volumes of financial and economic data for decision making.
Funding Scheme	FP7-ICT-2011-8
Project Website	http://www.newsreader-project.eu/
Project Coordinator	Prof. dr. Piek T.J.M. Vossen VU University Amsterdam Tel. + 31 (0) 20 5986466 Fax. + 31 (0) 20 5986500 Email: piek.vossen@vu.nl
Document Number	Deliverable D2.2
Status & Version	FINAL
Contractual Date of Delivery	January 2015
Actual Date of Delivery	February 3, 2015
Type	Report
Security (distribution level)	Public
Number of Pages	102
WP Contributing to the Deliverable	WP2
WP Responsible	EHU
EC Project Officer	Susan Fraser

Authors: Zuhaitz Beloki[1], German Rigau[1], Aitor Soroa[1], Antske Fokkens[2], Kees Verstoep[2], Piek Vossen[2], Marco Rospocher[3], Francesco Corcoglioniti[3], Roldano Cattoni[3], Stefan Verhoeven[4], Mathijs Kattenberg[5]

Keywords: system design, big data, scaling NLP

Abstract: This deliverable describes the second version of the System Design framework developed in NewsReader to process large and continuous streams of English, Dutch, Spanish and Italian news articles. The deliverable describes two alternative architectures to automatically process massive streams of daily news and reconstruct longer term story lines of events. Both architectures are complementary and are meant to be used on two different settings (batch and streamiing processing, respectively). The deliverable also describes a number of experiments were carried out to investigate where potential bottlenecks are found while passing text through our NLP pipeline. We also present the integration of the KnowledgeStore into a HPC cloud infrastructure as a first important step to enhance the scalability of the repository.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	December 2014	Structure of the deliverable set	Aitor Soroa	All
0.2	December 2014	NAF2SEM architecture	Piek Vossen	naf2sem
0.3	January 2015	NAF2SEM processing	Piek Vossen	naf2sem
0.4	January 2015	Added Haddop and provenance sections	Antske Fokkens	2,5,appendix1
0.5	January 2015	First final version	Aitor Soroa	all
0.6	3 February 2015	Changes after internal revision by SynerScope	Aitor Soroa	all
0.7	3 February 2015	Check by coordinator	VUA	-

Executive Summary

This deliverable describes the second version of the **System Design** framework developed in NewsReader to process large and continuous streams of English, Dutch, Spanish and Italian news articles. Deliverable D2.1 "System Design, draft" described the basic architecture of the NewsReader system, including the representation schemas and the first version of the distributed pipeline. This deliverable is a continuation of the work described in D2.1 and it describes the work performed under the second year of the project.

Within the project we have developed annotation schemes for representing linguistic information at many levels, from basic annotations such as tokenization and POS tagging to complex models for representing events and participants. In this deliverable we will describe the latest addition to the Grounded Annotation Framework (GAF), specifically those related to modeling the representation of provenance information. Provenance is important to know what the source of information is, so that users can compare perspectives from different sources, or verify whether the represented information is correct. The result of the processing pipeline is a set of documents annotated using the NLP Annotation Format (NAF), also developed within the project. These NAF documents are then converted to GAF representations, removing duplicates and harmonizing the information that come from different documents. In the deliverable we describe how this process is actually performed.

Processing massive quantities of data requires designing solutions that are able to run distributed programs across a large cluster of machines. We have developed two alternative architectures for NLP distributed processing, for a batch and streaming computing scenarios, respectively. In the deliverable we describe each of the solutions, detailing the main components of each one. This batch architecture was used to process 1.3 Million articles for the January 2015 Hackathons and user evaluation.

The deliverable also present a set of scripts developed within the project that quickly create a basic cluster of nodes for distributed processing. All the software, including the NLP modules, as well as the basic components of the clusters is publicly available and is distributed under open licenses.

Finally, we present the integration of the KnowledgeStore into a HPC cloud infrastructure as a first important step to enhance the scalability of the repository.

Contents

1 Introduction and scope **11**

2 Efficiency Experiments **13**
2.1 Performance per module 13
2.2 A more detailed analysis of the performance per module 15
2.3 Observations 19

3 Architectures for parallel NLP processing **21**
3.1 Linguistic modules for Event Detection 22
3.2 Frameworks for Big Data analysis 23

4 Streaming architecture using Storm **25**
4.1 Newsreader processing cluster 25
4.1.1 NLP processing topology 28
4.1.2 MongoDB integration 30
4.1.3 VM from scratch 33
4.2 Experiments 34
4.3 Batch and Streaming Processing 34
4.4 Non linear topologies 37

5 Batch architecture using Hadoop **39**
5.1 Hadoop 39
5.2 Cascading 39
5.3 Scaling up news processing 40
5.4 Processing the car data 41
5.5 Future work and possible improvements 42

6 Modeling Provenance and Attribution **45**
6.1 Provenance and Attribution 45
6.2 Attribution annotations 46
6.3 Proposed attribution models 47
6.3.1 Requirements 48
6.4 Modeling attribution 50

7 Conversion of NAF to SEM **55**
7.1 Architecture for processing NAF to SEM 55
7.2 Performance of the NAF to SEM processing on the automotive data set . . 58

8 The KnowledgeStore **61**
8.1 Setup and Configuration 61
8.2 Experiments 61

9 Conclusion and future work **62**

A Processing step per phase for individual modules **65**

B Alternative proposals for modeling attribution **75**
B.1 Proposal 1: Attribution applies to the statements 77
B.2 Proposal 2: Attribution applies to the relation between source and statement 80
B.3 Proposal 3: Attribution is assigned by reifying relations 81
B.4 Concluding remarks 83

C NewsReader architecture for distributed NLP processing **85**
C.1 VM from scratch 86
C.1.1 Create a basic cluster 86
C.1.2 Making copies of worker VMs 88
C.1.3 Virtual machines XML definitions 90
C.1.4 Booting and stopping the cluster 90
C.1.5 Deploying worker VMs into different host machines 91
C.1.6 Knowing which machines are connected to the boss VM 91
C.2 Running the topology 91
C.2.1 Topology XML specification 92
C.3 Troubleshooting and tools 93
C.3.1 Synchronizing the modules 93
C.3.2 Document queue 94
C.3.3 Starting and stopping the services 98

List of Tables

1 NLP modules of the NewsReader event detection framework for several languages. 22
2 LP modules of the NewsReader pipeline, with the pre- and post-requisites. 29
3 Processing time for a batch of 100 documents on a cluster comprising 6 worker nodes. 35
4 Statistic for batch processing. Throughput measures the ratio between the number of documents and the elapsed time. 36
5 Statistic for streaming processing for linear and non linear topologies. . . . 37

1 Introduction and scope

This deliverable describes the second version of the **System Design** framework developed in NewsReader to process large and continuous streams of English, Dutch, Spanish and Italian news articles. The goal of the NewsReader project is to automatically process massive streams of daily news in 4 different languages to reconstruct longer term story lines of events. For this purpose, the project will extract events mentioned in the the news articles, the place and date of their occurrence and who is involved. Based on the extracted knowledge, NewsReader will reconstruct a coherent story in which new events are related to past events. The research activities conducted within the project strongly rely on the automatic detection of events, which are considered as the core information unit underlying news and therefore any decision making process that depends on news articles.

The NewsReader architecture for NLP processing integrates a large number of tools which relate to various research areas such as Information Retrieval and Extraction, Natural Language Processing, Text Mining or Natural Language Understanding. Within the project we distinguish two main types of linguistic processing: intra-document linguistic processing and cross document linguistic processing. Intra-document linguistic processing involves the linguistic processing of a single text document, and comprises steps such as tokenization, lemmatization, part-of-speech tagging, parsing, word sense disambiguation, Named Entity and Semantic Role Recognition for all the languages in NewsReader. Besides, Named entities are linked as much as possible to external sources such as Wikipedia and DBpedia. Cross document processing involves steps such as document clustering or linking textual expressions, i.e. mentions, from several documents which refer to the same entity or event, creating together chains of coreferring mentions.

Within the project we have developed annotation schemes for representing linguistic information at many levels, from basic annotations such as tokenization and POS tagging to complex models for representing events and participants. Intra-document processing modules represent the annotations using NAF, the NLP Annotation Format developed within the project. NAF serves as a shared model that allows these NLP tools to cooperate. Events and their relations are represented using GAF [Fokkens *et al.*, 2013], an extremely compact annotation framework that allows the elimination of duplication and repetition. GAF is used to detect event identity, complete incomplete descriptions, and chain and relate events into plots (temporal, local and causal chains). GAF formally distinguishes between mentions in NAF and instances in the Simple Event Model (SEM, [van Hage *et al.*, 2011]). This way, it allows the abstraction over different mentions of the same events, participants, places and times results in a single representation of an instance with links to the places where it is mentioned in the news.

As said before, both NAF and GAF are described in deliverable D2.1: "System Design - draft". In this deliverable we will describe the latest addition to GAF, specifically those related to modeling the representation of provenance information. Provenance is important to know what the source of information is, so that users can compare perspectives from different sources, or verify whether the represented information is correct. We also describe the process of converting GAF representation from the NAF documents as produced by

the basic intra-document NLP processing modules.

Processing massive quantities of data requires designing solutions that are able to run distributed programs across a large cluster of machines. Besides, issues such as parallelization, distribution of data, synchronization between nodes, load balancing and fault tolerance are of paramount importance. The NewsReader project will follow a streaming computing paradigm, where documents will arrive at any moment and have to be processed continuously. The project foresees the processing of huge amounts of textual data at any rate. In this deliverable we describe the scaling solutions that have been implemented during the second year of the project. We have implemented two alternative architectures for distributed event detection. The first architecture is an enhancement version of the architecture implemented in the first year of the project and is meant to be used on a streaming computing scenario. The second architecture is an heavily optimized version of the processing pipeline, and it is focused on a batch processing setting.

The result of the day-by-day processing of large volumes of news and information is stored in the KnowledgeStore central data repository, which plays a crucial role in the NewsReader architecture. We present the integration of the KnowledgeStore into a HPC cloud infrastructure as a first important step to enhance the scalability of the repository.

This deliverable is structured as follows. We start in Section 2 describing some experiments carried out to analyze processing times spent by each module and locate potential bottlenecks. Section 3 describes the differences among the processing architecture developed in the first year with the solutions presented in this deliverable, and provides a brief description of the main frameworks used to linguistically analyze big quantities of data. Sections 4 and 5 explain the streaming and batch approaches implemented during this year, and describe several experiments carried out in each of them. Section 6 describes the modeling of provenance within GAF, whereas Section 7 explains the conversion of the information represented by means of NAF annotations to GAF. Section 8 describes the activity of porting the KnowledgeStore on the SURFsara HPC-Cloud infrastructure, a necessary step to integrate the KnowledgeStore repository to the processing architecture. Finally, section 9 draws some conclusions.

2 Efficiency Experiments

In the first year of NewsReader, we developed a fully functioning architecture to process English data. In the second year, our efforts mainly focused on improving this architecture and in particular in boosting its performance. A number of experiments were carried out to investigate where potential bottlenecks are found while passing text through our NLP pipeline. This section presents the outcome of these experiments. In Section 2.1, we describe a set of experiments that tests performance of individual modules on documents of different length. Section 2.2 provides a more detailed analysis of individual modules by diving into the internal structure of the modules. It distinguishes between the time needed to convert data to and from NAF, the time needed to start up the module or communication to external resources and the time to carry out the actual NLP task. Section 2.3 presents the main observations from the experiments described in this section.

2.1 Performance per module

In order to identify potential bottlenecks in scaling, it is useful to know how different modules respond to larger amount of data. In the first non-parallelized stream setup, documents are passed through the pipeline individually. Several experiments were carried out that test the performance of each NLP module in relation to document size. Figure 1 represents the processing time in seconds per document. The experiment was run on a single core of 8 GB memory.

We expect at least a linear increase in processing time when the amount of data increases, because all modules in this experiment perform their analysis sequentially. The analyses on these relatively short articles are mainly flat. This indicates that most time is not spent on the analysis itself, but rather on initiating the module (which takes an equal amount of time for each document that is analyzed). Figure 2 presents the results of a similar experiment using larger documents on a 2 core of 10 GB machine. Here, we do observe a clear linear increase in processing time as documents become larger. These results confirm our expectation and reveal that the modules are generally well-behaved and processing time can be estimated relatively reliably based on average document length in a set.

Figures 3 and 4 represent memory usage on a single core 8GB and dual core 10 GB machine respectively. Many modules operate on a term or sentence level and therefore memory size can remain relatively stable as the document length increases. For several modules, memory use does not increase at all when the document becomes longer. The language model they use takes up most of the memory and this is not influenced by the amount of data that is analyzed. Modules that do make use of information found all over the document reveal (unsurprisingly) a correlation between memory use and document length. This is particularly clear for the word sense disambiguation and event coreference modules, where there is a linear relation between document length and used memory.

The semantic role labelling module is by far the most costly module in the pipeline both as far as processing time and as far as memory use are concerned. This can mainly

be explained by the fact that this module also carries out syntactic parsing, a task that is notorious for its complex modules and (theoretically) exponential growth in ambiguity as sentence length increases.

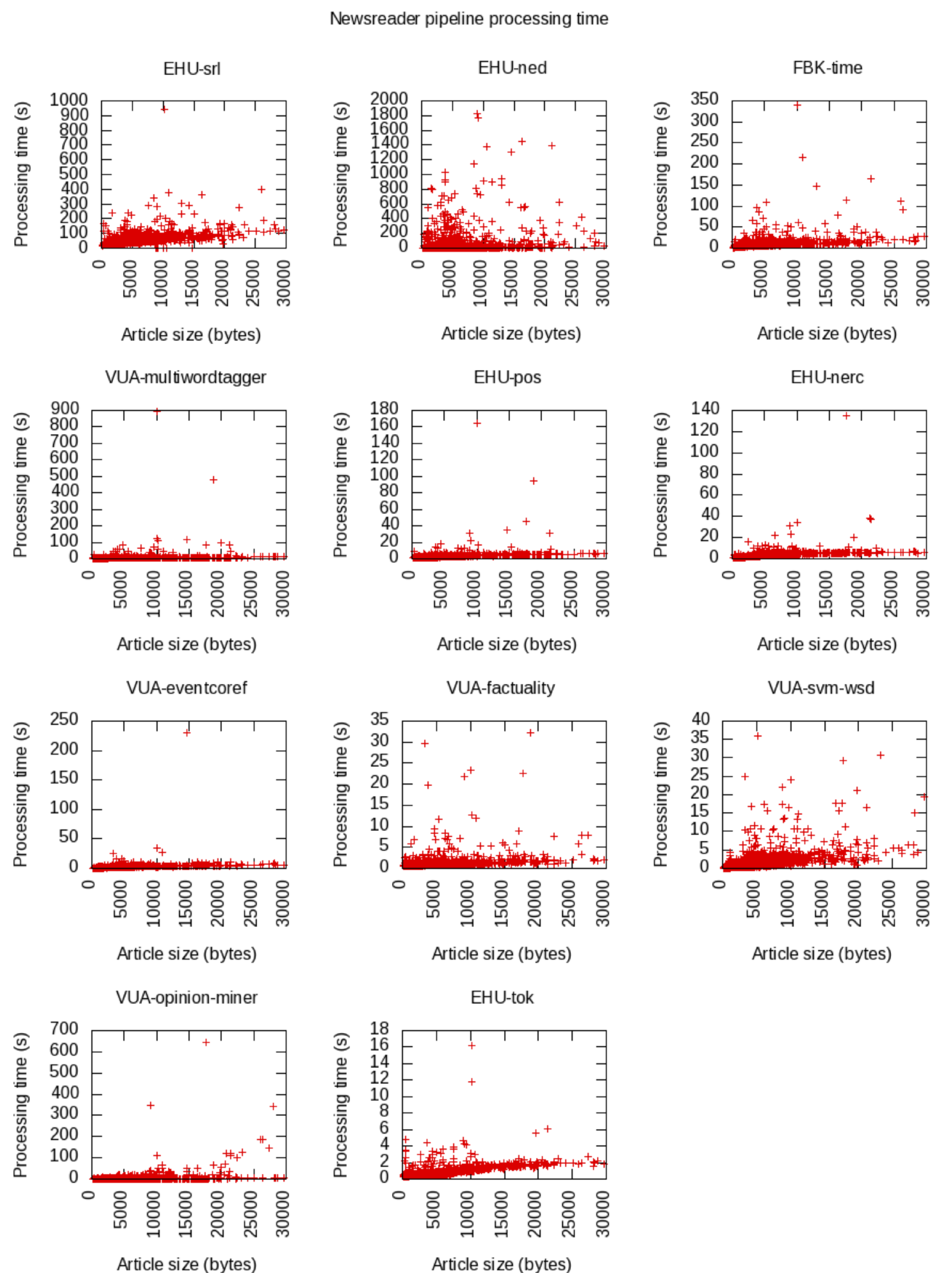


Figure 1: Processing time of articles of different size

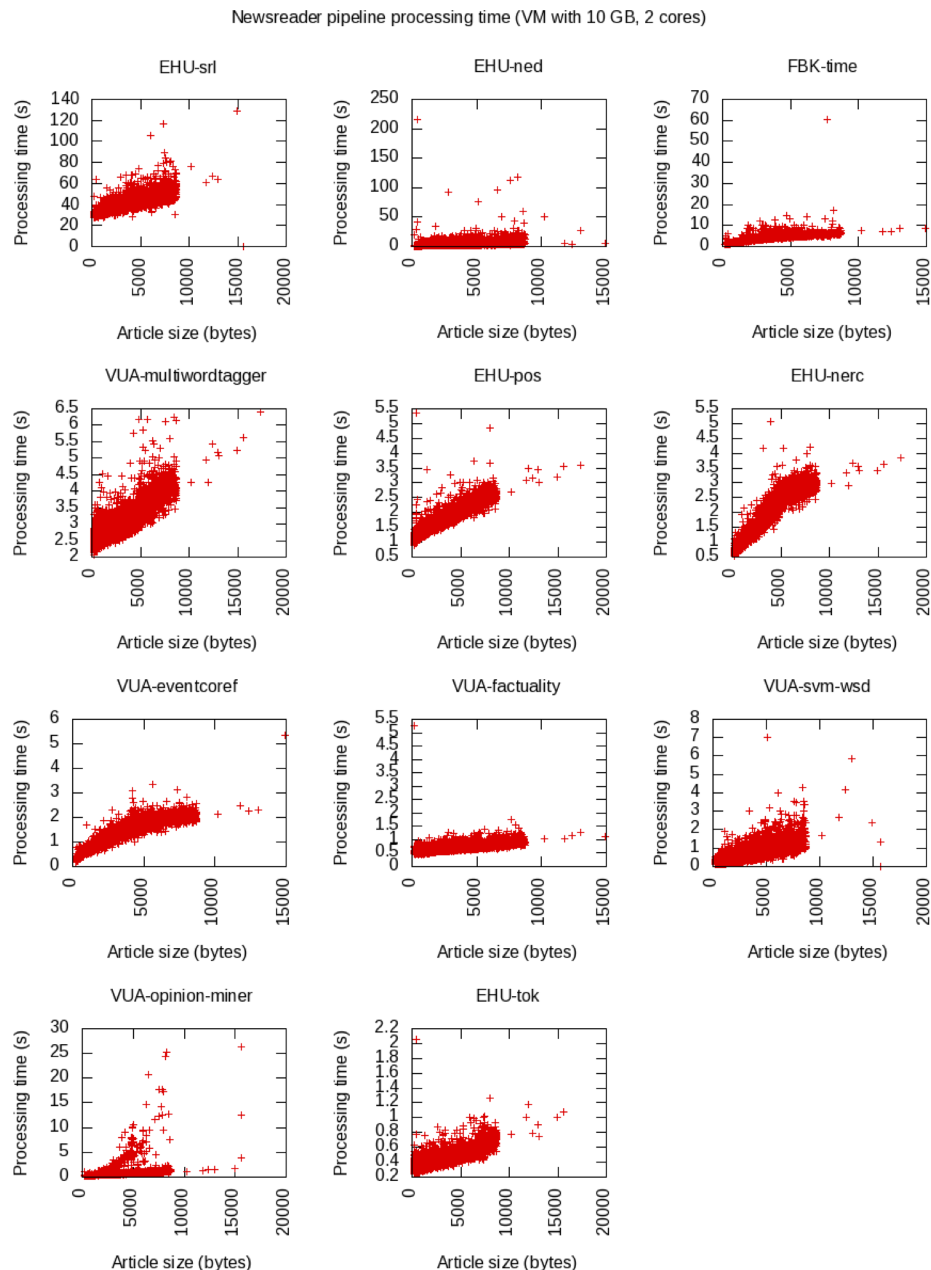


Figure 2: Processing articles of different size, 10G 2core

2.2 A more detailed analysis of the performance per module

The relation between processing time and document length indicates that several modules seem to spend a significant amount of their time on starting up the process. This can be observed in the relatively long processing time that even short documents require for some modules (e.g. semantic role labelling). We therefore carried out an internal investigation of processing time of our modules. This analysis investigates the following processing times:

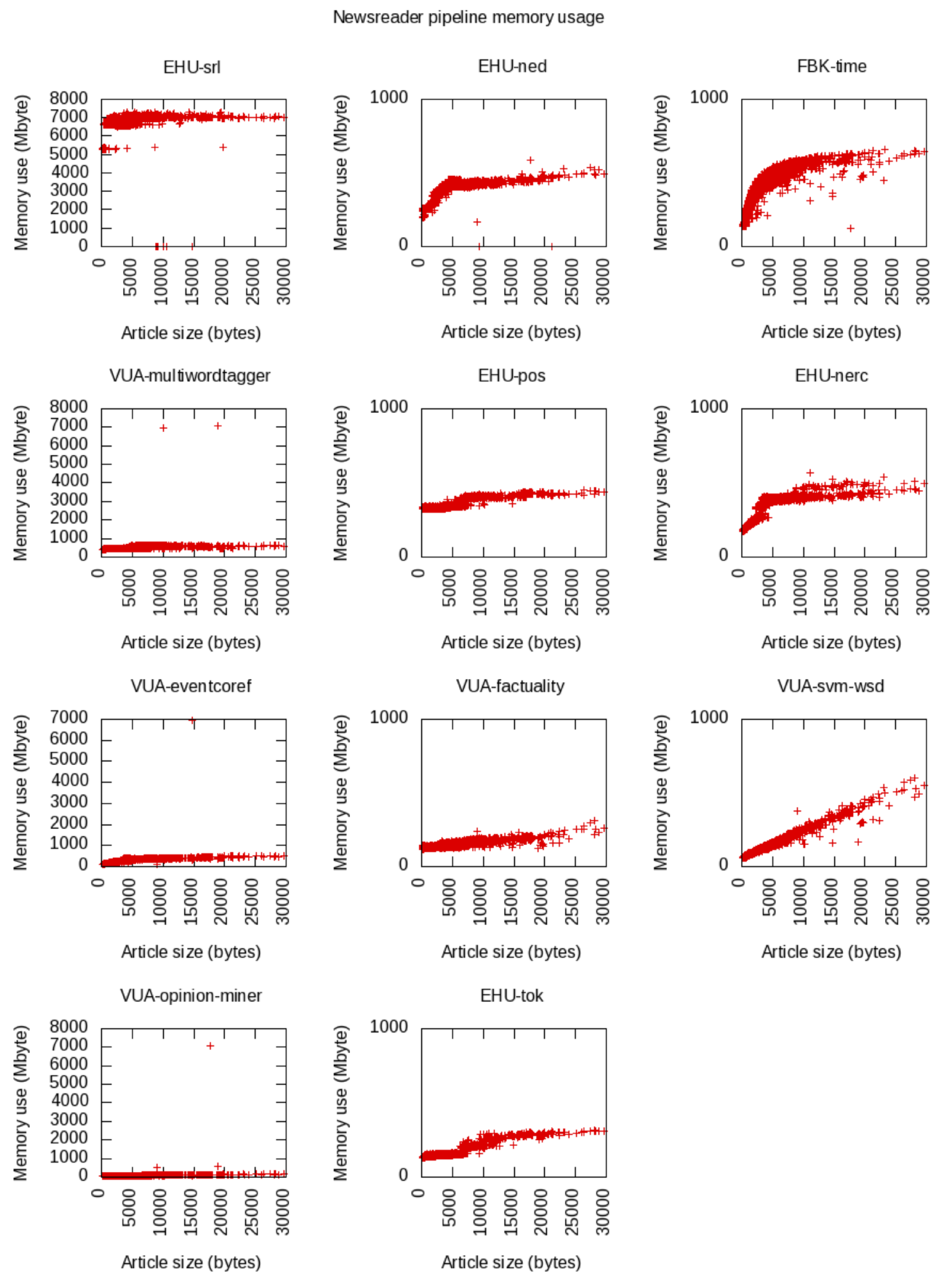


Figure 3: Memory used for processing articles of different size

- The time of the entire process from start to the end
- The startup time, from start to main executable
- Time to construct NLP task object or load language models
- Time to interpret input NAF file

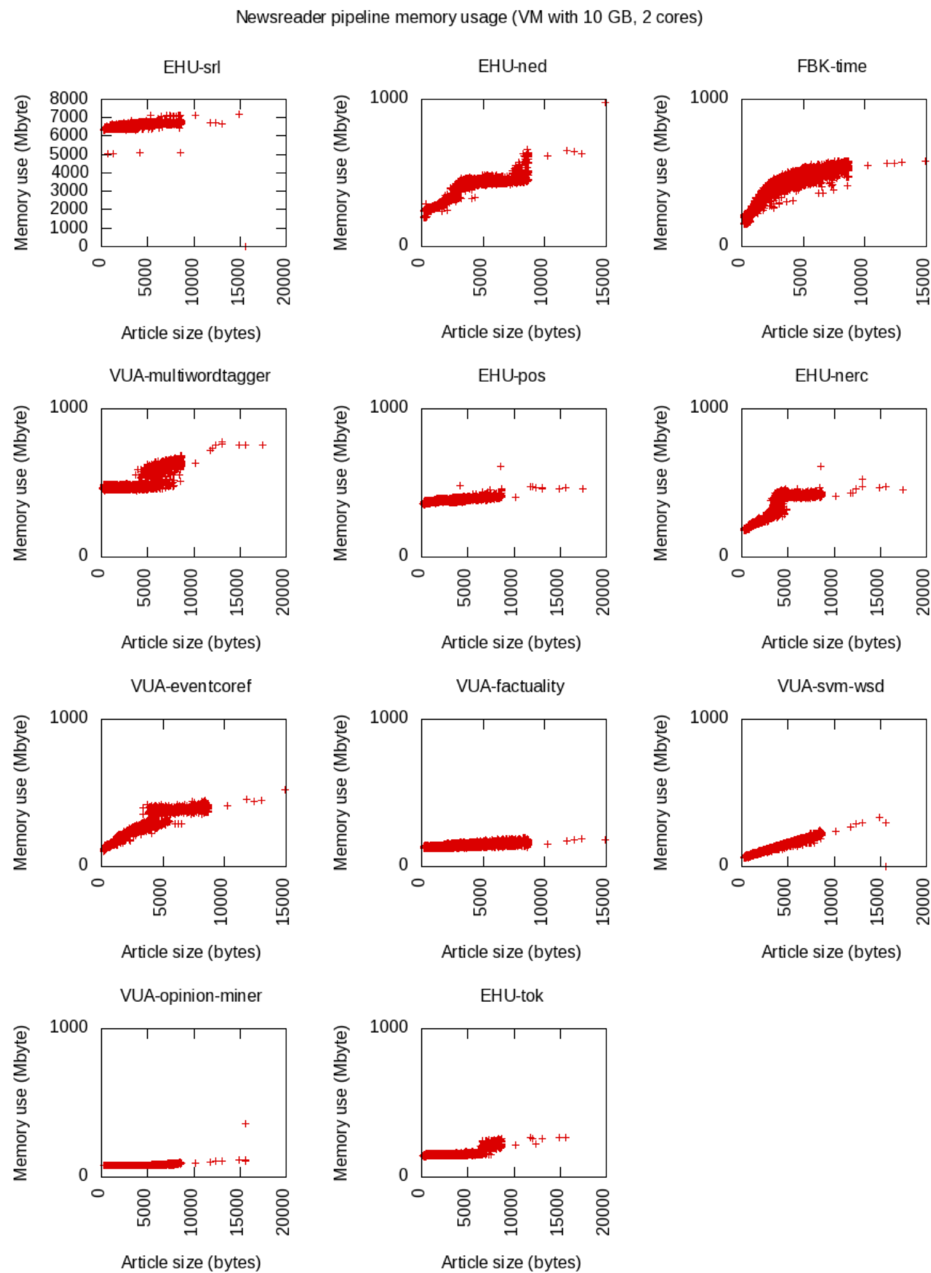


Figure 4: Memory used for processing articles of different size, 10G, 2core

- The core time carrying out the actual NLP task
- The time needed to create the output NAF from the output of the core task
- The end time from the last command to the overall end of the process

Figures 5 and 6 provide an overview of the average time of these steps for each module. The modules marked with a * had a slightly different division of subtasks. Details can be

found in Appendix A, Figures 32 and 34, respectively.

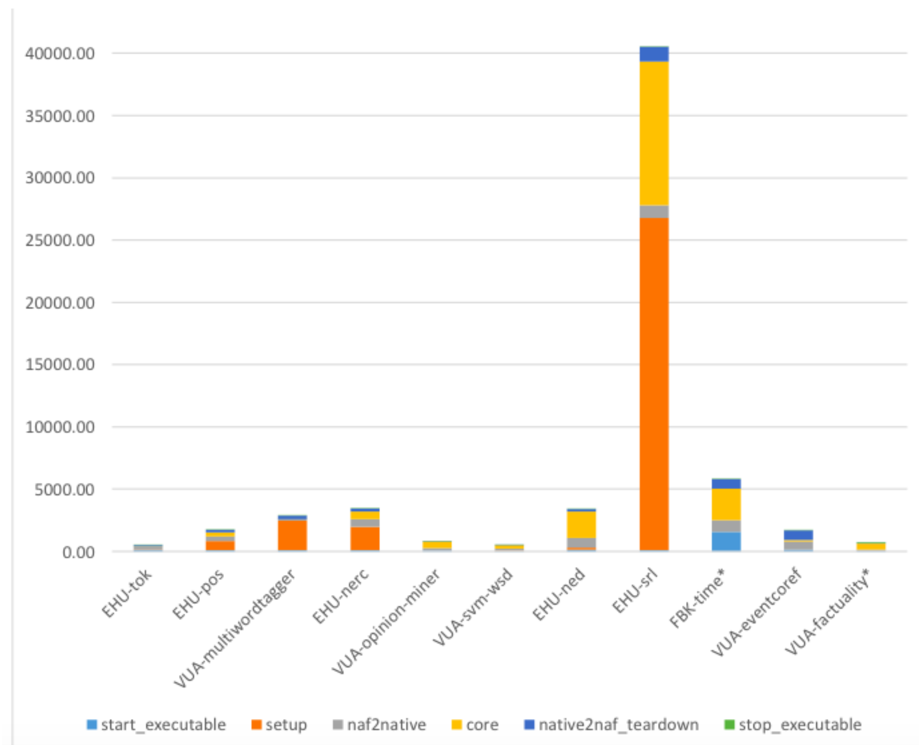


Figure 5: Processing time for individual steps inside NLP modules in ms (figure)

component	nr_docs	wall	start_exec	setup	naf2native	core	native2naf	stop_exec
EHU-tok	1122	527.58	119.98	42.50	202.86	22.87	123.34	16.04
EHU-pos	1122	1741.77	122.32	732.46	365.67	310.94	193.11	17.28
VUA-multiwordtagg	1122	2908.51	111.57	2348.00	140.55	0.21	294.29	13.89
EHU-nerc	1122	3466.30	118.69	1856.09	624.12	600.26	247.17	19.97
VUA-opinion-miner	1111	817.68	66.60	0.00	206.91	525.73	3.00	15.44
VUA-svm-wsd	1122	551.56	35.49	30.82	159.78	299.27	4.69	25.83
EHU-ned	1122	3400.82	152.59	150.39	773.17	2123.27	206.26	18.07
EHU-srl	119	38646.20	118.57	26678.46	1021.03	11528.26	1171.59	23.70
FBK-time*	119	5835.26	1591.92	0.00	920.78	2522.79	798.28	1.50
VUA-eventcoref	119	1691.87	126.39	0.00	660.34	132.51	757.82	14.82
VUA-factuality*	119	738.62	5.39	0.00	135.65	507.78	7.33	82.48

Figure 6: Processing time for individual steps inside NLP modules in ms (table)

The results are largely in line with what we have observed in the experiments illustrating overall processing time of modules depending on document length. Components such as the semantic role labelling module, the named entity recognizer, POS-tagger and multiwordtagger reveal relatively long setup times where language models are loaded. In fact, the time required to initiate the module is on average longer than the actual processing time. Time required to convert data from and to NAF also has a visible impact, though

this is limited to around 1 second per conversion in the most extreme cases. Nevertheless, this time adds up and it may be worthwhile to look into improving efficiency at this end.

2.3 Observations

The analyses presented above reveal that the NLP modules are relatively well behaved, in the sense that they very little outliers in processing time or memory use are observed and their performance can be estimated quite reliably based on document length. The semantic role labelling module requires significantly more memory and processing time than any other module. Its average processing time is about one and half time as long as the time all other components need together. We can thus gain in overall processing time if we parallelize this step in our pipeline. About two thirds of the time of the semantic role labelling module is needed to load the language model. This indicates that a significant speed up can be achieved if we load the language model once for processing a batch of documents.

3 Architectures for parallel NLP processing

Extracting events from raw text is a complex task which requires state of the art NLP processing at many levels. Besides, NewsReader requires such technology to be applied on realistic volumes of text within hard time constraints. Those requirements have lead us to adopt scalable architectures for performing parallel and distribute NLP processing. The parallelization can be effectively performed at several levels, from deploying copies of the same linguistic processor (LP) among servers to the reimplementation of the core algorithms of each module using multi-threading, parallel computing. This last type of fine-grained parallelization is out of the scope of the project, as it is unreasonable to expect a reimplementation of all the NLP processors and algorithms needed to perform such a complex task as mining events. In NewsReader we rather aim to processing huge amount of textual data by defining and implementing an architecture for NLP which allows the parallel processing of documents.

In the first year of the project we implemented a pipeline where each document was sent to a single virtual machine containing all the NLP processing modules, which were executed following a pipeline fashion, i.e., one module after another. Thus, the complete analysis for each document was performed inside a single VM. The integration of the different NLP components inside each VM was accomplished using the Storm framework (c.f. Section 3.2), which connected the pipeline steps according to a pre-defined topology[1]. This setting allowed us to process more than 100.000 documents in a timely manner. However the setting proposed had some drawbacks:

- Each VM is independent from each other, and there exist no synchronization among them. Documents have to be manually split into batches, and each batch has to be sent to the VMs manually.
- The framework was only capable to process batch of documents, and it can not deal with scenarios such as a random but continuous arrival of documents.
- The framework can only run linearized pipelines. As a consequence, it is not possible to implement non linear topologies under this framework.

For the second year of the project we have implemented two different architectures for distributed event detection, described in Sections 4 and 5, respectively. The first architecture is an enhancement of the architecture implemented in the first year of the project. As such, it relies on virtual machines and the Storm framework to process documents . However, unlike the previous version this new architecture implements a real distributed pipeline, where one single document can be processed on several nodes within a cluster.

The architecture presented in Section 5 is an heavily optimized version of the processing pipeline following a standard MapReduce model and implemented in a Hadoop environment (c.f. Section 3.2). This environment focuses in the efficient utilization of computational resources and prioritization of MapReduce jobs and are the standard computation

[1] This setting is thoroughly explained in project deliverable D2.1.

Language	#	Modules
English	15	TOK, POS, NERC, Parse, Coref, Opinion, WSD, NED, SRL, time, eCoref, tempRel, causalRel, Fact
Dutch	12	TOK, POS, NERC, Coref, Opinion, WSD, PMatrix, FNet, NED, SRL, time, eCoref
Italian	9	TOK, POS, NERC, Chunk, Coref, dep, time, eCoref, Fact
Spanish	11	TOK, POS, NERC, Parse, dep, Coref, WSD, NED, SRL, time, eCoref

Table 1: NLP modules of the NewsReader event detection framework for several languages.

model used today in mainframe computing, where system availability was a critical and scarce resource.

The main difference between the architecture described in Section 4 and the approach described in Section 5 is that whereas the latter is focused to work on a batch setting, the former is meant to work on a streaming scenario. Batch processing requires all of the input data to be completely available on the input store, e.g. HDFS , before any computation is started. The framework process the input data and the output results are available only when all of the computation is done. In contrast, streaming computing expects documents to arrive to the pipeline in any moment, and aims at reducing the overall latency, that is, the expected elapsed time needed to process one single document. In principle, streaming computing allows to minimize the delay that prevent users from gaining important insights affecting their analysis. Worse, it could provide an incomplete picture and therefore it can lead to make wrong decisions.

3.1 Linguistic modules for Event Detection

This section briefly explains the Event Detection framework that is being developed in the NewsReader project. The project aims to process large and continuous streams of English, Dutch, Spanish and Italian news articles, and extract relevant information about the events mentioned in the text. It also recognizes the major participants of the occurred events, with a special focus on achieving cross-lingual semantic interoperability.

Many NLP modules have been developed within the NewsReader project to perform state of the art event detection in several languages. Many of those modules are based on third party tools, some have been developed from scratch for the project. The whole description of the modules that constitute the NewsReader framework for event detection is described in project Deliverable D4.4.4, “Event Detection, version 2”.

NewsReader uses an open and modular architecture for NLP as a starting point. Text-processing requires basic and generic NLP steps such as tokenization, lemmatization, part-of-speech tagging, parsing, word sense disambiguation, named-entity and semantic role recognition, etc. for all the languages within the project. Semantic interpretation involves the detection of event mentions and those named entities that play a role in these events, including time and location relations. This implies covering all expressions and meanings

that can refer to events, their participating named entities, place and time relations. It also means resolving coreference relations for these named entities and relations between different event mentions. As a result of this process, the text is enriched with semantic concepts and identifiers that can be used to access lexical resources and ontologies. For each unique event, we will also derive its factuality score based on the textual properties and its provenance. Table 1 shows the NLP modules used within NewsReader for four languages.

The data-centric architecture of the IXA pipes relies on a common interchange format in which both the input and output of the modules needs to be formatted to represent and filter linguistic annotations: the NLP Annotation Format (NAF[2] [Fokkens *et al.*, 2014]). NAF is a standoff layered representation of the analysis of a whole series of NLP modules ranging from tokenization, part-of-speech tagging, lemmatization, dependency parsing, named entity recognition, semantic role labeling, event and entity-coreference to factuality and opinions. NAF is a document based representation and it is compliant with the Linguistic Annotation Format (LAF [Ide *et al.*, 2003]).

3.2 Frameworks for Big Data analysis

The two implementations that we have implemented for the NewsReader event detection rely on two basic frameworks for Big Data analysis: *Apache Hadoop* and *Storm*. *Apache Hadoop*[3] is a framework designed to perform large scale computations that is able to scale to thousands of nodes in a fault-tolerant manner. It is probably the most widely used framework for large scale processing on clusters of commodity hardware. Hadoop implements *MapReduce*, a programming model for developing parallel and distributed algorithms that process and generates large data sets. Hadoop is the basis for a large number of other specific processing solutions such as Mahout[4] for machine learning or Giraph[5] for graph processing, to name but a few.

As said before, Hadoop follows a *batch* processing model, where computations start and end within a given time frame. In a *streaming computing* scenario [Cherniack *et al.*, 2003], however, the processing is open-ended. Thus, the program is designed to process documents forever while maintaining high levels of data throughput and a low level of response latency.

Storm[6] is an open source, general-purpose, distributed, scalable and partially fault-tolerant platform for developing and running distributed programs that process continuous streams of data. Storm is agnostic with respect to the programming model or language of the underlying modules, and, thus, it is able to integrate third party tools into the framework.

[2] http://wordpress.let.vupr.nl/naf/
[3] http://hadoop.apache.org/
[4] https://mahout.apache.org/
[5] urlhttp://giraph.apache.org/
[6] http://storm.incubator.apache.org/

The main abstraction structure of Storm is the *topology*, a top level abstraction which describes the processing node that each message passes through. The topology is represented as a graph where nodes are processing components, while edges represent the messages sent between them. Topology nodes fall into two categories: the so called *spout* and *bolt* nodes. *Spout* nodes are the entry points of a topology and the source of the initial messages to be processed. *Bolt* nodes are the actual processing units, which receive incoming text, process it, and pass it to the next stage in the topology. There can be several instances of a node in the topology, thus allowing actual parallel processing.

The data model of Storm is a *tuple*, namely, each *bolt* node in the topology consumes and produces tuples. The tuple abstraction is general enough to allow any data to be passed around the topology.

In Storm, each node of the topology may reside on a different physical machine; the Storm controller (called *Nimbus*) is the responsible of distributing the tuples among the different machines, and of guaranteeing that each message traverses all the nodes in the topology. Furthermore, *Nimbus* performs automatic re-balancing to compensate the processing load between the nodes.

Section 4 describes our solution to Big Data processing using *virtualization*, Apache Storm and a set of NLP tools organized in a data-centric architecture. The description of such NLP tools is the subject of the next section.

4 Streaming architecture using Storm

This section describes the approach for streaming processing implemented in the project. The streaming computing architecture requires that the computation is not limited to any timeframe. Instead, the pipeline is always waiting to the arrival of new documents, and one such new document arrives the NLP processing starts. This setting allow NewsReader to rapidly ingest, analyze, and correlate information as it arrives from real-time sources.

We implemented a distributed architecture for parallel processing of documents, comprising many processing nodes and a controller node that orchestrates the document processing flow among the nodes. In this setting a document is typically analyzed in different processing nodes, and, even, the analysis stages each documents goes through can be executed in parallel. The main characteristics of the implemented streaming pipeline are the following:

- Following a streaming architecture approach, documents may arrive at any time into the pipeline, and the NLP processing starts as soon as new documents appear.
- The computation is distributed, that is, different stages of the pipeline may be processed on different machines.
- The pipeline has an unique entry point. Documents are sent to the pipeline using this single entry point, and the distribution of the NLP processing among the machines is done automatically.
- Likewise, the processed documents are left into a single output queue, which in turn is used to populate the KnowledgeStore repository with new information.
- The pipeline is agnostic with regard of the NLP components. The only requirement to integrate new NLP components is that of consuming and producing NAF documents.
- The topology —the definition of the NLP modules and its placement into the pipeline— is described in a declarative way, and thus easily modified.
- The pipeline allows the processing of non linear topologies. Whenever two or more modules have no inter-dependencies, they can be executed in parallel for a single document. We will describe non linear topologies in Section 4.4.
- The pipeline is easily scalable. It is very easy to include more computer power into the pipeline by adding more machines. The pipeline will automatically rebalance itself and include the new machines, which will be thus integrated into the pipeline cluster.

4.1 Newsreader processing cluster

The NewsReader streaming pipeline is meant to be run and deployed into a cluster of machines. As in the first version of the pipeline, the current implementation relies on

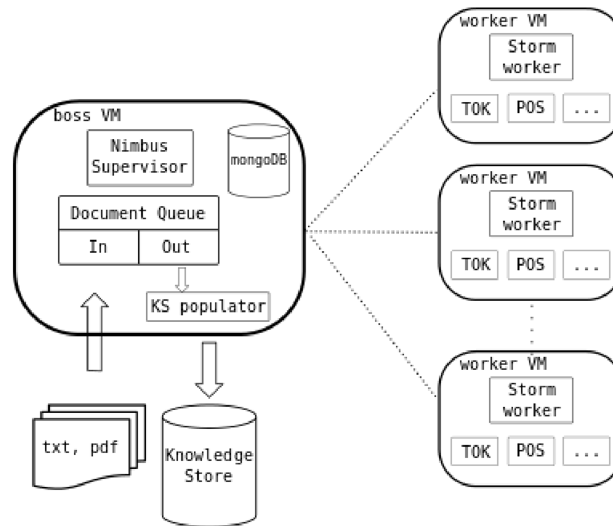


Figure 7: Main architecture of the NewsReader architecture for streaming processing.

virtual machines (VM) that contain all the required modules to analyze the documents. The use of VMs is justified as they allow the installation of linguistic processors, which are typically complex software packages that require many dependencies to be met to perform their tasks. In NewsReader all the NLP modules, along with all the dependencies, are installed into a VMs, which is then copied and deployed into clusters of computers. Using VMs also allows the replication of the linguistic analysis results, which is a is very important goal of the project. One LP module applied to a particular input text has to produce the same output regardless the software framework (machine, operating system, etc.) where it is installed. Therefore, special care has to be taken on guaranteeing that the same version of the LP modules, along with the exact same dependencies, are deployed among machines. Finally, virtualization is a widespread practice that increases the server utilization and addresses the variety of dependencies and installation requirements. Besides, virtualization is a 'de-facto' standard on cloud computing solutions, which offer the possibility of installing many copies of the virtual machines on commodity servers.

Figure 7 shows the general overview of the distributed pipeline. The pipeline architecture consists of a number of VMs: one *boss* node and several *worker* nodes. We will see both VM types in turn.

The *boss* node

The *boss* contains a document queue where input documents stored until the linguistic processing begins. On a streaming computing scenario the pipeline can receive documents at any time, with varying frequency rates. Sometimes the documents will arrive at a slow pace and can be immediately consumed by the pipeline. However, on some occasions many documents may arrive at the same time or within a tight timeframe, and the pipeline may not be able to process them. The queue will streamline the documents according to the

arrival time, and they will wait there until the pipeline is ready for consuming them.

The *boss* node supervises the execution of the different stages of the processing pipeline. Once a certain step of the pipeline ends, the *boss* decides which *worker* node to use for the next pipeline stage. As a consequence, a document may be analyzed on different *worker* nodes as it passes through the pipeline stages. This synchronization is automatically performed by the *Nimbus* component of the STORM framework.

Because the documents may be analyzed on different nodes, it is very important that the information among *worker* VMs can be shared quickly. In the first version of the pipeline the document themselves where serialized into an XML string and passed from one stage to the next in the form of a tuple. The second year of the pipeline uses a NoSQL database instead, which is used to store all the intermediate results of the processing. The database, called *MongoDB*, is also supervised by the *boss* node in the pipeline.

Finally, when the processing is over the *boss* node stores the processed NAF document into and output queue. A process pulls those documents from the queue and populates the KnowledgeStore with its contents. Once again, using a document queue allows streamlining the documents until they can be consumed by the KnowledgeStore repository.

The *boss* node implements a REST service that accept documents to the processing pipeline. For instance, the following command sends the document `doc.xml` to the processing pipeline:

```
% curl --form "file=@doc.xml" http://BOSSIP:80/upload_file_to_queue.php
```

where `BOSSIP` is the IP address of the *boss* VM.

***worker* nodes and NLP modules**

The cluster contains many *worker* nodes that actually perform the document processing. A cluster will typically contain many *worker* nodes, therefore allowing many instances of the NLP modules running in parallel. The *worker* nodes contain copies the same NLP processors, which are synchronized from the *boss* node at *worker* creation time, and can be regularly updated.

The NLP modules of the pipeline are developed and tested using a single VM hosted at the EHU servers, called the IXA *master* node. The node contains all the NLP modules used in the pipeline. New versions of the NLP modules, which typically fix bugs and implements new features, are deployed and tested into the *master* node. The module integration into the NewsReader pipeline is also tested, and scripts are provided to automatically check the validity of the output produced by each module.

The distributed cluster contains scripts to synchronize the NLP modules from the IXA *master* node. Instead of having all the *worker* nodes synchronizing with the *master* node, the synchronization is performed in two steps:

1. The *boss* node synchronizes with IXA *master* and retrieves the last version of the modules and topology specifications.

```
<topology>
  <cluster componentsBaseDir="/home/newsreader/components"/>
  <module name="EHU-tok" runPath="EHU-tok.v21/run.sh"
          input="raw" output="text"
          procTime="1"/>
  <module name="EHU-pos" runPath="EHU-pos.v21/run.sh"
          input="text" output="terms"
          procTime="2" source="EHU-tok"/>
  <module name="EHU-nerc" runPath="EHU-nerc.v21/run.sh"
          input="terms" output="entities"
          procTime="11" source="EHU-pos"/>
<!-- ... -->
</topology>
```

Figure 8: Excerpt of the module specification document.

2. The *worker* nodes synchronize with the *boss*.

The distributed cluster will typically reside on a different location that the *master*. Thus, this setting minimizes the internet traffic between physically distant nodes, while maintaining the bulk of the traffic inside the machines on the same cluster.

Inside each *worker* the modules are managed using the Storm framework for streaming computing, where each LP module is wrapped as a *bolt* inside the Storm topology (c.f. Section 3.2). When a new tuple arrives, the *bolt* calls an external command sending the tuple content to the standard input stream. The output of the LP module is received from the standard output stream and passed to the next node in the topology. Each module thus receives a NAF document with the (partially annotated) document and adds new annotations into it. The tuples in our Storm topology consist of a document identifier, which univocally identifies a NAF document as stored in the MongoDB database (c.f. Section 4.1.2).

4.1.1 NLP processing topology

The NLP processing comprises 15 modules that perform all the required tasks for document event extraction. Table 2 shows the NLP modules, along with the pre- and post requisites of each of them, indicating also which NAF layers each one of them consume and produce (in brackets).

The order in which the modules are executed is described by means of a Storm topology, a graph of calculation that represents the steps a document has to go through to be fully analyzed. The module information and the relative order of module execution is declaratively described in an XML document, as shown in Figure 8. The root `<topology>` element contains a `<cluster>` sub-element, that describes the root directory where all NLP modules are installed, followed by one `<module>` element per NLP component.

Each `<module>` element has the following attributes:

Module	Description	Input (NAF layer)	Output (NAF layer)
TOK	Tokenizer, Sentence splitter	Raw text (`raw`)	Tokens (`text`)
POS	POS tagger	Tokens (`text`)	Lemmas, POS tags (`terms`)
NERC	Named Entity Recognition	Lemmas, POS tags (`terms`)	Named entities (`entities`)
Parse	Constituency parser	Raw text (`raw`)	Parse trees (`constituency`)
Coref	Coreference resolution	Entities, Parse trees (`entities, constituency`)	Coreference relations (`coreferences`)
Opinion	Opinion detection	Entities, Parse trees (`entities, constituency`)	Opinion holders (`opinions`)
WSD-ukb	Word Sense Disambiguation	Lemmas, POS tags (`terms`)	Synsets (`terms`)
WSD-ims	Word Sense Disambiguation	Lemmas, POS tags (`terms`)	Synsets (`terms`)
NED	Named Entity Disambiguation	Tokens, Lemmas, Entities (`text, terms, entities`)	Disambiguated entities (`entities`)
SRL	Dependency parsing, Semantic Role Labeling	Lemmas, POS tags (`terms`)	Dependencies, Semantic Roles (`deps, srl`)
time	Time expressions	Lemmas, Entities, Parse trees (`terms, entities, constituency`)	Time expressions (`timeExpressions`)
eCoref	Event coreference	Lemmas, Semantic roles (`terms, srl`)	Event coreferences (`coreferences`)
tempRel	Temporal relations	Lemmas, Entitites, Parse trees, Coreferences, Semantic roles, Time expressions (`terms, entities, constituency, coreferences, srl, timeExpressions`)	Temporal relations (`temporalRelations`)
causalRel	Causal relations	Lemmas, Entitites, Parse trees, Coreferences, Semantic roles, Time expressions, Temporal relations (`terms, entities, constituency, coreferences, srl, timeExpressions, temporalRelations`)	Causal relations (`causalRelations`)
Fact	Factuality	Lemmas (`terms`)	Factuality indications (`factualityLayer`)

Table 2: LP modules of the NewsReader pipeline, with the pre- and post-requisites.

- `name`: the module name.
- `runPath`: the path relative to `componentsBaseDir` of the module executable.
- `input`: the NAF layer(s) that the module consumes.
- `output`: the NAF layer(s) that the module produces.
- `procTime`: the relative percentage of processing time that this particular module spends.
- `source`: the previous module in the pipeline. The first module in the pipeline has no `source` attribute.

The topology is fully specified by following the parent chain as specified by the `source` attribute of each module. The topology builder, which loads and executes a topology into the cluster, reads the modules specification document and creates a computation graph

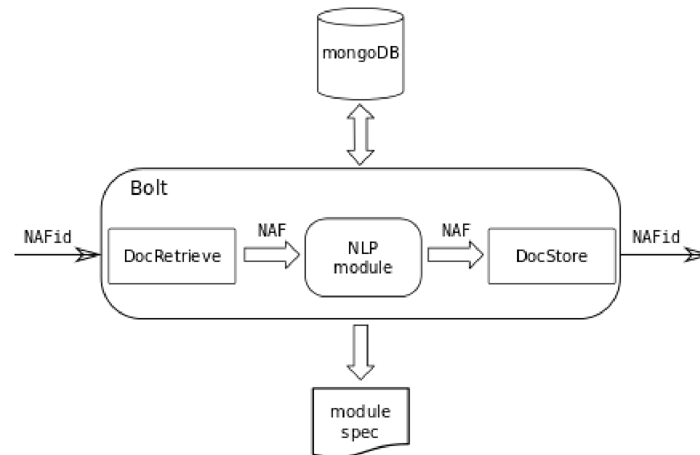


Figure 9: MongoDB integration.

which follows the required order. It also checks that the topology is well formed, and that the graph has no cycles.

The `procTime` attribute describes the relative time this module spends when analyzing the document. The value is used for tuning the pipeline and specify how many copies of a particular module will be executed in parallel. The main idea is that the most demanding modules, according to the processing time, are the best candidates to be replicated, so that many instances are executed in parallel. This value is of course unknown beforehand; in our case, we have executed a version of the pipeline with one single copy for each module over a single set of documents, and we have measured the elapsed time spent on each module. This time has been used to fill the `procTime` attribute of the modules.

The number of instances for each module is calculated according to the following formula:

$$p = \frac{t \cdot N - 1}{T} + 1 \quad (1)$$

where t is the is the percentage of elapsed time of the module (the `procTime` attribute), T is the maximum processing time percentage in the pipeline and N is the number of *worker* nodes.

4.1.2 MongoDB integration

In the distributed architecture documents may be analyzed on several *worker* nodes as they pass through the different stages of the pipeline and, as a consequence, partially analyzed NAF documents have to be shared among *worker*s. We use a NoSQL MongoDB database to store the partially annotated NAF documents, so that the impact of sharing the NAFs is minimized.

MongoDB is a document oriented NoSQL database. It is schema free and contains Databases, Collections and Documents. We store all NAF documents into a collection,

and each specific NAF layer is stored as a MongoDB document in the collection. We also create two separate collections, one for storing the NAF header metadata and one for storing the processing logs.

Figure 9 depicts the integration of MongoDB into the NLP components. Each NLP component is wrapped inside a Storm *bolt* which consumes and produces messages in form of tuples. In our case, the tuples consist in a NAF document identifier, which unequivocally identifies a partially annotated NAF as stored in the MongoDB database. The *bolt* also needs the module specification as explained in Section 4.1.1, so that it knows which NAF layer the underlying NLP module consumes and produces.

The *bolt* node queries the MongoDB database and re-creates a subset of the NAF document by retrieving just the NAF layers (and its dependencies) required by the module. The NAF document is then sent to the analysis engine, which produces new annotations layers or modifies previous ones. The new annotations are then stored again into MongoDB, and the newly created NAF document identifier is passed to the next stage of the processing pipeline. Note that in this setting the NLP modules receive only the minimal information needed by them to perform their work. In other words, no time is spent in parsing large NAF layers that are not required by the NLP modules.

Structure of MongoDB database In this section we will briefly describe the internal structure of the MongoDB database as used in the distributed pipeline. We designed the database structure so that the most frequent database operations can be performed as efficient as possible. The main requisite for the database organization is thus that of allowing quick access to a large number of annotated NAF documents.

The main information unit in MongoDB is the *document*. Documents are stored in BSON format, a format similar to JSON. Documents are grouped into *collections*, which are, loosely speaking, similar to tables in relational databases. MongoDB can deal with many databases, each one comprised by many collections.

We store each NAF annotation layer into a separate MongoDB collection, and within the collection each MongoDB document corresponds to a unique NAF document. That is, the database contains as many collections as NAF layers, and within each collection there are as many MongoDB documents as NAF documents. Besides, we allow the possibility of storing partial documents into MongoDB; for instance, we can split a NAF document into many parts (e.g., paragraphs or sentences) and store each piece into a separate MongoDB document. Figure 10 shows the organization of NAF documents into MongoDB collections and documents.

Each MongoDB document has the following attributes:

- `_id`: this is a mandatory attribute, the document identifier. We create the `_id` value by concatenating the original document ID, the paragraph number and the sentence number.

- `doc_id`: This is the unique identifier of the original document.

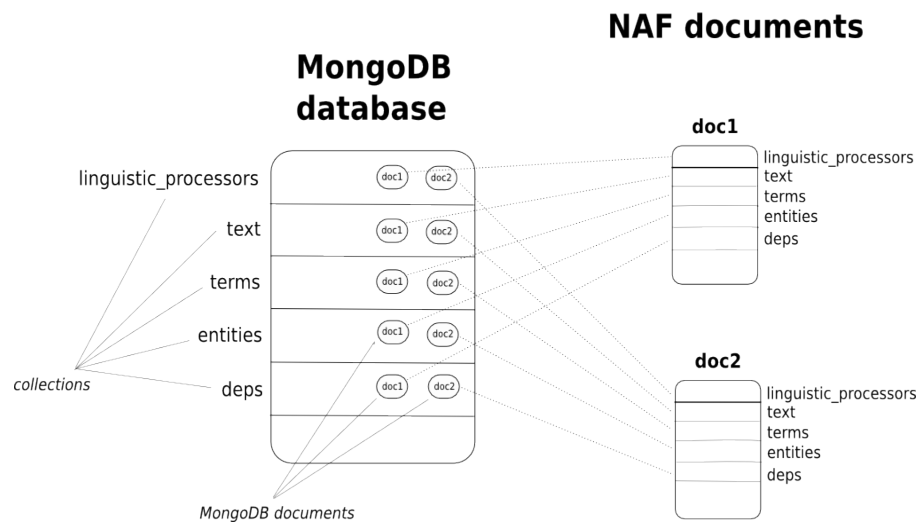


Figure 10: Structure of the MongoDB database.

- `annotations`: This attribute contains a list of MongoDB subdocuments, each of them describing a NAF annotation. The type of annotation depends on the collection it belongs.

Storing NAF documents into MongoDB requires representing the NAF content (an XML document) in BSON, a JSON dialect natively used in MongoDB. The main rule is to use key-value attributes and basic data types like strings or numbers to represent the XML attributes, and BSON objects to map XML elements. Cross layer references in NAF are represented by a list of string elements, each element describing the identifier of the target element.

As with any other database management system, handling indices correctly is essential to perform efficient queries. As mentioned before, this database will be part of a distributed system to process big amount of text documents. Being the performance one of the most important aspects of the system, appropriate indices are required to query the database efficiently. An important feature of our system is that there is only one type of read query, in which all the annotations of a specific layer of a specific NAF document are retrieved. Taking into account that the required information is always found in a single MongoDB document, we only need to know its identifier. Please notice that the layer name is not an attribute of a MongoDB document, but it defines the collection into which the document is found. Therefore, the only thing we need to perform the queries efficiently is to index the `_id` field of the database. Similarly, our system only performs one kind of write query. Each module of the processing pipeline writes a full layer of a NAF document in a database document.

With this structure, we avoid two potential problems of inappropriate MongoDB designs. One of them is the document growth. If the MongoDB documents are updated and they exceed the allocated space, MongoDB will relocate the document on disk. We organize the documents in a way that they won't be changed over the time. The other

important thing is the atomicity of operations. In MongoDB, operations are atomic at document level. Thus, we keep all the writes atomic writing one single document at a time.

4.1.3 VM from scratch

We have developed a set of scripts with the aim of automatically create a fully working cluster for distributed NLP processing. We call these scripts "VM from scratch", as they create and configure the required virtual machines. The scripts are in a github repository, at this address:

`https://github.com/ixa-ehu/vmc-from-scratch`

Creating a cluster using the scripts involves three main steps:

1. Create a basic cluster with the boss node and one worker node.
2. Make as many copies as required of the worker node.
3. Deploy the worker copies among different hosts and run the cluster.

The full documentation of the "VM from scratch" scripts are described in appendix C. Now we will just depict the main steps to create a cluster using the scripts:

Create a basic cluster

The first step is to create the basic cluster using the `create_basic_cluster.pl` script. The script will create two VM images, one for the *boss* node and one for the first *worker* node (called `workervm1` in the example). It is executed as follows:

```
% sudo create_basic_cluster.pl --boss-ip 192.168.122.111 --boss-name bossvm \\
  --worker-ip 192.168.122.112 --worker-name workervm1
```

The script needs some parameters, such as the IP addresses and host names of the *boss* and first *worker* nodes.

Once the first nodes are created, one need to log into the *boss* VM and retrieve all the software into the *boss* and *worker* node (including the NLP processors from the IXA *master* node). This is automatically done using the `/root/init_system.sh` script located at the *boss* node.

When the scripts finalizes the basic cluster is finally created and configured, and can be used to start analyzing documents. However, it does not make much sense to use a cluster with a single worker node. Therefore, the next step is to create the required copies of the worker VMs, using the `./cp_worker.pl` script. This script created as many copies of the *worker* node as needed, which are integrated into the processing cluster.

The last step is to login into the *boss* node and load a topology to start analyzing documents. The topology loader needs a topology definition as defined in Section 4.1.1 and also

the number of *worker* nodes of the cluster. This number is used to create many instances of the NLP modules in the topology, so that effective parallel execution is achieved.

It is worth noting that more *worker* nodes can be added to the topology at any moment. For this, just create a new *worker* using the `cp_worker.pl` script, and then login to the *boss* node and run a *rebalance* command, which will include the newly created worker when deploying the NLP modules among machines.

4.2 Experiments

This section describes a series of experiments made within the project to test the effectiveness of the distributed cluster when analyzing documents. We have tested the cluster on different settings and have measured the performance gain obtained by parallelizing the processing on each of them.

The main goal of the experiments is to assess the suitability of the NewsReader distributed pipeline when processing large quantities of documents in a reasonable time. In particular, we want to answer to the following research questions:

- How does the distributed architecture perform in a batch *and* streaming settings?
- What is the gain of using non linear topologies?

The distributed pipeline presented in this Section is focused to streaming scenarios, but it can also be used for batch processing nonetheless. If we want to analyze a large batch of documents, we can send them all to the input queue and the documents will be produced by the analyzing pipeline in a batch fashion. The first question focuses on analyzing the architecture behavior on both settings (streaming and batch) so that we can learn which configuration is best for each case.

The second question focuses on measuring the gain of using non linear topologies for both streaming and batch scenarios. As we will see, the distributed pipeline allows the definition of non linear topologies, where two or more modules can process the same document at the same time. We will test the use of non linear topologies and measure its main benefits.

Finally, the third question focuses on the overhead (or gain) that certain architectural decisions pose on the overall system performance. In particular, we want to measure the impact of the MongoDB and Storm systems in the system.

The next sections will analyze these aspects in turn.

4.3 Batch and Streaming Processing

We first tried the topology on a batch setting. For this, we selected a set of 100 documents and sent them to a cluster comprising 6 different workers. We tried several alternatives, regarding the cluster parameters:

- Baseline: a single instance of each NLP module.

Setting	Elapsed time (min)	Gain
Baseline	260.67	–
ALL_6	81	+69.74%
SRL_6	68.05	+74.58%
p_6	73.22	+72.65%
MONO	55.3	**+79.34%**

Table 3: Processing time for a batch of 100 documents on a cluster comprising 6 worker nodes.

- ALL_6: 6 instances of each module.
- SRL_6: 6 instances of the SRL module; the rest of modules have one single instance. We chose to parallelize the SRL module because it is the most demanding module regarding processing time.
- p_6: we calculated the number of instances for each module according to equation (1)[7].
- MONO: this setting mimics the *monolithic* approach taken in the first year of the project. In this setting each document is wholly analyzed on a single *worker* node, following a pipeline architecture. Note that this setting corresponds to having 6 instances of each module, but at any time there is only one module being executed in each *worker* node.

Table 3 shows the elapsed times for the 100 documents on the different settings. With no parallelization the documents need circa 4 hours and a half to be wholly analyzed. The table shows that creating many instances of the most demanding module (SRL_6) is preferable than having many instances for all modules (ALL_6). This result indicates that in the ALL_6 setting the NLP modules are competing to obtain CPU cycles on each worker node, incurring a overall penalty in performance. Using the equation to calculate module instances alleviates this problem, but it is still 2 percentual points worse than the SRL_6 setting, as shown in the table.

All in all, Table 3 clearly shows that the maximum gain is obtained by following the MONO setting for batch processing. In this setting the CPUS of the nodes are always working, and therefore there is small room for improvement by using a distributed architecture among machines. In fact, the results show that MONO represents an upper-bound to the overall elapsed time in the processing pipeline, when following a batch processing scenario.

With this results at hand, we processed a two batches of documents comprising 2000 and 1000 documents of medium size, respectively. The latter batch is a subset of the former. We used a cluster comprising 10 *worker* nodes under the MONO setting.

[7]Page 30.

Setting	Docs	Elapsed time	Throughput	Proc. time	Latency (doc/sent/token)
Batch	2,000	1819.47	1.10	15130.78	7.14 / 0.21 / 8.40×10^{-3}
Batch	1,000	719.4	1.39	6519.12	6.66 / 0.18 / 7.48×10^{-3}

Table 4: Statistic for batch processing. Throughput measures the ratio between the number of documents and the elapsed time.

We use two traditional metrics to judge the extent to which the parallelization is achieved: latency and throughput. The latency is the time taken to process an individual data set (document, sentence, word), while the throughput is the aggregate rate at which the data sets are processed and is calculated as the ratio between the number of documents and the total time spent analyzing them (elapsed time). Note that since multiple documents may be pipelined or processed in parallel, latency and throughput are not directly related.

Table 4 shows the elapsed times and latencies for the document batches, including the elapsed time, as well as the processing time, i.e., the sum of the times spent by each module and document. Documents have an average latency of 7.14 minutes to be fully analyzed although some documents caused some modules to spend a very large amount of time (the maximum time needed was 67.83 minutes).

Processing 2000 documents on a single node would require more than 10 days (5 days for 1000 documents); using the cluster, this elapsing time falls to 20 hours (resp. 10 hours), that is, we achieve a parallelism ratio of 8.5. One would expect a parallelism ratio of 10, as we have this number of nodes working on parallel. However, there are two important factors that determine not reaching to the maximum ratio:

- The last documents do not finalize at the same time in the machines. Many *worker* nodes remain on an idle state until the last document(s) of the batch is analyzed. If those last documents are large, a drop in the parallelism ratio occurs.
- The input documents have to be distributed into ten partitions of the input queue before the computation begins. However, this partitioning is done randomly and regardless of the size of the documents. As a consequence, some partitions contain many big documents, while some others contain only small documents. A particular *worker* node pulls documents from a single partition, and therefore there could be idle some nodes while some others have still many documents to consume.

The last factor is certainly an implementation restriction of the software used to implement the input queue. The results of this experiments clearly suggest that we need to test alternative software packages for implementing the queue.

Next, we want to try the distributed pipeline on a streaming computing scenario. In this setting, documents arrive to the processing pipeline at any time, and they have to be analyzed as fast as possible. We have implemented a Poisson process that emulates the arriving of documents randomly within a time frame. A Poisson process is useful to emulate events which occur individually at random moments, but which tend to occur

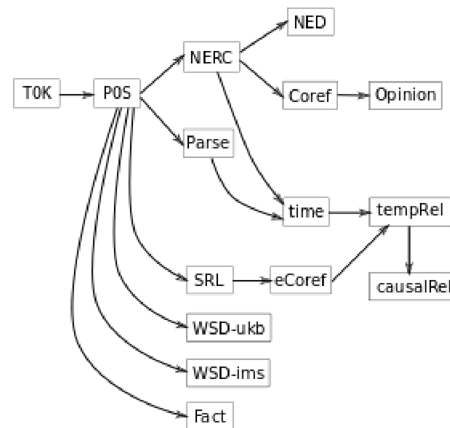


Figure 11: Non linear topology.

at an average rate when viewed as a group. It has one main parameter, called the *rate parameter*, that indicates the average rate of events (documents) per unit of time. The batch experiment suggests that analyzing a document needs between one and one and a half minutes on average, and therefore we set the rate parameter to 1000 documents within a time frame of 33 hours.

Setting	Docs	Proc. time	Latency (doc/sent/token)
Linear	1,000	4620.90	5.00 / 0.14 / 5.78×10^{-3}
Non linear	1,000	5421.33	2.78 / 0.08 / 3.13×10^{-3}

Table 5: Statistic for streaming processing for linear and non linear topologies.

The first line of Table 5 shows the processing time and latencies for the same set of 1000 documents used in the batch processing. This table omits the elapsed time and throughput measures, as we intentionally maintain the cluster idle when the queue is empty and waiting to the arrival of new documents. Comparing to the batch scenario, the documents need less time to be processed. This can be explained by the fact that in a batch processing scenario the CPUs on each node are always at a maximum processing rate, and therefore some modules suffer a penalty in terms of page faults and context switching. As a consequence, the overall latency is slightly better in a streaming scenario. We expected no significant gain in latency because the modules are executed on a following a linear pipeline architecture and thus the elapsed time required for a single document is more or less the same on both settings.

4.4 Non linear topologies

The distributed pipeline offers the choice of implementing non linear topologies. Linear topology executes all the processing steps following a pipeline architecture, i.e., one module after another. In contrast, non-linear topologies allow many processing stages to be

executed at the same time for a single document. By implementing non-linear topologies we expect that the overall document latency to decrease, as many processing stages are executed at the same time.

Two or more modules may be executed in a non-linear fashion whenever there is no dependency between them. For instance, according to the pre-requisites of each module shown in Table 2 one can see that the NERC and SRL modules can be executed non linearly, as they both depend on the `terms` NAF layer. On the other side, both modules have to be executed after POS, as the latter is the module which produces the `terms` layer.

Figure 11 depicts the non linear topology used in our experiments. The topology is defined by following the dependency chain of the NLP modules described in Table 2. Implementing non linear topologies require that modules can work with the required subsets of the NAF document needed by the module. Fortunately, this is trivially accomplished by using MongoDB to store intermediate NAF documents, as the wrappers retrieve only the required NAF layers for a certain NLP module.

It makes little sense to run experiments of non linear topologies on a batch processing scenario. As said before, in this setting the cluster nodes are always working at full capacity, so we expect no overall gain by executing the NLP modules for a single document in parallel. However, we expect a performance boost in latency when using non linear topologies on a streaming processing scenario. When a document arrives, its processing can be distributed among the idle machines of the cluster, and therefore the average time needed by each document until it is fully analyzed should drop significantly. The second row of Table 5 confirms our intuition. Although the processing time using non linear topologies is higher, the overall latency dropped almost to a half, that is, a document needs half the time to be analyzed.

Looking at the figure 11, we can see that there can be up to 6 modules running in parallel. Therefore, one would expect more than a 50% gain in the overall latency when using non linear topologies. However, the time required by each NLP module is very unbalanced. In fact, the SRL module takes almost the 60% of the overall processing time for a document, and, therefore, it represents a bottleneck of the overall processing. Although all other modules finish first, they have to wait to the SRL to end in order to complete the document analysis. Note also that having many instances of the SRL module does not alleviate this problem, as the module works linearly for each document. One possible solution to this problem is to implement different granularities in the NLP modules, that is, make the modules able to analyze fragments of documents such as sentences, paragraphs or word contexts. This way, many instances of the same module can be applied in parallel for a single document, thus reducing the overall latency. This is an option we want to investigate further during the third year of the project.

5 Batch architecture using Hadoop

The Storm architecture described in the previous section is particularly suitable for real time processing. This is the architecture that is meant to process news articles as they come in, analyze them, relate them to what is already known in the KnowledgeStore and add new knowledge. However, there are many situations in which large batches of data need to be processed and real time processing is not necessary. For instance, when we fill the KnowledgeStore with information from news over the last several years to get our background information for the news that is newly arriving, we are processing large batches. The same applies in many practical cases, such as preparing data for Hackathons or evaluations during the project. Even if we update news on a daily basis, this can be handled by batch processing. Because of these scenarios, alternative approaches that can be more efficient when processing batches are of interest to NewsReader.

A collaboration with SURFsara led to a proposal and implementation of a Hadoop architecture that can efficiently process large batches of newspaper text with the NewsReader tools. This alternative architecture (which makes use of the same NLP modules and still uses the NAF representation to pass information between modules and represent our output) has allowed us to use a significantly larger amount of data for the Hackathons and user evaluation of the second project year (a total of 1.3 Million articles in the second year compared to 63K in the first year). This section provides a brief introduction and description of the setup.

5.1 Hadoop

Hadoop[8] ([Venner *et al.*, 2014; White, 2009]) is a framework that supports processing of very large sets of data distributed across clusters. Hadoop can be used to partition data and computation and scale from a single server to thousands of servers using simple programming models. It furthermore has the advantages of being robust and fault tolerant and running on commodity hardware. Disadvantages are that the APIs are implemented in Java and there is no extensive support to run normal binaries. For the NewsReader pipeline, a solution had to be designed and implemented that allows for modules programmed in various languages to be combined.

5.2 Cascading

Placing a complex pipeline consisting of a variety of modules with different requirements into the MapReduce framework ([Dean and Ghemawat, 2008], the processing part of Hadoop) is not necessarily trivial. Cascading[9] was used to facilitate this task. Cascading is specifically designed to support complex workflows on Hadoop without diving into the complexity of MapReduce jobs. The current Hadoop architecture creates a Java object for each task in the pipeline. This object takes a path to the NLP module, a path to a

[8] `http://hadoop.apache.org/`
[9] `http://www.cascading.org`

scratch directory and a NAF input file as its arguments.[10] Any module that can be called from the command-line taking standard input, a path to the module itself and a path to a temporal or scratch directory and then produces a NAF file as standard output can be integrated in the cascading architecture. The implementation includes a 10 minute time out for each module to reduce lost time on files that lead to errors. In case of failure, the latest produced NAF file is stored so that information obtained in previous processing steps is not lost.

Most modules were adapted so that they could be integrated into the architecture in a similar way as they are used independently or when integrated in Storm. The only exception is the module for named entity disambiguation which communicates with DBpedia. Rather than starting up DBpedia every time the module was called, DBpedia was installed on a separate server to which the module calls.

5.3 Scaling up news processing

At the time of processing the car dataset, SURFsara had 86 nodes, 8 cores, 64GB RAM and 16TB disks available. Initial investigations on the performance of NLP modules in this settings were carried out. Figure 12 illustrates how much time individual modules spent on 500 documents from an initial test set. The figure shows that modules are generally well behaved, with the occasional outlier.

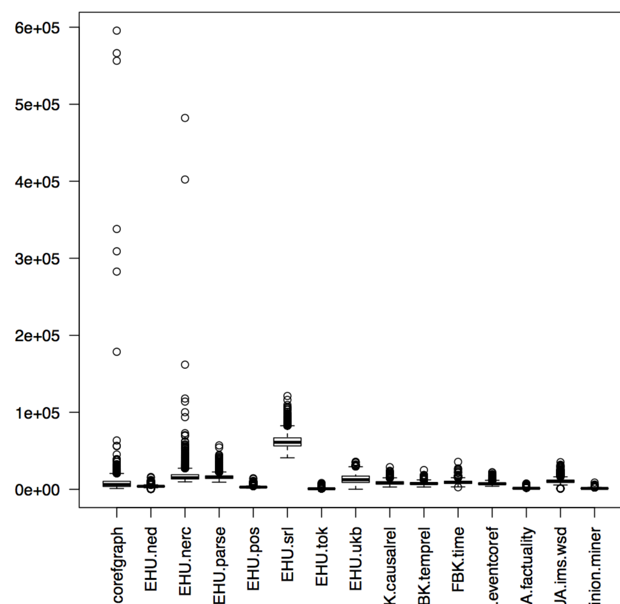


Figure 12: Time individual NLP modules spent on processing documents

[10]This basic setup applies to most modules in the pipeline, some require additional settings.

Based on the first evaluation results, we made an estimation of the requirements for processing the news that comes in on a single day.

- LexisNexis estimation: approximately 1 million articles per day
- We need 170 seconds per document on average for 1 core with approximately 8GB RAM
- Approximately 508 documents can be processed per core per day
- SURFsara has 680 cores on Hadoop (will be 1440): 350K to 730K documents per day at SURFsara
- 50,000 core hours per million documents:
  - We need 2100 cores for a day
  - SURFsara can handle a day of news within 3 days (will be 1.5 days)[11]
  - 1 computer will need 5 years

5.4 Processing the car data

The Hadoop architecture was used to process the 1.3 million articles about the car industry that are used during the Hackathon and user evaluation. Even though Hadoop is specifically designed to help distribute jobs over several nodes, the size of the jobs sent to Hadoop and particular settings can have an impact on the speed and efficiency of the processing. In Hadoop, a job (in our case batch of data) is divided over several workers on currently available nodes. Several programs can be run at the same time and Hadoop will distribute the available nodes among them. Each node has several workers that carry out the actual tasks. The load per worker can be defined as a setting when starting a task. While processing the 1.3 million car articles, this was set to 10 - 20 items per worker.

While going through a set of serialized jobs, the next job starts as soon as the previous job has finished. In our dataset, it may happen that the occasional long file or outlier in processing time results in one or two workers still carrying out some tasks while the others are done. This can result in idle nodes that are simply waiting for the job to finish on other nodes. In order to eliminate this scenario, we use two parallel runs. If nodes are idle, because a small part of the job in one run is not finished yet, these nodes can be taken up by a job by the other run.

Processing this data has revealed several ways in which we may further improve efficiency and the overall process. The car data was processed at the Hadoop cluster of SURFsara which has a rather conservative limit on the number of workers per node (only 4). We also used a cluster at the computer science department of VU University that does not impose such restrictions and can handle twice as much data per node. Second,

[11] Provided the full Hadoop capacity can be used for this task.

assigning 10-20 items per worker may not be the most optimal setting, but it proved to be reliable and we decided to stick to it for completing the processing of these datasets. Future experiments can provide further insight into what the best division is. Third, when one of the workers fails due to excessive memory usage, the entire job fails. This happens only occasionally and with a relatively low setting of number of items per worker, the amount of data that is affected remain limited. Nevertheless, the setup should be fixed so that only the document creating the problem fails and all other documents that happen to be part of the same batch remain.

In the previous section, we calculated that SURFsara could handle 1 million news items within 3 days. This estimation would hold under the assumption that no other users are running jobs on the cluster. Processing the 1.3 million articles in this set took approximately 11 days.

5.5 Future work and possible improvements

Despite the straight-forward implementation of the Hadoop architecture, it can currently still be a hassle to update the pipeline or to add a new module. This is mainly due to the fact that the NLP modules are not specifically designed to be run on Hadoop and therefore some requirements Hadoop imposes are not respected. We therefore created a protocol that outlines these requirements. Modules developed in the future should all respect the requirements outlined in this protocol. We aim to revise existing modules as much as possible, so that they too can easily be integrated in the cascading implementation for NewsReader and other architectures in general. Furthermore, the settings presented above (10-20 items per working) were used, because they quickly proved to be stable and we wanted to make sure the data could be processed without difficulty. In addition to removing the conservative limits set by SURFsara, we may gain in efficiency by further optimizing the number of items per worker.

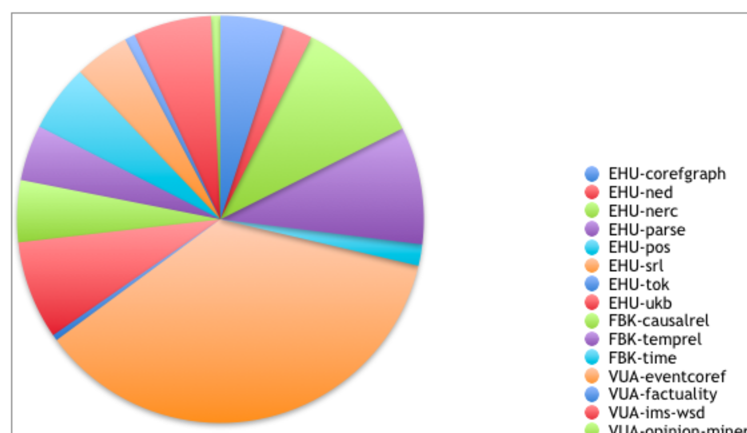


Figure 13: Comparison of time needed per module

Figure 13 reveals the time individual modules take on average. Just like already ob-

served in the Storm architecture, the semantic role labeling module is the most time consuming module in the pipeline. This can partially be explained by the complex language model that has to be initiated each time the module is called. If modules are implemented in Java, it is relatively straightforward to initiate a language module once for a batch of data. However, most of our modules use Python implementations which means that reimplementations of the software are required. It is therefore not feasible to address this problem for all modules within the project. We do hope to address this issue for modules that are already implemented in Java and this can be done relatively easily and the semantic role labelling module for which this change would have the most significant impact.

6 Modeling Provenance and Attribution

Deliverable D2.1 [Beloki *et al.*, 2014] provides a complete overview and motivation of the representation formats we use for representing our linguistic analyses and the final output in RDF. In this section, we describe the latest additions to these representations. In particular, we describe the model we developed for representing provenance information. The section is organized as follows. First, we will briefly outline what is understood by provenance and explain the two kinds of provenance that need to be modeled. We then provide an overview of the attribution values that are annotated in text and explain the requirements for the model. Finally, we present the model that we will use to relate attribution values to statements.

6.1 Provenance and Attribution

For several of NewsReader's goals, it is important to know what the source of information is. Users may want to compare perspectives from different sources, or they want to be able to verify whether the analysis is correct. We distinguish between direct sources (the paper, publisher or author of an article) and quoted sources (sources that are mentioned in the text, e.g. *According to Jerry Yang*, *President Dieter Zetsche says...*). In this document, we will consider the former sources (paper, publisher, author) as part of provenance information. We will use the term *attribution* to refer to the latter (the quoted sources).

Provenance will be modeled using the PROV-O [Moreau *et al.*, 2012]. The PROV-O is specifically designed to model the provenance of data. It allows us to model the data, the processes and agents involved in producing data. We will use PROV-O to model all provenance related information coming from metadata (i.e. publisher, author of text), but also to model information on NLP processing. Deliverable D2.1 describes provenance modeling in NAF.

When texts mention a certain source, we will not use the PROV-DM to indicate this. Though sources mentioned in text also indicate provenance of information, the processes, data and roles of agents involved are different from those commonly modeled by PROV-O. Moreover, there is a factor of uncertainty in the provenance assigned, because it is identified by NLP analyses. Finally, source indications can introduce speech acts which are events just like the 'data' they produce. Sources introduced in the text will therefore be modeled using SEM+ [van Hage *et al.*, 2011].

We can link sources modeled through SEM+ (a quoted source) to sources modeled in PROV if we want to create complete overviews of perspectives or opinions from a specific source (e.g. indicate both what is published by the NY Times and information other sources say comes from the NY Times). There are, however, several possible ways of doing this. They will be described in the coming subsections.

6.2 Attribution annotations

This section provides a brief outline of the linguistic annotations that indicate attribution values and are provided in our gold standard and the relevant output of NLP tools. The annotation guidelines [Tonelli *et al.*, 2014] specify three attribution related values and some special cases. The annotations apply to events and are directly linked to an event mention.

The attribution related values are:

- Polarity (POSITIVE, NEGATIVE, UNDERSPECIFIED): is the event affirmed or denied?
- Interpreted time (FUTURE, NON-FUTURE, UNDERSPECIFIED): does the event take place in the future from the source's point of view or not?
- Certainty (CERTAIN, PROBABLE, POSSIBLE, UNDERSPECIFIED)

In addition three "special cases" can be indicated:

- Part of a conditional if-clause
- Part of a conditional main clause
- A generic statement

For reasons of space and clarity, we will only address the general (non special) attribution values here. In addition, the following information is also provided:

- There is an opinion-mining module that can identify the source (opinion-holder), target (the opinion) and the judgment (positive or negative opinion)
- The Semantic Role Labeling module and Predicate Matrix allow us to identify FrameNet frames and roles. The Event-and-Situation Ontology distinguishes different kinds of attribution verbs (cognition, speech acts, etc).

We can use this output to link individual events to a specific source. If events are included in an opinion's target, the opinion holder is the source. If they are the ARG2 (using more generic propBank roles for reasons of generalization) of a so-called attribution verb, the ARG1 is the source. Attribution verbs and identified opinions also provide indications about the attribution values mentioned above (polarity, certainty, time), but this will be taken into account in the interpretation of attribution.

The current attribution model with its three main properties is relatively simple. It is likely that it will be extended in the future to provide more detailed information. To avoid an ever increasing amount of triples indicating attribution that needs to be attached to relations between events and other entities or event mentions (depending on the model chosen), we define complex attribution values that can incorporate one or more attribution properties. An overview of the current values is given in Figure 14. As we consider more aspects of attribution, more values will be added. These complex values allow us to represent all attribution related information in one triple.

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix nwrontology: <http://www.newsreader-project.eu/ontologies/> .

nwrontology:polarityPOS a  rdfs:property .
nwrontology:polarityNEG a  rdfs:property .
nwrontology:attributionCERTAIN a rdfs.property .
nwrontology:attributionPROBABLE a rdfs:property .
nwrontology:attributionPOSSIBLE a rdfs:property .
nwrontology:tenseNONFUTURE a  rdfs:property .
nwrontology:tenseFUTURE a rdfs:property .

nwrontology:attrPOSCERT rdfs:subPropertyOf nwrontology:polarityPOS .
nwrontology:attrPOSCERT rdfs:subPropertyOf nwrontology:attributionCERTAIN .
nwrontology:attrPOSPROB rdfs:subPropertyOf nwrontology:polarityPOS .
nwrontology:attrPOSPROB rdfs:subPropertyOf nwrontology:attributionPROBABLE .
nwrontology:attrPOSPOSS rdfs:subPropertyOf nwrontology:polarityPOS .
nwrontology:attrPOSPOSS rdfs:subPropertyOf nwrontology:attributionPOSSIBLE .
nwrontology:attrNEGCERT rdfs:subPropertyOf nwrontology:polarityNEG .
nwrontology:attrNEGCERT rdfs:subPropertyOf nwrontology:attributionCERTAIN .
nwrontology:attrNEGPROB rdfs:subPropertyOf nwrontology:polarityNEG .
nwrontology:attrNEGPROB rdfs:subPropertyOf nwrontology:attributionPROBABLE .
nwrontology:attrNEGPOSS rdfs:subPropertyOf nwrontology:polarityNEG .
nwrontology:attrNEGPOSS rdfs:subPropertyOf nwrontology:attributionPOSSIBLE .

nwrontology:attrCERTNF rdfs:subPropertyOf nwrontology:polarityNONFUTURE .
nwrontology:attrCERTNF rdfs:subPropertyOf nwrontology:attributionCERTAIN .
nwrontology:attrPROBNF rdfs:subPropertyOf nwrontology:polarityNONFUTURE .
nwrontology:attrPROBNF rdfs:subPropertyOf nwrontology:attributionPROBABLE .
nwrontology:attrPOSSNF rdfs:subPropertyOf nwrontology:polarityNONFUTURE .
nwrontology:attrPOSSNF rdfs:subPropertyOf nwrontology:attributionPOSSIBLE .
nwrontology:attrCERTF rdfs:subPropertyOf nwrontology:polarityFUTURE .
nwrontology:attrCERTF rdfs:subPropertyOf nwrontology:attributionCERTAIN .
nwrontology:attrPROBF rdfs:subPropertyOf nwrontology:polarityFUTURE .
nwrontology:attrPROBF rdfs:subPropertyOf nwrontology:attributionPROBABLE .
nwrontology:attrPOSSF rdfs:subPropertyOf nwrontology:polarityFUTURE .
nwrontology:attrPOSSF rdfs:subPropertyOf nwrontology:attributionPOSSIBLE .

nwrontology:attrPOSNF rdfs:subPropertyOf nwrontology:polarityPOS .
nwrontology:attrPOSNF rdfs:subPropertyOf nwrontology:attributionNONFUTURE .
nwrontology:attrPOSF rdfs:subPropertyOf nwrontology:polarityPOS .
nwrontology:attrPOSF rdfs:subPropertyOf nwrontology:attributionFUTURE .
nwrontology:attrNEGNF rdfs:subPropertyOf nwrontology:polarityNEG .
nwrontology:attrNEGNF rdfs:subPropertyOf nwrontology:attributionNONFUTURE .
nwrontology:attrNEGF rdfs:subPropertyOf nwrontology:polarityNEG .
nwrontology:attrNEGF rdfs:subPropertyOf nwrontology:attributionFUTURE .

nwrontology:attrPOSCERTNF rdfs:subPropertyOf nwrontology:attrPOSCERT .
nwrontology:attrPOSCERTNF rdfs:subPropertyOf nwrontology:attrPOSNF .
nwrontology:attrPOSPROBNF rdfs:subPropertyOf nwrontology:attrPOSPROB .
nwrontology:attrPOSPROBNF rdfs:subPropertyOf nwrontology:attrPOSNF .
nwrontology:attrPOSPOSSNF rdfs:subPropertyOf nwrontology:attrPOSPOSS .
nwrontology:attrPOSPOSSNF rdfs:subPropertyOf nwrontology:attrPOSNF .
nwrontology:attrNEGCERTNF rdfs:subPropertyOf nwrontology:attrNEGCERT .
nwrontology:attrNEGCERTNF rdfs:subPropertyOf nwrontology:attrNEGNF .
nwrontology:attrNEGPROBNF rdfs:subPropertyOf nwrontology:attrNEGPROB .
nwrontology:attrNEGPROBNF rdfs:subPropertyOf nwrontology:attrNEGNF .
nwrontology:attrNEGPOSSNF rdfs:subPropertyOf nwrontology:attrNEG POSS .
nwrontology:attrNEGPOSSNF rdfs:subPropertyOf nwrontology:attrNEGNF .

nwrontology:attrPOSCERTF rdfs:subPropertyOf nwrontology:attrPOSCERT .
nwrontology:attrPOSCERTF rdfs:subPropertyOf nwrontology:attrPOSF .
nwrontology:attrPOSPROBF rdfs:subPropertyOf nwrontology:attrPOSPROB .
nwrontology:attrPOSPROBF rdfs:subPropertyOf nwrontology:attrPOSF .
nwrontology:attrPOSPOSSF rdfs:subPropertyOf nwrontology:attrPOSPOSS .
nwrontology:attrPOSPOSSF rdfs:subPropertyOf nwrontology:attrPOSF .
nwrontology:attrNEGCERTF rdfs:subPropertyOf nwrontology:attrNEGCERT .
nwrontology:attrNEGCERTF rdfs:subPropertyOf nwrontology:attrNEGF .
nwrontology:attrNEGPROBF rdfs:subPropertyOf nwrontology:attrNEGPROB .
nwrontology:attrNEGPROBF rdfs:subPropertyOf nwrontology:attrNEGF .
nwrontology:attrNEGPOSSF rdfs:subPropertyOf nwrontology:attrNEG POSS .
nwrontology:attrNEGPOSSF rdfs:subPropertyOf nwrontology:attrNEGF .
```

Figure 14: Basic overview of properties that model attribution

6.3 Proposed attribution models

We have explored several ways to model attribution values in RDF. The first proposals we considered assume that attribution values are modeled on the level of *instances*. They provide an indication whether something happened or will happen or not and how certain

this is according to a specific source. Attribution values can also be related to *mentions*. In this case, the attribution values of the actual event can be identified by retrieving where the event was mentioned and what attribution values were related to these mentions. We will follow the latter option in NewsReader. We will first outline the requirements, then we describe how we model attribution on a mention level. A full overview of the alternative proposals modeling attribution on the instance level is provided in Appendix B.

6.3.1 Requirements

The model needs to be able to account for several phenomena. First, certain aspects of an event may have different attribution values as others. The following examples illustrates this:

(1) Willem will not be at the NewsReader meeting in October.

(2) Thomas will be there.

In this example, *there* refers to the meeting. We thus have one event (the meeting), but two attribution values, namely `nwrontology:attrPOSCERTF` and `nwrontology:attrNEGCERTF`. We need to know what exactly is confirmed about this meeting (Thomas's presence) and what is denied (Willem's presence). It nevertheless may also happen that attribution values are assigned to an entire event (e.g. *the NewsReader meeting was cancelled* assigns certain negative polarity to the NewsReader meeting). The model must be able to handle both these scenarios.

Second, the model needs to be able to deal with the scope of attribution values. Negation and uncertainty typically only have scope over a specific syntactic relation or, on the interpretation level, over a specific relation between an event and another entity. Consider the following two examples:

(3) Anna believes that Kim did not kill John with a knife.

(4) Linda says Sandy did not kill Harry.

In sentence (3), the default interpretation would be that Anna believes that Kim killed Harry, but that it was not with a knife. In the case of (3), the negation can either be about Sandy being the killer (but Harry was killed), about the event being killing (but Sandy did do something to Harry, e.g. hurt him or (try to) rescue him), or about it being Harry who was killed (but Sandy did kill someone else). Context is needed to determine which part of the statement is being denied, but it is highly unlikely that the negation concerns more than one of the components or especially all three. We should be able to model the scope of the negation, if known, or to be able to indicate underspecified negation if it is not clear where the negation applies to.

Finally, different perspectives may be given on the same triple. A specific perspective is always connected to the source that expresses the perspective. The model should be able to handle this.

The following properties and requirements guided the design of the model proposed here:

```
:NGsell {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell a sem:Event, fn:Commerce_sell , eso:Selling .
}

:NGsell_r1 {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell fn:Commerce_sell@Seller dbp:resource/Chrysler .
}

:NGsell_r2 {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
        fn:Commerce_sell@Goods nwr:data/cars/entities/Liberty_SUVs .
}

:NGexpect {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect a sem:Event, fn:Expectation .
}

:NGexpect_r1 {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect  fn:Expectation@Cognizer  dbp:resource/Chrysler
.
}

:NGexpect_r2 {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect sem:hasTarget :NGsell .
}

:NGsay {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say a fn:Statement .
}

:NGsay_r1 {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say
             fn:Statement@Speaker dbp:resource/Dieter_Zetsche .
}

:NGsay_r3 {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say
            sem:hasPlace nwr:/data/cars/entities/DaimlerChrysler_Innovation_Symposium .
}

:NGsay_r2 {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say sem:hasTarget :NGexpect .
}

:NGsymposium {
nwr:/data/cars/entities/DaimlerChrysler_Innovation_Symposium sem:hasPlace dbp:resource/New_York_City .
}

:NGsay2 {
nwr:madeupexampletext.xml#ev132Say a fn:Statement .
}
:NGsay2_r1 {
nwr:madeupexampletext.xml#ev132Say fn:Statement@Speaker dbp:resource/Manfred_Bischoff .
}

:NGsay2_r2 {
nwr:madeupexampletext.xml#ev132Say    sem:hasTarget :NGsell .
}
```

Figure 15: Triples modeling instances of basic example sentences

1. Statements need to be related to the entities making them.
2. Triples are derived from documents (through NLP analyses) and the relation to this document is indicated through `gaf:denotedBy`
3. There can be asymmetry between event and participation relations: we want "predicate" both on event instances and on event participation (as well as location, time, etc.)
4. There may be news reporting that a person changed his or her mind (e.g. *Initially Anna said that Kim killed John, but later she stated that Kim could not have done it.*)

The model will be illustrated using the example sentence:

(5) Chrysler expects to sell 5,000 diesel Liberty SUVs, President Dieter Zetsche says at a DaimlerChrysler Innovation Symposium in New York.

In order to illustrate how different perspectives are expressed, we also represent the following made up sentence, next to the sentence above.

(6) Manfred Bischoff said that Chrysler will probably not sell 5,000 diesel Liberty SUVs.

Figure 35 provides the simplified RDF representation including all relations between events and entities of the two example sentences after a correct analysis. This illustration only represents the instances. Relations between instance and mention are represented in Figure 16.

6.4 Modeling attribution

As mentioned above, we will model attribution on the level of mentions. The instance level represents events, other entities and the relations between them. Different statements can be made about these events and relations: one person may claim something is definitely true, where someone else claims that it is false. We even may have the same person making different statements concerning the truth of an event or his role in it. Given our assumption that we should not aim to identify what is true or not, but merely what different sources say is true or not, attribution is inherently related to statements. Modeling attribution on a mention level captures this intuition.

In the example sentences, two statements are being made about Chrysler selling 5,000 SUVs. One statement expresses that Chrysler expects to do so, the other that they will probably not. We thus have Chrysler having a certain expectation, which is stated by Dieter Zetsche. Manfred Bischoff makes the statement about the same sell event with other attribution values. Recall that the links between instances to their mentions are given in Figure 16. Figure 17 presents the triples that model attribution values for the example sentences.

```
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_17 .

:NGsell_r1 gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_17_r1 .

:NGsell_r2 gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_17_r2 .

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect
gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_16 .

:NGexpect_r1 gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_16_r1 .

:NGexpect_r2 gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_16_r2 .

dbp:resource/Chrysler gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_t_243 .

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say
gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_11 .

:NGsay_r1  gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_11_r1 .

:NGsay_r2 gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_11_r2 .

dbp:resource/Dieter_Zetsche gaf:denotedBy nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_t_252 .

nwr:madeupexampletext.xml#ev132Say gaf:denotedBy nwr:madeupexampletext.naf#mention_pr_132 .

:NGsay2_r1 gaf:denotedBy nwr:madeupexampletext.naf#mention_pr_132_r1 .

:NGsay2_r1 gaf:denotedBy nwr:madeupexampletext.naf#mention_pr_132_r2 .

dbp:resource/Manfred_Bischoff gaf:denotedBy nwr:madeupexampletext.naf#mention_t_2045 .

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
gaf:denotedBy nwr:madeupexampletext.naf#mention_pr_133 .

:NGsell_r1 gaf:denotedBy nwr:madeupexampletext.naf#mention_pr_133_r1 .

:NGsell_r2 gaf:denotedBy nwr:madeupexampletext.naf#mention_pr_133_r2 .
```

Figure 16: Links between instances and mentions basic examples

By placing statements that represent attribution coming from the same source, we can straightforwardly model who or what the source of the statement and its attribution is. The relations to the source are given in Figure 18.

An overview of how both attribution values and sources are modeled for our example sentences is given in Figure 19.

The examples and picture above illustrate that the model can capture alternative points of view regarding the attribution of the same event. It is also straightforward to extract all view points expressed by a specific source. When the scope of assigned attribution values is known, this can easily be modeled by assigning different attribution values to individual roles in an event. The case of underspecified scope as in the example *Sandy does not believe Kim killed Harry* can currently not be modeled. Recall that, without context, we do not know whether Sandy believes Harry was killed, but not by Kim, whether Sandy believes Kim killed someone else or whether Sandy believes Kim did not kill Harry, because he did

not end up dead, etc. In this case, we would ideally be able to model that according to Sandy something is probably not true, but we do not know what part of the statement this applies to exactly. We are currently working on a solution to handle such cases.

```
:NGChryslerAtt {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_17 nwr:hasAttribution nwrontology:attrPOSPROBF
.

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_17_r1
nwr:hasAttribution nwrontology:attrPOSPROBF .

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_17_r2
nwr:hasAttribution nwrontology:attrPOSPROBF .
}

:NGZetscheAtt {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_16 nwr:hasAttribution nwrontology:attrPOSCERTNF
.

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_16_r1
nwr:hasAttribution nwrontology:attrPOSCERTNF .

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_16_r2
nwr:hasAttribution nwrontology:attrPOSCERTNF .

:NGChryslerAtt nwr:hasAttribution nwrontology:attrPOSCERTNF .
}

:NGBischoffAtt {
nwr:madeupexampletext.naf#mention_pr_133 nwr:hasAttribution nwrontology:attrNEGPROPF .

nwr:madeupexampletext.naf#mention_pr_133_r1 nwr:hasAttribution nwrontology:attrNEGPROPF .

nwr:madeupexampletext.naf#mention_pr_133_r2 nwr:hasAttribution nwrontology:attrNEGPROPF .
}

:NGWAWAtt {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_11 nwr:hasAttribution nwrontology:attrPOSCERTNF
.

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_11_r1
nwr:hasAttribution nwrontology:attrPOSCERTNF .

nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.naf#mention_pr_11_r2
nwr:hasAttribution nwrontology:attrPOSCERTNF .

nwr:madeupexampletext.naf#mention_pr_132 nwr:hasAttribution nwrontology:attrPOSCERTNF .

nwr:madeupexampletext.naf#mention_pr_132_r1 nwr:hasAttribution nwrontology:attrPOSCERTNF .

nwr:madeupexampletext.naf#mention_pr_132_r2 nwr:hasAttribution nwrontology:attrPOSCERTNF .
}
```

Figure 17: Triples that model attribution

```
:NGZetsche prov:wasAttributedTo dbpedia:Dieter_Zetsche .
:NGChrysler prov:wasAttributedTo dbpedia:Chrysler .
:NGBischoff prov:wasAttributedTo dbpedia:Manfred_Bischoff .
:NGWAWAtt prov:wasAttributedTo dbpedia:Ward's_Auto_World .
```

Figure 18: Triples that model attribution

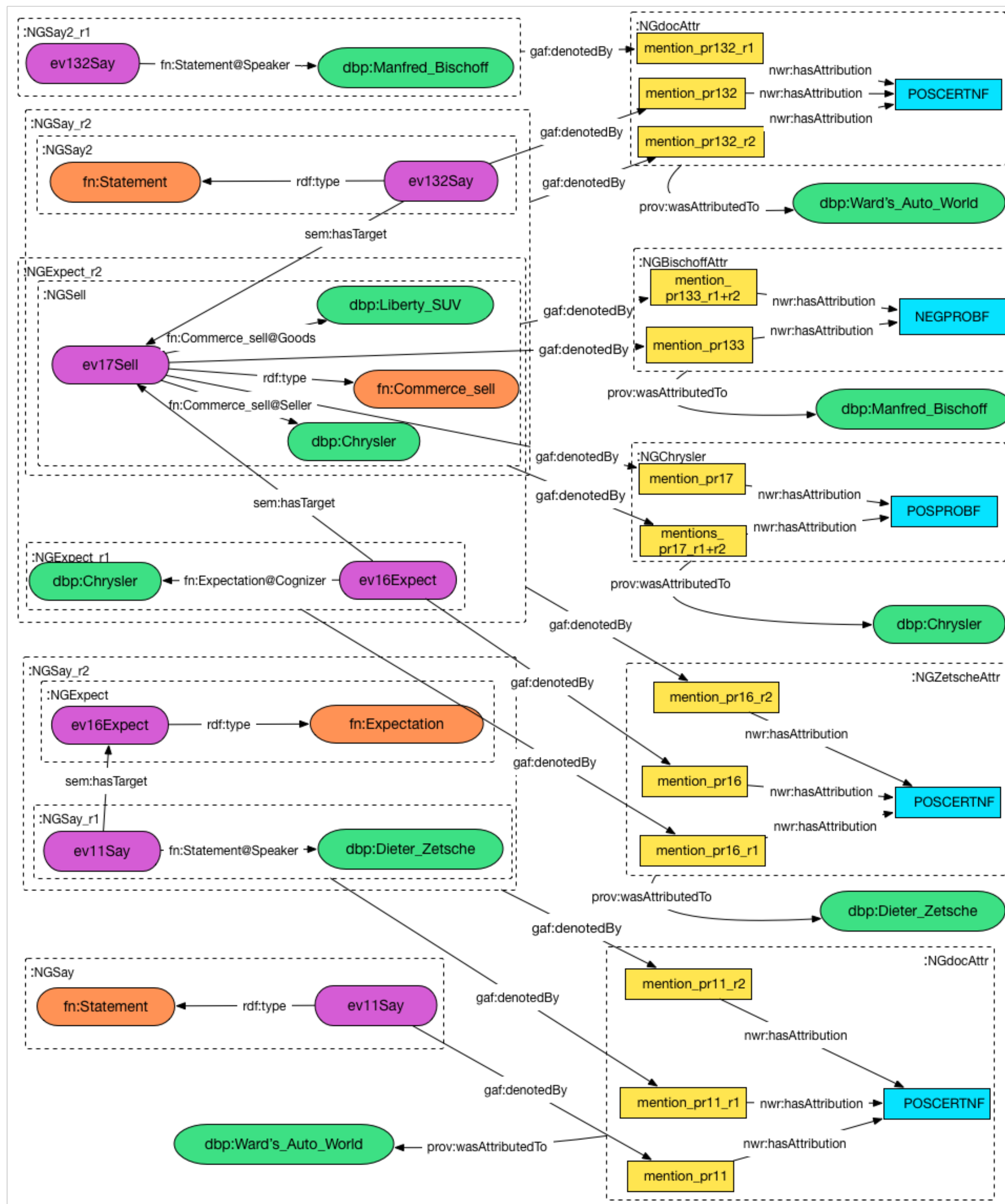


Figure 19: Illustration of different attribution values and perspectives applied by various sources

7 Conversion of NAF to SEM

The processing of text to RDF roughly consists of 3 phases:

1. Natural language processing to generate the NAF layers
2. Cross-document event coreference to convert NAF to RDF
3. Populate the KnowledgeStore with NAF and RDF

In this section, we describe the second phase after the NLP-NAF has been generated. We first describe the architecture of this phase and next the performance of processing 1.3M NAF files for the automotive industry.

7.1 Architecture for processing NAF to SEM

The NAF2SEM processing results in RDF-TRiG files that contain instances of events, entities and owl:DateTimeDescriptions as well as SEM relations between events, their participants, places, and time. SEM relations are stored as named graphs. The graph URIs are used to store factuality values and provenance relations for each SEM relation based on the GAF model. In Figure 20, an example is shown of the RDF-TRiG that is produced.

The second phase of the overall process is divided into two steps, shown in Figure 21.

1. Create binary object files with all SEM event related data in separate Time Description folders.
2. Compare all object files in the same Time Description folder to establish cross-document event instances and create a single RDF-TRiG file per folder with all the data.

In the first step, we first obtain all instances as SEM objects with unique URIs and all SEM relations between these objects from a NAF file, in Figure 21 NAF-1 and NAF-2. From these SEM objects and relations, we create so-called CompositeEvent objects for each event instance. A CompositeEvent object contains a single SEM event object and all the other SEM objects and SEM relations related to this SEM event. All event instances are bound to a time description, which can be a year, a month or day or a combination of disjunct times. For each NAF file, we create as many binary object files as there are different time descriptions associated with the events. The object files are stored in event Time Description folders that correspond with the associated time description. In Figure 21, the two NAF files result in object files in the folders e-2011-01 and 2-2011-01-03.

The Time Description Folders are grouped in separate folder making a distinction between the 3 main classes of events distinguished in NewsReader: source-introducing events, grammatical events and contextuals. We use a classification of FrameNet frames to distinguish these different types of events. Events that have no frame as an external

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix time:  <http://www.w3.org/TR/owl-time#> .
@prefix eso:   <http://www.newsreader-project.eu/domain-ontology#> .
@prefix gaf:   <http://groundedannotationframework.org/gaf#> .
@prefix nwrontology: <http://www.newsreader-project.eu/ontologies/> .
@prefix owl:   <http://www.w3.org/2002/07/owl#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sem:   <http://semanticweb.cs.vu.nl/2009/11/sem/> .
@prefix fn:    <http://www.newsreader-project.eu/ontologies/framenet/> .
@prefix nwrdata: <http://www.newsreader-project.eu/data/> .

<http://www.newsreader-project.eu/instances> {
    <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#ev20>
            a               sem:Event , fn:Placing , eso:Placing , nwrontology:SPEECH_COGNITIVE ;
            rdfs:label      "file" ;
            gaf:denotedBy  <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=138,143> ,
            <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=327,332> .
    <http://www.newsreader-project.eu/time/20111103>
            a               time:DateTimeDescription ;
            time:day        "---03"^^<http://www.w3.org/2001/XMLSchema#gDay> ;
            time:month      "--11"^^<http://www.w3.org/2001/XMLSchema#gMonth> ;
            time:unitType  time:unitDay ;
            time:year       "2011"^^<http://www.w3.org/2001/XMLSchema#gYear> .
    <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#tmx2>
            a                 time:Interval ;
            rdfs:label        "2011" ;
            gaf:denotedBy     <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=359,363> ,
            <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=364,365> ,
            <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=365,366> ,
             <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=367,371> ;
            time:inDateTime   <http://www.newsreader-project.eu/time/20111103> .
    <http://www.newsreader-project.eu/data/cars/entities/KEIPER_GmbH_\%26_Co_KG>
            a               nwrontology:ORGANIZATION ;
            rdfs:label      "KEIPER GmbH &amp; Co KG" ;
            gaf:denotedBy  <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=118,137> ,
            <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=307,326> .
}
<http://www.newsreader-project.eu/provenance> {
    <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#tr2>
            gaf:denotedBy  <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=93,143> .
    <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#pr6,rl11>
            gaf:denotedBy  <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=307,332> .
    <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#fv01>
            gaf:denotedBy  <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=138,143> ,
            <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#char=327,332> .
}
<http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#tr2> {
    <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#ev20>
            sem:hasTime  <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#tmx1> .
}
<http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#pr6,rl11> {
    <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#ev20>
            sem:hasActor             <http://www.newsreader-project.eu/data/cars/entities/KEIPER_GmbH_\%26_Co_KG> ;
            eso:translocation-theme  <http://www.newsreader-project.eu/data/cars/entities/KEIPER_GmbH_\%26_Co_KG> ;
            <http://www.newsreader-project.eu/ontologies/framenet/Placing@Agent>
                    <http://www.newsreader-project.eu/data/cars/entities/KEIPER_GmbH_\%26_Co_KG> ;
            <http://www.newsreader-project.eu/ontologies/propbank/A0>
                    <http://www.newsreader-project.eu/data/cars/entities/KEIPER_GmbH_\%26_Co_KG> .
}
<http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#fv01> {
    <http://www.newsreader-project.eu/data/cars/2013/01/29/57M5-WHY1-DXF1-N1JY.xml#ev20>
            <http://www.newsreader-project.eu/ontologies/value/hasFactBankValue>
                    "CT+" .
}
```

Figure 20: Example of the TRiG format for a single event from the car data set

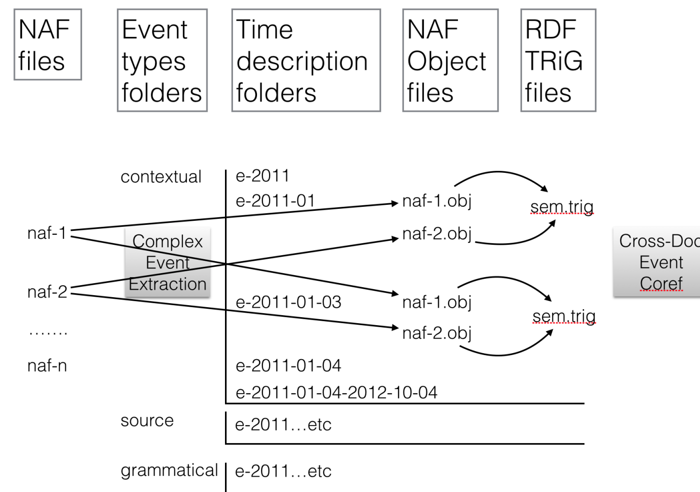


Figure 21: **Overview of the NAF to SEM process**

reference are considered to be *contextual* by default. Likewise, the result structure consists of 3 main folders: source, contextual and grammatical, with a range of Time Description subfolders and multiple object files within each Time Description Folder.

In the second step, we load all the object files from each subfolder and compare the events to determine cross-document coreference. The details of the algorithm to determine coreference are described in [Rospocher *et al.*, 2015]. If two events are corefential, all the information is merged. If not, it is kept separate. Finally, all the SEM objects, SEM relations and GAF relations are serialized to a single RDF-TRiG file per folder.

Since the first step takes NAF streams as input, it can be included into the regular processing pipeline of NewsReader. This process can be parallelized because each NAF file will generate different unique object files. Parallel processing needs to share the Time Description folders or the folders need to be merged in a separate step. The second step needs to process all the objects files in the same folder and can only be parallelized per folder. For processing the car data set, we used parallel processes for both steps.

In the 3rd year of the project, we will update this step by direct interaction with the KnowledgeStore. In that case, we do not create an object file but we keep the composite events of a NAF file or stream in memory. The second step of the processing then consists of querying the KnowledgeStore for events that potentially describes the same event as the target CompositeEvent does. The results of the SPARQL request will be represented as CompositeEvents in memory in the same way as they are now read from the object files. Comparing the new target event with the given events then can result in either a match or not. If there is a match, the new information is merged with the given information. If there is no match, the event is saved into the KnowledgeStore as a new event.

The new architecture is shown in Figure 22. In this design, we also include a further division in topics that can come from a topic detection module. This means that each event

is not only classified by the 3 main event types in NewsReader but also for different topic labels. In Figure 22, we show a subdivision of contextuals in *takeover* events. The CompositeEvent extraction generates a candidate event that is labeled as *contextual-takeover* and bound to the time description *2011-01-03*. A SPARQL request to the KnowledgeStore results in two given events that have the same topic and the same time description. The Crossdocument Event Coreference function now compares the new event with the given events to determine identity. If identical, the new information is merged with one of the given events and the NAF mentions for the event are updated. If not identical, it is considered as a new *takeover* event and added as such to the KnowledgeStore.

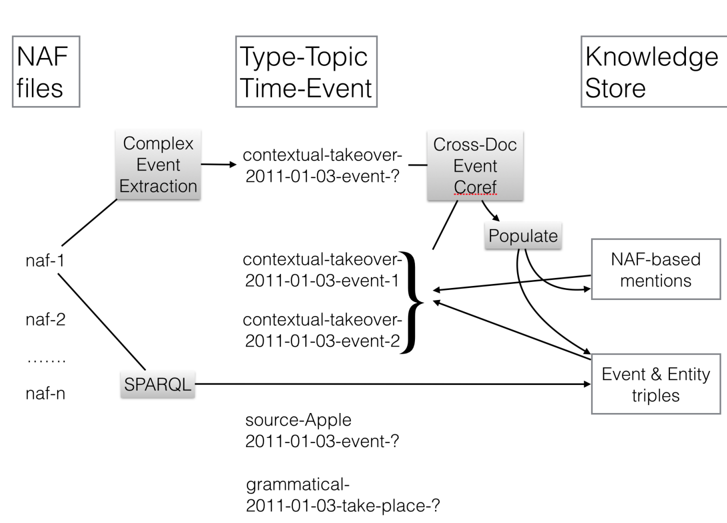


Figure 22: **Overview of the NAF to SEM process with direct integration with the KnowledgeStore**

The new architecture does not require any storage of intermediate result and can be integrated directly into the NewsReader pipeline. Furthermore, we can fine-tune the SPARQL requests to the KnowledgeStore to select the given events for comparison. This can be events for the same topic, same time description and involving the same participants.

7.2 Performance of the NAF to SEM processing on the automotive data set

We obtained 1.3 M news-articles from Lexis Nexis. Each text has been packed in "raw" NAF and annotated with the standard english pipeline. To this end a Hadoop set-up has been made on the "DAS" supercomputer of the Computer-science faculty. The pipeline produced for each news-article a NAF-encoded file. In (gzip) compressed form the processed NAF files take up 95 GB of disk space. In Figure 23, you can see how the 1.3M NAF files are distributed over publication date. The volume is evenly distributed except for the year 2012, which has twice the volume of news compared to the other years.

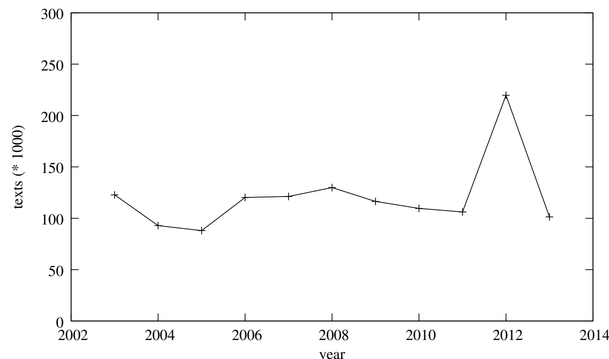


Figure 23: Number of news-texts as a function of year

As explained in the previous section, the final NAF files are analyzed to obtain the CompositeEvent objects for each single NAF file. These objects are stored in binary files in separate folders, the so-called Time Description Folders, based on the time-stamp associated to the event. All the events from a single NAF file are then divided over different object files, each corresponding to different time-stamps. Different NAF files get different unique object files based on the name of the NAF file but they are stored in the same Time Description Folder for matching time-stamps. Hence, a NAF-file may produce multiple object-files and a Time Description folder may contain object-files from multiple NAF-files that share the same time-stamp.

The ClusterEventObjects Java program produced 10.4 M "object" files in 1.03 M folders (taking up 226 GB of disk-space).

To produce the "obj" files, a set of 15 VM's had been set up. On each VM four instances of the Java program ran as parallel processes. The NAF files were distributed over the 15 VM's and the folders with the produced obj files were collected and merged in a single directory structure so that the object files with the same time-description folder across the different VMs are merged into a single time-description folder. This step took about seven days due to the intense moving of files.

The MatchCompositeEvent Java program then reads all objects files from a Time Description Folder, compares the CompositeEvents and creates a single RDF-TRiG file per folder. To this end the merged folder-structure was written on a physical (spindle) hard-disk with capacity of 1 GB. Because it turned out that disk-access was the most time-consuming part of the process, only a single process performed the merge. It took 64 hours to process the 4.5 M "contextual" object files.

The next table shows the overview statistics for the processing as an indication of the infrastructure load. From the processing, we learned that the HPC cluster was not optimized for handling many different files: copying, merging, compressing, decompressing, etc. Especially in step 1, the process was delayed because of the file handling, such as merging the Time Description Folders created by the 15 VMs. A lot of processing time can still be gained by using a shared file system across the different VMs.

Input data	
NAF number	1.3M.
NAF diskspace compressed	95GB
Output data	
OBJ number	10.4 M (4.5M. contextual; 2.5M. Grammatical; 3.4M. source)
OBJ diskspace	227GB
RDF-TRiG number	1.03M. (510K. contextual; 194K. grammatical; 328K. source)
RDF-TRiG diskpace compressed	5.44GB (4.4GB contextual; 744MB grammatical; 296MB source)
Step 1 ClusterCompositeEvents	
Parallel	15 VM's; four parallel processes each
CPU	1 CPU core per process
Total time	app. zeven days (CPU-time not counted)
Step 2 CompareCompositeEvents	
Parallel	1
CPU/process	1
Total time	60 hours for contextual

8 The KnowledgeStore

This section describes the activity of porting the KnowledgeStore on the SURFsara HPC-Cloud infrastructure. SURFsara[12] is a Dutch foundation that provides supercomputers, collocation, networks and high-end visualisation to academic institutions. This activity has been conducted in the context of the Enlighten Your Research (EYR4[13]) program.

The motivations underlying the porting of the KnowledgeStore in the SURFsara infrastructure are multiple: first, we intended to test the portability of the KnowledgeStore application outside the FBK infrastructure on which it was developed. Second, installing the KnowledgeStore on the SURFsara facilities has been a first important step with respect to enhancing its scalability. Third, we aimed at check the usability of the ported KnowledgeStore for one of the actual NewsReader use-cases.

8.1 Setup and Configuration

The KnowledgeStore is built on top of Java jdk1.7.0_51 and uses the Java packages Hadoop 1.0.4, HBase 0.94.10, Zookeeper 3.4.5, Bookkeeper 4.2.2 and Omid (FBK development over version 0c24b16). In addition it utilizes the Virtuoso Open-Source Edition 7.0.1.

The KnowledgeStore is a distributed application. The system ported on SURFsara HPC-Cloud utilizes a set of five virtual machines (VMs) configured with the Linux CentOS release 6.5 (Final) OS. The VM1 hosts the HBase, Hadoop and Zookeeper masters, as well as the Omid transaction server. VMs from 2 to 4 host the HBase, Hadoop and Zookeeper slaves. The VM5 hosts the Virtuoso triple store and the KnowledgeStore front-end (also known as the KnowledgeStore server).

The KnowledgeStore installation package is a tarball file including a portion of the file system that is self-contained: the tarball comprises all the required software (including java) as well as additional directories to store the data that will be created by the packages at run time. It is worth noticing here that the software can run without root permissions. The tarball includes a script to configure automatically the 5 VMs once decided their role.

8.2 Experiments

The KnowledgeStore ported on SURFsara HPC-Cloud has been successfully populated with a small test set of 20 NAFs from the World Cup domain utilized for the Hackathon held on June, 2014.

In term of performance, preliminary experiments on both the population and lookup functionalities indicate a increase of the time spent of a factor around 10 with respect to a similar setup running on the FBK infrastructure.

The reason of such degradation of the performance is related to the features of the infrastructure at SURFsara HPC-Cloud cluster. It is based on a centralized File System: all the machines in such cluster use the same File System through NFS; machines have

[12]https://www.surfsara.nl/
[13]https://www.enlightenyourresearch.net

no local File Systems. This is a choice underlying the HPC-Cloud cluster whose target is to maximize the computation power and not the I/O throughput. On the other hands the KnowledgeStore is structurally based on a distributed architecture (Hadoop and HBase are natively distributed in order to achieve scalability and fault tolerance). What happens with the KnowledgeStore ported on SURFsara HPC-Cloud is that all its daemons running on the five VMs utilize the same centralized File System via NFS. In addition, also all the machines in the HPC-Cloud cluster utilize the same File System. Therefore it is highly probable that the File System becomes overloaded and the bottleneck of the application.

The port of the KnowledgeStore on SURFsara HPC-cloud has been a very interesting activity to develop such application on a typical cloud infrastructure, showing the KnowledgeStore portability and scalability. We plan to do a further step in porting the KnowledgeStore on another SURFsara infrastructure, the *Hadoop* cluster: since it offers Hadoop and HBase as native packages, it appears to be much better suited for the intrinsic distributional nature of the KnowledgeStore.

9 Conclusion and future work

This deliverable describes the second version of the **System Design** architecture developed in NewsReader to process large and continuous streams of English, Dutch, Spanish and Italian news articles.

We described the work carried out to accurately measure the spent time of each NLP module, including the times spent on each stage of the processing (initialization, conversion from NAF to the native format, etc). This is a necessary step towards the optimization of each NLP module, which helps us to focus on the critical aspects of each NLP module.

We have developed two alternative architectures for NLP distributed processing, for a batch and streaming computing scenarios, respectively. In the deliverable we describe each of the solutions, detailing the main components of each one. We also present a set of scripts developed within the project that quickly create a basic cluster of nodes for distributed processing. All the software, including the NLP modules, as well as the basic components of the clusters is publicly available and is distributed under open licenses.

The deliverable also describe the latest additions to the Grounded Annotation Framework (GAF) that is used within the project to accurately represent the events and participants extracted from the texts. In particular, we report the representation of provenance information within GAF, which allows comparing perspectives from different sources about some particular event or fact.

The result of the processing pipeline is a set of documents annotated using the NLP Annotation Format (NAF), developed within the project. These NAF documents are then converted to GAF representations, removing duplicates and harmonizing the information that come from different documents. In the deliverable we describe how this process is actually performed.

We also describe the first steps towards the integration of the KnowledgeStore into a HPC cloud infrastructure, an important aspect necessary to enhance the scalability of the

repository.

In the future we want to further optimize the pipeline by explore the use of different granularities, that is, exploiting the fact that NLP modules work at different type of granularity. For instance, a POS tagger works at a sentence level, the WSD module works at paragraph level, whereas a coreference module works at a document level. We want to experiment splitting the input document into pieces of the required granularity, so that the NLP modules can quickly analyze those pieces. In fact, the experiments carried out with non-linear topologies in Section 4.4 suggests that there is an upper bound in the performance gain when performing document-level processing. The reason is that some modules spend a large amount of time processing the document, thus representing a bottleneck in the processing. Therefore, we could expect a significant performance boost if we were able to run many instances in parallel of those processors at a finer level of granularity, i.e., sentences or paragraphs.

One important aspect is, however, the times spent by the modules in the initialization phase. The experiments of Section 2 clearly suggest that some modules take too much time initializing, and that much time is wasted by performing a full initialization each time the module is called. If we want to experiment with different granularities (which would require calling the module more times, thus suffering the initialization penalty more times), we have to seriously consider converting some modules architectures into a server/client model.

This second year we have mainly focused on the parallelization of NLP processing. Although the work on WP2 was meant to finish on this second year, we consider that some additional effort has to be made within WP2 in order to create a system where all the components are integrated. This would include the integration with the cross document processing modules developed on WP5, including integrated access to the KnowledgeStore to retrieve all the required information, as well as the integration to the Decision Support

A Processing step per phase for individual modules

This section provides an overview of the experiments carried out on individual modules. The results of the average time spent by a module are based on these outcomes. An overview is given in the Figures below.

EHU-tok

Repo: https://github.com/NLeSC/ixa-pipe-tok/tree/instrumented

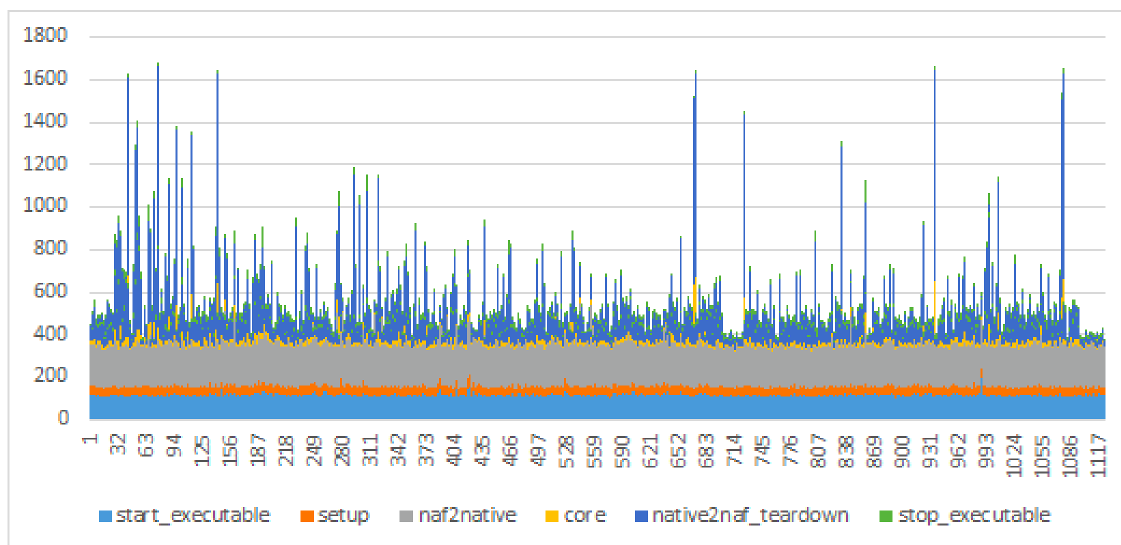


Figure 24: Ixa-pipe tokenizer processing time analysis

EHU-pos

Repo: https://github.com/NLeSC/ixa-pipe-pos/tree/instrumented

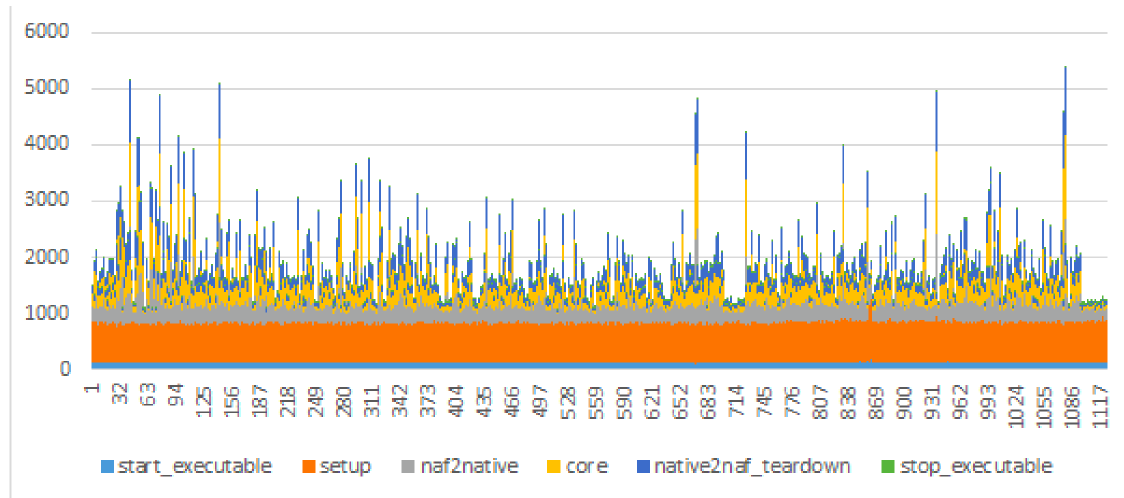


Figure 25: Ixa-pipe POS tagger processing time analysis

VUA-multiwordtagger

Repo: https://github.com/NLeSC/MultiWordTagger/tree/instrumented

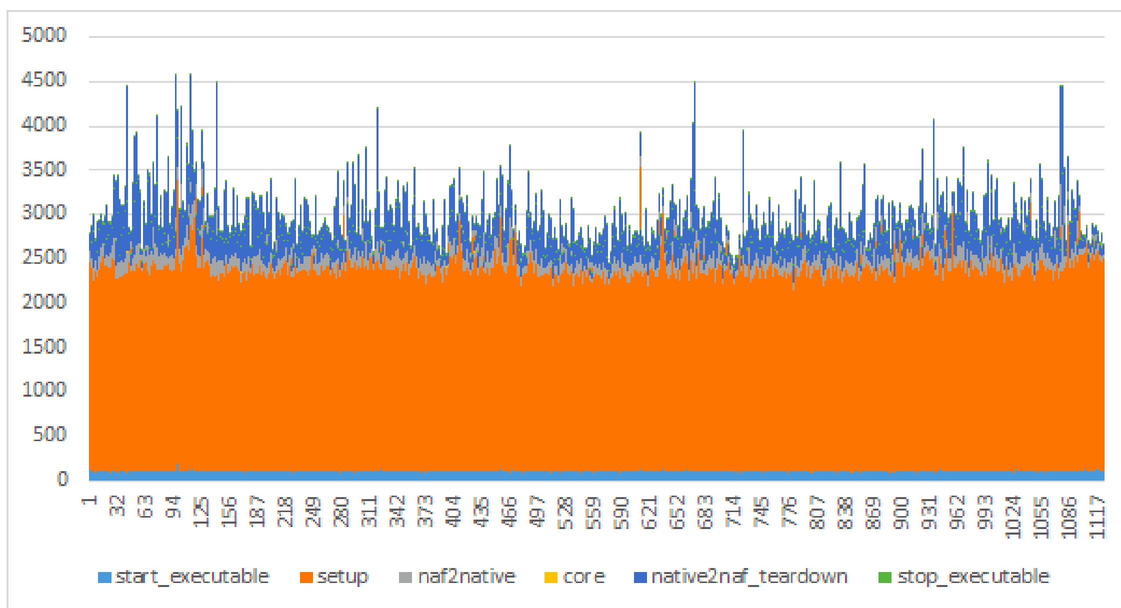


Figure 26: VUA multiword-tagger processing time analysis

EHU-nerc

Repo: https://github.com/NLeSC/ixa-pipe-nerc/tree/instrumented-1.0.0

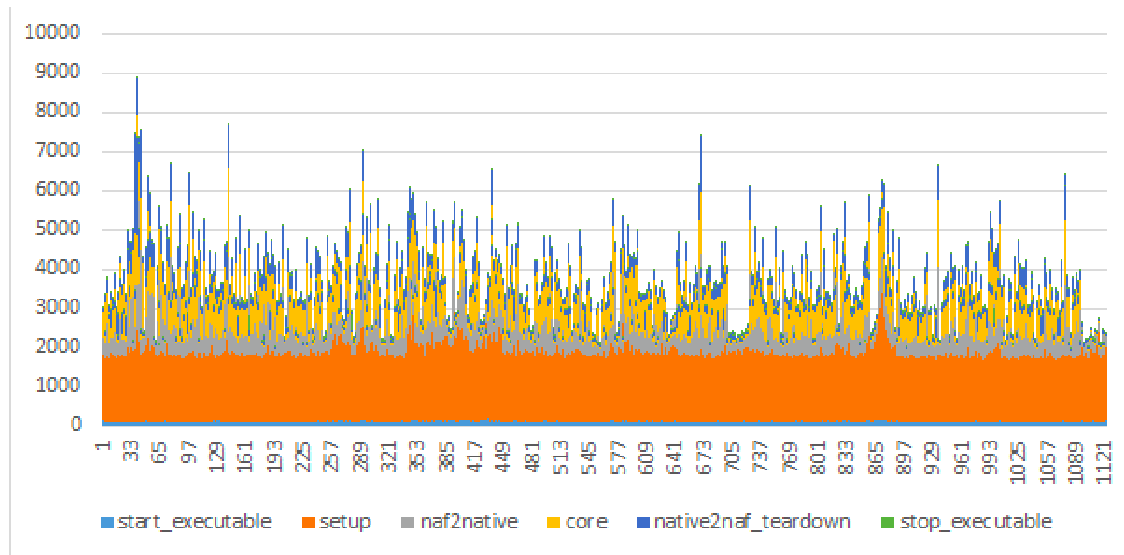
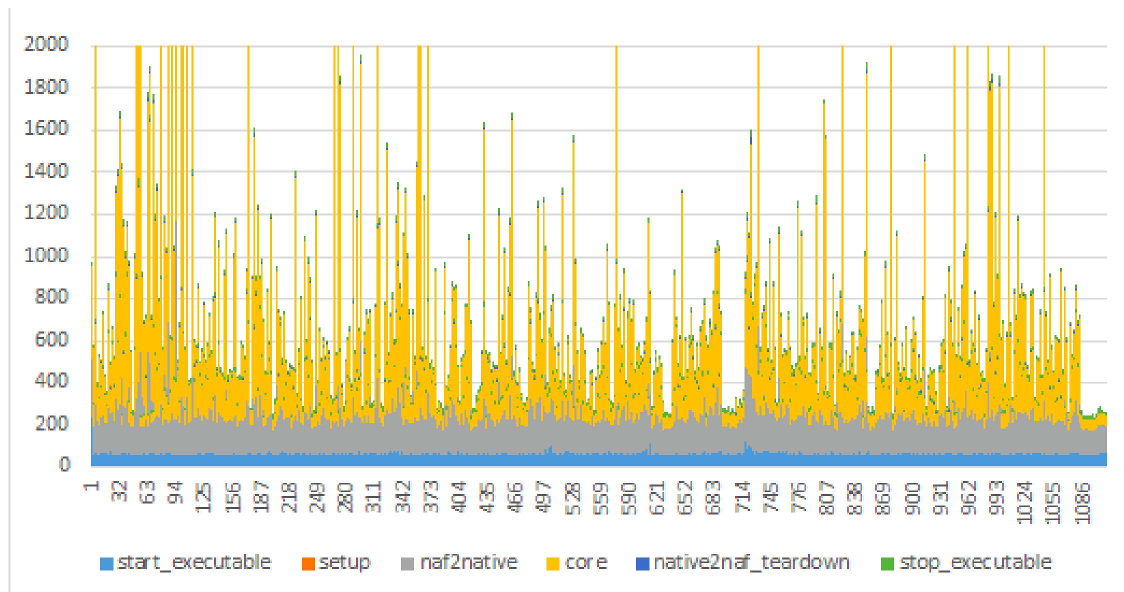


Figure 27: Ixa-pipe NERC tagger processing time analysis

VUA-opinion-miner

Repo: https://github.com/NLeSC/opinion_miner_deluxe/tree/instrumented



`Y axis clipped to 2000ms.`

`A timeout of 60 secounds was used.`

Figure 28: VUA opinion miner processing time analysis

VUA-svm-wsd

Repo: https://github.com/NLeSC/svm_wsd/tree/instrumented

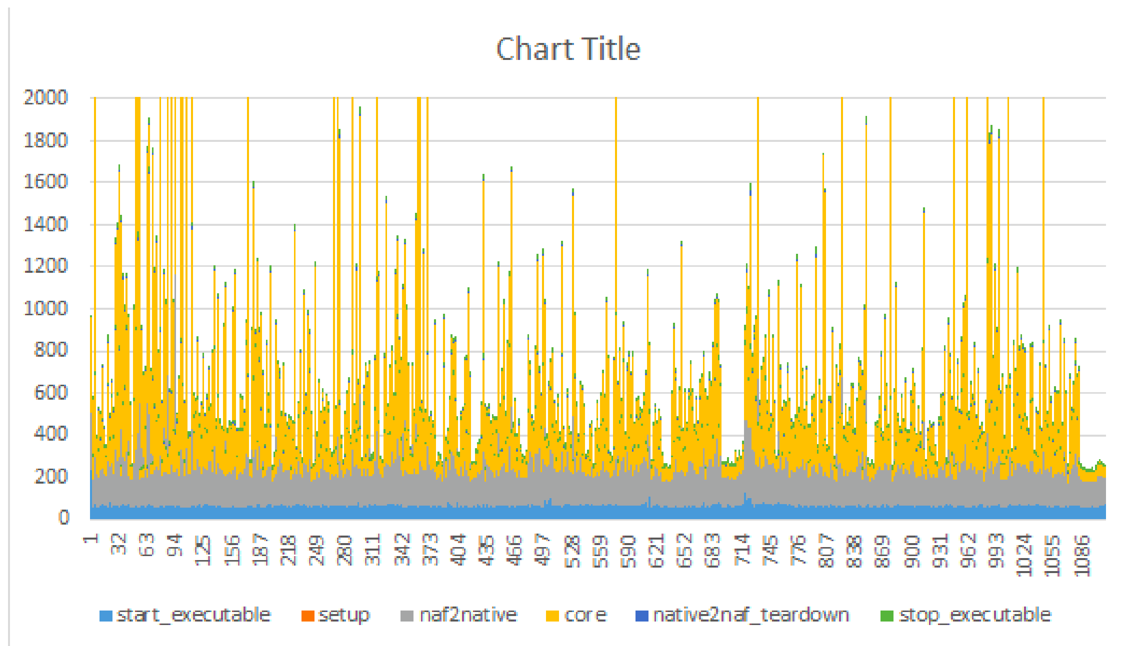
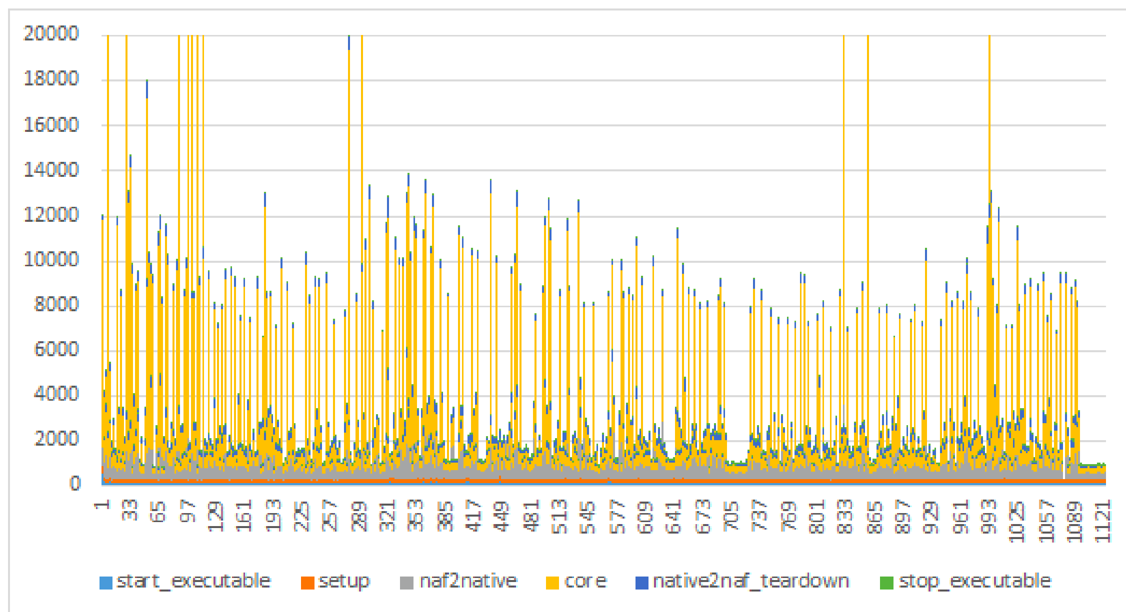


Figure 29: VUA svm-wsd processing time analysis

EHU-ned

Repo: https://github.com/NLeSC/ixa-pipe-ned/tree/instrumented-spot06



Y axis clipped to 20000 ms

Running in 4G VM with DBpedia spotlight using 4G, so some swapping was occurring.

Figure 30: Spotlight-based NED processing time analysis

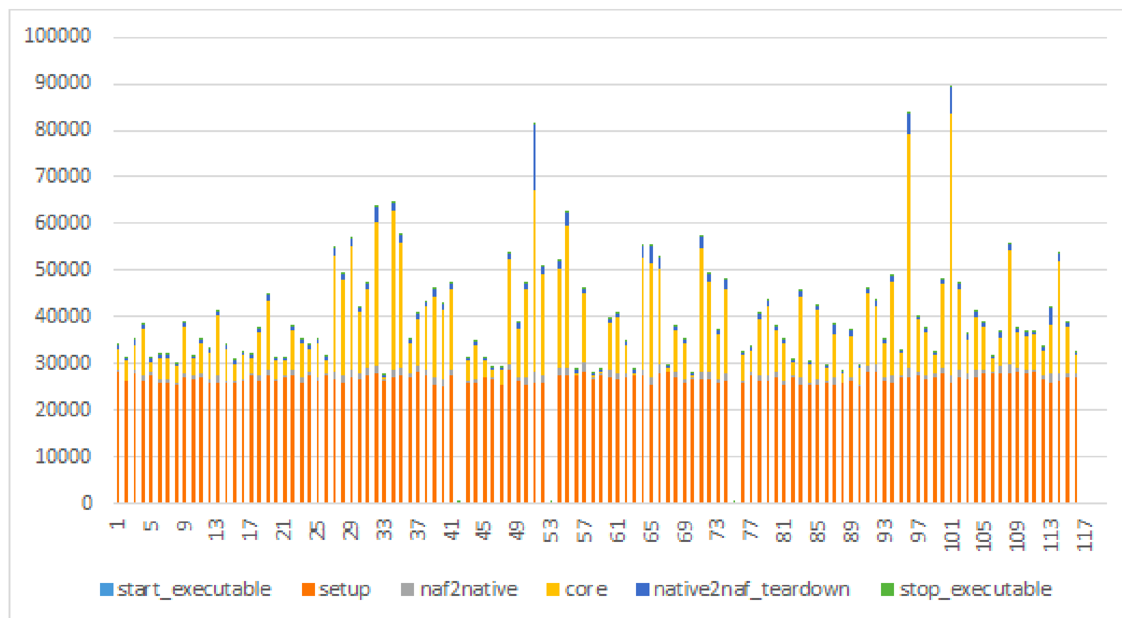


Figure 31: MATE tool-based SRL and dependency parsing processing time analysis

FBK-time

Repo: https://github.com/NLeSC/newsreader-component-wrappers/tree/instrumented/FBK-time

Each step is done in with an executable and was timed.

1. naf2native, convert naf to native txp format
2. Yamcha, http://chasen.org/~taku/software/yamcha/ runs with model
3. Timepro, part of http://textpro.fbk.eu/
4. Native2naf, convert native 2 naf format

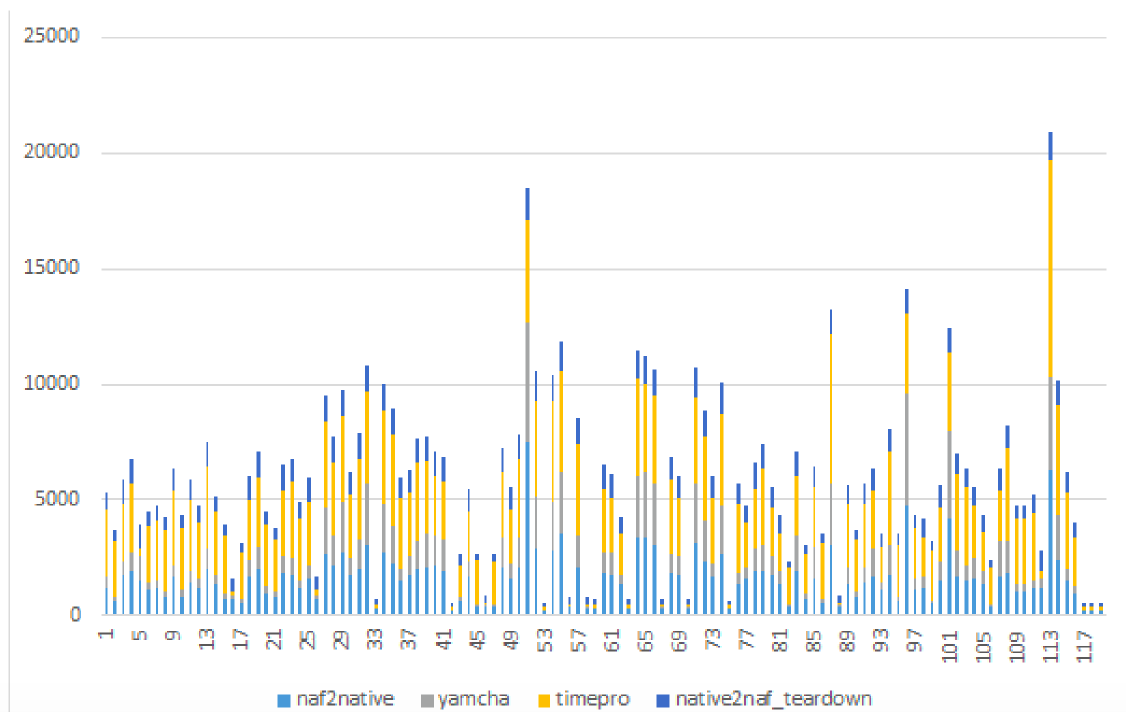


Figure 32: TimePro processing time analysis

VUA-eventcoref

Repo: https://github.com/NLeSC/EventCoreference/tree/instrumented

Added missing deps to pom.xml so it would compile.

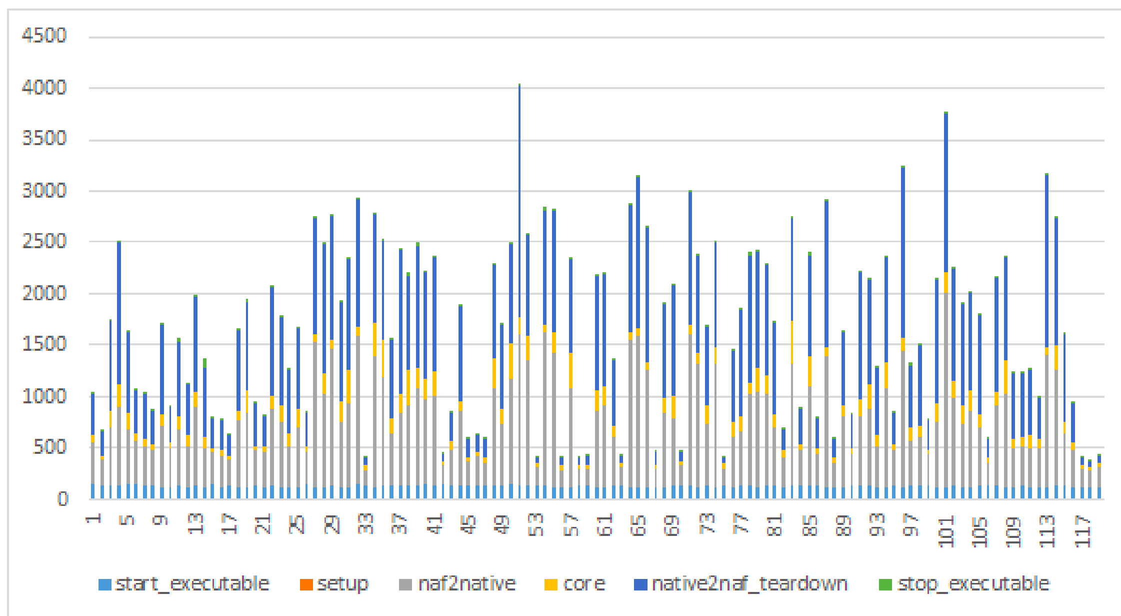


Figure 33: VUA event coreference processing time analysis

VUA-factuality

Repo: https://github.com/NLeSC/newsreader-component-wrappers/tree/instrumented/VUA-factuality

Updated perl to 5.10.1 as it gave segmentation fault.

Run.sh does several steps, these where timed:

1. Start_executable, writes file with current timestamp
2. Naf2native
3. core, mallet csv2classify
4. Sort, sortmalletoutput.pl
5. native2naf + removes temporary files

The classify step is a Java application from mallet (http://mallet.cs.umass.edu/), the rest are perl scripts.

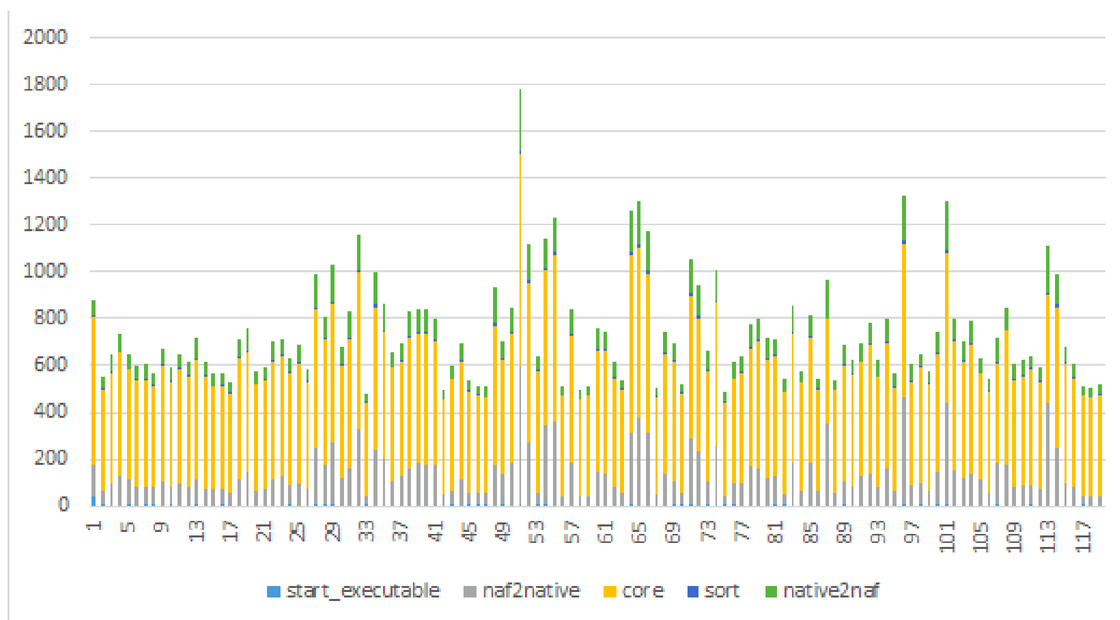


Figure 34: VUA-factuality processing time analysis

B Alternative proposals for modeling attribution

This section describes the first proposals we considered for modeling attribution, which model this on an instance level. If we relate attribution to instances, we can only treat scope correctly if we assign attribution values to triples rather than events. For instance, the sentence *Chrysler sells Liberty SUVs*, the attribution values apply to the following triples:[14]

```
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
            a              sem:Event , fn:Commerce_sell , eso:Selling .
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
            eso:possession-owner_1   dbp:resource/Chrysler ;
            fn:Commerce_sell@Seller dbp:resource/Chrysler ;
            fn:Commerce_sell@Goods nwr:data/cars/entities/Liberty_SUVs .
```

rather than just the event *sell*. This applies to all proposals that model attribution at the instance level.

We will use the same example sentences as in Section 6 to illustrate the proposals outlined below. For convenience, they are repeated below:

```
:NGsell {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
            a              sem:Event, fn:Commerce_sell , eso:Selling ;
            fn:Commerce_sell@Seller dbp:resource/Chrysler;
            fn:Commerce_sell@Goods nwr:data/cars/entities/Liberty_SUVs .
}

:NGexpect {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect
  a  sem:Event, fn:Expectation ;
           fn:Expectation@Cognizer  dbp:resource/Chrysler ;
   sem:hasTarget :NGsell .
}

:NGsay {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say
   a fn:Statement ;
            fn:Statement@Speaker          dbp:resource/Dieter_Zetsche ;
            sem:hasPlace  nwr:/data/cars/entities/DaimlerChrysler_Innovation_Symposium ;
    sem:hasTarget :NGexpect .
}

:NGsymposium {
nwr:/data/cars/entities/DaimlerChrysler_Innovation_Symposium
   sem:hasPlace dbp:resource/New_York_City .
}
```

Figure 35: Triples modelling basic example sentences

(7) Chrysler expects to sell 5,000 diesel Liberty SUVs, President Dieter Zetsche says at a DaimlerChrysler Innovation Symposium in New York.

[14]For reasons of space and because the examples are mainly meant to illustrate relations, we will not define the namespaces in these examples.

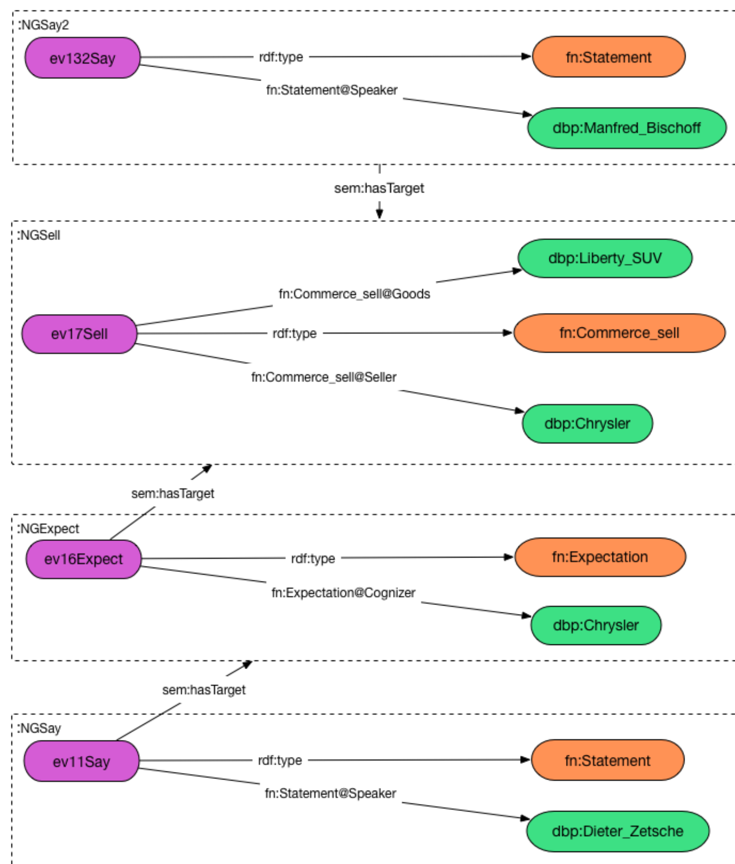


Figure 36: Illustration of triples modelling basic example

(8) Manfred Bischoff said that Chrysler will probably not sell 5,000 diesel Liberty SUVs.

Figure 35 provides the simplified RDF representation including all relations between events and entities of the sentence in (7) after a correct analysis.

The additional (made-up) sentence in (8) adds the following triples (note that Bischoff makes a statement about the same selling event):

```
:NGsay2 {
nwr:madeupexampe#ev132Say
   a fn:Statement ;
            fn:Statement@Speaker          dbp:resource/Manfred_Bischoff ;
   sem:hasTarget :NGsell .
}
```

Figure 36 provides an illustration of the triples of the original example expressions and the additional made up example.

B.1 Proposal 1: Attribution applies to the statements

The first proposal assumes that attribution applies to statements. The attribution values are linked to the named graph that includes the statements they apply to. The triples below provide the attribution values for the basic structure presented in Figure 35. Figure 38 illustrates how these triples are integrated in the representation of the two example sentences.

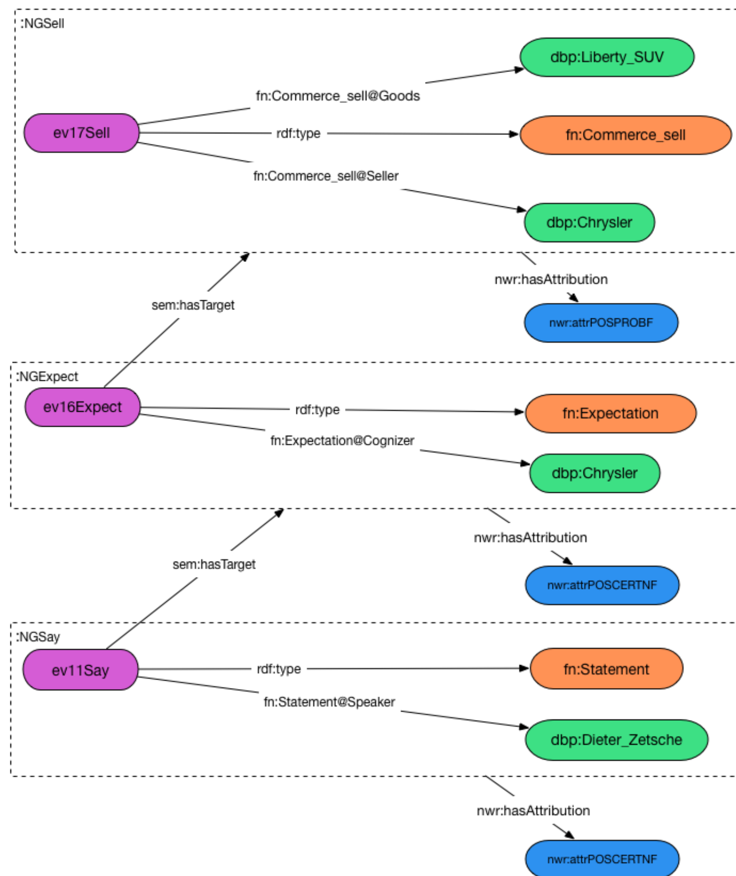


Figure 37: Illustration of triples modelling basic example, proposal 1

```
:NGsell nwrontology:hasAttribution nwrontology:attrPOSPROBF .
:NGexpect nwrontology:hasAttribution nwrontology:attrPOSCERTNF .
:NGsay nwrontology:hasAttribution nwrontology:attrPOSCERTNF .
```

Consider the additional example sentence that expresses an alternative opinion on Chrysler selling 5,000 SUVs. If we use the same named graph to express the target of the expectation event in :NGexpect and the saying event in :NGsay2 as is done in Figure 36, we cannot distinguish which attribution values are assigned by whom. Each statement that assigns alternative attribution values to an existing statement therefore needs to have its own target. Figure 39 illustrates what this looks like (the statement by Dieter Zetsche is omitted for reasons of space) and Figure 38 provides a graphic illustration.

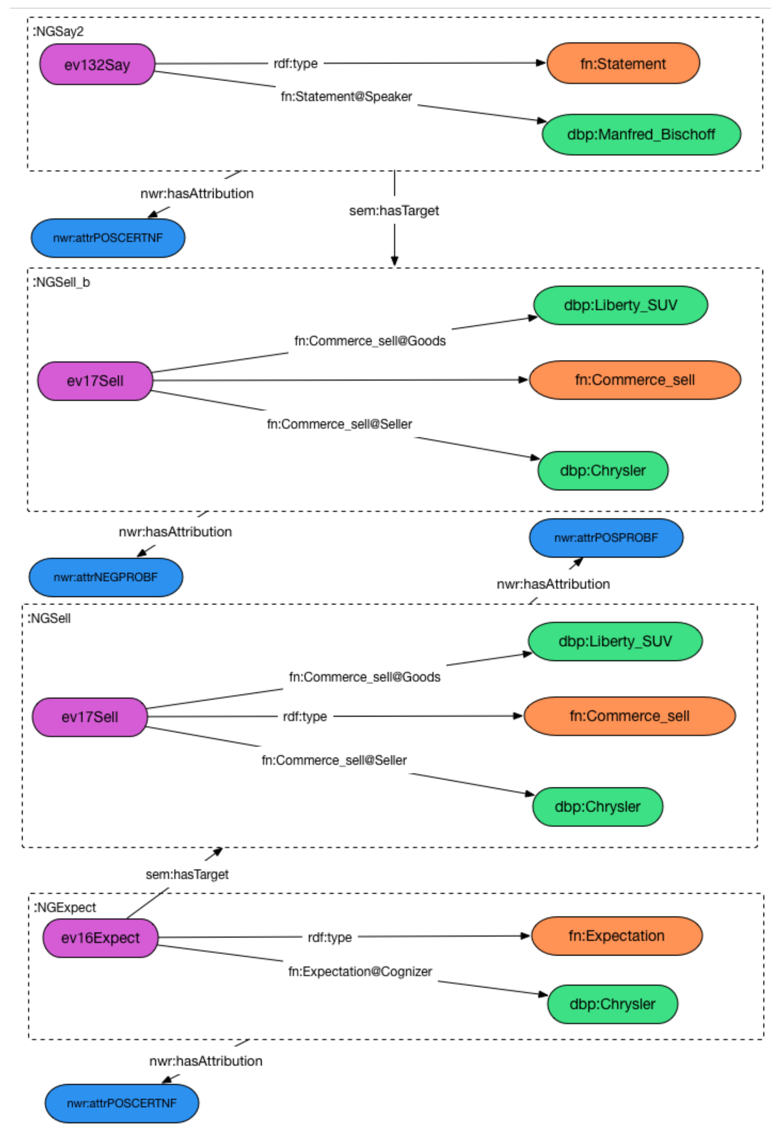


Figure 38: Alternative views proposal 1

```
:NGsell {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
            a              sem:Event, fn:Commerce_sell , eso:Selling ;
            fn:Commerce_sell@Seller dbp:resource/Chrysler;
            fn:Commerce_sell@Goods nwr:data/cars/entities/Liberty_SUVs .
}

:NGsell_b {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
            a              sem:Event, fn:Commerce_sell , eso:Selling ;
            fn:Commerce_sell@Seller dbp:resource/Chrysler;
            fn:Commerce_sell@Goods nwr:data/cars/entities/Liberty_SUVs .
}


:NGsay2 {
nwr:madeupexampe#ev132Say
   a fn:Statement ;
           fn:Statement@Speaker          dbp:resource/Manfred_Bischoff ;
   sem:hasTarget :NGsell_b .
}

:NGexpect {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect
  a  sem:Event, fn:Expectation ;
           fn:Expectation@Cognizer  dbp:resource/Chrysler ;
   sem:hasTarget :NGsell .
}


:NGsell nwrontology:hasAttribution nwrontology:attrPOSPROBF .
:NGsell_b nwrontology:hasAttribution nwrontology:attrNEGPROBF .
:NGexpect nwrontology:hasAttribution nwrontology:attrPOSCERTNF .
:NGsay2 nwrontology:hasAttribution nwrontology:attrPOSCERTNF .
```

Figure 39: Triples modelling alternative perspectives under proposal 1

```
:NGsell {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
            a              sem:Event, fn:Commerce_sell , eso:Selling ;
            fn:Commerce_sell@Seller dbp:resource/Chrysler;
            fn:Commerce_sell@Goods nwr:data/cars/entities/Liberty_SUVs .
}

:NGexpect {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect
  a  sem:Event, fn:Expectation ;
          fn:Expectation@Cognizer  dbp:resource/Chrysler .
}

:NGexpectTarget {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect
   sem:hasTarget :NGsell .
}

:NGsay {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say
   a fn:Statement ;
            fn:Statement@Speaker          dbp:resource/Dieter_Zetsche ;
            sem:hasPlace  nwr:/data/cars/entities/DaimlerChrysler_Innovation_Symposium .
}

:NGsayTarget {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say
   sem:hasTarget :NGexpect .

}

:NGexpectTarget nwr:hasAttribution nwr:attrPOSPROBF .
:NGsayTarget nwr:hasAttribution nwr:attrPOSCERTNF .
```

Figure 40: Attribution modelling on example with Proposal 2

B.2 Proposal 2: Attribution applies to the relation between source and statement

This proposal reflects the idea that attribution values really say something about the relation between the source and the statement attribution applies to. It captures this idea by linking attribution values to the triple that links the speech or cognitive verb and the statement. In the example, attribution values are assigned to the relation between *expect* and Chrysler selling SUVs and, respectively, *say* and Chrysler's expecting to sell SUVs.

Attribution values of statements that are made directly by the author are modelled on the relation between these statements and the metadata (this part is left out of the picture for reasons of space and clarity).

Just like for the first proposal, we consider the additional expression that assigns different attribution values to Chrysler selling SUVs. In this case, the statement about selling cars need not be repeated: a new *saying* event that has the selling event as its target is added and the alternative attribution values are linked to this relation as shown in Figure 41. Figure 42 provides a graphic illustration.

```
:NGsell {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev17Sell
            a              sem:Event, fn:Commerce_sell , eso:Selling ;
            fn:Commerce_sell@Seller dbp:resource/Chrysler;
            fn:Commerce_sell@Goods nwr:data/cars/entities/Liberty_SUVs .
}

:NGexpect {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect
  a  sem:Event, fn:Expectation ;
           fn:Expectation@Cognizer  dbp:resource/Chrysler .
}

:NGexpectTarget {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect
   sem:hasTarget :NGsell .
}

:NGsay2 {
nwr:madeupexampe#ev132Say
   a fn:Statement ;
            fn:Statement@Speaker          dbp:resource/Manfred_Bischoff .
}

:NGsay2Target {
nwr:madeupexampe#ev132Say
   sem:hasTarget :NGsell .
}

:NGsay {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say
   a fn:Statement ;
            fn:Statement@Speaker          dbp:resource/Dieter_Zetsche ;
            sem:hasPlace  nwr:/data/cars/entities/DaimlerChrysler_Innovation_Symposium .
}

:NGsayTarget {
nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say
   sem:hasTarget :NGexpect .

}

:NGexpectTarget nwr:hasAttribution nwr:attrPOSPROBF .
:NGsay2Target nwr:hasAttribution nwr:attrNEGPROBF .
:NGsayTarget nwr:hasAttribution nwr:attrPOSCERTNF .
```

Figure 41: Attribution modelling on example with Proposal 2, alternative views

B.3 Proposal 3: Attribution is assigned by reifying relations

The final way we may model attribution at an instance level is by using reification. Reification can generally be used instead of named graphs to model a property of a triple. Generally speaking, named graphs are preferable when statements are made about groups of triples. When statements are mainly made about individual triples, reification becomes an interesting alternative. The second proposal has the disadvantage that the triple indicating the `sem:hasTarget` relation needs to be placed in a named graph only to be able to make statements about this triple. As a result, we may end up with a model where the ac-

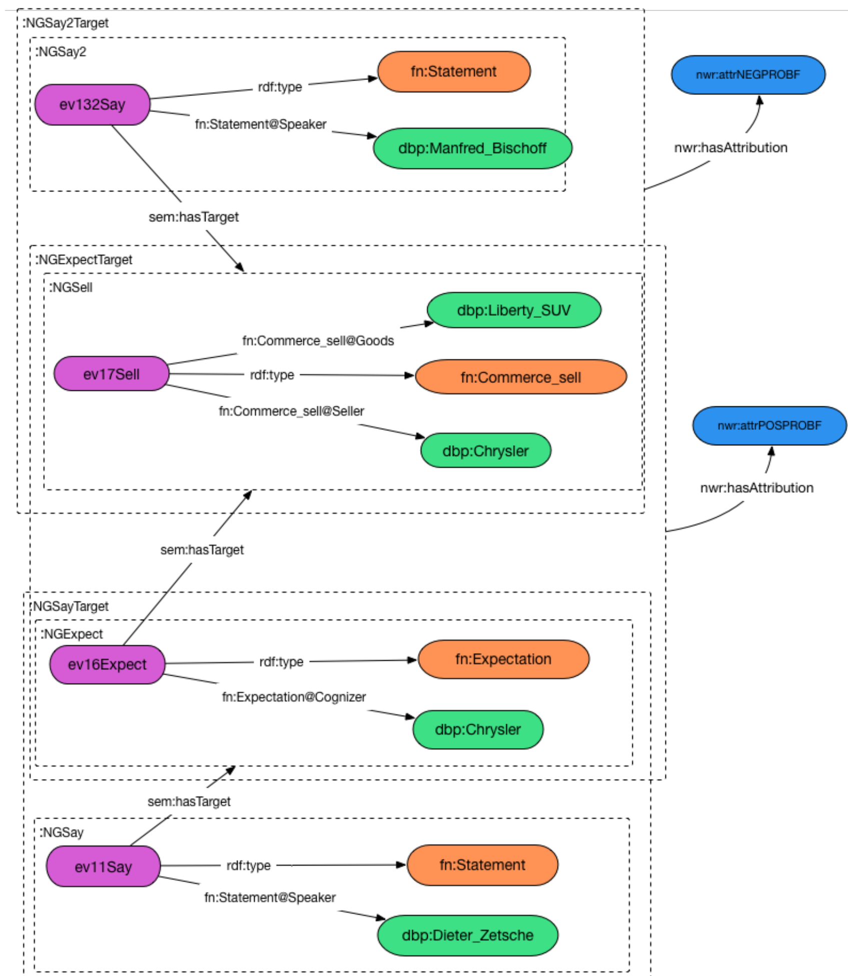


Figure 42: Alternative views proposal 2

tual triples are embedded in several layers requiring complex triples to retrieve information about attribution.

The representation below illustrates how `sem:hasTarget` can be expressed through reification:

```
:targetRel16 a sem:targetRel ;
sem:hasSource nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev16Expect .
sem:hasObject :NGsell .

:targetRel11 a sem:targetRel ;
sem:hasSource nwr:/data/cars/2003/01/01/47VH-FG30-010D-Y3YG.xml#ev11Say .
sem:hasObject :NGexpect .

:targetRel132 a sem:targetRel ;
sem:hasSource nwr:madeupexampe#ev132Say .
sem:hasObject :NGsell .
```

Attribution values can be linked directly to a specific `sem:targetRel`:

```
:targetRel16 nwr:hasAttribution nwrontology:attrPOSPROBF .
:targetRel11 nwr:hasAttribution nwrontology:attrPOSCERTNF .
targetRel132 nwr:hasAttribution nwrontology:attrNEGPROPBF .
```

This is illustrated in Figure 43.

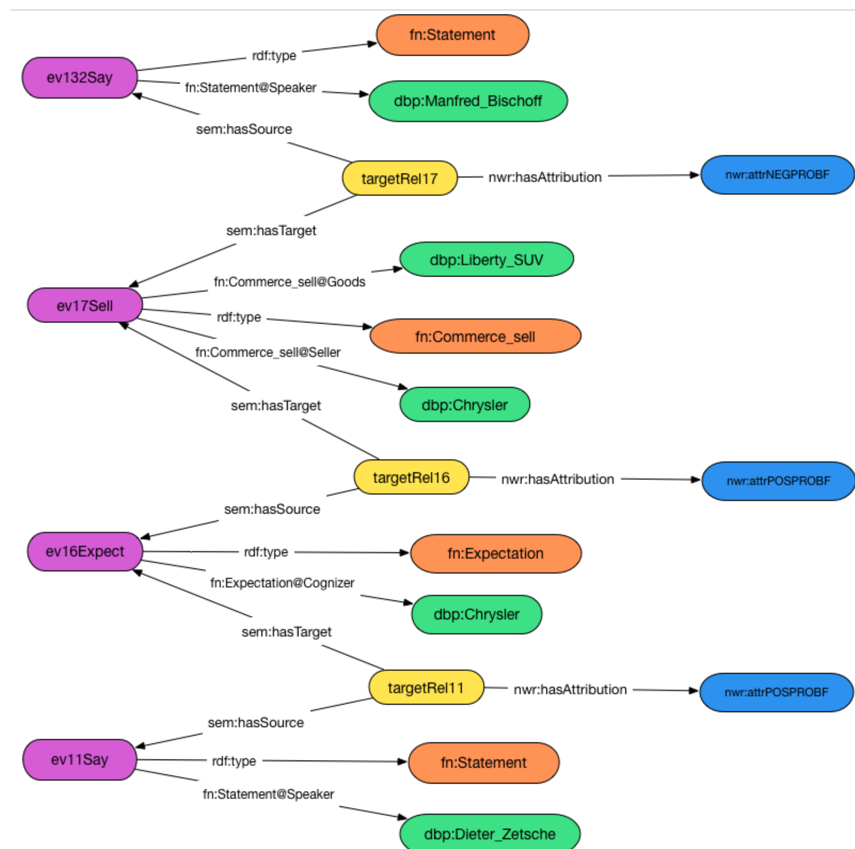


Figure 43: Alternative views proposal 3

B.4 Concluding remarks

Considering we have decided to model attribution at the mention level, there is no need to elaborate extensively on the discussions we have had comparing these proposals. We will therefore only briefly mention the conclusions we came to before the mention level modeling was proposed. Modeling attribution at the target level has the advantage that it captures the intuition that attribution is related to both the statement and its source. Because of the disadvantage that multiple embeddings can lead to, it seems like the reification solution is preferable in this case, though this may depend on the number of embeddings we actually observe (it is likely that they are quite rare). Duplicating triples as done in the first proposal also seems suboptimal. However, if we take into consideration that it is in general a hypothesis that two statements talk about the same thing (based on the

outcome of our algorithms), this may be an advantageous solution if we adopt an approach where we always create new instance when finding an event and then use the `owl:sameAs` (or a similar) relation to indicate that they are the same. In that case, the first proposal could quite naturally be integrated in our model. We currently believe attribution is best modelled at a mention level, but if we cannot resolve some of the open issues, we may turn back to these proposals.

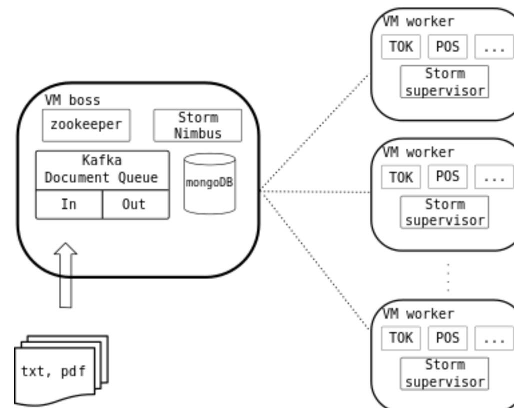


Figure 44: Main architecture of the NWR distributed pipeline

C NewsReader architecture for distributed NLP processing

This document describes the distributed pipeline implemented within the NewsReader project. The pipeline follows an streaming computing architecture, where the computation is not limited to any timeframe. Instead, the pipeline is always waiting to the arrival of new documents, and one such new document arrives the NLP processing starts.

Figure 44 shows a general overview of the NewsReader distributed pipeline. The processing NLP modules and the software infrastructure for performing parallel and distributed streaming processing of documents is packed into virtual machines (VM). Specifically, we distinguish two types of VM into the cluster:

- One "boss" node, which is the main entry of the documents and which supervises the processing.
- Several "worker" nodes, which actually perform the NLP processing.

The boss node is the entry point for input documents. It contains a document queue where the input documents are stored and wait until they enter the pipeline. It also supervises the execution of the different stages of the processing pipeline. When the NLP processing is over, the boss node receives the processed NAF document into the output queue, and writes it in a specified directory.

The boss node implements a REST service that accept documents to the processing pipeline. For instance, the following command sends the document `doc.xml` to the processing pipeline[15]:

```
% curl --form "file=@doc.xml" http://BOSSIP:80upload_file_to_queue.php
```

[15]In this document we will show many examples which are commands to execute from the CLI. If the command has to be executed from the host machine, we will use the '%' character in the beginning of the line. If the command is meant to be executed from the boss machine, we will use the '$' character.

Once the NLP processing is finished, the final NAF will be stored inside the boss VM, under the `docs/output` directory. In principle those documents should be sent to the KnowledgeStore (KS), but at the time being this last module which populates the KS with the processed NAFs is not ready.

The worker nodes are the nodes where the NLP modules are executed. There can be many worker nodes in a cluster, and therefore there can be many instances of the modules running in parallel.

This document is a reference guide that covers the most important steps to build a NewsReader architecture from scratch.

C.1 VM from scratch

We distribute a set of scripts with the aim of automatically create a fully working cluster for distributed NLP processing. We call these scripts "VM from scratch", as they create and configure the required virtual machines. The scripts are in a github repository, at this address:

`https://github.com/ixa-ehu/vmc-from-scratch`

The latest version of the documentation is also in the same repository, under the `doc` directory.

Creating a cluster using the scripts involves three main steps:

1. Create a basic cluster with the boss node and one worker node.
2. Make as many copies as required of the worker node.
3. Deploy the worker copies among different hosts and run the cluster.

The subsections below will describe how to use the provided scripts to achieve these goals in turn.

C.1.1 Create a basic cluster

The first step is to create the basic cluster using the `create_basic_cluster.pl` script. Please note that the VM images created by these scripts are huge (9 Gb memory), as they contain all the NLP modules of the Newsreader processing pipeline. Likewise, the machine to install those VMs need to have a large amount of RAM memory, as each VM, particularly the worker nodes, need circa 10Gb of memory to run.

The script is executed as follows[16]:

[16] In ubuntu machines, install the following packages before running this script

- libvirt-bin
- libguestfs-tools
- qemu-kvm

```
% sudo create_basic_cluster.pl --boss-ip 192.168.122.111 --boss-name bossvm \\
  --worker-ip 192.168.122.112 --worker-name workervm1
```

The script needs four parameters:

- `--boss-ip` (**required**) the IP address of the boss VM[17].
- `--boss-name` (**required**) the hostname of the boss VM
- `--worker-ip` (**required**) the IP address of the first worker VM created by this script.
- `--worker-name` (**required**) the hostname of the worker VM.
- `--master-ip` (**optional**) the IP address of the IXA master machine. This machine is where the latest version of all NLP modules are.

Use the `--help` option to obtain information about the script's parameters (this option is present in all scripts described here).

The script will perform the following steps for creating the basic cluster:

- Copy an empty VM image from the IXA servers.
- Create the boss VM image using the empty VM image. In the example above, the hostname of the boss VM is "bossvm" and its IP address is `192.168.122.111`.
- Create the worker VM image using the empty VM image. In the example above, the hostname of this worker VM is "workervm1" and its IP address is `192.168.122.112`.
- The VMs have an unique user "`newsreader`" whose password is "`readNEWS89`". The user can `sudo` any root commands inside the VMs.

The next step is to turn on both VMs and start a synchronization process so that the required software (both the system software as well as the NLP modules) is properly installed in the newly created VMs.

The above script creates two XML definition files with the specification of the boss and worker virtual machines[18]. To (locally) run the VMs, run the following commands as root from the host machine:

```
% virsh create nodes/bossvm.xml
% virsh create nodes/workervm1.xml
```

Once the machines are up and running, we ssh into the boss VM.

besides, you'll need to run

```
% sudo update-guestfs-appliance
```

[17]In this example and in the examples below we will use private IP addresses for the boss and worker IPs. See section C.1.5 for more information about IP addresses.

[18]See section C.1.3 on the XML definitions of the VMs.

```
% ssh newsreader@192.168.122.111
(pass: readNEWS89)
```

Once logged into the boss VM, run the following:

```
$ sudo /root/init_system.sh
```

This command prepares installs all required software into both virtual machines (boss and worker). Specifically, the script performs the following steps:

- Install the required software into the boss VM. This involves the installation, among others, of the following packages:
  - *Zookeeper*
  - *Storm*
  - *Kafka*
  - *MongoDB*
  - *Puppet*
- Synchronize the boss VM with the IXA master VM to obtain all the NLP modules.
- Synchronize the worker VM with the boss VM to obtain a copy of all the NLP modules.
- Start the *puppet* agents on both the boss and worker VMs.

When the scripts finalizes the basic cluster is finally created and configured. The boss VM image is stored in the file `nodes/bossvm.img` and the worker image is `nodes/workervm1.img`. The cluster is already ready for processing documents, although it does not make much sense to use a cluster with a single worker node. Therefore, the next step is to create the required copies of the worker VMs.

C.1.2 Making copies of worker VMs

The next step is thus to copy the worker VM and create new worker nodes in the cluster. Before doing this, however, the boss and the worker VMs must be shut down. There are several ways to shut down the cluster. One possible way is, being into the boss VM, is to execute the following:

```
$ sudo pdsh -w workervm1 poweroff
$ sudo poweroff
```

The first command remotely shuts down the worker VM, whereas the second command shuts down the boss VM itself. In any case, check that the VMs are properly shut down by executing the `sudo virsh list` command from the host machine. The command should return an empty list of VMs:

```
% sudo virsh list
 Id    Name                           State
----------------------------------------------------
```

Once the cluster shut down, one can make as many copies as wanted of the worker nodes. The script cp_worker.pl accomplishes this task:

```
%  sudo ./cp_worker.pl --boss-img nodes/bossvm.img \
  --worker-img nodes/workervm1.img 192.168.122.102,workervm2
```

The script needs two parameters with the images of the boss VM and the first created worker VM (workervm1 in our example). Then, the script accepts a space separated list of IP,hostname pairs. The above example just creates a copy of the worker VM, called workervm2, whose IP address will be 192.168.122.102. Of course, it is very important that each VM receives a different hostname and IP address.

If we were to create more copies of the VM, we can just run the script again, the only requisite being that the cluster VM can not be running. For instance, the following command will create two more worker nodes:

```
%  sudo ./cp_worker.pl  --boss-img nodes/bossvm.img \
  --worker-img nodes/workervm1.img \
  192.168.122.103,workervm3 192.168.122.104,workervm4
```

After this step we need to boot the cluster manually. Section C.1.4 explains how to power on and power off the whole cluster at any time.

Creating copies of the worker without stopping the cluster The requirement of having to stop the cluster before making copies of the workers may too hard to fulfill, in the case the cluster is running and processing documents. Therefore, we present a method to create worker nodes without the need to stop the whole cluster. However, this method needs a copy of a worker image, and it is very important that this particular image is not running.

For creating a worker without stopping the cluster, execute the following:

```
%  sudo ./cp_worker.pl  --boss-img nodes/bossvm.img \
  --worker-img nodes/workervm1.img \
  192.168.122.105,workervm5
```

This will create the new workervm5 node with IP address 192.168.122.105. Now, we need to log into the boss node, and manually add this new address to the /etc/hosts file:

```
% sudo echo "192.168.122.105 workervm5" >> /etc/hosts
```

The next step is to stop the topology (see section C.2) and run it again with the proper number of workers/CPUs.

C.1.3 Virtual machines XML definitions

The virtual machines specifications are defined in a XML document, one document per VM. These XML specification document is stored along with the VM image, i.e., if the image is in `nodes/workervm1.img`, the corresponding XML specification is `nodes/workervm1.xml`. You may want to manually edit the XML specification file to change some important things:

- The number of CPUs assigned to the VM (line 6, `/domain/vcpu` element). By default VMs are assigned 1 CPU but this can be easily changed. In fact, **we recommend to increase the number of CPUs per worker whenever possible**.
- The memory assigned to the VM (line 4, `/domain/memory` element). By default VMs are assigned 10Gb of RAM (needed mostly by the NED module).
- The exact location of the image files (line 22, `/domain/devices/disk/source` element). If you plan to deploy the worker VM on a different machine, you must change this value to the directory where the worker image file will be.

C.1.4 Booting and stopping the cluster

Boot the boss VM by running this into the host machine:

```
% sudo virsh create nodes/bossvm.xml
```

The host needs some time to start all the required services. To be completely sure that the machine is ready, run the following command inside the boss machine until it produces some output. First ssh into the boss VM, and then:

```
$ ps -aux | grep zoo | grep java
```

Once the boss is ready, turn on the worker nodes running the following inside the host machine:

```
% sudo virsh create nodes/workervm1.xml
% sudo virsh create nodes/workervm2.xml
...
```

The above commands will start the worker VMs in turn. Each VM needs some time to boot up (mostly due to the required time in loading the NED module). Section C.1.6 explains how to know which machines are connected to the boss VM.

C.1.5 Deploying worker VMs into different host machines

In principle the worker VMs can be executed in any host machine, as far as the host has a 64 bit CPU. The main requirement is that the IP of the worker VM, as specified when creating the VM image, is accessible from within the boss VM. Likewise, the boss IP has to be accessible from the worker VM.

In the examples on this document all the IPs are private IPs, and therefore not accessible from outside the host machine. If you need to copy the worker VMs among several host machines, perhaps the best way is to use a bridged network configuration (not explained here). Alternatively, you can use iptables for allowing external access through ssh to the VM. In any case, talk with the IT people in your laboratory to correctly set-up the environment.

C.1.6 Knowing which machines are connected to the boss VM

When the cluster is up and running, we can consult any time which worker machines are connected to the boss. This is easily done by querying the STORM server on the boss VM. From inside the boss VM, run the command line "elinks" web browser:

```
$ elinks localhost:8080
```

Another option is to connect to the boss VM from the host machine, by putting the address `http://ip_of_boss:8080`. In the running example used in this document the boss IP is 192.168.122.111, we should put the address `http://192.168.122.111:8080` into the web browser.

C.2 Running the topology

The steps described in section C.1 will create a cluster that is ready to start the NLP processing. For this, we need first to load a topology into the cluster. The topology is an abstraction that describes processing steps each document passes through. In principle the cluster can run any topology, the only requisite being that the NLP modules described in the topology are installed in the worker VMs.

Topologies are declaratively described in an XML document. There are already some topologies in the bossvm `opt/topologies/specs` directory. In this example, we will run the topology described in `opt/topologies/specs/test.xml`.

Apart from the topology definition, running the topology requires knowing the number of worker nodes in the topology, and the number of CPUs used. In the running example of this document we are using three workers. Suppose we also use two CPUs per worker[19]. So, the total of CPUs assigned to worker nodes is 6. This parameter is very important because it fixes the maximum parallelization we can achieve. If we have 6 CPUs working on our cluster, it makes little sense to have more than 6 copies of any NLP module running in parallel.

[19]Section C.1.3 describes how to change the number of CPUs assigned to a node

Inside the boss VM, run the following to run the topology:

```
$ opt/sbin/run_topology.sh -p 6 -s opt/topologies/specs/test.xml
```

This will load the topology and, as a consequence, the cluster will be ready to accept and process documents.

The script also accepts another optional parameter to specify the main configuration of the topology, the default value being `opt/etc/topology/topology.conf`. This configuration file is seldom changed within the newsreader project, and specifies values such as the port and addresses of various services such as zookeeper, kafka, mongoDB, etc. However, there is one parameter that need to be checked. The "host" values (`zookeeper.host`, `kafka.host`, `mongodb.host`) need to point to the hostname of the boss node (by default, `bossvm`).

C.2.1 Topology XML specification

The topology executed by the cluster is declaratively defined in an XML document. Here is an excerpt of a small topology:

```
<topology>
  <cluster componentsBaseDir="/home/newsreader/components"/>
  <module name="EHU-tok" runPath="EHU-tok/run.sh"
          input="raw" output="text"
          procTime="10"/>
  <module name="EHU-pos" runPath="EHU-pos/run.sh"
          input="text" output="terms"
          procTime="15" source="EHU-tok"/>
  <module name="EHU-nerc" runPath="EHU-nerc/run.sh"
          input="terms" output="entities"
          procTime="75" source="EHU-pos"/>
</topology>
```

The `<cluster>` element specifies the base directory of the NLP modules. Each module is described by a `<module>` element, whose attributes are the following:

- `name` (**required**): the name of the module.
- `runPath` (**required**): the path relative to `componentsBaseDir` where the module resides.
- `input` (**required**): a comma separated list of NAF layers required by the module as input.
- `output` (**required**): a comma separated list of NAF layers produced by the module.

- `source` (optional): the previous module in the pipeline. If absent, the attribute will get the value of the immediately preceding it according to the XML tree. If the module is the first node in the XML tree, and the source attribute is absent, the attribute gets no value at all.
- `procTime` (optional): the percentage of time this particular module uses when processing a document.
- `numExec` (optional): the number of instances of the module that will run in parallel.

The topology is defined by looking at the chains of `source` attributes among the modules, which have to define a directed acyclic graph. The module without parents is considered to be the first module in the pipeline. Likewise, the module without children is the last module in the pipeline. In principle the declaration may define non linear topologies, but at the moment its use is not properly tested, and thus not recommended.

The last two attributes are very important and require some tuning in order to obtain the maximum efficiency when processing the documents. There are two ways of calculating these factors. If the `numExec` parameter is present, this particular number of instances will be used. If `numExec` is absent, the system will use the `procTime` information, along with the number of nodes/CPUs to automatically calculate the number of instances of each module. The main idea is to execute as many copies as possible of the most resource demanding modules. Finally, if both `numExec` and `procTime` are absent, the system will run one single instance of the module.

C.3 Troubleshooting and tools

This section describes the most usual tasks performed by the cluster administrator to manage and monitorize the cluster, as well as when facing many problems that may occur. In principle the cluster should run flawlessly and without errors; it should just accept documents via the boss input port or inside the boss machine itself, and the documents should get processed, put in the output queue of the boss machine, and written to the `docs/output` in the boss machine. However, if something goes wrong there are some useful commands to help understanding what is happening.

C.3.1 Synchronizing the modules

The NLP modules are constantly being developed and upgraded into the IXA master machine. Obtaining the latest version of the modules from the master node into the cluster involves two steps. First, synchronize the boss node with the IXA master node; then, synchronize each worker node with the boss node.

For synchronizing the boss node with the IXA master, log in into the boss VM and run the following:

```
$ sudo ./update_nlp_components_boss.sh
```

For updating the workers there are two main options. One option is to log in into each of the workers and execute the following:

```
$ sudo ./update_nlp_components_worker.sh
```

The other is to update the worker nodes from inside the boss node, using the `pdsh` command. Being logged into the boss machine, run the following for each worker node:

```
$ sudo pdsh -w ip_of_worker /home/newsreader/update_nlp_components_worker.sh
```

C.3.2 Document queue

The boss VM contains two document queues, for input and output NAF documents, respectively. The input queue stores the documents to be processed until they are consumed by the NLP pipeline. The output queue stores the final NAF documents.

In principle, the usage of the queues is wholly automatic. The user sends a document to the input queue, and eventually the processed document is enqueued into the output queue. However, it is important to have some basic notions about how these queues are implemented in case we want to consult the contents of a particular queue, or if we want to manually send or retrieve documents.

The queues are implemented using “Apache Kafka”[20]. Kafka is a messaging system that can be distributed among many machines and that provides fast and reliable access to the queue contents. Queues in Kafka are organized into *topics* and *partitions*. In NewsReader we have two topics, called “`input_queue`” and “`output_queue`”, respectively, and each topic has one unique partition. Once a documents is sent to the queue, it stays there for a specific time frame, which by default is set to seven days. The input queue is attached to a consumer that continually pulls documents from the queue and sends them to the first stage of the processing pipeline. It also maintains an *offset* to the last consumed document in the queue.

Inside the boss VM there are some tools which can help with the management of the queues. Currently there exist the following tools:

`send_to_queue`

This tool is useful for feeding the input queue with documents from inside the boss VM. The usage is as follows:

```
$ opt/sbin/push_queue -f docs/input/input.xml
```

Please note that documents can be sent to the input queue from outside the boss VM, by using the following command:

```
% curl --form "file=@input.xml" http://BOSSIP:80/cm_upload_text_file.php
```

[20] http://kafka.apache.org/

this is meant to be the principal way to send documents into the NewsReader pipeline. The `send_to_queue` tool is an auxiliary tool for manually feeding the input queue with new documents.

`ls_queue`

Sometimes it is useful to know which documents are present in a queue (input or output), and the `ls_queue` command does exactly this. This is a typical example of usage (from inside the boss VM):

```
$ opt/sbin/ls_queue
doc18.txt  1209  9   input_queue
doc20.txt  1025  10  input_queue
doc23.txt  2020  11  input_queue
doc24.txt  347   12  input_queue
doc26.txt  789   13  input_queue
doc27.txt  257   14  input_queue
doc28.txt  1675  15  input_queue
doc29.txt  9837  16  input_queue
doc20.txt  721   17  input_queue
```

The tool displays 4 columns for each document in the queue, namely, the document name, the size, the offset number, and the queue name.

The above example showed the documents of the input queue waiting to be processed. The "`-a`" switch shows all the documents in the queue. Remember that Kafka keeps all documents in the queues for some time (by default, seven days). The following command shows all the documents in the queue.

```
$ opt/sbin/ls_queue -a
doc1.txt   762   1   input_queue
doc0.txt   4678  2   input_queue
doc3.txt   9876  3   input_queue
doc4.txt   189   4   input_queue
doc6.txt   875   5   input_queue
doc7.txt   8989  6   input_queue
doc8.txt   6735  7   input_queue
doc9.txt   9875  8   input_queue
doc18.txt  1209  9   input_queue
doc20.txt  1025  10  input_queue
doc23.txt  2020  11  input_queue
doc24.txt  347   12  input_queue
doc26.txt  789   13  input_queue
doc27.txt  257   14  input_queue
doc28.txt  1675  15  input_queue
```

```
doc29.txt  9837   16  input_queue
doc20.txt  721    17  input_queue
```

It is also possible to list the documents of the output query, using the "`-q`" command switch to select the queue name:

```
$ opt/sbin/ls_queue -q output_queue
doc1.txt_065a4fe5686a240b0569d35c6f04b025.naf 222715  21   output_queue
doc0.txt_0ff7e7f4e55ad14f085aada286f0b718.naf 219339  22   output_queue
doc3.txt_2ab255aac76c7b7110be1fe7cf9bfd0e.naf 214548  23   output_queue
doc4.txt_3762e590fe5ba62e96845491acda0656.naf 213492  24   output_queue
doc6.txt_98aef58c8b9b824006dd3c5a0d4d87cb.naf 212985  25   output_queue
doc7.txt_d0af8c39cd227492e6d6cf656be19491.naf 211940  26   output_queue
doc8.txt_45d6c827fbab1ef8f211a2a0ed2d448c.naf 210390  27   output_queue
doc9.txt_5cd1fbedf874d25391ba78ea789ebe90.naf 209767  28   output_queue
doc18.tx_621f460498502b42d4a9ddf2401349f9.naf 209104  29   output_queue
```

Note that these examples show documents consumed from the input queue are already processed and stored in the output queue.

`flush_queue`

Once the documents are processed, they are stored in the output queue. The boss machine contains a consumer that continuously pulls from the output queue and stores the documents in the `docs/output` directory. However, sometimes it is interesting to dump all the current content of a queue (input or output) into a directory. The `flush_queue` accomplishes this task.

```
$ opt/sbin/flush_queue -q output_queue -o path_to_docs
```

The parameters of the script are the following:

- `-o` (**required**): the directory to dump the documents.
- `-f` (optional): by default the script will not overwrite any existing document in the output directory. Set this parameter to overwrite the documents.
- `-q` (optional): the queue to flush. If omitted, the script dumps the document from the `output_queue` queue.

Logs

The cluster maintains a log of all the documents when passing over the several stages of the pipeline. Each time the document processing starts, or when it enters some particular stage of the topology (in any worker machine), a log entry is created. Besides, if a document processing fails a record is created with the name of the document and the module which caused the error.

All the log records are stored into the *mongoDB* database, and are accessible from within the boss node. To access the logs, one has to first enter the so called mongo shell. Being logged into the boss machine, run the `mongo` command:

```
$ mongo
MongoDB shell version: 2.4.6
connecting to: test
> use newsreader-pipeline
switched to db newsreader-pipeline
> db.log.find()
```

the `find` command will show all entries of the log database. Each log entry has the following fields:

- `_id`: internal ID.
- `tag`: the entry type. Possible values:
  - `DOC_START` -¿ the document "doc_id" enters the pipeline
  - `DOC_BOLT_RCV` -¿ the document "doc_id" enters the bolt "module_id"
  - `DOC_BOLT_EMT` -¿ the document "doc_id" exits the "module_id"
  - `DOC_BOLT_FAIL` -¿ the document name "doc_id" failed in bolt "module_id"
  - `DOC_FAIL` -¿ the document "doc_id" failed
  - `DOC_ACK` -¿ the document "doc_id" worked as expected
- `doc_id`: the document name.
- `module_id`: the NLP module.
- `hostname`: the node name.
- `timestamp`: the time stamp in mongoDB format.

Find documents with errors, also displaying the bolt which caused the error and the worker node

```
> db.log.find( { tag: "DOC_BOLT_FAIL" }, { _id:0, doc_id:1, module_id:1, hostname:1 }
{ "doc_id" : "4MSN-CTV0-TX2J-N1W8_8bb9805598b55e10850852165981897f.xml",
  "module_id" : "FBK-time", "hostname" : "workervm1" }
{ "doc_id" : "4MR6-N790-TX33-71WM_bab251443f3e6869306c87f8173e11c4.xml",
   "module_id" : "EHU-corefgraph", "hostname" : "workervm2" }
{ "doc_id" : "4MRN-CMP0-TX37-G2P3_82436b4a93ef1826a280763ac2838a74.xml",
   "module_id" : "EHU-corefgraph", "hostname" : "workervm1" }
{ "doc_id" : "4MRP-5X50-TX2J-N2H8_d83f60a3c7a5e783b8e84fd5d13a9b69.xml",
   "module_id" : "EHU-corefgraph", "hostname" : "workervm3" }
...
```

Find successfully analyzed documents

```
> db.log.find( { tag: "DOC_ACK" }, { _id:0, doc_id:1 } )
{ "doc_id" : "4MR1-81M0-TWX2-W2WD_f5341438336a8f739fd3bb2192a3374b.xml" }
{ "doc_id" : "4MR2-D9G0-TX52-F26P_5484c46f08a6242518fa389f8f1e4851.xml" }
{ "doc_id" : "4MRC-JSD0-TX2J-N1V0_7a5dc0f87c195104251969c9ca100c6c.xml" }
{ "doc_id" : "4MRK-J360-TX2J-N39T_f51f321d7a7a60d8c802bfece498027b.xml" }
...
```

C.3.3 Starting and stopping the services

There are a number of scripts in the boss VM (under the `opt/init.d` directory) to start and stop many services. It is convenient to stop the puppet agent before stopping services:

```
$ sudo /etc/init.d/puppet stop
```

Likewise, the puppet agent needs to be restarted after the services are running again:

```
$ sudo /etc/init.d/puppet start
```

Here is a list of the provided scripts to start or stop the services. All these scripts require one command, `start` or `stop`, which starts or stops the service, respectively:

- `kafka_server`: it needs a parameter, `start` or `stop`. The script starts or stops the kafka service (document queues), respectively.
- `zookeeper_server`: start or stop the zookeeper service.
- `storm_boss_server`: start or stop the STORM nimbus service.
- `boss_servers`: this script start or stops all services in the boss VM.

- `wipe_all_boss`: this script is special and erases all documents of the queues. **Use it with care**! It is recommended to first dump the documents of the queues in some temporary directory before erasing the queues.

The worker services can also be started/stopped:

- `worker_servers`: start or stop all the services in the worker VM. Run this script inside the worker VM. Alternatively, run this script from inside the boss VM using the `pdsh` tool. Inside the boss VM, write the following:

```
$ sudo pdsh -w ip_of_worker /home/newsreader/opt/sbin/worker_servers stop
$ sudo pdsh -w ip_of_worker /home/newsreader/opt/sbin/worker_servers start
```

References

[Beloki *et al.*, 2014] Zuhaitz Beloki, German Rigau, Aitor Soroa, Antske Fokkens, Piek Vossen, Marco Rospocher, Francesco Corcoglioniti, Roldano Cattoni, Thomas Ploeger, and Willem Robert van Hage. System design, 2014.

[Cherniack *et al.*, 2003] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

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

[Fokkens *et al.*, 2013] Antske Fokkens, Marieke van Erp, Piek Vossen, Sara Tonelli, Willem Robert van Hage, Luciano Serafini, Rachele Sprugnoli, and Jesper Hoeksema. GAF: A grounded annotation framework for events. In *Proceedings of the first Workshop on Events: Definition, Dectection, Coreference and Representation*, Atlanta, USA, 2013.

[Fokkens *et al.*, 2014] Antske Fokkens, Aitor Soroa, Zuhaitz Beloki, Niels Ockeloen, German Rigau, Willem Robert van Hage, and Piek Vossen. NAF and GAF: Linking linguistic annotations. In *To appear in Proceedings of 10th Joint ACL/ISO Workshop on Interoperable Semantic Annotation (ISA-10)*, 2014.

[Ide *et al.*, 2003] Nancy Ide, Laurent Romary, and Éric Villemonte de La Clergerie. International standard for a linguistic annotation framework. In *Proceedings of the HLT-NAACL 2003 Workshop on Software Engineering and Architecture of Language Technology Systems (SEALTS)*. Association for Computational Linguistics, 2003.

[Moreau *et al.*, 2012] Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV Data Model. Technical report, 2012.

[Rospocher *et al.*, 2015] Marco Rospocher, Piek Vossen, Tommaso Castelli, and Agata Cybulska. Event narrativemodule, version 2. NewsReader Deliverable 5.1.2, 2015.

[Tonelli *et al.*, 2014] Sara Tonelli, Rachele Sprugnoli, and Manuela Speranza. Newsreader guidelines for annotation at document level nwr-2014-2. Technical report, 2014.

[van Hage *et al.*, 2011] Willem Robert van Hage, Véronique Malaisé, Roxane Segers, Laura Hollink, and Guus Schreiber. Design and use of the Simple Event Model (SEM). *J. Web Sem.*, 9(2):128–136, 2011. `http://dx.doi.org/10.1016/j.websem.2011.03.003`.

[Venner *et al.*, 2014] Jason Venner, Sameer Wadkar, and Madhu Siddalingaiah. *Pro Apache Hadoop*. Apress, 2014.

[White, 2009] Tom White. *Hadoop: the definitive guide: the definitive guide.* " O'Reilly Media, Inc.", 2009.