Natural Language Engineering: page 1 of 23. © Cambridge University Press 2017
doi:10.1017/S1351324917000092

A scalable architecture for data-intensive natural language processing[†]

ZUHAITZ BELOKI, XABIER ARTOLA and AITOR SOROA

IXA NLP Group, University of the Basque Country (UPV/EHU), Donostia-San Sebastián
e-mail: zuhaitz.beloki@ehu.eus, xabier.artola@ehu.eus, a.soroa@ehu.eus

(*Received 25 April 2016; revised 7 February 2017; accepted 8 February 2017*)

Abstract

Computational power needs have greatly increased during the last years, and this is also the case in the Natural Language Processing (NLP) area, where thousands of documents must be processed, i.e., linguistically analyzed, in a reasonable time frame. These computing needs have implied a radical change in the computing architectures and big-scale text processing techniques used in NLP. In this paper, we present a scalable architecture for distributed language processing. The architecture uses Storm to combine diverse NLP modules into a processing chain, which carries out the linguistic analysis of documents. Scalability requires designing solutions that are able to run distributed programs in parallel and across large machine clusters. Using the architecture presented here, it is possible to integrate a set of third-party NLP modules into a unique processing chain which can be deployed onto a distributed environment, i.e., a cluster of machines, so allowing the language-processing modules run in parallel. No restrictions are placed a priori on the NLP modules apart of being able to consume and produce linguistic annotations following a given format. We show the feasibility of our approach by integrating two linguistic processing chains for English and Spanish. Moreover, we provide several scripts that allow building from scratch a whole distributed architecture that can be then easily installed and deployed onto a cluster of machines. The scripts and the NLP modules used in the paper are publicly available and distributed under free licenses. In the paper, we also describe a series of experiments carried out in the context of the NewsReader project with the goal of testing how the system behaves in different scenarios.

1 Introduction

A vast amount of textual information is produced every day through diverse channels such as traditional newspapers and social media sites. Linguistically analyzing this huge amount of text requires new architectures and paradigms that overcome the limitations of traditional pipeline-based processing. For instance, the Newsreader European project[1] requires automatically processing streams of daily news in four

[†]This work has been partially funded by the NewsReader (FP7-ICT-2011-8-316404) project. Zuhaitz Beloki's work is funded by a PhD grant from the University of the Basque Country.
[1] http://www.newsreader-project.eu/

languages with the aim of extracting events and their participants as mentioned in the text. The ultimate goal of the project is to assist professionals to make well-informed decisions based on the knowledge that NewsReader can offer them. Therefore, analyzing huge amounts of textual data in a timely manner becomes of paramount importance for the project.

Documents are traditionally analyzed offline following a *batch* setting, which requires all of the input data to be completely available on the input store before any computation is started. Typically, input data are processed and the output results are available only when all the computation is completed. Batch processing aims at optimizing the overall *throughput* of the system, that is, the overall time spent on processing all the documents. In contrast, *streaming* computing (Cherniack *et al.* 2003) expects documents to arrive at any moment, and aims at reducing the elapsed time needed to process one single document. Streaming processing is thus focused on optimizing the *latency*, in order to process one single document in the shortest possible time. In this paper we present a scalable architecture for language processing which is able to perform on both batch and streaming scenarios.

Complex NLP applications require basic linguistic processing modules (Part of Speech (POS) tagging, Named Entity Recognition and Classification (NERC), syntactic parsing, coreference resolution, etc.) to be able to further undertake more complex tasks. These basic NLP modules are used as building blocks to form complex processing chains required for end-user applications such as Information Extraction, Question Answering or Sentiment Analysis.

Building scalable NLP applications requires designing solutions that are able to run distributed programs in parallel and across large machine clusters. The parallelization can be effectively realized at several levels. The most effective way to achieve full parallelization is probably to re-implement all the core algorithms and procedures of linguistic processors, and adapt them to follow some well-known paradigms like *MapReduce* jobs (Dean and Ghemawat 2008). This way, NLP can benefit of the whole potential offered by large-scale computation frameworks such as Apache Hadoop.[2] However, NLP modules are complex pieces of software that are implemented using a variety of programming languages, and which often require the integration of third party libraries and dependencies to work properly. Feasible as it might be, it would take a tremendous amount of time to adapt each NLP module of a processing chain and re-implement it following the *MapReduce* paradigm. Besides, on a rapid evolving area such as NLP, where new algorithms and tools are developed continuously, there is a considerable risk of components becoming obsolete because new tools perform the task better and more efficiently.

Our approach is therefore different. Having a set of third-party modules that perform some language-processing task, we aim at integrating them into a unique processing chain which can be deployed onto a distributed environment. The distributed environment will allow running the language-processing modules in parallel using a cluster of machines. We put no restrictions on the NLP modules apart of being able to consume and produce linguistic annotations following a common format,

[2] https://hadoop.apache.org/

that in our case is NAF (cf. Section 3). We show the feasibility of our approach by integrating two linguistic processing chains for English and Spanish, respectively.

Typical NLP suites and tools usually require users to perform complex compilation/installation/configuration procedures in order to use such modules. In research, and also in industry, acquiring, deploying or developing such base-qualifying technologies is an expensive undertaking that redirects their original central focus. In research, much time is spent in the preliminaries of a particular experiment trying to obtain the required basic linguistic annotation, whereas in an industrial environment SMEs see their already limited resources taken away from offering products and services that the market demands. The architecture presented here can be easily installed and deployed onto a cluster of machines. This way, it offers SMEs and research laboratories the possibility of effectively using cluster nodes and CPUs to perform fast linguistic processing. All the components of the system (core technologies, as well as all the NLP processors) are publicly available under free licenses. Besides, we have developed a set of scripts that allow the building of a whole distributed architecture from scratch.

This paper is structured as follows. In Section 2, we present the IXA pipes tools, which are the actual language processing modules used in the experiments described in Section 6. Section 3 briefly describes NAF, the common linguistic annotation format required by the system. Next, in Section 4, we present the overall architecture of the system, describing its main characteristics and the most important components, while in Section 5 its integration with MongoDB, a NoSQL database is described. Section 6 is devoted to explain the experiments realized and the results obtained in them. Finally, some related work is depicted in Section 7, and conclusions and future plans are explained in Section 8.

2 Linguistic processing chains for NLP applications

The distributed architecture relies on a set of NLP modules that actually perform the linguistic processing tasks. The IXA pipes tools (Agerri, Bermudez and Rigau 2014)[3] is a multilingual suite for performing NLP analysis. IXA pipes provide tools that are simple and ready to use, portable, modular, efficient, accurate and distributed under a free license. They are based on the Apache OpenNLP API,[4] and rely on machine learning algorithms to perform several NLP tasks, yielding state of the art results in many of them (Agerri and Rigau 2016). As in Unix-like operating systems, the IXA pipes consist of a set of processes that communicate through standard input and output streams, in a way that the output of each process feeds directly as input to the next one. The Unix pipeline metaphor has been applied for NLP tools by adopting a very simple and well-known data-centric architecture, in which every module/pipe is interchangeable for another one as long as it reads and produces the required data format. The IXA pipes are designed to minimize or eliminate any installation/configuration/compilation effort and are distributed under the Apache 2.0 license, which is free and commercially friendly (Agerri *et al.* 2014).

[3] http://ixa2.si.ehu.es/ixa-pipes
[4] http://opennlp.apache.org

```
<NAF>
 <text> <!-- text (token) layer -->
  <wf id="w1" offset="0" length="4">John</wf>
  <wf id="w2" offset="5" length="6">taught</wf>
  <wf id="w3" offset="12" length="11">mathematics</wf>
  <wf id="w4" offset="24" length="2">in</wf>
  <wf id="w5" offset="27" length="3">New</wf>
  <wf id="w6" offset="31" length="4">York</wf>...
 </text>
 <terms> <!-- term layer (POS, lemmas, WSD, ...) -->
  <term id="t1" lemma="John" pos="R">
   <span><target id="w1"/></span>
  </term>
  <term id="t2" type="open" lemma="teach" pos="V">
   <span><target id="w2"/></span>
  </term>...
 </terms>
 <!-- entity layer (NERC and NED) -->
 <entities>
  <entity id="e1" type="person">
   <references> <!--John-->
    <span><target id="t1"/></span>
   </references>
  </entity>
  <entity id="e2" type="location">
   <references> <!--New York-->
    <span><target id="t5"/><target id="t6"/></span>
   </references>
   <externalReferences> <!-- link to DBpedia -->
    <externalRef reference="http://dbpedia.org/page/New_York_City"
                 confidence="0.8"/>
   </externalReferences>
  </entity>...
 </entities>
</NAF>
```

Fig. 1. Excerpt of a NAF document showing the text, term, and entity layers.

Table 1 shows the IXA pipes English modules of the NLP processing chain used for the experiments in this paper. The processing chain comprises fifteen modules and perform tasks such as tokenization (TOK), POS tagging, Word-Sense Disambiguation (WSD), NERC and Name Entity Disambiguation (NED), constituent parsing, coreference resolution, Semantic Role Labeling (SRL), and so on. Some modules are part of the IXA pipes toolkit, whereas others are NLP processors adapted to work within the IXA pipes framework, the so-called third-party tools.[5] See Agerri *et al.* (2016) for a complete description of the NLP modules.

3 NAF: An NLP Annotation Format

The experiments described in this paper involve the integration of many NLP tools into a common framework. One key issue to achieve this integration is the

[5] http://ixa2.si.ehu.es/ixa-pipes/third-party-tools.html

Table 1. *IXA-Pipes English modules, including pre- and post-requisites*

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	Tokens (`text`), POS tags (`terms`)	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`)

The modules marked with † follow a server/client architecture.

definition of a common annotation format that guarantees the correct interoperability among the tools. In this work, we use the so-called NLP Annotation Format (NAF) (Fokkens *et al.* 2014), which is designed to be a standard format for exchanging information between linguistic processing tools.

NAF follows the main principles of LAF as outlined in Ide, Romary and de La Clergerie (2003). Like LAF, NAF aims at maximum flexibility, processing efficiency and reusability. It is a layered, extensible format where each tool incrementally adds its output while maintaining all the information that was present in its input. NAF has shown to be suitable for linguistically annotating documents in many languages, and it has been successfully used on several projects such as Kyoto,[6] OpeNER,[7] and NewsReader.

NAF comprises several annotations over a text at different linguistic 'levels' (morphosyntactic, syntactic, and semantic). The levels refer to different types of linguistic information, which can be different groupings of linguistic entities (e.g., tokens versus terms versus chunks), relations between linguistic entities (e.g., dependencies and semantic roles), or information about a specific linguistic entity (e.g., disambiguated word sense).

The characteristic of being multi-layered makes NAF to be particularly well suited to work on a distributed and parallel environment. Processing modules represent their output in different NAF layers and usually they do not modify the annotations of the lower layers. Therefore, several processors can create new annotations to the same document in parallel as described in Section 5.2.

The most basic level in NAF is the text layer that assigns identifiers to tokens in the text. The term layer defines basic terms (lexical units) that consist of one or more tokens in the case of multiword expressions. Further layers (chunks, entities, etc.) typically consist of one or more terms. `span` elements are used to refer to specific elements in lower layers. For instance, in NAF, multiword expressions are described by a single term, which spans to the IDs of the tokens that compose the expression. Figure 1 shows an excerpt of a NAF document comprising three layers: text, terms, and entities. Apart from this, NAF provides layers to represent the output of common NLP tasks such as chunking, dependency parsing, semantic role labeling, coreference, time expressions, temporal relations, causal relations, and factuality.

4 A distributed architecture for language processing

The distributed architecture for parallel processing of documents comprises many processing nodes and a controller node that orchestrates the document processing flow. Documents are typically analyzed in several processing nodes, and the analysis stages each document goes through are carried out in parallel.

The main characteristics of the implemented architecture are the following:

- The computation is distributed, that is, different stages of the processing chain may be realized on different machines.

[6] http://www.kyoto-project.eu
[7] http://www.opener-project.eu

- The system has a unique entry point. Documents are sent to an input queue, which is located in a given node, and the distribution of the processing among the different machines is done automatically.
- Likewise, the processed documents are delivered to a single output queue.
- The topology—the definition graph describing the NLP modules and their dependencies—is described in a declarative way, and thus it can be easily modified.
- New documents may arrive to the chain at any time, and the linguistic processing starts as soon as possible.
- The architecture allows processing documents using 'non-linear' topologies. Whenever two or more modules have no inter-dependencies, they can be executed in parallel for a single document.
- The system is easily scalable. It is straightforward to include more computer power into the distributed architecture by adding more machines. The system will automatically rebalance itself and include the new machines, which will be thus integrated into the processing chain.

The processing chain is meant to be deployed onto a cluster of machines for execution. The current implementation relies on virtual machines (VMs) that contain all the required modules to process the documents. Virtualization is a widespread practice that increases the server utilization and addresses the variety of dependencies and installation requirements. Besides, it is a 'de-facto' standard on cloud-computing solutions, which offer the possibility of installing many copies of the virtual machines on commodity servers. In our architecture, all NLP modules, along with the dependencies they have, are installed into a VM, which is then copied and deployed into clusters of computers.

We use Apache Storm[8] to integrate and orchestrate the NLP modules of the processing chain. Storm is a framework for streaming computing whose aim is to implement highly scalable and parallel processing of data streams. Storm processing is based on *topologies*, a top-level abstraction that declaratively describes the processing that each document goes through. It is important to note that in Storm you can have several instances of each topology node, thus allowing the actual parallel processing. Following the so-called *parallelism hint*, it is possible to specify how many instances of each topology node will be actually running.

Storm nodes fall into two categories: the so called *spout* and *bolt* nodes. *Spout* nodes are the entry points of the topology. *Bolt* nodes are the actual processing units, which receive documents, process them, and pass the analyzed documents to the next stage in the topology. In our case, the *bolt* nodes are wrapper programs that receive input documents, call the actual NLP modules for processing, and send the output *tuples* (the pieces of information transferred from one bolt to another) to the next stage in the topology. In Storm, each node of the topology may reside on a different physical machine; the Storm controller (called *Nimbus*) is the responsible to

[8] https://storm.apache.org/

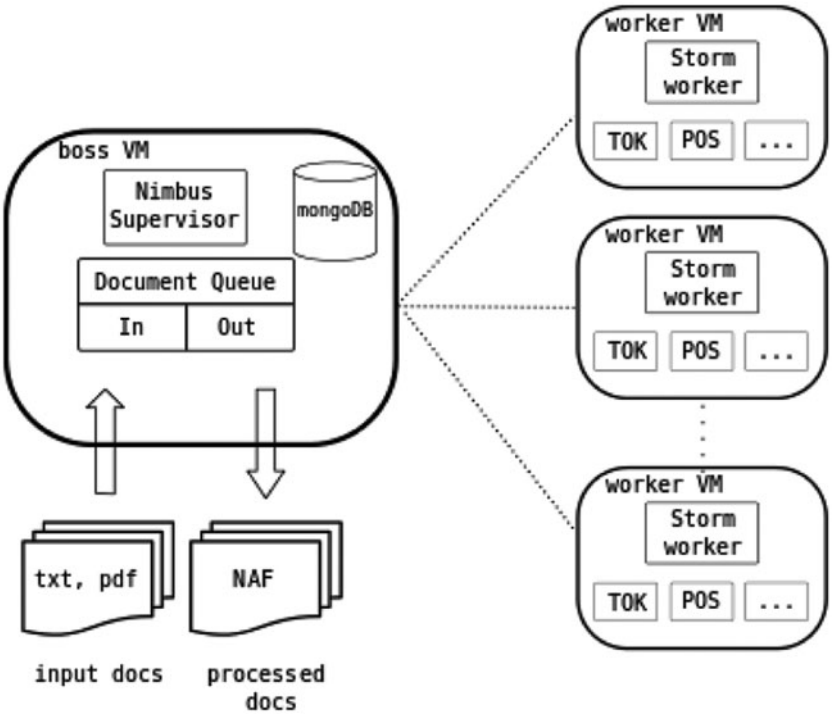

Fig. 2. Main architecture of the distributed architecture for streaming processing.

distribute the tuples among the different machines, and guarantees that each tuple undergoes all the nodes in the topology.

Figure 2 shows the general overview of the distributed architecture. The architecture consists of a number of VMs: one *boss* node and several *worker* nodes. We will see both VM types in turn.

4.1 The boss *node*

The *boss* contains a document queue where input documents are stored until the linguistic processing begins. On a streaming computing scenario, the system 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 system. However, on some occasions many documents may arrive at the same time or within a tight timeframe, and the system may not be able to process them immediately. Documents will thus be temporarily stored in the input queue, and they will wait there until the processing chain is ready for consuming them.

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

Because the documents may be analyzed on different nodes, it is very important that the information is efficiently shared among *worker* VMs. We use a *MongoDB* database to store all the intermediate results of the processing, as explained in Section 5.

Finally, when the processing is over, the *boss* node stores the processed NAF document in an output queue.

4.2 Worker *nodes*

Worker nodes are where the actual document processing is performed. A cluster will typically contain many *worker* nodes, therefore allowing many instances of the NLP modules running in parallel. Each *worker* node is contained into a VM, which contains instances (copies) of the NLP processors, including the required dependencies, the run-time environments and third-party libraries. The NLP modules are synchronized from the *boss* node at *worker* creation time, and can be regularly updated. The only requirement posed into the modules is that of consuming and producing NAF documents from the standard input and output streams, respectively. Each module is wrapped in a Storm *bolt*, which acts as a proxy and connects the particular module with the next stages of the processing chain. The wrapping process is described in the next section.

The majority of the NLP modules are executed on a document basis. However, when a NLP module follows a client/server paradigm, we initialize the server side once when setting up the VM, and configure the *bolt*s to launch new instances of the client side. Running modules in a client/server fashion has some important consequences. On the one side, it significantly reduces the overhead of module initialization, which is performed only once when the server is loaded. On the other side, running the server in the background reduces the available memory for the rest of the modules in the *worker* nodes, and, in fact, sometimes it is not viable to keep every NLP processor loaded and running on each node. Our setting allows creating dedicated *worker* nodes that contain only the processors specified by the user, and frees up the resources needed by those processors on the rest of the nodes. We did not use this setting in the current paper, and we leave the work of experimenting with dedicated *worker* nodes for future work.

Out of the fifteen modules used in our experiments (see Table 1), only one module is used as server/client, namely, the NED module, which needs 8 min to complete the initialization phase, and has a 10 GB memory footprint.

5 Wrapping NLP modules and MongoDB integration

In the distributed architecture documents may be analyzed on several *worker* nodes as they pass through the different stages of the processing chain and, as a consequence, partially annotated NAF documents have to be shared among *worker*s. To reduce the traffic as much as possible, we designed the system so that each NLP module receives only the required annotation layers and creates its output in the form of new annotation layers. We use a NoSQL MongoDB database

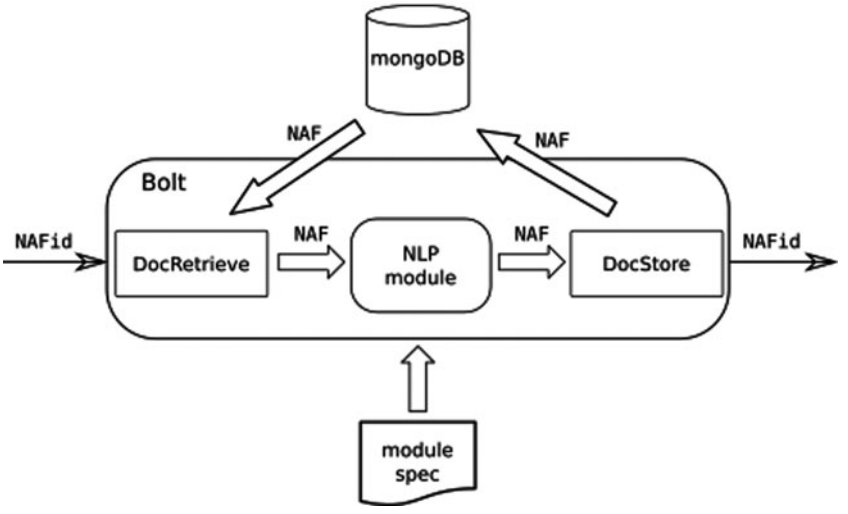

Fig. 3. *Bolt* wrapper around an NLP module.

to store all the annotation layers for partially annotated documents shared among the nodes. MongoDB is a schema free, document-oriented NoSQL database, which allows replication by maintaining local copies of the shared database on each node.

Instead of using a central repository for storing partially annotated documents, we could have send the required annotation layers as STORM tuples among the NLP modules. However, this would have considerably increased the overall traffic in the system. For instance, in Figure 6, the POS module sends the output (the NAF `terms` layer) to the modules immediately depending on it. However, the `terms` layer is also required by modules that are farther apart in the processing chain, like the NED module or the different modules dealing with time expressions (see Table 1). In the absence of a central database, the POS module would have to send its output to almost every module in the chain, increasing the overall traffic of the system.

Inside each *worker* the modules are managed using Storm, and each NLP module is wrapped as a *bolt* inside the Storm topology, as shown in Figure 3. The *bolt* receives a tuple from the preceding stage in the chain, which consists of a NAF document identifier that unequivocally identifies a partially annotated NAF document as stored in the central MongoDB database. The *bolt* also needs the module specification as explained in Section 5.2, so that it knows which NAF layer the underlying NLP module consumes and produces. Finally, the *bolt* keeps a log with the begin and end timestamps to measure the elapsed times spent by the modules.

When a new tuple arrives, the *bolt* node queries the MongoDB database and re-creates a subset of the NAF document by retrieving just the NAF layers (and their dependencies) required by the module. Then, the *bolt* node launches an instance of the NLP module, which analyzes the NAF document and produces new annotation layers and/or modifies previous ones. The new annotations are then stored again in the database, and the newly created NAF document identifier is passed to the next stage of the processing chain.

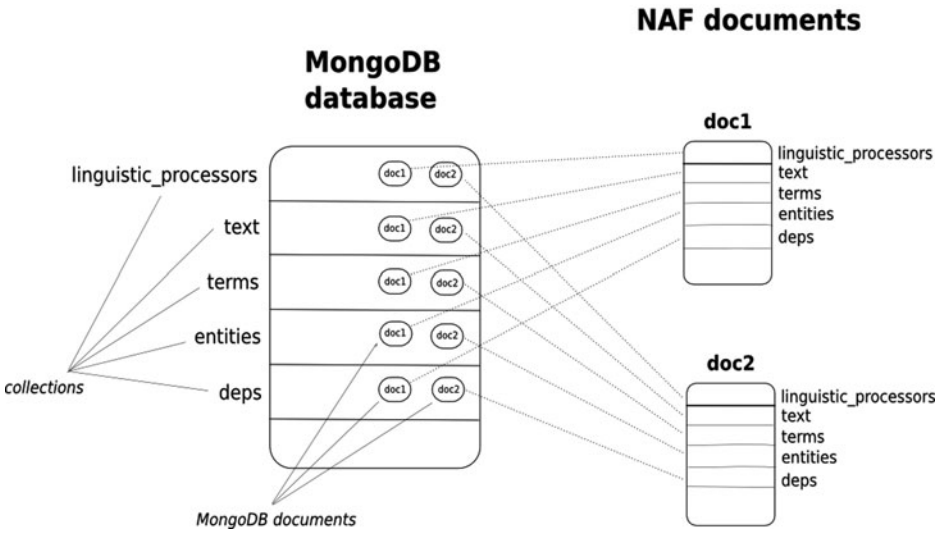

Fig. 4. Structure of the MongoDB database.

5.1 Integration of the MongoDB database

In this section, we will briefly describe the internal structure of the MongoDB database as used in the distributed system. We designed the database structure so that the most frequent database operations can be performed in the most efficient way. The main requirement for the database organization is thus to allow quick access to a large number of annotated NAF documents. The main information unit in MongoDB is the *document*. Documents are stored using BSON, 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 composed by many collections.

We store each NAF annotation layer in 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 in a separate MongoDB document. Figure 4 shows the organization of NAF documents into MongoDB collections and documents.

Storing NAF documents into MongoDB requires representing the NAF content (an XML document) in BSON. 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. An important feature of our system is that the queries retrieve complete NAF layers from a document (likewise, write operations store complete NAF layers). Note that the layer name is not an attribute of a

```
<topology>
  <cluster componentsBaseDir="/home/worker/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>
```

Fig. 5. Excerpt of the module specification document.

MongoDB document, but it determines the collection into which the document is found. Therefore, the only requisite to perform the queries efficiently is to index the documents according to its identifiers.

With this structure, we avoid two potential problems of inappropriate MongoDB designs. One of them is document growth. If documents grow and exceed the allocated space, MongoDB will relocate them on disk. We insert a single layer of annotations of a NAF document in each MongoDB document. Assuming that annotations will not change once they are stored in the database, we ensure that documents will not be grown 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, by writing one single document at a time.

5.2 Describing topologies

Topologies representing any NLP processing chain can be declaratively defined by means of an XML document, as shown in Figure 5. The `<topology>` root 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:

- `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 processing time of the module.
- `source`: the previous module in the processing chain. The first module in the chain has no `source` attribute.

The topology is fully specified by following the parent chain as specified by the `source` attribute of each module. The `input` and `output` attributes describe the NAF layers consumed and produced by the modules, and are used to determine the pre- and post-requisites a particular module has. The `procTime` attribute describes the relative time spent by this module when analyzing documents. The value is used for tuning the processing chain and specifying how many copies of a particular

module will be executed in parallel. The main idea is that the most demanding modules, according to their processing time, are the best candidates to be replicated, so that many instances are executed in parallel. This value is of course unknown beforehand, and so, we executed a version of the system with one single copy for each module over a small set of documents, and measured the elapsed time spent by each module. This time was used to calculate the `procTime` attribute of the modules.

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

$$p_i = \left\lceil \frac{t_i \cdot N}{T} \right\rceil \tag{1}$$

where t_i is the relative elapsed time by module i (the value of the `procTime` attribute), $T = \max t_i$ is the maximum elapsed time, and N is the number of *worker* nodes. Using this formula, the number of instances of each module increments as its workload is bigger. Also, dividing by the elapsed time of the slowest module, we limit the maximum number of instances to be the same as the number of nodes, avoiding to have more than one instance of a module running on the same node. Please note that the result of the formula is the ceiling of the division, ensuring that there exists at least one instance of the fastest modules.

For instance, our particular NLP processing chain comprises fifteen modules that perform all the required tasks for document event extraction. Table 1 shows the NLP modules, along with the pre- and post-requisites of each of them, indicating also which NAF layers each one of them consumes and produces (in brackets). We have defined two different topologies that represent our NLP processing chain. The first topology, which we call *linear*, executes all the processing steps serially, i.e., one module after another. In contrast, the *non-linear* topology allows many processing stages to be executed at the same time for a single document. 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 as shown in the *Input* column of Table 1, one can see that the NERC and SRL modules can be executed in parallel, 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 6 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 1. Note that the dependencies of the modules heavily depend on the particular techniques used when implementing the module. For instance, one would expect that the Fact module would need coreference chains and/or time expressions. However, the module is implemented as a supervised system that rely exclusively on features at lemma level (see Agerri *et al.* 2016).

Implementing non-linear topologies requires that modules can work with the required subsets of the NAF document they need. 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.

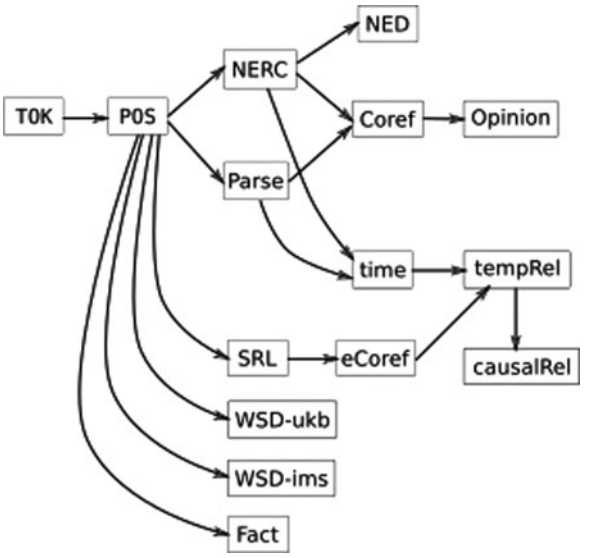

Fig. 6. English non-linear topology used in our experiments.

5.3 VM cluster from scratch

We have developed a set of scripts with the aim of automatically create a fully working cluster for distributed NLP. We call these scripts 'VM cluster from scratch', as they create and configure the required virtual machines. The scripts are publicly available and can be distributed freely.[9]

6 Experiments

This section describes a series of experiments made to test the effectiveness of the distributed cluster when processing large quantities of documents. We use two traditional metrics to judge the extent to which the parallelization is achieved: latency and throughput. Latency is the time taken to process an individual data set (document, sentence, and word), while 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[10]). Note that since multiple documents may be pipelined or processed in parallel, latency and throughput are not directly related. We also report speedup, the relative performance improvement when using the distributed architecture, and efficiency, the fraction of the optimal speedup achieved.

The processing chain is composed of the modules shown in Table 1. We have tested the distributed architecture on different settings and have measured the performance gain obtained by parallelizing the processing on each of them. In particular, we want to answer to the following research questions:

[9] https://github.com/ixa-ehu/vmc-from-scratch

[10] The elapsed time include the time spent initializing the NLP modules. However, the time spent initializing the VMs in the cluster is not considered.

Table 2. *English and Spanish datasets used in the experiments*

		Words			Sentences		
Name	Docs	N	μ	σ	N	μ	σ
en70K	70,000	16.7×10^6	238.1	191.9	688,692	9.8	8.5
en1K	1,000	874,799	874.8	86.4	35,536	35.5	10.0
en100	100	92,305	923.1	75.7	3,494	34.9	10.8
sp	1,873	989,168	528.1	383	28,331	15.1	11.6

For each dataset, we include the number of words and sentences, as well as the median (μ) and standard deviation (σ).

- How does the distributed architecture perform in both batch and streaming settings? What are the best configurations on each of these settings?
- Which is the gain of using non-linear topologies?

The distributed architecture 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 processed by the analyzing pipeline in a batch fashion. The first question focuses on analyzing the architecture's 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. In principle, non-linear topologies should decrease the overall document latency, as many processing stages are executed at the same time. We will test the use of non-linear topologies and measure its main benefits.

Table 2 shows the document sets used in our experiments. The documents are News articles from the car industry domain, and were collected within the NewsReader project. The en70K dataset contains small and medium documents, with 9.8 sentences on average. The en1K dataset is a subset of en70K comprised of larger documents, with a sentence average of 35.5. We used en1K for our main experiments in both the batch and streaming settings, and the whole en70K dataset to prove the feasibility of our distributed architecture when tested on a moderately large dataset. Finally, the en100 dataset is a subset of en1K, which we use as a development dataset to select the best configuration for batch processing (see below). Our experiments are mainly focused on small and medium News documents. We did experiment processing larger documents, but many times the processing failed due to some NLP module crashing (particularly, the semantic modules at the end of the chain). One way to overcome this problem and to make the processing more robust is to automatically split large documents into smaller ones. This would also improve the performance on non-linear systems, as shown in Section 6.2.

Regarding the hardware, the experiments were performed on a cluster composed of 8 nodes, each one with 16 CPU cores (E5-2640 2.00 GHz) and 128 GB RAM. All nodes have 6TB of local storage, and have access to a common HDD

Table 3. *Processing time for the en100 dataset on a system with 6 worker nodes. Times are measured in minutes. Best result in bold.*

Setting	Elapsed time	Gain (percent)
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**

following the NFS protocol. The nodes are connected by means of a Ethernet 1 GB bridge.

The document input and output queues are managed using *Apache Kafka*.[11] Kafka is a distributed, horizontally scalable, and fault-tolerant system and allows building real-time streaming data pipelines to get data between different systems or modules.

6.1 Batch processing

We first tried the topology on a batch setting. For this, we created a system with six different VMs deployed in the cluster. We tried several alternatives, regarding the number of module instances executed on each worker node:

- Baseline: A single instance of each NLP module.
- ALL_6: Six instances of each module.
- SRL_6: Six instances of the SRL module, while 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: The number of instances for each module is calculated according to equation (1).
- MONO: In this setting, each document is wholly analyzed on a single *worker* node, following a pipeline architecture. Note that this setting corresponds to having six 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 en100 dataset on the different settings. With no parallelization the documents need circa 4 h 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 an overall penalty in performance. Using the equation to calculate module instances alleviates this problem, but it is still five points worse than the SRL_6 setting.

[11] https://kafka.apache.org/

Table 4. *Several statistics regarding batch processing experiments*

Dataset	Elapsed time	Throughput	Proc. time	Speedup	Efficiency (percent)
en1K	719.4	1.38	6,480.22	9.01	90.1
en70K	38,159.47	1.83	31,328.13	8.21	82.1

Times are measured in minutes. Throughput measures the ratio between the number of documents and the elapsed time.

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 these results at hand, we processed the en1K and en70K datasets using a system with five VMs, each one having two host CPUs assigned. In total, the system contained 10 *worker* nodes (two nodes on each VM), and followed the MONO setting described above.

Table 4 shows the elapsed times for the document batches, including processing time, i.e., the sum of the times spent by each module and document. The en1K dataset take 6.5 min on average to be fully analyzed, although some documents caused some modules to spend a very large amount of time (the maximum time needed was 51.8 min). As for the en70K dataset, the average processing time of documents is lower (4.48 min, the maximum being 29.35 min).

Processing 70,000 documents on a single node would require more than 217 days (5 days for 1,000 documents); using the cluster, this elapsing time falls to 26 days (resp. 10 h), that is, we achieve a speedup of 8.2 (resp. 9.01), representing an efficiency of 82 percent (resp. 90.1 percent).

By analyzing the results to better understand the gap between the obtained efficiency and the theoretical maximum, we concluded the following:

- There are 10 *workers* in the cluster, deployed into five different machines (five VMs having two CPUs each). As a consequence, NLP modules compete for the shared resources on the same physical machine, which causes a drop in the performance.
- The last documents do not finalize simultaneously. Many *worker* nodes remain on an idle state until the last documents of the batch are analyzed. The larger these last documents are, the more the speedup drops.
- 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 may contain many big documents, while some others contain only small documents. A particular *worker* node pulls documents from a single partition, and therefore some nodes could be idle while some others

Table 5. *Processing times and latencies for streaming processing using linear and non-linear topologies*

Setting	Dataset	Proc. time	Latency (doc/sent/token)
Linear	en1K	4,620.90	$5.00/0.14/5.78\times10^{-3}$
Non linear	en1K	5,421.33	$2.78/0.08/3.13\times10^{-3}$

Times are measured in minutes.

have still many documents to consume. This factor is an characteristic of the software used to implement the input queue (*Apache Kafka*), and so, the results of these experiments suggest that we need to test alternative software packages for implementing the queue.

6.2 Streaming processing

Next, we want to try the distributed architecture on a streaming computing scenario. In this setting, documents arrive to the processing chain 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 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 1,000 documents within a time frame of 33 h.

We used the same system used in the batch processing, namely, ten *worker* nodes distributed in five VM (with two host CPUs assigned to each of them). The number of instances of the NLP modules are calculated following formula (1).

The first line of Table 5 shows the processing time and latencies for the en1K dataset. 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 batch processing 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 following a pipeline architecture, and thus the elapsed time required for a single document is more or less the same on both settings.

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.

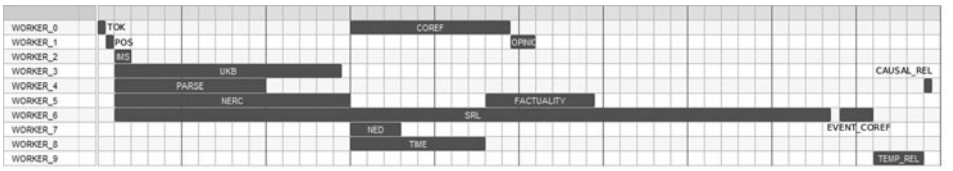

Fig. 7. Distribution in time of modules when processing one document using the non-linear topology.

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 processed.

Looking at Figure 6, we can see that there can be up to six modules running in parallel. Therefore, one would expect more than a 50 percent 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 percent of the overall processing time for a document, and, therefore, it represents a bottleneck of the whole 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 do 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, to 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 in the future.

Figure 7 shows the distribution in time of the NLP modules among worker nodes when processing one document using the non-linear topology in the distributed architecture. The system records the begin and end timestamps of each module, as well as the worker node where it was executed, and stores all this information in the NAF document header. The figure shows that as soon as the POS module ends in `WORKER_1`, several modules are ran in parallel, the last one being the causalRel module in `WORKER_4`.

6.2.1 Results on a Spanish dataset

In another experiment, we processed the sp Spanish dataset consisting of 1,800 documents by deploying a processing chain into the same system of ten *worker* nodes. The Spanish processing chain comprised ten modules, ranging from tokenization, POS tagging, constituency parsing, dependency parsing, NERC, and NED to coreference analysis, SRL, and recognition of time expressions. Again, the details of each of the modules can be found in Agerri *et al.* (2016).

Table 6. *Statistic for batch processing in the Spanish dataset*

Setting	Dataset	Docs	Elapsed time	Throughput	Proc. time	Speedup
Batch	sp	1,873	1,338.85	1.40	9,730.57	7.10

Throughput measures the ratio between the number of documents and the elapsed time.

The results, as depicted in Table 6, show a speedup of 7.27 and, overall, a similar throughput as in the English case.

7 Related work

Nowadays, there exist many NLP packages that provide easy and ready-to-use tools for linguistic analysis in many languages. For example, the Stanford CoreNLP[12] is a suite that provides many tools for processing English texts. Freeling (Padró and Stanilovsky 2012) provides multilingual processing for a number of languages, including Spanish and English. GATE (Cunningham 2002) is a well-known framework for text annotation that integrates many NLP processors using a common data model. The NLP components in GATE act as building blocks with which to define complex NLP pipelines. Mechanisms and tools to create and manually annotate corpora, visualize data, and evaluate NLP tools are also offered. Apache UIMA[13] is a middleware architecture for processing unstructured information, similar to GATE. The IXA pipes (Agerri *et al.* 2014) follow a similar modular architecture, where basic blocks are combined to build custom linguistic processing chains.

This paper extends previous work presented in Agerri *et al.* (2015). The previous work covered a basic system to process documents in a distributed environment following a batch model. However, the overall architecture presented in this paper is new. The input and output queue system, temporal MongoDB databases to reduce traffic, separation of boss and worker VM types, non-linear topologies to reduce latency, real-time streaming processing mode, automatic calculation of the number of instances of each NLP component depending on the resources available, and the option to create dedicated VMs are some of the concepts presented exclusively in this paper. Moreover, the system is now publicly available and can be distributed freely.

Many of the NLP processing solutions have been adapted to work with large quantities of data. GATE provides cloud services following a Platform-as-a-Service (PaaS) model via GateCloud (Tablan *et al.* 2012). GateCloud is a closed solution, and require users to adopt a subscribtion plan to use it. TextServer (Padró and Turmo 2015) and ILLINOISCLOUDNLP (Wu *et al.* 2014) offer a similar service, but in these cases the processing modules are preinstalled on the server following a Software-as-a-Service (SaaS) model. UIMA-AS (UIMA Asynchronous Scaleout)

[12] http://nlp.stanford.edu/software/corenlp.shtml
[13] http://uima.apache.org/

provides mechanisms for facilitating the deployment of UIMA applications into parallel systems. UIMA-AS was developed before the advent of modern big data approaches, and therefore it relies on custom built technologies for achieving parallelization. Maybe because of this, experiments on parallelization in UIMA applications showed that UIMA-AS does not scale well, particularly in streaming scenarios (Epstein *et al.* 2012).

The work presented in Derivière, Hamon and Nazarenko (2006) is one of the first attempts to develop an architecture to linguistically annotate web documents in a distributed environment using existing NLP tools. However, the system does not perform any load balancing or synchronization, as all the modules are executed in one node exclusively.

Cocytus (Evans, Asahara and Matsumoto 2008) is a system built on top of Inferno, a hosted operating system derived from Unix. Its main goals are offering a format-independent and parallel processing environment for NLP. Cocytus implements a MapReduce algorithm to distribute Inferno shell commands over the network nodes, but does not require NLP modules to follow the MapReduce paradigm. It uses a custom format, called TreePaths, for sharing annotations among NLP modules. TreePaths are mostly focused on representing syntactic trees, and are optimized to perform operations like s-expression searching. However, it is not clear how to represent more complex annotations, particularly those that deal with ambiguity issues.

KOSHIK (Exner and Nugues 2014) is a more recent batch-oriented unstructured natural language content processing framework, implemented on top of Apache Hadoop. It offers a layered annotation model suitable to process data in a distributed environment. To add a new processing tool to the framework the module is added as a library and implements a given function. It does not offer any solution to stream-processing needs. A similar work has been undertaken for Spanish (Otero *et al.* 2014). They run a pipeline composed of several linguistic processing modules in a distributed environment using Apache Hadoop. They basically install a copy of the pipeline in each server and distribute the input documents among them. The experiments presented in the paper reflect a speedup of 32 by using a cluster of 68 nodes. Note that this corresponds to an efficiency of 47.1 percent, far from the results obtained in our architecture.

Nesi, Pantaleo and Sanesi (2015) present another distributed framework for executing NLP-related tasks in a parallel environment. They developed an architecture in which any general GATE application could be executed in a Hadoop environment. In this case, they created a GATE application for extracting keywords and keyphrases from unstructured text. As a result, the speedup by using five nodes compared to a single-node setup was of 2.18, a modest efficiency of 43.6 percent.

Most the works described above focus exclusively on batch processing of large quantities of text, and do not consider the streaming approach we follow in our system. However, streaming processing is a demanding approach that is required nowadays for applications such as reputation management or technology forecasting, which are continuously searching for new data from sources like news or social media.

8 Conclusion and future work

In this paper, we have presented a scalable architecture for language processing that enables scaling up text analysis to process huge amounts of textual information. Unlike most works in the area, the architecture presented here is able to work on both batch and streaming scenarios.

NLP components are installed on virtual machines and deployed among cluster of nodes. We use the Storm infrastructure to integrate and orchestrate the NLP modules of the linguistic chains. The linguistic chain is defined on a declarative way, and the system automatically uses all the computing nodes to analyze the documents in parallel. The architecture enables defining non-linear topologies, where two modules can process the same document at the same time. However, the experiments showed that non-linear topologies are useful only on streaming scenarios, where nodes are usually idle until a new document enters into the processing chain.

The architecture allows the installation of any NLP component, as long as it uses NAF as a format for representing the annotations. NAF has been defined to allow many modules annotating the same document at the same time, and has been proved useful for distributed processing.

We used the IXA-pipes tools to create two fully processing chains for English and Spanish, comprising fifteen and ten modules, respectively, and obtained significant performance gains when deploying the modules in several computer nodes. We believe that among the free, commercially friendly, and multilingual toolkits, IXA-pipes performs competitively in terms of evaluation and performance. Besides, and together with the third-party tools added, IXA-pipes provides one of the most, if not the most, comprehensive ready-to-use NLP pipelines currently available under a free and permissive license such as Apache License 2.0.

Experiments with non-linear topologies have shown that some NLP modules act as bottlenecks and hinder achieving optimal latency rates. In the future, we want to experiment with different levels of granularity in the processing chain. For instance, a POS tagger works at sentence level, the WSD module works at paragraph level, whereas a coreference module works at 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, thus increasing the overall processing speed. Dividing the documents into small pieces of different granularities would also help making the system more robust, as it would allow the linguistic chain to process very large documents without crashes.

References

Agerri, R., Aldabe, I., Beloki, Z., Laparra, E., Rigau, G., Soroa, A., van Erp, M., Fokkens, A., Ilievski, F., Izquierdo, R., Morante, R., van Son, C., Vossen, P., and Minard, A.-L. 2016. Event detection, version 3. NewsReader Deliverable 4.2.3.

Agerri, R., Artola, X., Beloki, Z., Rigau, G., and Soroa, A. 2015. Big data for natural language processing: a streaming approach. *Knowledge-Based Systems* **79**: 36–42.

Agerri, R., Bermudez, J., and Rigau, G. 2014. IXA Pipeline: efficient and ready to use multilingual NLP tools. In *Proceedings of the 9th Language Resources and Evaluation Conference (LREC2014)*, Reykjavik, Iceland.

Agerri, R., and Rigau, G. (2016). Robust multilingual named entity recognition with shallow semi-supervised features. *Artificial Intelligence* **238**: 63–82.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. 2003. Scalable distributed stream processing. In *CIDR 2003 – First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA.

Cunningham, H. 2002. Gate, a general architecture for text engineering. *Computers and the Humanities* **36**(2): 223–54.

Dean, J., and Ghemawat, S. 2008. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* **51**(1): 107–13.

Derivière, J., Hamon, T., and Nazarenko, A. 2006. A scalable and distributed nlp architecture for web document annotation. In *Advances in Natural Language Processing*, pp. 56–67. Springer.

Epstein, E. A., Schor, M. I., Iyer, B. S., Lally, A., Brown, E. W., and Cwiklik, J. 2012. Making watson fast. *IBM Journal of Research and Development* **56**(3): 15.

Evans, N., Asahara, M., and Matsumoto, Y. 2008. Cocytus: parallel NLP over disparate data. *TAL* **49**(2): 271–93.

Exner, P., and Nugues, P. 2014. KOSHIK: a large-scale distributed computing framework for NLP. In *Proceedings of the 3rd International Conference on Pattern Recognition Applications and Methods*, pp. 463–70.

Fokkens, A., Soroa, A., Beloki, Z., Ockeloen, N., Rigau, G., van Hage, W. R., and Vossen, P. 2014. NAF and GAF: linking linguistic annotations. In *Proceedings of 10th Joint ACL/ISO Workshop on Interoperable Semantic Annotation (ISA-10)*.

Ide, N., Romary, L., and de La Clergerie, É. V. 2003. 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.

Nesi, P., Pantaleo, G., and Sanesi, G. 2015. A distributed framework for NLP-based keyword and keyphrase extraction from web pages and documents. In *Proceedings of the 21st International Conference on Distributed Multimedia Systems DMS '15*, Hyatt Regency.

Otero, G., Pichel, J., García, M., Abuín, J. M., and Fernández, T. 2014. Análisis morfosintáctico y clasificación de entidades nombradas en un entorno Big Data. *Procesamiento del Lenguaje Natural* **53**: 17–24.

Padró, L., and Stanilovsky, E. 2012. Freeling 3.0: towards wider multilinguality. In *Proceedings of the Language Resources and Evaluation Conference (LREC '12)*, Istanbul, Turkey, ELRA.

Padró, L., and Turmo, J. 2015. Textserver: cloud-based multilingual natural language processing. In *Proceedings of the IEEE International Conference on Data Mining Workshop (ICDMW)*, IEEE, pp. 1636–39.

Padró, L., and Turmo, J. 2015. Textserver: cloud-based multilingual natural language processing. In *Proceedings of the 15th IEEE International Conference on Data Mining Workshop (ICDMW '15)*, Atlantic City, USA, IEEE, pp. 1636–39.

Tablan, V., Roberts, I., Cunningham, H., and Bontcheva, K. 2012. GATECloud.net: a platform for large-scale, open-source text processing on the cloud. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical, and Engineering Sciences* **371**(1983).

Wu, H., Fei, Z., Dai, A., Sammons, M., Roth, D., and Mayhew, S. D. 2014. Illinoiscloudnlp: text analytics services in the cloud. In *Proceedings of International Conference on Language Resources and Evaluation (LREC)*, pp. 14–21.