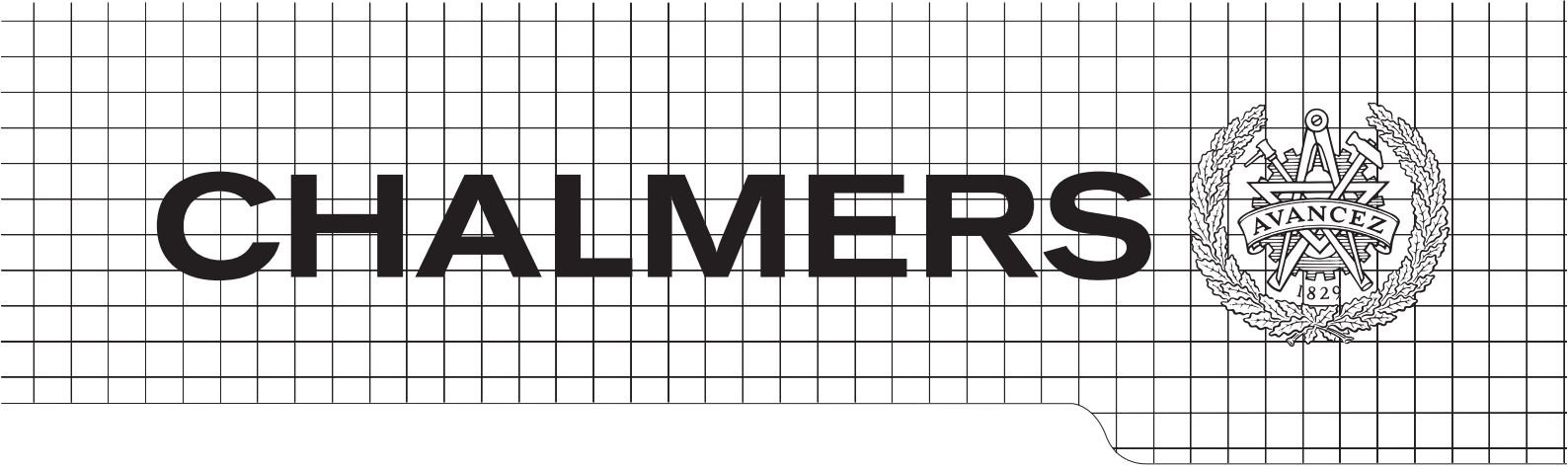

Evaluation of Document and Search Query Processing Frameworks

TOBIAS SVENSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science & Engineering
Göteborg, Sweden, March 2014

Evaluation of Document and Search Query Processing Frameworks

Tobias Svensson

Examiner: Bengt Nordström

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, March 2014

Abstract

As search becomes a vital cornerstone of any organization and as expectations and demands on findability and search steadily increase, there is a need for high-performance, scalable and simple Text Processing Frameworks to implement document processing solutions. Today, there are many open source solutions available to this end.

In this thesis, the processing frameworks GATE, UIMA, OpenPipeline, Hydra and Storm are analyzed and compared. We investigate the impact of parallelism and distribution on throughput and performance.

Additionally, the possibilities and demands of performing Natural Language Processing tasks on real-time search queries is analyzed. The feasibility of using the processing frameworks for this task is investigated and the results are discussed. Finally, recommendations are made for which kind of system to implement for different use cases and improvements to existing systems are suggested.

Contents

1

Introduction

Today, search is a vital feature for any organization. With the large amount of data that is generated every day in today's society, search is more important than ever. A good search infrastructure can allow employees to quickly find the information they need in a large information database generated from several sources such as documents, files, web pages, presentations, employee information and more.

To provide search solutions across such varied data sources, many Text Processing Frameworks have been developed over the last years. Today there exist many variations of such frameworks, each with its own architecture and advantages. To gain an understanding of what makes a Text Processing Framework efficient, an in-depth study is needed. Most have been developed for continuous processing of data streams, but is it also possible to use them in real-time applications with strict timing demands?

1.1 Background

To provide the demands described above, there is a need for software solutions to solve the search problem. Findwise is a Search consultancy company which provides a Search and Text Processing Architecture as shown in Figure 1.1.

In this architecture, the Content Collection and Refinement section processes documents and information to extract properties and search information. The processed and searchable data is then sent to the Search Core, which can be used by various user applications and interfaces to provide search functionality. The Content Refinement stage is where the text content of collected document is subject to Natural Language Processing, which is implemented by a Text Processing Framework (TPF).

The essence of a Text Processing Framework is the ability to analyze, augment and process textual information in a uniform fashion. Its core is a pipeline which consists of (pluggable) stages, where each of them is often dedicated to a single task (e.g. white

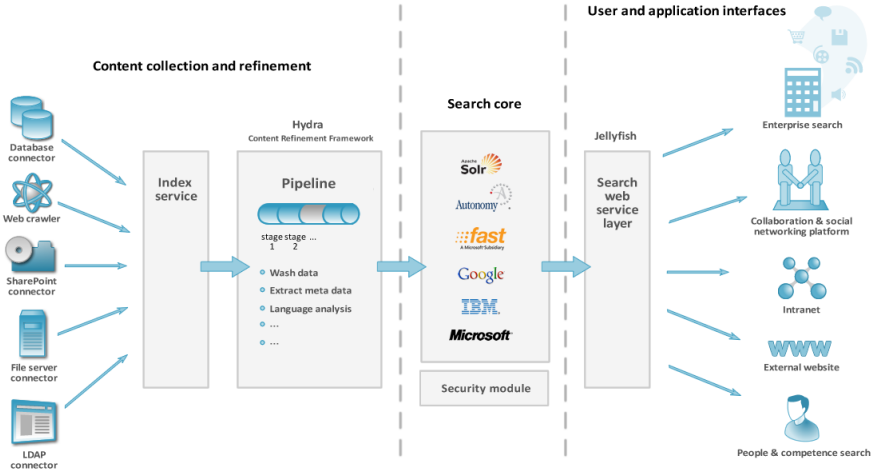

Figure 1.1: Findwise Search and Text Processing Architecture

space tokenization, copy one document field to another, extract date, etc.). In addition, the Processing Framework needs to be extendible - which means it can easily be adapted to new domains, languages or tasks, as well as scale to large number of document, work in a distributed environment and do not crash.

While there are a number of such frameworks available, two of them are well-known - the open source frameworks for document processing, *GATE* and *Apache UIMA*. In addition to these frameworks which have native support for linguistic analysis there exist a number of open source solutions with pipelines which have the potential to be extended with third-party components and even have built-in NLP capabilities. Two such examples are *OpenPipeline*, and *Hydra* which is currently being developed at Findwise. In addition there is also *Storm*, a distributed real-time computation system which can be used as a text processing framework.

1.2 Problem Description

The problem as described by Findwise is that there has been recent interest in new fields of usage for their Text Processing Frameworks which would impose stricter requirement for these frameworks. However, there is not enough knowledge about the characteristics of the available frameworks, which makes it hard to evaluate their current ability to meet these restrictions and to make informed decisions about future development efforts.

The task of this thesis is to evaluate these five pipelines for the use of document/text processing from the point of view of scalability, robustness, stability and performance. This can be done in the form of a Test Framework which tests these pipelines. There should be at least one or more tasks which will be performed by the pipelines (such as Named Entity Recognition, or data extraction, etc.). The thesis should take into consideration the technical requirements which Findwise has with respect to the processing

pipelines and verify which of them are fulfilled.

We are also interested in evaluating the possibility of using these pipelines for the purpose of Search Query Processing. There is an interest in performing more in-depth analysis of search queries to extract valuable information such as e.g. person names, organization names, locations file types or base forms of words. To perform these steps, the question is if the same NLP components which are used for Text Processing can be re-used. Therefore it is of interest to evaluate the performance of these pipelines and compare with the requirements imposed on Search Query processing software.

1.3 Purpose

The thesis will attempt to answer these research questions:

1. What are the differences between the Text Processing Frameworks in terms of scalability, robustness, stability and performance?

2. Can any of these Text Processing Frameworks also be used for search query processing?

To answer question 2, the thesis will need to collect requirements for the query processing use case.

2

Literature

2.1 Natural Language Processing

Natural Language Processing (NLP) is at the core of any search solution. It is a field which appears in a vast number of software applications, such as artificial intelligence, text or speech interpretation, language translation and most forms of human-computer interaction. NLP occurs in any situation where human text or speech, in any language, needs to be understood, interpreted or otherwise processed.

The field of NLP has its roots in the late 1940's. One of the earliest articles in the field is the influential 'Translation' article by Warren Weaver[1]. This article is can be seen as one of the first steps into the area of NLP, as it describes the possible use of powerful computers to solve the tasks before such computers existed. The article was followed rapidly by the publication of a well-known article by Alan Turing [2] describing the Turing Test, which attempts to determine a machines ability to exhibit intelligent behavior, a test which involves language processing.

NLP is a vast field of which only a subset is used in the topic of text processing for search applications, in which the task can be described as processing documents to generate terms and information which can be used to increase the quality of results to search queries.

In this section, we will describe the NLP steps we have used in this project. These are also the steps which are most commonly occurring in any NLP solution used for search.

2.1.1 Tokenization

Tokenization is usually the first step performed after retrieving the source text[3]. When tokenizing a text, the text is split into separated usable chunks called *tokens*. Tokens are usually single words. A first attempt at a basic tokenizer using regular-expressions might simply split an English text on whitespaces. More complicated solutions are required

to solve issues such as punctuations at the end of words or abbreviations such as *can't*. For some languages, such as Chinese, the task is much more tricky as the text is not separated into words by whitespaces.

Tokenization results can be different depending on the language and the NLP task being performed. For example, the English word *can't* may be split differently using different tokenizers. In OpenNLP, the *english.Tokenizer* stage splits *can* and *not* into two tokens for use with subsequent grammar parsing, whereas *SimpleTokenizer* stage simply splits on whitespaces and special characters. Other techniques might be applied in this step, such as *Case alterations* or *Stopword removals*.

2.1.2 Sentence splitting

Tokenization of a text, in certain situations, can have the disadvantage of obfuscating paragraphs and sentences. For example, in a standard article, a new paragraph is often introduced by adding a new line and indentation before the first word, both of which are lost in tokenization. Sentence Splitting divides the source text (or tokens, depending on the application used) into sentences, so that subsequent stages can produce more accurate results.[3]

In the standard OpenNLP pipeline, the Sentence Splitting stage occurs before the Tokenization stage.

2.1.3 Part-of-Speech tagging

Part-of-Speech (POS) tagging is the process of determining the POS for each token, such as if it is a verb, noun, adjective or otherwise[3]. Having this information available can increase the quality of results in later steps.[3] It can help distinguish between e.g. the adjective *nice* and the french city *Nice*. For the same reason it will also help with Named Entity Extraction, see 2.1.4.

2.1.4 Named Entity Recognition

Named Entity Recognition (NER) is the process of recognizing named entities such as *persons*, *organizations* or *locations*.[3]

Of the steps described in this section, NER is the first step which produces information than can be used in standard search applications to improve search results.

2.2 Application pipelines

In this report we will regularly refer to application *pipelines*. A *pipeline* within the software field generally refers to a series of processing stages, in which a subsequent stage depends on the results of previous stages, such that an item being processed through the pipeline must be fed through the stages in a predetermined order.

2.2.1 Synchronous vs Asynchronous pipelines

We define a *synchronous* pipeline to be a pipeline in which only one stage may be processing a job at any given point in time. In such a pipeline, a job must typically be fully processed before another job can be sent into the pipeline.

We define an *asynchronous* pipeline to be a pipeline in which more than one stage may be processing jobs at the same time. This can be due either to running stages in parallel using separate threads, or due to the pipeline supporting distributed execution across multiple computers, or nodes. In such a pipeline, typically there is no limit to the number of jobs that can currently be in the processing queue for any of the stages, thus a job can be submitted at any time regardless of the current state of the stages of the pipeline and the jobs within the pipeline.

2.2.2 Pipeline throughput

When looking at the performance of a pipeline, the most important metric is the *throughput* of the pipeline as a system. The throughput is defined by BusinessDictionary as 'Productivity of a machine, procedure, process or system over a unit period'[4]. In the context of NLP, we are interested in the average amount of documents processed per second. However, since documents can vary in length and complexity, we would require a different metric. Looking at the amount of bytes processed is not feasible as various data sources can have different compression and encoding, so we generalize this to the average amount of *characters* processed per second.

The throughput τ_S of a pipeline stage S can be defined as [5]:

$$\tau_S = \frac{C}{T_S} \tag{2.1}$$

where C is the total amount of characters processed and T_S is the total amount of time used for processing for stage S.

The total throughput τ for a pipeline with N stages is determined by its slowest stage[5]:

$$\tau = MIN\left(\tau_0, \tau_1, ..., \tau_N\right) \tag{2.2}$$

From this we can gather that the slowest stage in a pipeline will bottleneck the performance of the rest of the pipeline.

Note also that these definitions assume that the stages are running asynchronously. If a set of sequential stages are synchronous, i.e. at any given time a maximum of one of the stages may be processing, this set of stages may be considered as a single stage for the purposes of these definitions.

2.2.3 Parallelism and throughput

Using the formulas in the previous chapter, we can derive formulas for throughput of pipelines with parallel instances of stages.

By increasing the number of parallel instances of the slowest stages, we can increase the total throughput of the system. If a stage S has throughput τ_S and we run P parallel instances of it, then the maximum throughput through this part of the pipeline is $P\tau_S$. We could define the increase in throughput as:

$$\frac{P\tau_S}{\tau_S} = I \tag{2.3}$$

Any method of communication between and maintenance of parallel instances will add some overhead to the system as a whole. But, as long as throughput of document transmission between the stages, τ_O, is larger than $P\tau_S$, the increased throughput of the system is not affected (even though the total workload for processing a single document may be substantially higher):

$$\frac{MIN(P\tau_S, \tau_O)}{\tau_S} = I \tag{2.4}$$

Note that this does not account for other consequences of the increased resource usage. If the maximum I/O speed of a memory or disk device is reached, or problems such as disk trashing or memory swapping occur, the system can slow down and Tmax may increase enough to decrease the total throughput.

We now revise the formulas in 2.2.2 to account for pipelines with stages running in parallel, but excluding performance drawbacks of the increased number of processes.

When running P parallel instances of stage S, the total computation time of the stage as a whole is determined by the parallel instance with the longest computation time T, so from (2.1) we get:

$$\tau_{PS} = \frac{C}{MAX\left(T_{S1}, T_{S2}, ..., T_{SP}\right)} \tag{2.5}$$

As the execution time of the stage increases, the difference in computation time between stages proportional to the total computation time will decrease. We can thus generalize (2.5) for long execution times. Recall from equation 2.1 that T_S is the total computation time of stage S. Under parallel execution we can use the approximation:

$$\tau_{PS} = \frac{C}{T_S/P} = P\tau_S \tag{2.6}$$

Let P_X be the number of parallel instances of stage X. From (2.2) we get the following expression for the throughput of a parallel pipeline with N stages:

$$\tau = MIN\left(P_0\tau_0, P_1\tau_1, ..., P_N\tau_N\right) \tag{2.7}$$

2.3 Real-time & Stream Processing

In Computer Science, two commonly used terms for describing processing systems are *Real-time processing systems* and *Stream processing system*

A *Real-time processing system* refers to a system where it is critical that a result is delivered within a certain time-frame. Simply put, a real-time system has a strict timing requirement on the time from submission of a task to the delivery of its result.

A *Stream processing system* refers to a system which operates on a data input with the following properties[6]:

- The size of input data is unbounded
- Input data may become available at any time
- The system processes input data continuously over a long period of time

In other words, a Stream processing system operates on a *data stream* with high volumes of data. Due to the nature of such a system, it is not possible to put any strict timing demands, as the volume of data may vary over time. Instead, important factors for such a system are availability, reliability and scalability.

3

Method

3.1 Text Processing Frameworks

In this section we will describe the Text Processing Framework (TPF) applications which have been studied in this project.

Common aspects of all chosen TPS is that they are open-source and are wholly or partially developed in Java.

3.1.1 GATE

GATE[1] (General Architecture for Text Engineering) is a TPF which began development as part of an R&D program started in 1995[7]. Since then, GATE has been continuously in development. GATE has seen use in large corporations, research programs and undergraduate studies. Many third-party plugins have been developed for GATE.

GATE comes with an IDE which allows easy configuration of a text processing pipeline by combining and configuring available plugins.

3.1.2 Apache UIMA

The Apache Unstructured Information Management Architecture (UIMA)[2] is a component software architecture designed to enable analysis of unstructured information[8]. It is one of the most widely used publicly available NLP solutions, being the first TPF to become an OASIS standard[9]. In 2011 IBM used the Watson computer to compete in an subsequently win a competition of *Jeopardy!*. Watson used UIMA to process real time

[1]GATE is available at http://gate.ac.uk/. The version used in this project is Release 7.1 (Nov 30th 2012).

[2]UIMA is available at http://uima.apache.org/. The version used in this project is 2.4.0 (Nov 15th 2012).

information[10]. A wide selection of applications are available and many search engines such as Solr have support for Apache UIMA.

3.1.3 OpenPipeline

OpenPipeline[3] is an open-source software for crawling, parsing and analyzing documents, which can be used to tie together incomplete NLP solutions. Development began near the beginning of 2008[11]. OpenPipeline comes packaged with some basic stages and a GUI for creating a text processing pipeline from available stages.

3.1.4 Storm

Storm[4] is a distributed real-time computation system suitable for any kind of stream processing. It is an open-source project developed by Nathan Marz and is used within many companies and projects, including Twitter and Groupon. While Storm is developed in Java, the API can be extended to any programming language.

Dependencies

To enable its distributed behavior Storm uses *Zookeeper*, a distributed coordination service useful for maintaining distributed applications. The Zookeeper project began as a subproject of Hadoop, but has now branched off and become a stand-alone open source software product. Storm also uses *Nimbus*, a toolkit providing common cloud computing features.

3.1.5 Hydra

Hydra[5] is a open-source, distributed processing framework for search solutions, which has been in development at Findwise since 2012.

The goals of the Hydra project are to provide a distributable, stable, scalable and reliable solution for text processing. Each stage runs in a separate JVM and stages which crash or enter faulty states will be restarted automatically to a working state.

Dependencies

Hydra uses *MongoDB*, a NoSQL[12] database with high performance and built-in distribution support, to store data and for communication between the different nodes of a Hydra cluster.

[3]OpenPipeline is available at http://openpipeline.com/. The version used in this project is 0.8.4.
[4]Storm is available at http://storm-project.net/. The version used in this project is 0.8.4.
[5]Hydra is available at GitHub. (https://github.com/Findwise/Hydra)

3.2 Apache Solr

The search engine used in this project is Apache Solr, an open-source enterprise search server based on the Lucene Java search library. Solr is designed to be scalable and fault tolerant, with an extensive plugin architecture for in-depth customization of most of its features[13].

Communication with Solr is provided via a REST interface, see 4.1.1. Documents are submitted to Solr as a set of fields. Each field can contain any value, being a number or string, or a list of values. When searching, a selection algorithm compares the search terms to the fields of each documents and assigns each document a score based on how accurately the search term matches the document. The documents with the highest scores will be returned as a result to the search query.

3.3 Research

Prior to initiating any design work, the fields of NLP and testing which were expected to be applied to the projects were researched.

For the field on NLP, we focused on researching the standard applications and methodologies that are common to most NLP solutions. The goal of the research was to identify which NLP techniques (stages) would be used during testing and how to implement these efficiently and fairly between the various TPF.

For the field of testing, research focused primarily on which test metrics and methodologies would yield the most interesting and useful results. Existing scheduler implementations were examined, such as the scheduler for the Linux operating system[14], to gain an understanding of which the most critical metrics are. Resource usage collection strategies were researched specifically for the Linux operating system.

3.4 Requirements

Due to the nature of the testing methodology of this project, it was hard to define any requirements to work towards as we were more interested in the comparative differences between the TPF. For this reason it was decided to leave out requirements for most tests.

When testing a TPF's feasibility for use in search query processing, a time limit was necessary to define the maximum time before a response on the search query from the server. After taking into account perceived factors such as usability, quality of the results and perceived UI responsiveness, we decided to use the requirement of one second from the time the search query was sent to the time the response is received. This requested limit on responsiveness would likely change between products, use cases and environments.

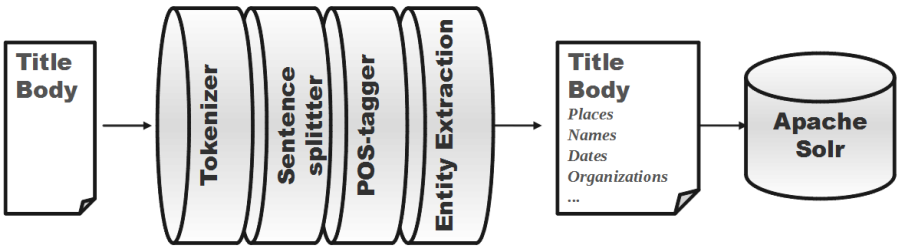

Figure 3.1: Target Pipeline

3.5 Test methodology

3.5.1 Target pipeline

In order to be able to fairly compare the performance of the TPF's, it is vital that the pipeline and pipeline stages used are exactly the same for each TPF. An overview of the target pipeline can be seen in figure 3.1. A document, consisting of a *title* and a *body*, is sent into the pipeline which consists of four NLP stages, see 2.1. The pipeline yields annotations as results, which are exported to Apache Solr, see 3.2.

3.5.2 OpenNLP

The stages in all the TPF need to be as similar as possible, so to this end we have chosen the Apache OpenNLP Library as the NLP solution for our processing stages.

OpenNLP is a set of NLP stages available for free use. It is widely supported by most TPF which is one of the main reasons that OpenNLP was chosen as the NLP solution. OpenNLP uses machine learning algorithms to perform common NLP tasks. Each stage requires a *model* which needs to be generated in advance, by training the machine learning algorithm with example data. There are ready-made models for most stages available for download, which are what we use in our target pipeline.

Although interestingly, OpenNLP does not require the POS results for the NER stage, meaning the POS tags are not used in any of the subsequent stage in our target pipeline, the POS stage is still included in our target pipeline as it is a very common processing stage.

3.5.3 Metrics

Throughput

We measure the *throughput* of a pipeline as the average amount of documents that finish execution per second. The throughput is an indicator of how efficient the TPF is at feeding documents through the various stages of the pipeline. The higher the throughput, the faster the TPF can, on average, finish processing a set of documents.

End-to-end time

We measure the *end-to-end time*, or response time, of a job as the time taken from sending of the job to the pipeline until the finished results are retrieved back from the pipeline. This metric is an indicator on how fast one can expect to get a result when submitting a document to the NLP framework. It is also an indicator of fairness of the pipeline and its stability in efficiency, since if the end-to-end times vary wildly for jobs of the same size under the same load there may be a problem with either of those two factors.

Starvation

We look for *starvation* issues in a pipeline by measuring the delay times until a job is sent to the next stage. In a TPF with fair scheduling, jobs are processed in the order they enter the waiting queue, which implies equal waiting times for all jobs depending on the length of the queue. In a TPF which does not have fair scheduling, unlucky jobs may be stuck waiting for a very long time.

Resource usage

An important aspect for the performance of each TPF is the amount of computer resources they use during processing. To measure this, the resource usage of the system as a whole will be measured per second. The metrics collected will be CPU usage, memory and swap file usage and maximum available space for each, I/O operations for each disk device, and I/O operations for each network device.

For more accurate and informative results it would be preferable to measure only the resource usage of the specific processes. As each application to be tested is a Java application, this could be performed by measuring the resource usage of each respective JVM. However, while this is fine for those involved TPF which only use one Java process, it is a much more difficult task to do this for Storm and Hydra, as these run several JVM's which may be destroyed and recreated under certain conditions. Thus it was determined that the resource usage of the system as a whole would give a sufficiently detailed overview of the performance of each TDF, such that the extra effort of measuring usages for each JVM was not justified.

3.5.4 Test methodologies

Performance test

The Performance Test is a stress test in which documents are constantly sent to the TPF at high loads such that the TPF is operating at maximum efficiency. This test is designed to test the maximum throughput and efficiency of the TPF, while also exposing their behavior under heavy load. The test is performed by pushing jobs to the TPF until a specified amount of jobs are either being processed, or in a buffer of incoming jobs waiting to be processed, depending on if the target pipeline is synchronous or asynchronous.

Query processing test

We are interested in the scenario in which a certain number of queries is submitted at the same time. The *Query Burst Test* was developed to investigate this case. The test submits a *pack* of documents simultaneously in bursts and verifies the end-to-end time for each submitted query. The size of the document pack is increased after each submission until a failure is detected, in which case the pack size is decreased again, where a failure is defined as the response being received after the specified time limit as discussed in 3.4.

The test makes smaller increments and decrements over time, eventually stabilizing at a constant pack size which we use to determine the search query burst that the TPF can be expected to handle.

3.5.5 Distribution

Running Hydra in a distributed fashion is as simple as installing Hydra on each server with the same setup as on a standalone setup, then configuring each installation to use the same MongoDB host address. No further configuration changes are necessary. Hydra uses locks in the MongoDB database to ensure that all actions are synchronized between the various Hydra instances.

To run Storm in a distributed fashion, Storm and its dependencies must be installed on each machine. One machine acts as the master server, which is running a master instance of ZooKeeper, and also a Nimbus host. This server also runs ActiveMQ. The other two servers need only run storm and a slave instance of ZooKeeper.

3.5.6 Data sources

In the interest of measuring the effect the size of the submitted content has on the TPF, we run all tests several times with different data sources each time. The goal is to have a large and a small data source for performance tests, and a data source for query tests. The data sources used in the tests are:

- *Search query logs* - Logs from search queries made by users on production environments. Used for query testing.
- *Twitter feeds* - Twitter feeds were collected for some time and stored. Since each Twitter post has a 140 character limit, these are acceptable as small data sources. Used for performance tests.
- *Wikipedia articles* - Wikipedia articles between 15kb to 50kb as text files, as large data source. Used for performance tests.

4

System design

4.1 Network communication

When executing the tests, the test application itself should not impact the performance of the TPF which is being tested. The test application is therefore run on the local computer, while the TPF is run remotely on a server. This also mirrors a standard production environment. A problem which needed to be solved was how to actually send the documents over the network. The only TPF involved in the tests which provides this feature by default is Hydra. For other TPF, this had to be developed manually.

4.1.1 REST

REpresentational State Transfer (REST) is a standardized design design pattern for HTTP communication and resource sharing.

Since Gate, UIMA and OpenPipeline do not have any built-in network capabilities, each of these needs to be managed by a service capable of communicating with the test framework across the network. The method chosen for this has been to develop a REST service for each of these framework, which runs inside Tomcat[1]. REST was chosen due to the ease of interaction with such a system - requests can be made and data can be analyzed using most modern web browsers or any programming language capable of making HTTP requests.

4.1.2 ActiveMQ

For Storm, we decided to use an available component which allows sending and receiving documents using the message broker Apache ActiveMQ, an open-source JMS message

[1]Tomcat is a webserver application developed in Java, which can be installed on any Java-compatible computer to easily set up an HTTP webserver.

broker. A message broker is a service which receives messages to specified queues from *producers* and sends them to one of the available *consumers*.

4.2 Resource usage

Resource usage statistics are collected from Linux by reading the content of the /proc folder once per second and printing it to a log file. /proc is a virtual file system containing information on the hardware and processes of the Linux machine[15]. The log file is generated by parsing this date once per second and appending the information to a log file. This log file is then retrieved using SCP at the end of the test and saved together with the rest of the test results.

In the following subsections we will detail how each metric is collected.

4.2.1 CPU load

The path */proc/stat* contains various statistics about the system, including the amount of time the processor cores have spent in processing and idle states respectively, detailed for each core and summarized for all cores collectively. The increase in these values are sampled each second to calculate the CPU usage in percentage per second.

4.2.2 Memory and Swap file usage

Memory and swap file usage is read from the path */proc/meminfo* which contains detailed statistics about the current state of both, in kB.

4.2.3 Disk I/O

While /proc contains all information on current disk usage, the data detailing how much data has been read or written is slightly complicated to parse. The Linux application *iostat* summarizes the information neatly, thus the disk I/O statistics are read from the output of the command *iostat -d*. An example output looks like:

```
Linux 3.2.0-45-generic (vmpipetest)    06/04/2013    _x86_64_    (2 CPU)

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda              26.58       277.27       625.46    1195019    2695720
```

.

4.2.4 Network interface usage

The amount of bytes which have been transmitted and received through each network interface is collected from the following paths, in this case for the device *eth0*:

```
/sys/class/net/eth0/statistics/tx_bytes
/sys/class/net/eth0/statistics/rx_bytes
```

These files contain the bytes transmitted and received respectively as an integer.

4.3 Vagrant

Deploying all systems on a server can be a time consuming task. All systems and dependencies need to be installed and tested for each server. In addition, each time a component is updated it needs to be upgraded on each node.

To streamline the task of deployment we use the Vagrant deployment tool, which automates the task of 'rolling up' a new VM and configuring services and dependencies on it. With Vagrant, the VM configuration is specified in a Ruby file. A provisioning system is used where specific services and their settings are specified. Vagrant will then create and provision the entire VM with a single command line call. This greatly simplifies the testing process.

In the test framework, a Vagrant VM is specified which contains Gate, UIMA, OpenPipeline, Hydra, Storm and their required applications and dependencies. This is then deployed on each server used for testing.

4.4 Implementation

While an API has been developed for the test framework such that it can be reused for each framework, the implementation of the text processing varies greatly. The facilities available for extracting processing information from each document also vary from detailed to non-existent.

4.4.1 GATE

The *Application* containing the GATE pipeline is predefined using the GATE Developers Interface, the GUI accompanying GATE. When the test framework processing is initiated, the application is launched. To submit documents through the pipeline, a Corpus is created at initialization. Documents are converted to GATE's Document class and added to this Corpus which is then submitted through the Application. After this all the finished documents are retrieved and the Corpus is cleared for the next run. Since GATE contains no Solr export functionality, document export to Solr is delegated to a separate thread running in parallel with the execution thread.

After the pipeline has finished processing a document, the result is exported to Solr. To do this, all annotations need to be iterated through. Since annotations specify their span as start and end positions in the source text but do not contain the source string for the annotation itself, the source text must be kept in memory until the export is complete and each annotation string must be extracted from the source text using its span. It is possible to optimize this process by requesting only annotations of certain types.

4.4.2 UIMA

A Processing Engine Archive (PEAR) file containing the UIMA pipeline is generated client-side and packaged with the UIMA Service WAR to be deployed with Vagrant. The PEAR file is loaded when the server is initialized, which in turn loads all the pipeline stages into memory. Documents are submitted as plain-text one at a time. Since UIMA contains no Solr export functionality, document export to Solr is delegated to a separate thread running in parallel with the execution thread.

Annotations contain their source text which makes Solr export trivial.

4.4.3 OpenPipeline

The pipeline is stored in an XML file which is loaded by OpenPipeline on server startup. Each stage is loaded when the pipeline is 'started', which occurs automatically when the first item is sent through the pipeline.

Submission to Solr occurs as its own stage at the end of the pipeline.

4.4.4 Hydra

Hydra regularly polls its MongoDB server to look for new documents to process, and to look for updates to existing documents. To submit a document to Hydra, it is necessary to write the document directly to the MongoDB database. The document is then fed through the stages of the pipeline until it reaches an OutputStage, which in this case is the SolrOutputStage. When the document has finished its processing path through all relevant stages in the pipeline, it is moved to a separate MongoDB collection *oldDocuments*. The test framework scans for finished documents by polling this collection for updates. The finished document contains information on when each stage fetched the document and when each stage finished processing (touched) each document.

4.4.5 Storm

In Storm, a topology is built using Bolts and Spouts. A *Spout* generates data *tuples*. These tuples are then sent to *Bolts* which perform processing on the tuples and may emit further tuples after processing is finished. A spout can send tuples to number of bolts and a bolt can receive tuples from any number of spouts, making design of the topologies flexible.

The Storm topology is pre-compiled into a JAR file, which is uploaded to the server-side Storm client when the server is deployed.

With Storm, the choice of bolts affects the performance of the system. Each bolt which is added to a topology increases the overhead load and also increases waiting times. However having more bolts further pipelines the processing path which may result in an increase in throughput.

To communicate with Storm, the message broker ActiveMQ is used. ActiveMQ runs as a service on one of the deployed nodes. A spout is defined in the storm topology which reads messages containing documents to process from the input queue *storm_in*. At the

end of the topology, an output bolt writes messages containing information about a processed document, which are sent to the output queue *storm_out*. To send documents to Storm, the test framework converts the documents to messages and sends them to *storm_in*, then subscribes to the *storm_out* queue to receive finished jobs.

4.5 Generating results

At the end of the test, the test data is saved to disk as JSON data. From this data, graphs and statistics can be generated by a Java script. To generate charts, JFreeChart was used. Finally, the generated statistics are displayed in an HTML report page.

5

Results & Analysis

In this chapter we will detail and discuss the results gathered from our tests.

5.1 Test results

5.1.1 Throughput

These chart shows the average throughput, in documents per second, for each pipeline tested.

Twitter feeds

The throughput charts from the Twitter feed tests can be seen in Figure 5.1.

Gate and UIMA have roughly the same performance, although UIMA slightly outperforms Gate.

OpenPipeline is an interesting case. Since OpenPipeline has functionality for running several pipelines synchronously, this feature was tested by breaking out the Solr output stage and running it in a separate pipeline. As we can see, with a single pipeline, with small pieces of content, the throughput suffers, see 5.2.1. However, running more than one pipeline alleviates the issue and brings OpenPipeline up to roughly the same throughput as UIMA. Refer to the formulas in 2.2.3 for why throughput increases when one pipeline is divided into several parallel pipelines.

Hydra's communication overhead issues become readily apparent in this test with many small pieces of content as the throughput is very low, see 5.2.3.

Storm by default features the option of running several instances of the pipeline in parallel. Thus Storm was tested with one and two pipelines respectively. When running a dual pipeline, the throughput is doubled as is to be expected, see 2.2.3.

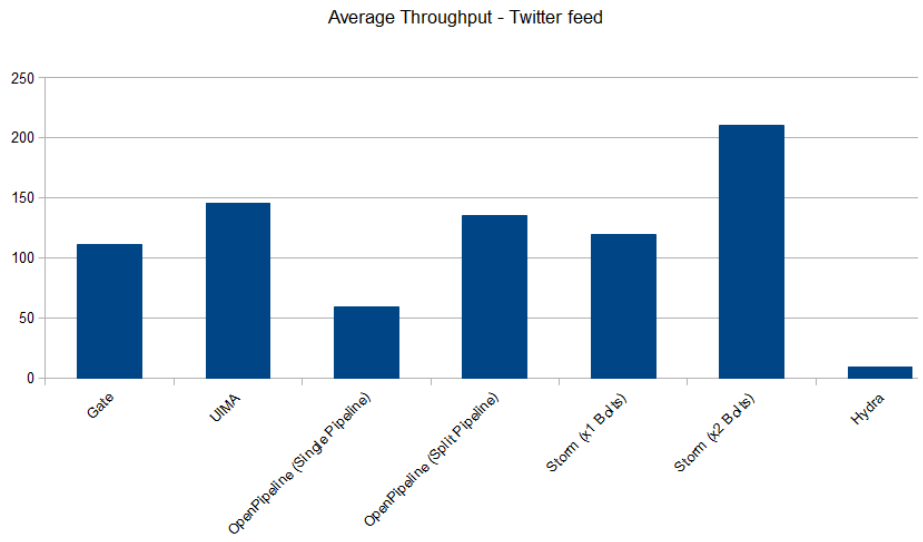

Figure 5.1: Average Throughput during tests with Twitter data

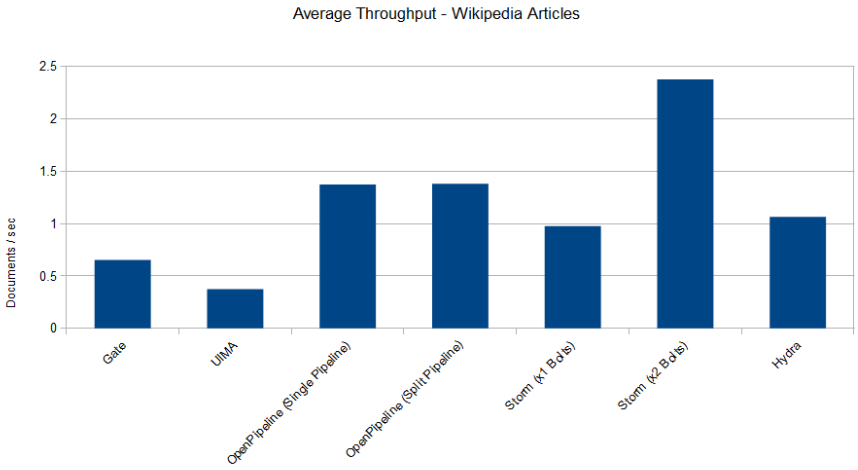

Figure 5.2: Average Throughput during tests with Wikipedia articles

Wikipedia articles

The throughput charts from the Wikipedia article tests can be seen in Figure 5.2. This chart shows interesting results when compared to Figure 5.1.

GATE and UIMA see slower performance as the document size increases, see 5.2.2.

For Hydra, the communication overhead has much less impact as there are fewer jobs to be communicated and thus fewer instances of communication overhead impact.

Finally, the OpenPipeline scheduling issues no longer have any significant effect, as

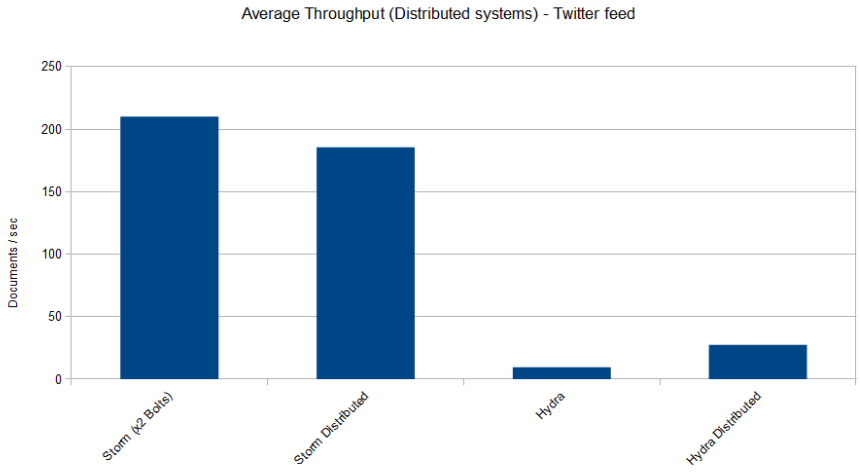

Figure 5.3: Average throughput, Twitter data, with and without distribution

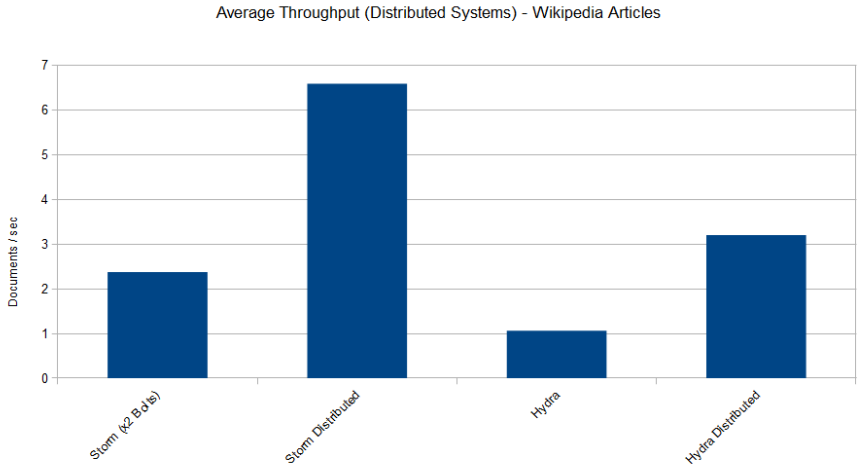

Figure 5.4: Average throughput, Wikipedia articles, with and without distribution

there are fewer jobs to schedule and thus less impact from scheduling issues.

Distributed tests

As can be seen in Figures 5.3 and 5.4, when running Hydra in a distributed fashion, the result is as expected in that the throughput increases according to the amount of nodes added. In these distributions tests three nodes were used, so the throughput is tripled.

For Storm, the effect is interesting. For the Wikipedia tests, Figure 5.4, the throughput is tripled during distribution. But for the twitter test, Figure 5.3, the throughput is *reduced* from 210 documents/s to 185 documents/s. The added overhead from com-

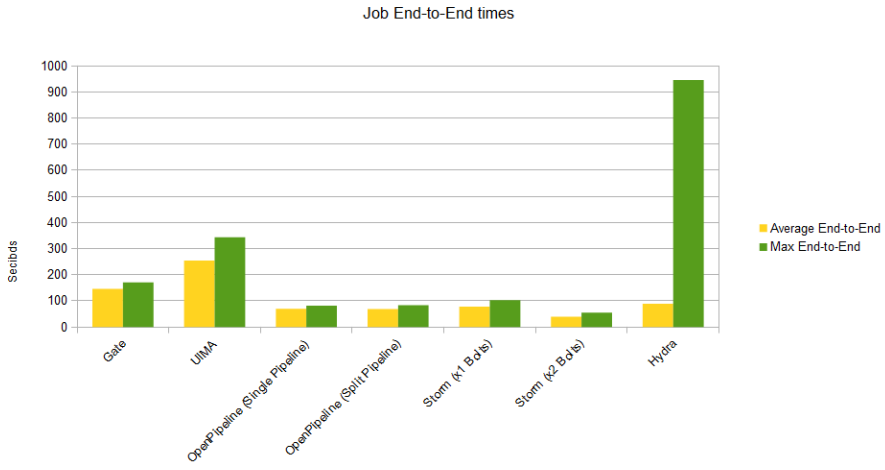

Figure 5.5: End-to-end times, average and worst-case

munication across several Storm nodes seems to be large enough to hinder performance when large volumes of small jobs are sent.

5.1.2 End-to-end

Figure 5.5 shows the end-to-end times of the performance test with Wikipedia data. The test was configured to keep 100 jobs running simultaneously, and this number affects the end-to-end times linearly. This chart is mostly the inverse of the Throughput chart with Wikipedia articles, where higher throughput equals lower end-to-end times. But there is one interesting outlier, and that is the Hydra worst-case end-to-end time. This outlier happens because some jobs encounter starvation due to the scheduling issues of Hydra, see 5.2.1.

5.1.3 Starvation

Starvation was measured for all pipelines during all tests. GATE, UIMA, OpenPipeline and Storm show no issues with starvation, while Hydra does.

The effect of starvation in Hydra can be seen most clearly for the NER stage. This stage has the longest average processing times and is thus the bottleneck stage. This means that it will be stage with the largest set of waiting incoming jobs, as it will receive jobs faster than it can output them.

Each data sample in Figure 5.6 is a document. The time axis represents how long the document waited from the moment it was output from the previous stage to the moment it began processing in the next stage. What we can see in this graph is that, while most jobs get processed within a few seconds, some unlucky jobs must wait several minutes to get processed. The order that the jobs are submitted to the waiting queue

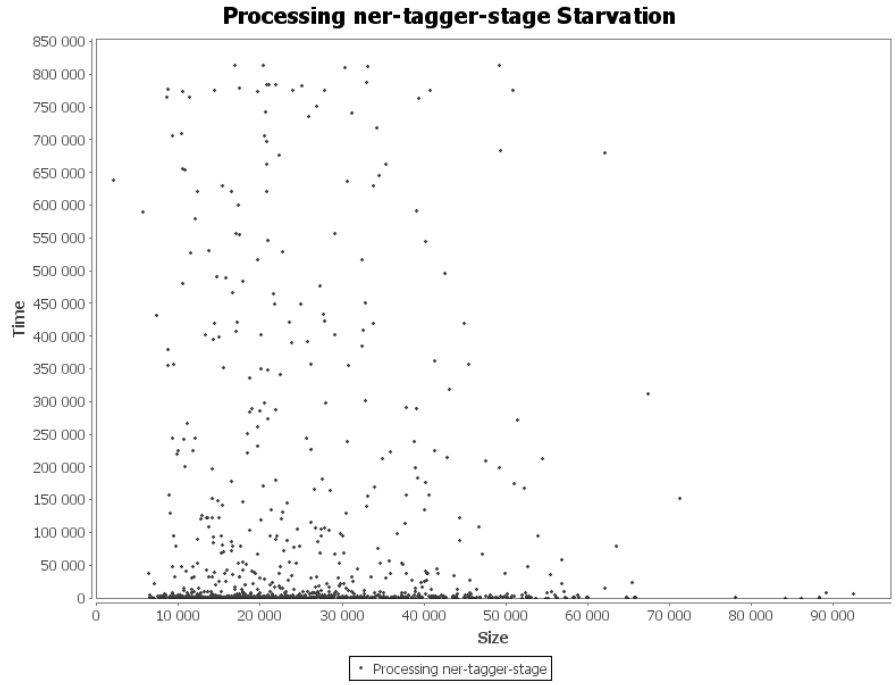

Figure 5.6: Starvation before the bottleneck stage in Hydra

is not considered when polling for unfinished documents, therefore this starvation issue occurs.

5.1.4 Resource usage

Resource usage was collected from each node during the tests with the help of NodePerf. Charts for the CPU Load are available in Appendix C. Charts were generated for other resource usage metrics, however they did not show any findings significant to the purpose of this report and thus have not been included.

CPU Load

The nodes used during tests were outfitted with dual-core processors, see Appendix A.2. As can be seen in the CPU charts, GATE and UIMA both average slightly above 50% CPU usage, fully utilizing one core. Storm averages above 90% as stages run in separate threads, thus all available processor resources can be utilized. Hydra shows lower CPU usage, averaging around 15% CPU load, due to delays caused by polling behavior as described in 5.2.3. OpenPipeline also shows some idle CPU time caused by scheduling, see 5.2.1.

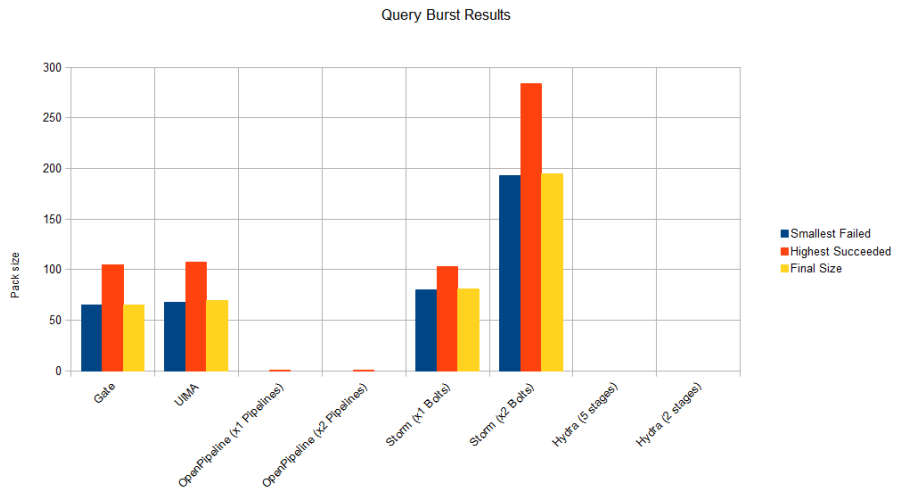

Figure 5.7: Results from the Query Burst tests

Other resource metrics

Memory usage charts showed increased memory usage for each instance of an OpenNLP stage opened and for each document currently being processed. However no significant difference could be noticed between the different TPF.

Naturally, distributed frameworks generate more network traffic. There were no noticeable difference between Storm and Hydra in this regard, as most of the traffic generated during tests consisted of the document contents.

Finally, Disk IO data showed no noteworthy impact on performance or any noteworthy difference between the TPF.

5.1.5 Query processing test

The results from the Query Burst Test described in 3.5.4 can be seen in Figure 5.7.

The results show the final stabilized packet size, and also the maximum packet size that was successfully sent plus the minimum packet size that experienced a failure. The packets had an end-to-end time requirement of 1 second.

The first thing which is immediately apparent from the results is that OpenPipeline and Hydra are not able to process queries within the time requirement at all.

For Hydra this is due to communication overhead as described in section 5.2.3, which means that even for a simple pipeline the query will not process on time.

For OpenPipeline the cause is a little more intricate. When sending a single query, the time requirement does not always fail. After studying the response times, it was discovered that the end-to-end time varies around the 1-second mark, however the total processing time is shorter than 200ms in all cases. The cause for this seems to be the Quartz scheduler as described in section 5.2.1.

Distributed tests

Remarkably, neither Storm nor Hydra succeed the Query Burst Test when run in a distributed fashion. For Hydra, distribution does not alleviate its communication overhead. For Storm however, the test does not succeed because when run in a distributed fashion, end-to-end times for jobs increase to approximately 20 seconds. The cause for this behavior could not be ascertained, but for the purposes of this report we consider this delay to be a communication overhead. Specifically, there seemed to be some waiting time, averaging 10 seconds, before jobs were sent to another node.

5.2 Performance impact factors

5.2.1 Scheduling

Ideally, to minimize the processing time on a set of documents and to maximize efficiency, each stage in a pipeline should always be working on processing a document. It is the task of the scheduling implementation for each TPF to ensure that this is the case. During testing, we discovered performance differences caused by the different scheduling implementations.

GATE and UIMA have no schedulers, as they process documents one-at-a-time. Thus it is up to the developer to submit the next document quickly.

OpenPipeline uses the Quartz scheduler and is able to execute several pipelines in parallel. Quartz schedules jobs by defining triggers which will start a job either at a specified time with millisecond precision, or with a certain periodicity, such as running a job once per second. By analyzing the CPU usage charts it was discovered that OpenPipeline goes idle for short periods of time even though documents are submitted continuously. This suggests that the scheduler is causing slight delays between documents by causing short periods during which the pipeline is idle.

Storm uses message passing to communicate the state of each stage, see 5.2.3. While analyzing Storm, all stages of the topology were constantly running at maximum efficiency, implying no idle times between tuples and an efficient scheduling behavior.

Hydra uses a polling behavior to discover document states, which causes idle times as described in 5.2.3.

5.2.2 Execution overhead

In this section we examine processing and communication overhead on execution of a document through each pipeline.

Hydra

For Hydra, we noticed a flat delay time for each stage. When a stage asks for a new document, the Hydra core must find and serialize the next document and send it to the stage which must then deserialize it. Currently the serialization format is JSON. When a stage has finished processing the document, these serializations are again repeated

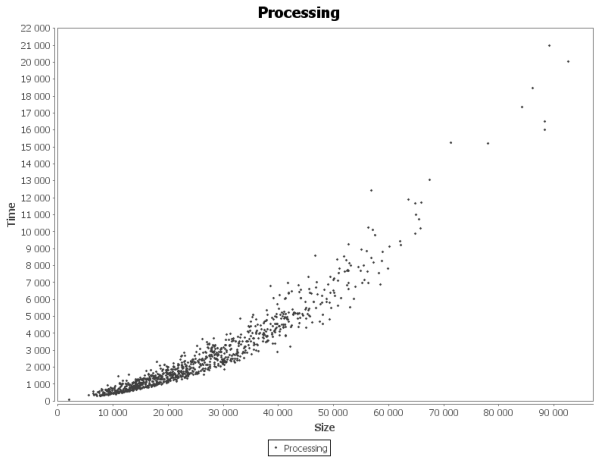

Figure 5.8: UIMA Processing times (in milliseconds)

to submit the document back to the Hydra core. Additionally, requests between the Hydra core and individual stages are performed using a REST interface which adds some overhead as described in 5.4

GATE & UIMA

An interesting effect of the execution overhead can be seen in GATE and especially UIMA, where the overhead increases polynomially as the document size increases, see Figure 5.8. This effect has been verified not to be apparent in the other TPF or other OpenNLP implementations, thus we conclude that this is not due to an increase in processing time by OpenNLP stages themselves. That leaves an execution overhead imposed by the TPF as the cause for the slowdown.

A common design choice between GATE and UIMA is that NLP processing results are represented as annotations covering spans of the source text. Each annotation is represented as a Java object, meaning the amount of objects increase as the document size increases. For each individual stage, the TPF data model must be traversed during data extraction for OpenNLP execution. The results in Figure 5.8 seem to indicate that these TPF have inefficient data models and therefore suffer from increased execution time for each stage proportional to the size of the document.

5.2.3 Communication overhead

For distributed pipelines, the communication between different nodes is another source of delay on text processing. Each event must be communicated between the different components and nodes, potentially increasing delay times and resource usage.

Hydra

Hydra is engineered in such a way that communication between stages is performed by updating properties of a document in the MongoDB database. When a stage fetches a document, it sets the *fetched* property to the time of fetching. When it has finished processing, it sets either the *touched* or *processed* property to the finishing time, depending on the stage, The largest source of delays is the fact that Hydra currently relies on *database polling* to notice these updates to documents.

A document is sent to a subsequent stage once it has been processed by the previous stage. Hydra regularly polls the MongoDB database for updates on unfinished documents and takes action once it notices a new update to a document. For this reason, once a document has finished processing through a stage, rather than instantly notifying the next stage, it must instead wait for Hydra to notice this update in MongoDB before it is sent to the next stage. This waiting time is the largest source of delay for Hydra. It is also a source of increased CPU load and network traffic.

For an example document, in this case a twitter post, the end-to-end time was measured to 4613ms. Of this time, 553ms was spent processing the document, meaning the total idle time for the document was 4060ms, which is 88% of the end-to-end time.

Storm

Storm relies on *message passing* for communication between its components. When a Spout or Bolt emits or finishes processing a tuple, it sends an *ack* message and also outputs the next tuple.

The message passing feature minimizes delay times between different stages, at the cost of increased CPU load and network traffic. However, if a topology is distributed across a storm cluster such that each worker node contains an instance of each Spout and Bolt, most messages can be sent to a target within the same worker node and do not need to be sent over the network, thus reducing the impact on network traffic.

5.2.4 Memory leaks

The tests have been designed to attempt to discover memory leaks for each TPF. The method for discovering these have been to operate the TPF under high load for a long period of time. However, no memory leaks of any remarkable impact could be discovered.

5.3 Development complexity

In practice, the choice of which TPF to use for a search solution does not only depend on their technical performance. Many organizations do not have high search loads, or for other reasons do not have strong requirements on the performance of the system. For these organizations, other factors may be more important, such as cost of implementation and maintenance. Therefore, in practice the difficulty of implementing and administrat-

Task	GATE	UIMA	OpenPipeline	Storm	Hydra
Stage Development	Easy	Easy	Medium	Hard	Easy
Configuration	Easy	Hard	Easy	Hard	Medium
Execution	Easy	Easy	Easy	Medium	Medium
Document Submit	Medium	Medium	Medium	Hard	Easy
Overall	Easy	Medium	Easy	Hard	Medium

Table 5.1: Implementation and Maintenance Complexity

ing a search solution will sometimes be a more important factor when deciding which TPF to use.

For this section we will look at the difficulty of implementing the target pipeline for each of our TPF. Our findings are summarized in Table 5.1 and explained in detail in the following sections.

5.3.1 Stage development

All of the TPF covered in this report share similar implementation techniques. To implement a stage, one needs to create the stage as a class which inherits a specific interface or abstract class, which the TPF will then notice upon launch. Although there are slight technical differences, the programming of a basic stage is mostly trivial.

For Storm, stages are implemented as Spouts and Bolts, which is slightly more complex as a technical understanding of the system is required to be able to develop an efficient implementation. For example, the developer needs to know in advance which fields are available as input from the previous bolt. Additionally, the bolt will be serialized and sent to workers which will marshal an instance of the class for each thread. Since many classes (such as OpenNLP stages) are not serializable, it is important not to instantiate such classes before serialization.

A complicating factor with OpenPipeline is that the program scans the JAR's in its classpath for stages at launch time. For a stage to be noticed, it needs to implement the correct interface but also needs to be pointed towards by a resource listing file. If any of these are not correctly configured the application will silently fail to notice the stage.

5.3.2 Pipeline configuration

GATE and OpenPipeline each provide a GUI which makes the creation and configuration of a pipeline easy, by piecing together available stages with a configuration for each. For other TPF, the process is more complicated. Hydra uses JSON configuration files to read the configuration for each stage. To submit a stage to Hydra, the pipeline is specified as a series of JSON files together with the corresponding JAR files containing the stages. A Java application is available for submitting these to Hydra.

UIMA is more complicated, as it takes pipelines as input in the form of a Processing Engine ARchive (PEAR) file, which must be generated by building an ANT project with a hierarchy of XML files, describing the stages, their order in the pipeline and their configuration including required files.

Finally, Storm poses the largest difficulty of successfully submitting a pipeline. A pipeline must be specified as a 'topology' of Spouts and Bolts. This topology needs to be implemented as a Java class which instantiates and configures each Spout and Bolt, including configuration of parameters for each, which cover the standard NLP parameters but also Storm-specific information such as requested parallel instances of each. This class is then compiled as a JAR file and submitted to an active Storm cluster. Additionally, the document submission method must be considered and implemented as a part of the pipeline. In our project, ActiveMQ was chosen for sending messages containing documents to the topology. This required an ActiveMQ-specific Spout to receive messages to the topology, and a similar Bolt to send back results.

5.3.3 Execution

Executing GATE and UIMA is done by creating a new instance of their respective main class. OpenPipeline is designed to be executed as a web service running on a provider such as Tomcat. Hydra and Storm are designed to run as stand-alone services on a server. Hydra requires a MongoDB instance, whereas Storm requires Zookeeper and some Java library dependencies.

5.3.4 Document submission

For GATE, UIMA and OpenPipeline, document submission is performed using a Java API which submits documents to an active instance of the application on the same machine. If one desires to submit documents over a network, this functionality must be added separately. This usually means wrapping the application with your own HTTP service. As OpenPipeline is already such a service, it would be required to extend the functionality of the existing service.

Hydra provides a MongoDocumentIO class for submitting documents to and reading documents from a local or remote MongoDB instance, making the submission process simple.

Storm comes with no built-in submission method and few default Spouts. The implementation method must be chosen and implemented manually, although some third-party solutions are available.

5.4 Communication Overhead

During the early testing phases of the project, when performing tests with the small documents obtained from Twitter, it was noticed that the throughput of the TPF for which we had developed a separate REST service for network transmission was unreasonably low and the CPU usage was unexpectedly low. It was quickly discovered that

this was due to the TPF processing documents faster than it was receiving them - the REST interface was simply transferring documents too slowly.

As described in 4.1.1, The REST interface is in practice a design structure based on HTTP. When making a HTTP request, the HTTP protocol adds a header to the data sent, increasing its size. It is also necessary to wait for an answer from the remote server to the request. These two factors significantly increase the delay time of each request. Since the REST client could only send one document per request, and since the same client was used for both sending and receiving documents meaning each of these operation would block the other, in the worst-case scenario the transmission rate was as low as 8 documents per second.

The issue was resolved by changing the REST service to receive a large amount of documents per request. We also create a separate client each for sending and receiving documents. However, the REST interface could still be a bottleneck, and still introduces a minor impact on resource usage as it needs to serialize and deserialize the transmitted data which is sent as JSON.

This can be compared to the choice of message brokering service for interaction with Storm. In this case we use ActiveMQ over TCP, which creates a TCP tunnel sending an unbounded byte stream, greatly increasing data transfer rates. The protocol can also be configured not to require waiting for a response from the server, eliminating wait times[16]. A simple transmission test showed rates of more than 5000 documents sent per second, a major improvement over the REST interface. For further transmission speed improvements, Java objects could be sent as byte streams instead of JSON strings, reducing data size and serialization processing times.

5.5 Extensions to Existing Software

During the project, the following additional modules were developed:

- An OpenNLP Bolt and a Solr-output Bolt for Storm.
- Four OpenPipeline stages, one for each OpenNLP stage.
- Four Hydra stages, one for each OpenNLP stage.
- The Linux runtime statistics collector NodePerf, developed in C++.

6

Conclusion

We have tested and analyzed the Text Processing Frameworks (TPF) with a focus on scalability, robustness, stability and performance. We have measured throughput, end-to-end times, starvation and the responsiveness of the system with a focus on feasibility for search query processing. Based on our findings, we give recommendations for which TPF to use in which use case, see 6.1.

We have analyzed the impact of pipeline design and architecture on throughput. Our research gives insight into how to design a pipeline for increased throughput of the system. We have found the biggest impact on the throughput of the system to be a combination of which stages are chosen for use in the pipeline and the TPF's support for parallelism of the individual pipeline stages.

6.1 Recommendations

Based on the findings throughout the results chapter, we make the following recommendations for which TPF to use in which use case.

- *Search query processing* - In this use case, we recommend to use GATE, UIMA or Storm in a non-distributed fashion. For fastest response times, choose Storm with several parallel instances.
- *Text processing with no performance requirements* - For this use case, we would recommend GATE or OpenPipeline, as they are the most easy-to-use TPF out of the 5 covered in this report and thus may reduce implementation and maintenance time.
- *Text processing with demands on availability* - For this use case, we recommend using Storm or Hydra in a distributed fashion, as both gracefully support fail-over in the case of a server going offline.

- *Text processing of data streams with demands on availability and performance* - For this use case we recommend using Storm in a distributed fashion as it is the most high-performance choice out of the 5 TPF covered in this report, while also providing reliability.

Bibliography

[1] W. Weaver, "Translation," July 1949.

[2] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950.

[3] G. S. Ingersoll, T. S. Morton, and A. L. Farris, *Taming text: How to find, organize, and manipulate it.* Manning, 2013.

[4] "Throughput definitions," May 2013. [Online]. Available: http://www.businessdictionary.com/definition/throughput.html

[5] A. Suleman, "Clarifying pipeline parallelism," July 2011. [Online]. Available: http://www.futurechips.org/parallel-programming-2/parallel-programming-clarifying-pipeline-parallelism.html

[6] S. Chakravarthy and Q. C. Jiang, *Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing.* Springer, 2009, vol. 36.

[7] H. Cunningham, D. Maynard, and K. Bontcheva, *Text processing with gate.* Gateway Press CA, 2011.

[8] D. Ferrucci and A. Lally, "Uima: an architectural approach to unstructured information processing in the corporate research environment," *Natural Language Engineering*, vol. 10, no. 3-4, pp. 327–348, 2004.

[9] "Uima specification," May 2013. [Online]. Available: http://uima.apache.org/uima-specification.html

[10] E. Epstein, M. Schor, B. Iyer, A. Lally, E. Brown, and J. Cwiklik, "Making watson fast," *IBM Journal of Research and Development*, vol. 56, no. 3.4, pp. 15–1, 2012.

[11] C. Kanaracus, "Openpipeline seeks to ease document prep for search," *PC World*, April 2008.

[12] "What is nosql?" May 2013. [Online]. Available: http://www.10gen.com/nosql

[13] "Solr," May 2013. [Online]. Available: http://lucene.apache.org/solr/

[14] J. Aas, "Understanding the linux 2.6. 8.1 cpu scheduler," *Retrieved Oct*, vol. 16, pp. 1–38, 2005.

[15] T. L. D. Project, "/proc," May 2013. [Online]. Available: http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html

[16] "using activemq: Performance tuning," May 2013. [Online]. Available: http://activemq.apache.org/performance-tuning.html

A

Test Procedure

A.1 Server specifications

The server used for testing is a Dell PowerEdge R720 with components:

```
Intel Xeon E5-2609 2.40GHz,10M Cache, 6.4GT/s QPI, No Turbo, 4C, 80W, DDR3-1066MHz
1600 MHz RDIMMs
12x 8GB RDIMM, 1600 MHz, Standard Volt, Dual Rank, x4
2x Heat Sink for PowerEdge R720 and R720xd
DIMM Blanks for Systems with 2 Processors
Intel Xeon E5-2609 2.40GHz, 10M Cache, 6.4GT/s QPI, No Turbo, 4C, 80W, DDR3-1066MHz
VFlash, 8GB SD Card for iDRAC Enterprise
6x 1TB, SATA, 3.5-in, 7.2K Hard Drive (Hot-plug)
PERC H710 Integrated RAID Controller, 512MB NV Cache
Dual, Hot-plug, Redundant Power Supply (1+1), 750W
Intel Ethernet i350 QP 1Gb Network Daughter Card
Intel Ethernet I350 DP 1Gb Server Adapter
RAID 5 for H710p, H710, H310 Controllers
iDRAC7 Enterprise
```

A.2 VM specifications

The test were performed on VM's created on the above server using Hyper-V Server 2012, with the following configuration for each VM:

```
Cores: 2
Memory: 4GB RAM
```

A.3 Test Procedure Checklist

- Reboot server VM
- Ensure required services are running
- Ensure no unnecessary services are running
- Wipe the Solr index
- Ensure NodePerformance collector is running
- Execute StressTest
- Check that correct number of jobs were sent and received
- Check that no errors occurred during execution
- Perform a basic data validation check in Solr

B

Test Framework Design

B.1 REST Service interface

Below is a specification of the REST service interface.

The following paths can be queried with a GET query:

```
/start - Initiates the application and begins processing
/stop - Closes application and stops processing documents
/document/poll - Returns a finished document, if available
```

The following paths can be queried with a POST query, combined with a JSON data string matching the specified pattern:

```
/document/addText - Add document from text source.
data: {"name":"jobname", "text":"Document text"}

/document/addLocalFile - Add document from local file.
{"file":"/path/to/file/on/server.xml"}

/document/addListText - Add list of documents from text sources.
[
  {"name":"jobname1", "text":"Document text"},
  {"name":"jobname2", "text":"Document text"}
]

/document/addListLocalFile - Add list of documents from local files.
[
  {"file":"/path/to/file/on/server.xml"},
  {"file":"/path/to/file/on/server.xml"}
]
```

C

CPU Resource usage per TPF

These charts show the recorded CPU usage for each TPF during the non-distributed test with Twitter data. The data was collected using NodePerf.

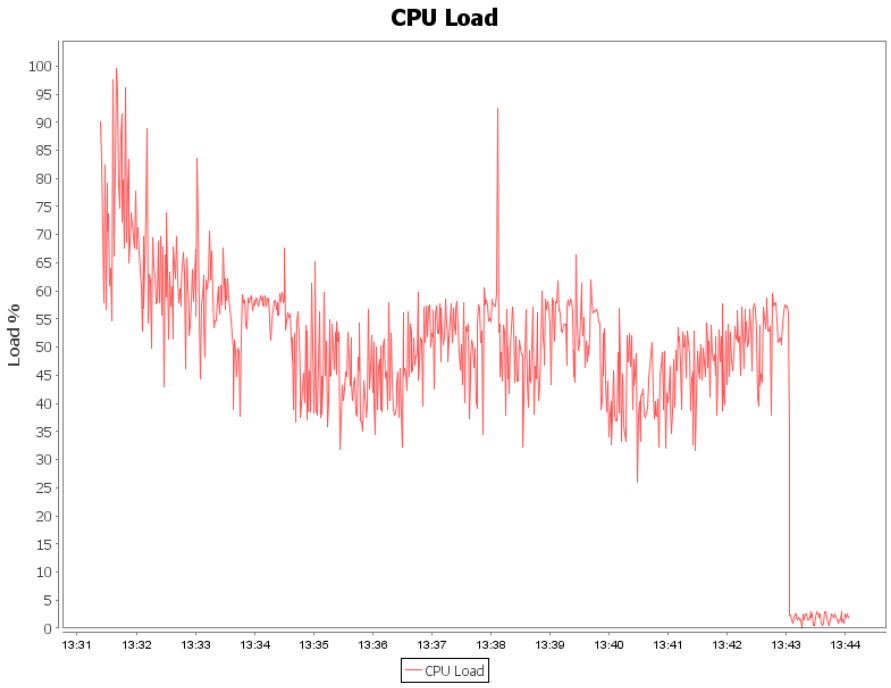

Figure C.1: UIMA

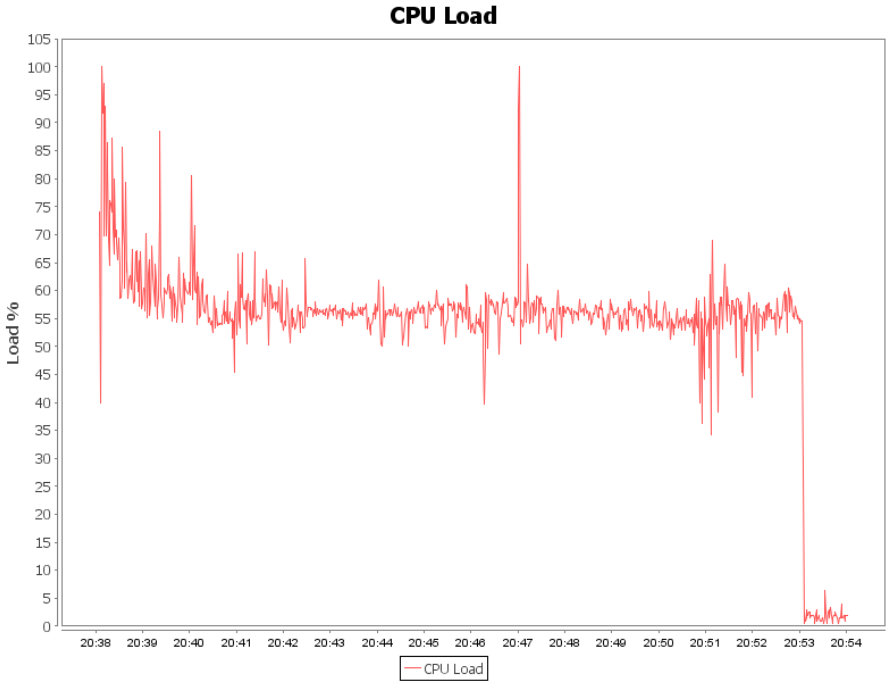

Figure C.2: Gate

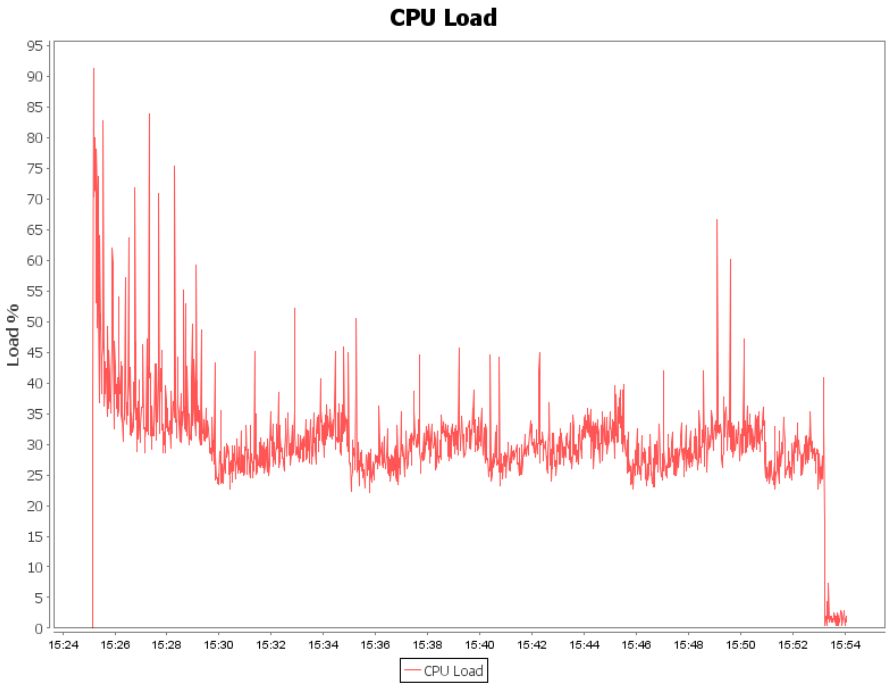

Figure C.3: OpenPipeline

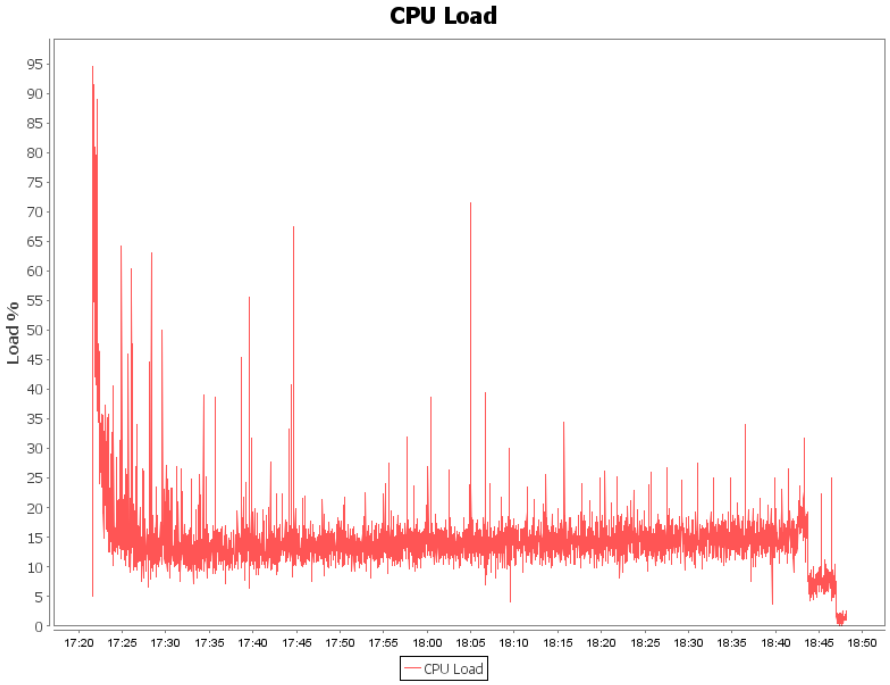

Figure C.4: Hydra

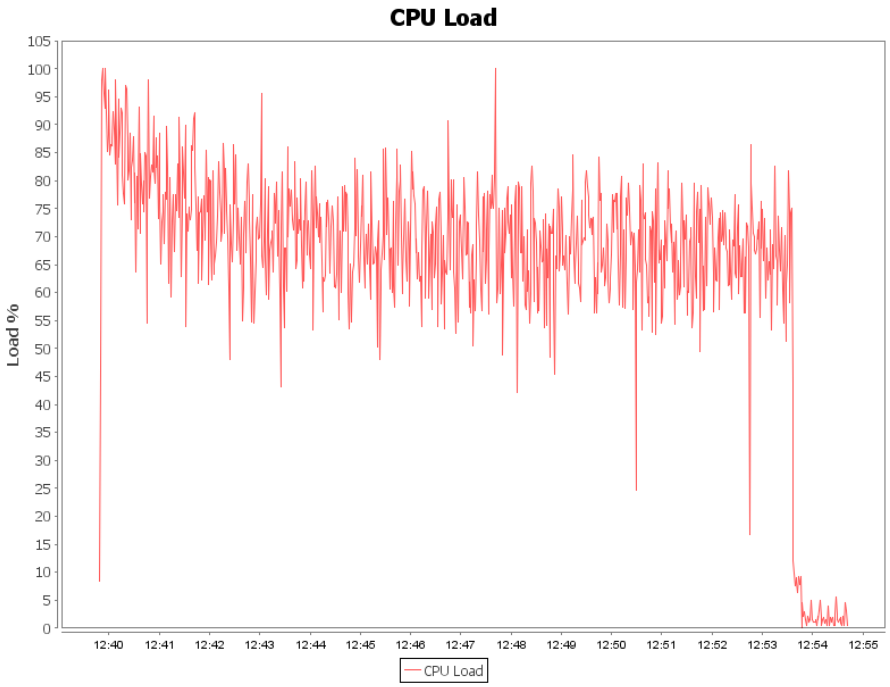

Figure C.5: Storm