Masaryk University
Faculty of Informatics

Replacing Logstash

Master's Thesis

Ondřej Lomič

Brno, Spring 2020

Masaryk University
Faculty of Informatics

Replacing Logstash

Master's Thesis

Ondřej Lomič

Brno, Spring 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ondřej Lomič

Advisor: doc. RNDr. Tomáš Pitner, Ph.D.

Acknowledgements

I would like to thank all the coworkers at Y Soft, past and current, mainly Zuzka, Andrij and Jakub, which made this thesis possible.

Abstract

Y Soft Corporation uses tool Logstash in its log processing pipeline, but it was proved inadequate for its high consumption of system resources and difficulty of configuring complex processing pipelines. Therefore this paper aims to replace Logstash with a better alternative.

At first, all requirements for the replacement are specified. Based on those there is a comparison of other existing tools. Because no suitable alternatives were found, a new custom solution was implemented to replace Logstash. Lastly, performance benchmarks were run to compare Logstash and its replacement.

The results show a significant improvement of specified properties, in range of an order of magnitude for the measurable ones, while not falling behind in other areas. The tool was successfully used to fully replace Logstash in Y Soft Corporation, while the core is publicly available and can be further expanded.

Keywords

Logstash, stream processing, C++

Contents

Introduction

Y Soft Corporation started using Logstash in 2017, as a part of a solution for log processing. The solution was designed in my bachelor thesis [1] and Logstash [2] was used to collect, parse and transform logs and to transfer both logs and metrics between the solution's components. At first, it has proven to be a quick and easy solution for this use-case, however, it was far from perfect. Firstly parsing and transforming logs was quite CPU intensive, moreover configuring more complex processing pipelines was very difficult. It was partly solved by generating Logstash configurations with a Python script, but such a solution is far from perfect. Secondly, although Logstash's high memory usage of nearly one gigabyte was tolerable on the server-side, it was too high to use Logstash as a lightweight collector on the client-side. It was still used since there was no better alternative, but both installation size and memory usage was higher by an order of magnitude than desired. And the fact that Logstash is quite CPU intensive was not helpful but was tolerable due to low traffic on the client-side. Despite all problems mentioned above, Logstash was still used since there was no clear alternative that would have solved its problems while providing the features that Logstash did, for example a persistent queue. However, when the solution for log processing became more widely used, Logstash's CPU consumption on the server-side became a bottleneck and a more efficient solution was needed.

Therefore this work aims to find a replacement for Logstash. The first step was to specify all required properties of such replacement, both functional and non-functional ones. Secondly, a thorough overview and comparison of other existing tools was done, based on the specified requirements, to either replace Logstash with an existing tool or to implement a custom solution. Since there was no suitable replacement, the latter option was chosen and a tool called Wolf was implemented. To compare the performance of both tools, a set of benchmarks was created, to evaluate both general usage as well as specific configurations used in Y Soft. Lastly, Wolf was used to replace Logstash in the log processing solution to confirm its real-world usability.

The first part of this work is a brief overview of Logstash, followed by a part focused on the decision making to implement Wolf instead of using an existing tool: specification of requirements, an overview of existing tools, its comparison and summary of advantages and disadvantages of implementing a new tool. The third part describes the design choices of Wolf implementation and differences in features in comparison to Logstash. Lastly, an evaluation of performance benchmarks and comparison of non-functional properties follows, together with final conclusion.

1 Logstash's overview

Logstash is open-source software developed as a part of ELK stack [3] (Elasticsearch, Logstash and Kibana) by Elasticsearch company [4]. The main product is Elasticsearch [5], a database mainly focused on full-text search using Lucene [6] library, but in the latest releases, a machine learning extension was also added [7]. Kibana [8] is a web browser-based visualizer, closely coupled to Elasticsearch, which enables the creation of graphs, browsing of stored data but also management and monitoring of Elasticsearch.

As the last part of ELK stack, Logstash [2] serves to collect/ship/index data into Elasticsearch database, but it supports many other outputs as well. It aims to provide a universal way to easily load data from any source and output them to another, while processing/altering it in between. It uses a system of input/output/filter plugins to do so, many of which are a part of Logstash base, but there are community-made plugins as well.

Logstash runs on JRuby [9], which is an implementation of Ruby atop of JVM, largely written in Java. That combines advantages of an easy to use programming language like Ruby with platform compatibility of JVM, however, it is comparably slower than both Java and Ruby[1], which affects Logstash's performance.

On the other hand, Logstash is actively developed with very good documentation [13] and its configuration is easy to learn and read, as long as it is simple enough. For example, there is a configuration which listens on TCP port, adds "Hello world!" message and outputs it to standard output and into a file:

1. [10, 11, 12] provides benchmarks for selected languages, however, a direct comparison is missing, they have to be compared manually. For example, binary-trees benchmark, N equals 21, fastest time: Java 8.25 seconds, Ruby 44.61 seconds, JRuby 61.62 seconds.

```
input {
  tcp { port => 9555 }
}
filter {
  mutate {
    add_field => { "message" => "Hello world!" }
  }
}
output {
  stdout {}
  file {
   path => "output.txt"
   codec => line { format => ": %{message}"}
  }
}
```

2 Selection of Logstash's alternative

There were several places in the Y Soft log processing pipeline where Logstash was used. Firstly, it was used as a part of a collector on the client-side, to normalize collected logs and to provide a persistent buffer, in case the connection to the server-side was interrupted. Especially the latter feature was important, to be able to keep collecting logs for prolonged periods of time without a connection to the server-side, without depleting the client's memory or discarding collected logs. However, Logstash's too high memory usage and CPU consumption meant it was usable, but not an ideal solution. Moreover, Logstash packaged together with Java made the final installation package over 400 MB large, which is again far from a lightweight log collector.

Secondly, Logstash was used in multiple variations on the server-side, to process collected logs and to index them into a database. There are two variants of the server-side monitoring, the large and the small one, however, it is enough to consider Logstash use-case within the small monitoring since it is essentially a combination of all Logstash use-cases within the large one and it comes with the same set of requirements. Logstash's main responsibility within the monitoring server was to process logs (to find specific logs using regex pattern matching and to extract metrics from them) and to store logs into a database or to drop them, based on the monitoring server variant and specific configuration. While it was initially quick and easy to get Logstash running with its wide range of plugins, the plugin system made it very difficult to configure the pipeline for specific purposes, which required functionality that was not covered by predefined plugins. That created complex configuration files, generated by a custom Python script, which for example contained snippets in Ruby language. Those were however without any syntax checking et cetera, therefore the configuration became really hard to debug and test. Moreover, Logstash's high CPU usage caused not just by using JRuby, but also by poor regex matching performance, quickly became a bottleneck and was the primary reason to replace Logstash.

Lastly, it was used to occasionally upload logs to the monitoring server and to generate data for performance tests, but these use-cases are not considered in further requirements specification since Logstash

is an adequate tool for these ad-hoc purposes and does not need to be replaced there.

This chapter contains a specification of functional and non-functional properties of Logstash's use-cases, an overview of third-party tools possibly suitable for Logstash replacement and their comparison and summary of the advantages and disadvantages of implementing a custom solution.

2.1 Specification of requirements

As mentioned above, only two Logstash's use-cases are further considered for the specification of requirements, the collector and the parser, since the other usages have no need for replacement or their requirements are already covered by the requirements of the selected two. Although both use-cases do have an intersection of common requirements, they are different enough that two separate tools could be used to replace Logstash, therefore their additional requirements are specified separately. However, one tool to replace Logstash would be highly prefered, not just because of the extra time needed to learn how to use two tools instead of one, but also because it allows to simply move functionality from the collector to the parser and vice versa.

2.1.1 Common requirements

The first requirements are that all tools used in Y Soft log processing pipeline have to be ***free to use*** and ***open-source***. Furthermore, they need to support the ***Linux*** operating system as well as the ***Windows*** one.

Secondly, to ensure adequate quality of used software, it should be ***proven***, to avoid early prototypes that could be soon discontinued. On the other hand, the replacement should also be actively used and ***maintained***, not to end up with legacy software without any further development. Moreover, well-written ***documentation*** should be present, to ensure quick and easy integration into the monitoring solution.

Lastly, as a good rule, selected software should be as ***simple to use*** as possible. The selected replacement should be easy to configure and

easy to run, not to make installation of both the collector and the parser unnecessarily complicated.

2.1.2 Collection requirements

The general idea behind collector requirements is that it should be as ***lightweight*** as possible, while providing necessary features.

The collector needs to be able to ***listen on TCP*** ports for incoming logs and metrics, be able to provide ***basic filtering and data manipulation***, and also enable data output via ***TCP or Kafka producer***, based on the collector's configuration. Lastly, a ***persistent buffer*** needs to be present, to be able to collect data even when a connection to the monitoring server is lost, without filling up the system memory or dropping collected data.

The requirements above are indeed satisfied by Logstash, however, its consumption of system resources is far from lightweight. Using Logstash to collect logs generated by expected load results in 5 to 10% of CPU usage and over one gigabyte of system memory usage. Also, Logstash alone (without Java, which is also needed) consists of over 12,000 files with a total size of over two hundred megabytes, therefore it takes around a minute just to extract it from an archive even on an SSD drive, which slows down installation and any manipulation with the installation package.

To define the requirement of lightweightness, the replacement should be ***more CPU efficient***, should have ***less memory usage*** and a ***smaller disk footprint*** than Logstash, at least ***an order of magnitude improvement*** is required.

2.1.3 Server processing requirements

While the focus behind the collector's requirements was lightweightness, ***performance*** is more important for the parser, the processing component of the monitoring server.

Logs are received through ***TCP or Kafka consumer***, based on the monitoring server configuration. The most intensive processing part is finding specific logs using regex matching, but the parser is used also for aggregating logs into specific metrics. That is for example counting them per host or calculating elapsed time between two specific logs,

advanced data manipulation is therefore needed. Finally, logs and metrics are outputted to a database using ***HTTP protocol*** for both used databases, or back to ***Kafka*** by its ***producer***.

The biggest bottleneck of Logstash's processing is the lack of ***efficient regex matching***. JRuby, as well as Ruby, Java, Python et cetera, uses backtracking[1] to evaluate regex matches, which causes increasing processing demand with an increasing number of regexes to evaluate. However multiple regexes can be combined into one deterministic finite automaton, resulting in constant time complexity with respect to the number of regexes, which significantly improves performance [14]. However, Logstash's high CPU usage by itself is less than ideal, a tool with more efficient CPU usage would also increase total throughput.

Although possible, configuring Logstash to perform regex matching and aggregation of logs into metrics was incredibly difficult. The resulting Logstash configuration was generated using Python script, was almost 1,000 lines long and hardly readable[2]. Moreover, raw Ruby snippets were used to help with limitations of Logstash plugins, however, these snippets are without any validation, therefore it is easy to make a syntax mistake while writing them. Also their functionality is hard to test since that requires starting up Logstash (which takes up to a minute) and feeding it test data, so the Ruby code is executed. Therefore the replacement should be ***configured via programming language***, not by a set of predefined plugins, to allow more manageable configuration of complex processing pipelines.

Lastly, although the biggest performance inefficiency of Logstash was regex matching, even without it the limiting factor of Logstash's throughput was its CPU usage. Therefore, we look for a more CPU efficient replacement, however in contrast to the collector, not to achieve a lower footprint, but to ***increase throughput***. And an ***improvement in an order of magnitude*** is again required.

---

1. Backtracking is a general algorithm which searches the whole state space, usually using a depth first search algorithm, therefore it has poor worst case time complexity equal to the size of the state space.
2. Parts of Logstash configuration with brief description are located in the appendix A.

Common requirements	
free to use, open-source	
both Linux and Windows support	
proven and maintained	
well written documentation	
simple to use	
Collector's requirements	**Parser's requirements**
TCP input	TCP and Kafka input
TCP and Kafka output	HTTP and Kafka output
basic filtering and data manipulation	configured via programming language
persistent buffer	efficient regex matching
lightweight	performant
an order of magnitude improvement of:	
CPU efficiency	throughput
memory usage	
disk footpring	

Table 2.1: Summary of requirements for Logstash's replacement

2.2 Comparison of found alternatives

Since the task of the collector and the parser is quite general, to listen for data, process it and output it, there are many tools that could be potentially used. On the other hand, there are other strict requirements (such as having a persistent buffer or really small binary size of the collector), therefore many of them are excluded from comparison right away.

The search and comparison itself was done during the summer of 2018, two years before completing this thesis, and since naturally there were numerous changes or new alternatives, an update is placed at the end of this section.

2.2.1 Tools excluded from the comparison

All paid alternatives are excluded from this comparison since we look for an open-source and free to use tool, however, one of them is worth mentioning. Syslog-ng [15] would satisfy most of the specified requirements and would be one of the ideal tools to replace Logstash. However, it is only available on Linux, the Windows version is a premium, ***paid*** one.

Neither Heron [16], Wallaroo [17], Rsyslog [18] nor Hindsight [19] has ***Windows support***, however, Hindsight does have it planned and could be promising in the future.

Benthos [20] could be a good alternative, fast and lightweight since it is written in Go, however, it was still in beta version and very rapid development in summer of 2018, therefore ***far from proven*** enough to depend on it.

There are also two tools that are being developed for long enough, but still unusable. Both Onyx [21] and Monix [22] have ***poor documentation***, which could be tolerable if they wouldn't be written in Clojure and Scala respectively. Since these languages are far from commonly used, at least the documentation should be good enough, although the lack of community and usage could explain why it is not.

On the other side of not being used enough, there are plenty of tools that are ***no longer maintained***. For example, Heka [23], the predecessor of Hindsight, was discontinued because of a bad design. But there is also S4 [24], SPQR [25], Facebook's Scribe [26], Tigon [27],

paid software	Syslog-ng
no Windows support	Heron, Wallaroo, Rsyslog, Hindsight
not used widely enough	Benthos, Onyx, Monix
no longer maintained	Heka, S4, SPQR, Scribe, Tigon, Hailstorm, Teknek, Concord, StreamBox, Saber, Swave, Bistro Streams
too complex	Apex, Flink, Gearpump, Samza, Spark, Storm, Kafka Streams, Akka Streams

Table 2.2: Software removed from comparison

Hailstorm [28], Teknek [29], Concord [30], StreamBox [31], Saber [32], Swave [33] and Bistro Streams [34]. Whether they are still maintained was determined using Github, none of those tools had any recent activity there.

There are also many tools that are ***too complex*** for this use-case, used in large-scale deployments, robust, distributed, but hard to configure, run and manage. Many of them require Zookeeper [35] for their state management or they have their own system of master/slaves, server/clients. All but one belong to Apache Software Foundation, namely Apex [36], Flink [37], Gearpump [38], Samza [39], Spark [40], Storm [41] and Kafka Streams [42]. The only exception being Akka Streams [43], which is part of its own Akka architecture.

At the end, there are six tools which satisfy the common requirements and are further discussed for use as a collector or a parser: Hazelcast Jet [44], FS2 [45], Flume [46], Fluentd [47], LogAgent [48] and Flowgger [49].

2.2.2 Collector's alternatives

Firstly, FS2 and Flowgger are missing a way to ***persistently buffer incoming data*** and there does not seem to be a simple enough way to add such functionality. All the remaining tools do have required inputs, ways to alter processed data and outputs, but LogAgent, which is missing TCP output, but that functionality could be implemented.

Therefore all of Hazelcast Jet, Flume, Fluentd and LogAgent could be used, however, none of them is really lightweight. Both Hazelcast Jet and Flume are written in Java, designed for distributed, robust deployments rather than as a lightweight collector, both at the edge of too complex for this use. Since they both use Java language, their performance would be better, but the use of JVM does not promise big improvement in ***memory usage***. Their ***installation size*** brings no significant improvement either. Even though both Hazelcast Jet (150 MB) and Flume (90 MB) do have a smaller footprint than Logstash (250 MB), all of them require Java Runtime Environment (225 MB), therefore even Flume lowers disk footprint just by one third.

On the other hand, Fluentd and LogAgent do lower memory usage significantly, however since they are written in Ruby and Javascript respectively, no significant improvement of ***CPU efficiency*** is expected. Their ***disk footprint*** is also not close to the required order of magnitude improvement compared to Logstash (475 MB), smaller being Fluentd (190 MB) with LogAgent far larger (340 MB).

Out of the compared tools, Fluentd would be the best one to replace Logstash. While it brings no significant improvement in CPU usage efficiency, at least it brings disk footprint down by more than half and would bring significant improvement in memory usage. However, it is still far from being the ideal replacement.

2.2.3 Parser's alternatives

In contrast to the collector, the parser does not need to be lightweight but needs to allow complex configuration by using a ***programming language*** instead of a configuration file, which excludes Flowgger and Fluentd.

For ***efficient regex matching*** a library RE2 [50] can be used, which has binding to Node.js, therefore can be used in LogAgent. There is also a port RE2/J, available for use in all of the other remaining tools, therefore all of them could be used.

However, LogAgent, being written in Node.js, does not promise the required increase in ***performance***. Moreover, its configuration is written in Javascript, which is not the most readable language.

FS2, being written in Scala, almost belongs to the same category as Monix and Onyx mentioned above, the category of not being used

enough with poor documentation and not using a widespread programming language. Although true, it is being used just enough and the documentation is not great, but also good enough. However, using it would still require to learn Scala and although it looks proven enough, it is still in beta version.

Hazelcast Jet is similar to FS2 in that it is still in a beta version and under development. However, it uses Java language and its documentation looks much better, therefore could be suitable for replacement. It lacks required HTTP output, but that could be implemented. On the other side stands Flume, also being written in Java, used for so much longer that it is almost under no development, however with poorer documentation. But it could be used for Logstash replacement as well.

2.2.4 Summary of alternatives

There are two tools that could replace Logstash as a parser in the monitoring server, Flume being developed for many years and still maintained, however, with poor documentation. On the other hand, Hazelcast Jet is still in beta but already usable, only missing HTTP output, which could be implemented.

However, there is no tool that would replace Logstash as the collector and satisfy all specified requirements. The best alternative would be Fluentd, which is a tool very comparable to Logstash. There are already many comparisons between those two tools [51, 52, 53], however using Fluentd wouldn't bring any significant improvement in disk footprint nor CPU usage efficiency, only in system memory usage.

Because no suitable solution was found for the collector, a decision to implement a custom solution was made. That, of course, brings its own disadvantages, which are discussed in the next section.

2.2.5 Comparison update as of 2020

There are a few notable changes since the comparison above was done during the summer of 2018. Although they are not relevant now, since Logstash was already replaced, the most notable are listed below for completeness.

	FS2	Flume	Fluentd	LogAgent	Flowgger	Hazelcast Jet
Collector's requirements						
persistent buffer	x				x	
disk size	-	315 MB	190 MB	340 MB	-	375 MB
CPU efficiency	-	Java	Ruby	Node.js	-	Java
memory usage	-	JVM			-	JVM
Parser's requirements						
complex configuration			x	Java-script	x	
CPU efficiency	Scala	Java	-	Node.js	-	Java

Table 2.3: Rating of tools selected for final comparison

Benthos went through rapid development and could be possibly used to replace Logstash both as a collector and as a parser. It is still missing a persistent buffer, but it would be possible to implement it. Also, there is a tutorial [54] which shows how to create custom plugins, which could be used to implement parser's complex pipeline.

Hazelcast Jet, a tool which was one of the candidates to replace Logstash as a parser, is also out of beta and could be used without any fear that its development will be discontinued.

Lastly, Trill [55] and Lightsaber [56] are newly created tools in the meantime, Lightsaber having the first commit on GitHub just a few days before writing this update. Written in C# and C++ respectively, both could be suitable replacements for Logstash in the future.

2.3 Pros and cons of a custom solution

The first and clear advantage is, if designed and implemented properly, custom software will exactly match the specified requirements that are demanded, meaning not only providing required features but just those and no unnecessary other ones. Also, when the requirements change and additional specific features are needed, it can be a lot more difficult to add them to a third-party tool.

On the other hand, the clear disadvantage is the time required not just to develop, but to test and maintain software. However, it is worth mentioning, that in this specific case, using such an improper tool as Logstash consumed more time, that it would take to implement a custom replacement. Also even when using a proper tool, a certain time is needed to learn it in order to use it.

Lastly, and also being one of the main reasons an implementation of custom software was chosen, there is a clear advantage of having one tool with a different configuration for both the collector and the parser, instead of two different ones. Because both use-cases are very similar, not only it does not add complexity, but it also decreases time to maintain the solution and mainly allows sharing configuration between the two tools, therefore functionality can be easily moved from the collector to the parser and vice versa.

3 Wolf implementation

Since no suitable Logstash's replacement as the collector was found, a new custom tool was implemented, named Wolf. It aims to replace the collector as well as the parser since both use-cases are very similar.

It is heavily inspired by Logstash, that the processing pipeline is built out of separate plugins. The main difference is that Wolf's pipeline is built using the programming language C++, not by using a configuration file. There are numerous advantages, for example, the syntax of the pipeline configuration can be checked even before compilation, using static analysis built into many integrated development environments. The specific design choices and their consequences are discussed further in this chapter, together with the most important features of Wolf.

3.1 Requirements

All the required properties of Wolf come from the specification of requirements, section 2.1. To summarize, Wolf needs to provide the same functionality as Logstash, while lowering CPU and system memory usage, as well as disk footprint.

On the other hand, Wolf is not required to implement all the functionality that Logstash provides, only that, which is necessary to replace Logstash's usage within Y Soft. That, for example, includes Logstash's monitoring API, support for multiple pipelines or numerous plugins.

3.2 Design choices

This section focuses on design decisions that had to be done before the implementation of Wolf started and cannot be changed afterwards. Four, the most important, are discussed: the selection of language the Wolf is written in, how the pipeline is configured, separation of the core library from the pipelines' configurations and a way to manipulate processed data. Advantages and disadvantages of each design choice are discussed, as well as possible alternatives.

3.2.1 Language selection

The root cause for the poor performance of Logstash is the fact that it is written in JRuby. Moreover, it is also the reason for the large installation size and system memory consumption. Therefore it was necessary to select such programming language, which would negate these problems.

Therefore a C++ language was selected. Not only it comes with exceptional performance, efficient memory usage and small binary size, but it is also well proven and has a large community. It is a very low-level language, close to C language, which enables writing of efficient and performant programs, but also the C++11 and later standards bring cross-platform compatibility and many useful language features, containers, functions etc. which make the development of C++ programs much faster and easier.

On the other hand, a big disadvantage is the lack of a package manager to easily add or remove libraries, or any standard for building libraries whatsoever. Many libraries use CMake [57] build tools, which also provides ways to include and build additional projects/dependencies, however, adding a library to a project is still a lot more complicated than for example using package manager Pip of the programming language Python.

There are certainly other languages that could be used instead of C++, e.g. Go, C# or Swift, but the main reason to choose C++ over those other languages was my personal experience. I already passed an advanced C++ seminar or our faculty and was familiar with that language, which made the development much faster with better quality, which would not be true for the other languages, which I haven't used at all.

3.2.2 Configuration as a code

The other biggest difference in comparison to Logstash is that the processing pipeline is not configured via a file with custom syntax, but directly using the programming language C++ before compilation.

There are many advantages to this approach, when done properly the configuration can be as readable and easy to use as the Logstash's one. It also brings the advantages of using a widely used syntax of

C++: IDE's static code analysis, error highlighting and automatic formatting, etc. Moreover, when needed, a full potential of the C++ programming language can be used to manipulate processed events, not being limited to use predefined plugins.

On the other hand, any change to the pipeline's configuration means that the resulting binary has to be recompiled, also to do so a C++ compiler and CMake is required. Logstash in comparison needs only one dependency to run the pipeline, the Java runtime environment. However, Logstash takes for our use-cases nearly one minute to start, and that time is needed whether the configuration was changed or not, while Wolf's pipeline has to be compiled just once.

3.2.3 Separation of pipelines' configurations

The same way the Logstash is a generic library that executes specific configuration files, such distinction was made for Wolf as well, although the configuration of Wolf is not a file, but a C++ project. To be more exact, the C++ project can contain multiple Wolf configurations, it is not one project per one configuration.

To achieve such separation, all common files and plugins are located inside a Wolf library [58], which is independently used by a public Wolf project [59], containing pipeline configurations used in benchmarks later in this thesis, as well as a private project containing configurations used in Y Soft.

3.2.4 Manipulation with the processed data

The last important decision was how to represent and manipulate the processed data. The same approach as Logstash's one was chosen here, to use a JSON format, except for one difference: the Logstash's event is always a JSON object, but Wolf forwards the event between its plugins as any JSON value. It can be an object too, but also a string, integer etc. That is firstly more intuitive and easy to use, for example, TCP input produces a string, not an object with a `message` key, where the incoming data is stored. It also leads to a more optimized pipeline since JSON object serialization/deserialization is not always needed, Wolf can manipulate with events being just string values.

A lot of focus was also given to choosing the best library to manipulate the JSON data. That is important since the library not only affects the speed of serialization/deserialization of JSON values but also provides an interface to use. The most performant [60] JSON C++ library is RapidJson [61], however, its interface is far from user friendly. The result would be a very fast pipeline, however, most plugins would be unnecessarily complicated and hardly readable only because of the way the JSON data are worked with. Therefore a TaoJson [62] library was chosen, being the fastest one that provides a user-friendly interface. The TaoJson library is about three to four times slower than the RapidJson one, however, still faster than most other ones. The JSON data manipulation is closely inspired by the way containers from the standard library are used, therefore it is easy to learn, use and read for any C++ developer.

3.3 Usage overview

This section gives an example of Wolf's pipeline configuration and shows a simple plugin implementation. It does not aim to be a complete description of all aspects, it is rather a brief overview of its usage. Selected Wolf's features are more thoroughly described in the next section.

3.3.1 Pipeline's configuration

Wolf's pipeline configuration has many similarities with the Logstash's one, they both use a set of plugins to define inputs, data manipulation and outputs. However, there are two key differences: Wolf's plugins are not divided into categories and they can have more than one input/output.

Logstash has four categories of plugins: inputs, outputs, filters and codecs, all separated into corresponding parts of the pipeline's configuration: the input, filter and output one. The codec plugins are used to define how to serialize/deserialize data when configuring inputs/outputs. All data from inputs are directed to the same set of filters (which mutate/alter the data more often than filter them), the resulting data go to the same set of outputs. Specific filters/outputs can be

selected using if-else decision branches. To showcase that, a simplified comparison of parser's configuration can be seen on a figure 3.1.

The main disadvantage of that approach is that when a filter plugin in the middle of the processing pipeline creates a new piece of data, e.g. metric containing number of processed events per second, such piece of data has to follow the same processing pipeline as all other data. However, such a piece of data is usually not processed the same way as the incoming data, therefore complicated if-else branches have to be created, to filter the newly created data.

On the other hand, instead of forcing all Wolf plugins to be part of one processing flow, and to filter data using if-else branches, Wolf's configuration can be represented as a directed graph, a plugin can have more than one input or output. That means that the data flow can be divided for different types of data, for example, generated metrics can be directly forwarded to a specific output, e.g. a file, without disrupting the flow of processed data. Also, multiple separate pipelines can be easily created, if there are multiple distinct processing flows.

Secondly, all Wolf plugins inherit from the same class, therefore they can be arbitrarily interchanged. For example, it is possible to firstly split incoming data only by new-line character and throw away too long events before the JSON is deserialized, which would be very complicated to do in Logstash, if even possible.

Table 3.1 shows examples of both Logstash's and Wolf's pipeline configuration, both providing the same functionality, counting processed events. The only difference is, that Wolf does not deserialize incoming JSON data, since it is not necessary.

3.3.2 Custom plugin implementation

All Wolf plugins inherit from the same class, `base_plugin`, but there are also three additional subclasses: `threaded_plugin`, `mutexed_plugin` and `threaded_mutexed_plugin`. These classes provide additional functionality to the `base_plugin`, to make it easier to implement plugins which require their own thread, a lock for mutual exclusion of processing threads, or both. Their usage with a simple example is shown below.

A typical simple plugin inheriting from the `base_plugin` class overrides the `prepare(json &&message)` method, which takes an rvalue ref-

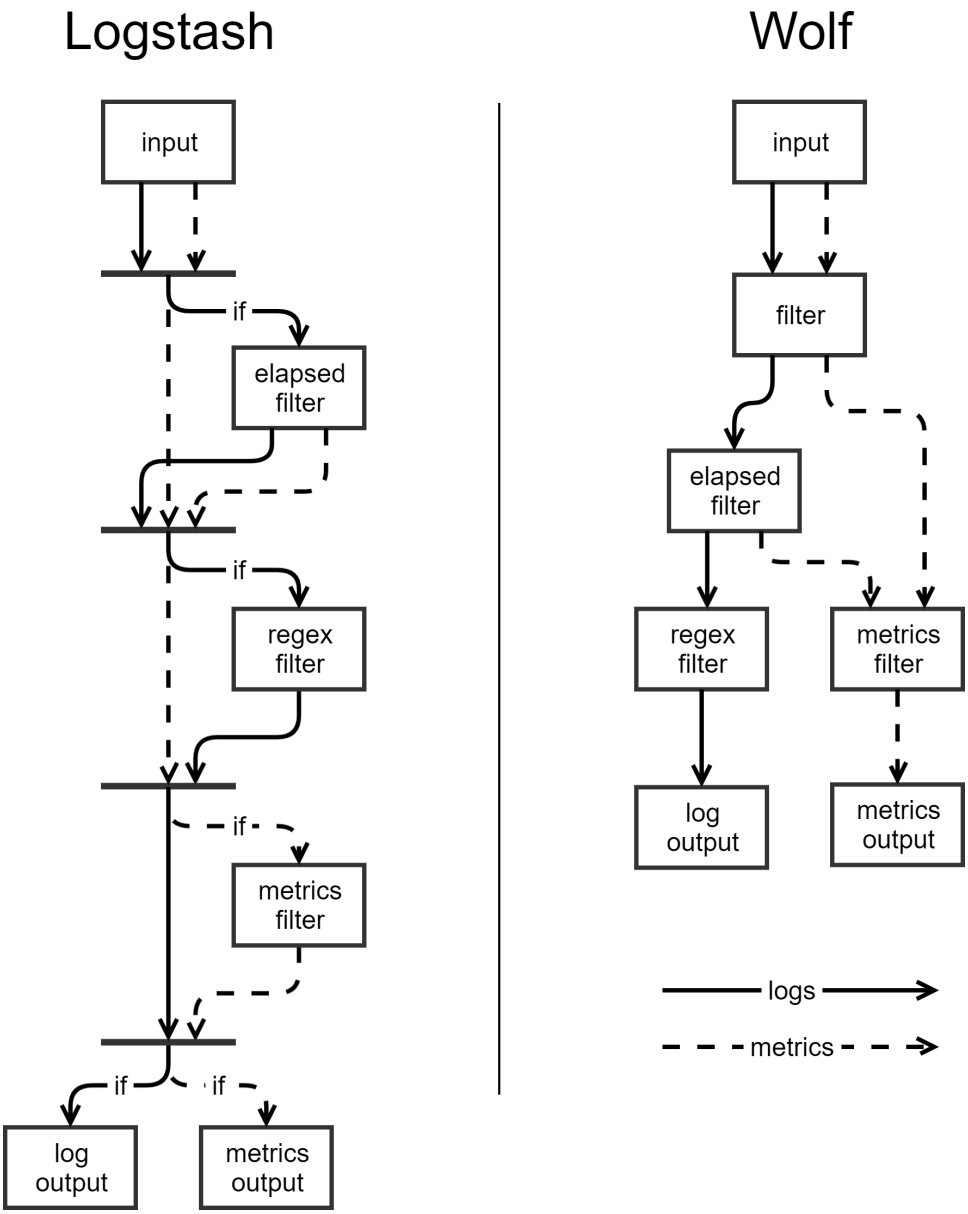

Figure 3.1: Difference between Logstash's of Wolf's pipeline

Logstash

```
input {
  tcp {
    codec => "json_lines"
    port => 9069
    type => "logs"
  }
}
filter {
  metrics {
    add_tag => "metrics"
  }
}
output {
  if [type] == "logs" {
    tcp {
      host => "localhost"
      codec => "json_lines"
      port => 9070
    }
  }
  if [type] == "metrics" {
    stdout {
      codec => rubydebug
    }
  }
}
```

Wolf

```
#include <wolf.h>

int main() {
  using namespace wolf;
  pipeline p;

  p.register_plugin(
    make<tcp::input>(9069),
    make<from::line>(),
    make<stats>()->metrics(
      make<to::string>(),
      make<to::line>(),
      make<cout>()
    ),
    make<to::line>(),
    make<tcp::output>(
      "localhost",
      9070
    )
  );

  p.run();
}
```

Table 3.1: Showcasing different syntax of Wolf and Logstash

erence of the processed event, applies its functionality and then passes it into the output method. The output forwards the event to the next plugin. When multiple outputs are needed, an output method overload with a name parameter is used to specify a different one. Also, a method to register such output when creating the pipeline is usually created, for convenience. Lastly, a constructor with custom parameters can be created, to allow configuration of the implemented plugin. An example of a plugin which splits the processing flow based on a specified condition is provided below.

```
class filter : public base_plugin {
 public:
  filter(std::function<bool(const json &)> condition) :
      condition(std::move(condition)) {}

  // method to register the second output with
  template<typename... Args>
  plugin filtered(plugin plugin, Args &&... args) {
    return register_named_output("filtered", std::move(plugin),
        args...);
  }

 protected:
  void process(json &&message) override {
    if (condition(message)) {
      output("filtered", std::move(message));
    } else {
      output(std::move(message));
    }
  }

 private:
  const std::function<bool(const json &)> condition;
};
```

The `threaded_plugin` subclass keeps the process method for incoming events, but also adds methods for managing a separate thread. The thread is automatically run when the pipeline is started, the `setup`

method can be used for one time initialization, then the `loop` method is executed repeatedly. Moreover, a sleeper helper class is provided when the execution of the thread should be delayed. In contrast to the `std::this_thread::sleep_for` method the sleeper's delay is automatically interrupted when the pipeline is stopped, therefore it's not necessary to wait until the thread wakes up. A plugin which uses a thread and logs the number of events per minute follows:

```
class stats : public threaded_plugin {
 protected:
  void process(json &&message) override {
    i++;
    output(std::move(message));
  }

  void loop() override {
    logger.info << i << " events per minute." << std::endl;
    i = 0;
    get_loop_sleeper().sleep_for(std::chrono::minutes(1));
  }

 private:
  std::atomic<unsigned long> i{0};
};
```

The usage of `mutexed_plugin` is the same as of a `base_plugin`, only the process method is guarded with a lock. However, the `threaded_mutexed_plugin` not only locks the process method, but `locked_loop` and `unlocked_loop` methods replace the `loop` method of the `threaded_plugin`. As the names suggest, both methods are executed in a loop, one while the common lock is locked and the other while it is not. Below is a file output, which shows such functionality by flushing the output stream once per second, not after each event.

```
namespace file {
class output : public mutexed_threaded_plugin {
 public:
  explicit output(const static_option<std::string> &file_name) {
    file.open(file_name->value(), std::ios_base::app | std::
        ios_base::binary);
  }

 protected:
  void process(json &&message) override {
    file << message.get_string();
  }

  void locked_loop() override {
    file.flush();
  }

  void unlocked_loop() override {
    get_loop_sleeper().sleep_for(std::chrono::seconds(1));
  }

 private:
  std::ofstream file;
};
}
```

3.4 Features

Three of Wolf's key features are discussed in this section. Firstly, the implementation of a persistent buffer is detailed, since it was highly optimized to allow buffering between any two plugins while not decreasing the performance when not needed. Secondly, the generic way to get a parameter either from a command line or from a processed event is discussed. The last feature is an alternative way to build Wolf using a docker image, which decreases the number of required dependencies in comparison to a native build.

3.4.1 Persistent buffering

The processing pipeline consists of multiple connected plugins. It typically starts with a plugin which receives data (e.g. TCP or file input) and ends with a plugin that writes altered data somewhere else (e.g. http output). However, all plugins inherit from the same class and there are no restrictions whether any plugin can or cannot receive/-generate new data, even in the middle of the pipeline. Therefore, there is a buffer in front of every plugin that can always receive data. That way, even when some plugins are blocked due to slow output or long processing time, no other plugins need to wait, their output is simply redirected to the buffer of the following plugin. A good example is a plugin which counts processed events and each minute creates another event which contains that statistic. Even when no events are processed due to filled output, this plugin still can emit new events into a buffer without blocking itself.

Wolf distinguishes two types of threads, processing and non-processing ones. The non-processing thread provides functionality for one specific plugin only, for example, each TCP input creates its own thread to receive incoming data. However, when a non-processing thread outputs an event to the next plugin, it is always buffered, so the thread is made available as soon as possible. The processing threads then periodically check all buffers and process any waiting events. These, on the other hand, buffer processed events only when necessary, when the following plugin is marked as full. That means that even though a buffer is located in front of all plugins, they are only used when necessary.

When an event is buffered, it is firstly simply put into a queue located in system memory, not to decrease performance when not necessary. That is because buffering events onto a disk is much slower, moreover the events have to be serialized first and deserialized when loaded back. However, when events do have to be buffered onto a disk, they are also compressed first, to decrease the buffer size. Therefore the buffer works differently when it fits into a system memory and when it does not.

The buffer is designed to be as non-blocking as possible for the threads that want to input data into it. Therefore, even in memory only mode, there are two queues, front and back one, both having a lock

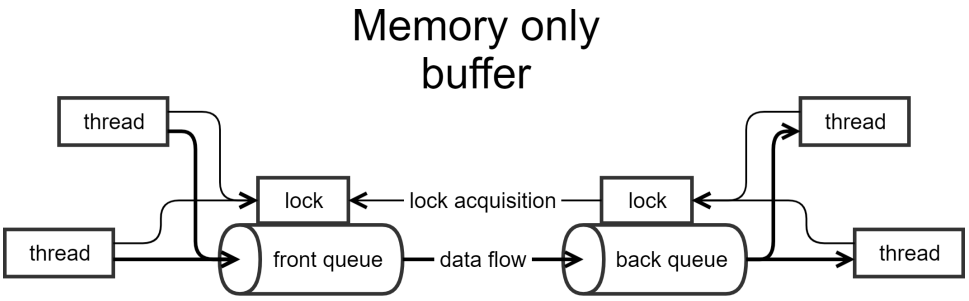

Figure 3.2: Schema of memory only buffer

(figure 3.2). When an event is to be inputted, the thread locks the front queue, inputs the event (assuming the queue is not full) and frees the lock. To retrieve an event the processing thread locks the back queue and when not empty it gets an event and frees the lock. In case the back queue is empty, it locks the front queue as well and swaps the queues, frees the front queue, and checks for emptiness again. That way, the threads that input data into a queue do not compete as much for access with the threads that retrieve the data, moreover, when a thread that retrieves data needs to lock the front queue, it is only for a brief time.

When the front queue gets full and the data has to be buffered onto a disk, the process of inserting and retrieving data from the queue gets much more complex, although it is designed using the same principle, that inputting data should be as non-blocking as possible. In addition to the file-system buffer provided by STXXL library [63], two more memory queues are added. An overview of all queue components can be seen on a figure 3.3, detailed description of how the data is inserted and retrieved is provided in the appendix B.

3.4.2 Generic parameters of plugins

Because the pipeline partly consists of generic plugins, for example, an HTTP output which posts a processed event to a URL, there should as well be a generic way to configure such plugins. Therefore Wolf provides a generic `option<type>` class, which can be used as a parameter of the plugin's constructor. There are three types of options that

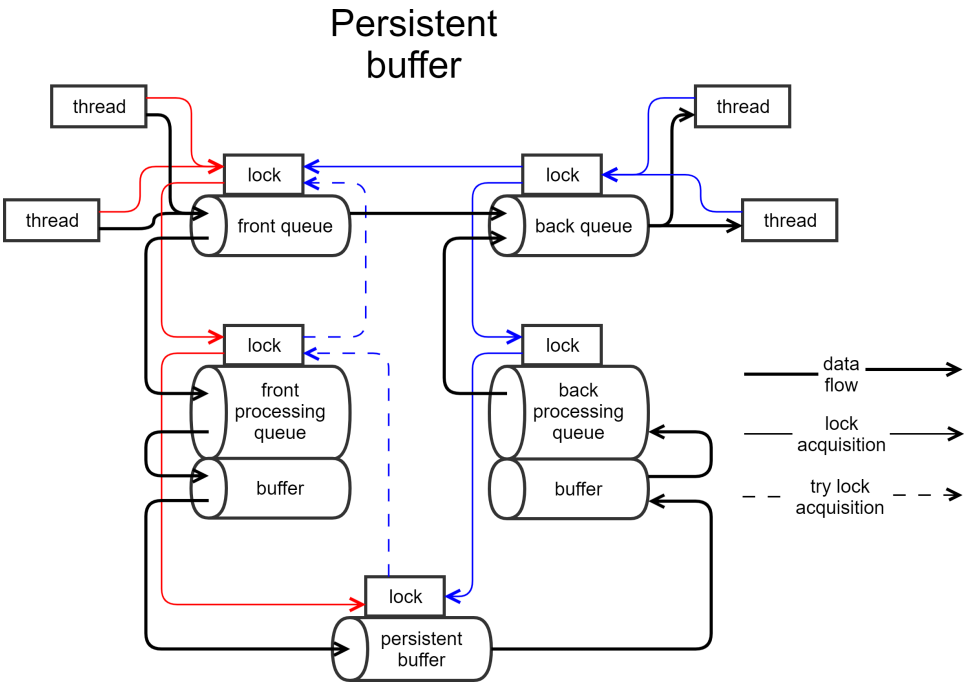

Figure 3.3: Schema of persistent buffer

can be then passed to a plugin when constructed. Firstly, the constant option is simply a static value set before the compilation itself.

Secondly, the input option can be used to pass a value into a plugin from the command line, when Wolf is started. When the input option is specified, it's description is also added to the help command, invoked by running the executable with `-h` option. The input options enable a user to configure certain parameters of the pipeline without the necessity to recompile it, which makes it much easier to use.

Lastly, the event option is used when the plugin needs to extract the parameter's value from each one processed event. An example would be an HTTP plugin, which posts an event to a URL directly specified in the processed event. When the event option is instantiated, a lambda function is passed into it, which takes the processed event and returns the required value.

However, for some use-cases, it is not desired to allow using the event option, for example when specifying a port on which TCP input listens for incoming data. Then, instead of the `option<type>`, the `static_option<type>` can be used, which is not a parent of the event option and therefore only the constant and input options are available to use.

3.4.3 Build using docker image

The last discussed feature is building Wolf using a docker image. But to begin with, the native build is made as easy to use as well. Most importantly, all dependent libraries are already specified in CMake files, therefore only two commands are needed to build Wolf: the CMake configure and build command. In the configure phase, all dependencies are downloaded and built from their GitHub repositories, which makes the build significantly easier to use. However, building Wolf natively still requires a specific C++ compiler and CMake version, which can be difficult to get, especially if you already need another version for different projects.

Therefore a docker image was created, which serves two purposes. The first one is, of course, the ability to build a Wolf pipeline without any dependencies (except for Docker), which is especially useful for building Wolf as a part of a continuous integration pipeline. Secondly, the docker image provides exact specification not only of the used

C++ compiler and CMake, but also of the whole environment. That is useful when troubleshooting errors while trying to build Wolf natively, because it is possible to compare both environments and determine the difference.

There are two distinct commands to build Wolf using Docker:

```
docker run --rm -v ${PWD}:C:\wolf lamanchy/wolf_win
docker run --rm -v ${PWD}:/wolf lamanchy/wolf_linux
```

one for each operating system. In case the current directory is empty, this command creates an empty Wolf project with an example of a pipeline configuration and a simple custom plugin. Otherwise, assuming the current directory contains Wolf project, it compiles provided pipelines and puts the resulting binaries into the `install` subdirectory.

3.5 Implementation details

This last section more closely describes some of Wolf's implementation details. Firstly, the internal classes are overviewed, together with some reasoning behind their design. Secondly, the flow of event processing is described. Overview and categorization of implemented plugins follows, together with a quick comparison to Logstash plugins. Lastly, there's a description of a pipeline's deployment.

3.5.1 Architecture overview

One of the most important principles when designing Wolf was readability of its pipeline. While it was clear, that it won't be as readable as Logstash's custom DSL language, a lot of effort was put to make it as readable as possible. For example, when referring to the `plugin` class, the class is actually named `plugin_base`, the `plugin` keyword is a name of `std::shared_ptr<plugin_base>` type. However, this specific distinction is for purposes of this thesis neglected, but more importantly, the design decision of making Wolf as easy to use as possible is apparent at multiple other places as well.

Class diagram (figure 3.4) shows all core classes of Wolf implementation, and to make it more readable, only public and protected methods and attributes are shown. Also, the `logger` class is used by

most of the other classes, so that relation is omitted for readability as well. A quick overview of each class follows.

Pipeline

The pipeline is created by passing `argc` and `argv` arguments, or by already created `options`. The options are parsed and then plugins can be registered into the pipeline. Finally when the pipeline is executed using `run` method, it starts all plugins and processing threads, then waits for plugins to end or for an interrupt to be received.

The `register_plugin` method has also an overload, which enables plugins to be registered in a much nicer way. The two following examples provide the same functionality:

```
p.register_plugin(
  make<cin>(),
  make<from::line>(),
  make<stats>(),
)
```

```
p.register_plugin(
  make<cin>()->register_output(
    make<from::line>()->register_output(
      make<stats>()
    )
  )
)
```

Pipeline status

This class contains the pipeline's configuration and status, which is shared by multiple other classes. Even though it is a globally shared state (which is often described as a bad design), it was used in this way specifically for that purpose, to globally allow only one pipeline to be created, and to check whether all input options were specified before that. This way, although it limits Wolf's potential (running multiple pipelines in multiple threads), the structure of pipeline's configuration is exactly defined and developer of such pipeline needs to abide the structure for sake of much better readability.

Plugin

This plugin, together with its three subclasses (`mutexed_plugin`, `threaded_plugin` and `mutexed_threaded_plugin`), is a base class to inherit from when implementing any custom plugin. The protected

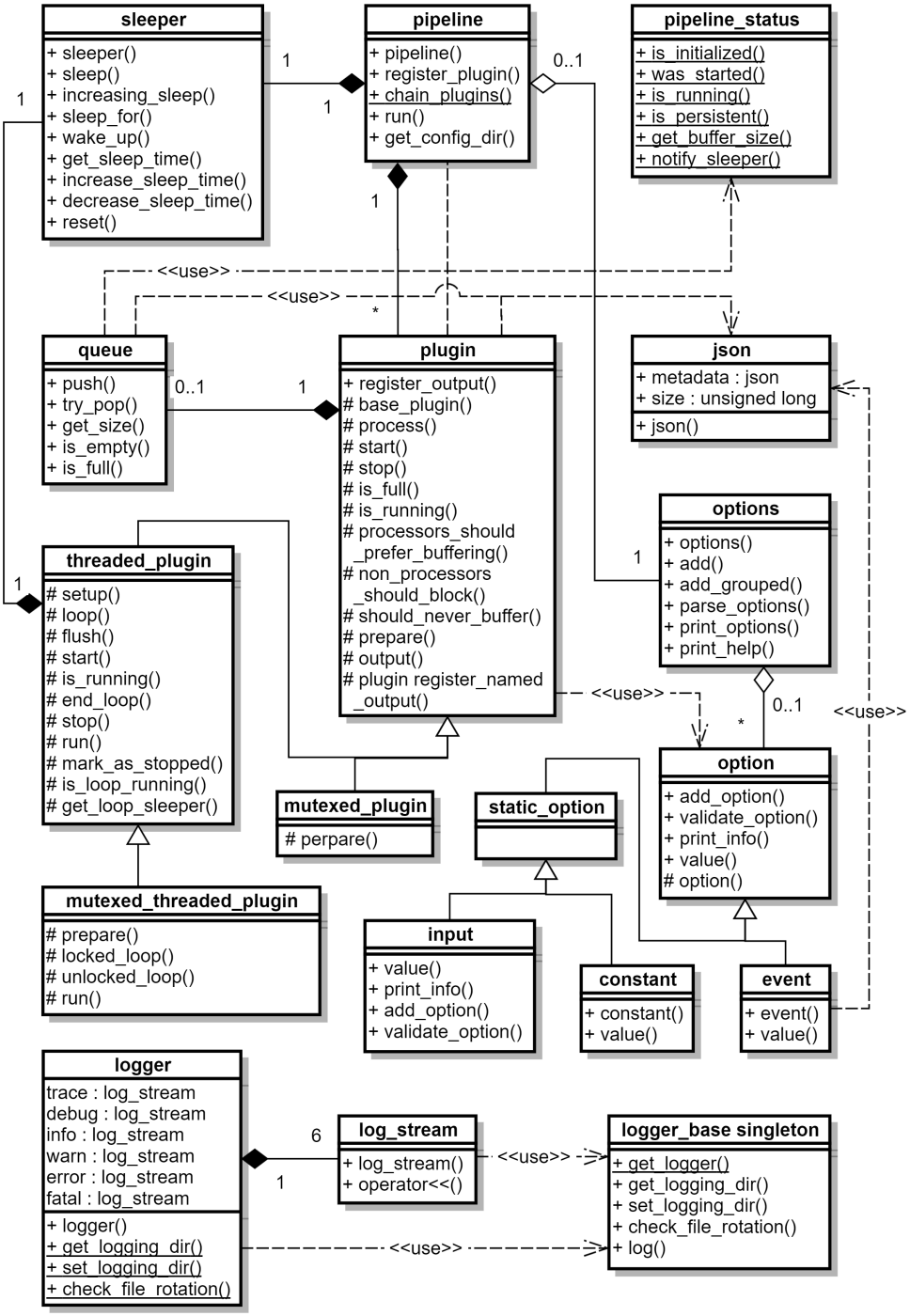

Figure 3.4: Class diagram

methods are those, which can be overridden to provide custom functionality. That is except three methods, which should be called when initializing plugin in its constructor and which alter the way events are buffered. It's:

`processors_should_prefer_buffering`	used at `mutexed` plugins, it's better to put outputted event into a buffer, rather then to process following plugins while holding a lock
`non_processors_should_block`	used at all input plugins, simply to stop receiving events when buffer is full
`should_never_buffer`	to disable buffering entirely for this specific plugin

Queue

While queue's `push` method works as expected, taking one `json` event and adding it into the queue, the `try_pop` method, instead of returning a popped event, takes a callback function as an argument. The function is used to process the popped event in case the queue is not empty, this way multiple threads can easily access the queue without complicated synchronization.

Options

The options class is used for definition and parsing of input parameters. It can be created before the creation of the pipeline to specify custom input parameters, otherwise, the pipeline will create it by itself.

Usage of the `option` class and it's subclasses was already discussed in section 3.4.2.

Sleeper

This is purely helper class, it is initialized with minimum and maximum sleep duration and provides methods to pause execution of the current thread. However, when sleep time is increased over 10 milliseconds, the thread is put to sleep using `condition_variable` and can be

waken up by another thread. That allows prolonged sleep durations with an immediate reaction when needed.

Json

This class is used to add two additional attributes to the original `taojson` class: `metadata` and `size`. The `metadata` parameter is a `json` as well, storing information provided by plugins, for example, a TCP port on which the event was received. The `size` parameter simply stores a number of actual events inside this specific JSON, since the JSON can be simply a string value containing many serialized events.

Logger

Lastly, a simple custom logger to be used by the whole application. It supports log file rotation and the logging itself is implemented using `std::stringstream`:

```
logger.info << "Successfully connected to " << host << ":" <<
   port << std::endl;
```

3.5.2 Event processing

As quickly mentioned in section 3.4.1, there are two types of threads, processing and non-processing ones. The non-processing threads are created as a part of `threaded_plugin` or `mutexed_threaded_plugin` and events created by these threads have a straightforward flow, they are directly outputted into a buffer of the following plugin, to be processed by the processing threads.

Those are created by `pipeline` class, their number is determined by an input parameter `-t, --threads`, which defaults to a number of CPU cores. Processing threads periodically check all plugins' buffers and process any waiting events (figure 3.5). Upon processing an event, their sleep duration is halved and is only increased (by doubling) after all plugins are processed. That results in a rapid decrease of any sleep time when there are events to process, and a gradual increase up to one second when there are not.

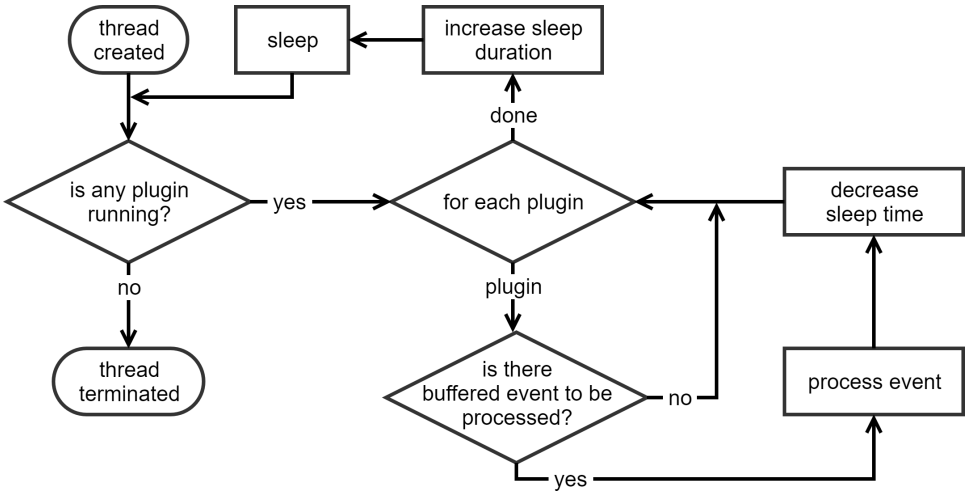

Figure 3.5: Life-cycle of processing thread

3.5.3 Implemented plugins

Table 3.2 shows all implemented plugins, that are a part of Wolf library, sorted by parent class. However, there is also another categorization, input and output plugins, which only receive/send events, have a `::input`/`::output` suffix, while deserialization/serialization plugins have `from::` and `to::` prefix.

There are in total 6 input plugins, 10 filter plugins and 5 output ones. When compared to Logstash's 48/41/50, it seems like quite a few, however, many of Logstash's output plugins are just thin wrappers over HTTP/TCP connection. For example, while Logstash provides a custom output for the InfluxDB [64] database, Wolf uses just a plain HTTP output. Also, many Logstash's filter plugins can be simply replaced by Wolf's lambda plugin, that alters processed event with a custom function. But still, Logstash does provide a lot more built-in functionality than Wolf.

3.5.4 Pipeline deployment

The result of Wolf compilation is a simple executable, that can be run from a command line and stopped by sending the interrupt signal. The first interrupt signal stops all inputs and Wolf exists after all buffered data are outputted, the second received signal throws away

parent class	plugins
plugin	copy, drop, filter, generator, lambda, regex, to::string, to::line, to::influx, to::compressed, from::string, from::line, from::compressed, tcp::output
mutexed_plugin	cout
threaded_plugin	stats, cin, http::output, kakfa::input, kafka::output, stream::input, file::input, tcp::input
mutexed_threaded_plugin	collate, elapsed, file::output, stream_sort, time_sort

Table 3.2: Implemented plugins

any buffered data and immediately stops the pipeline. This way, Wolf can be easily deployed for example on Windows using NSSM [65], which creates a service out of an executable and is able to stop it using the interrupt signal.

Except for custom input options, specified during pipeline configuration, there is a set of pipeline's internal options, `-h` or `--help` option, for example, makes Wolf to print all (custom and internal) options and exit. The others are:

`-c, --config_dir arg`	Path to configuraition files (default: executable's path)
`-l, --logging_dir arg`	Path to logs (default: executable's path)
`-p, --persistent`	Enables persistent buffering
`-b, --buffer_size arg`	Size of memory buffers, affects system memory usage as well as performance (default: 4096)
`-t, --threads arg`	Number of processing threads (default: number of CPU cores)
`--test`	Run Wolf and exit automatically after 1 second
`-h, --help`	Print help

4 Comparison of Wolf and Logstash

The last chapter focuses on verification that the replacement of Logstash by Wolf was successful. Firstly, a set of benchmarks was created to evaluate the performance of both tools. The benchmarks compare specific behaviour of Wolf in general scenarios (as a performance of persistent buffer) as well as the exact usage within Y Soft (the parser and collector configurations). That is followed by a comparison of both tools' usage, how simple they are to build, use, but also debug etc. Lastly, both tools are used to monitor a performance test of YSoft SafeQ[1], therefore providing a real usage comparison as well as final verification, that Wolf can fully replace Logstash.

4.1 Comparison benchmarks

All benchmarks included in this section can be run using a Python script located in a GitHub repository [66]. The script expects that CMake and C++ compiler is installed and runs only on a Linux operating system, however, those are all required dependencies. It downloads Logstash, Java and Wolf with all dependencies and compiles all necessary binaries. Then it automatically executes all benchmarks, collects needed metrics and generates graphs used below.

Both Logstash and Wolf had their configuration optimized to utilize multiple threads, Logstash memory buffer size was adjusted as well for better performance. All benchmarks receive and output data via TCP connection, Netcat[2] is used to generate data as well as to receive it. The communication uses localhost only, so the net transfer is not a bottleneck, and there are also multiple input Netcat instances to achieve saturation. Results shown below originate from Azure virtual machine F2s_v2 [68], which has two processing cores, 4 GB of system memory and SSD disk. Data used for processing is a sample of YSoft SafeQ logs collected from a performance test, therefore are a good representation of real-world data.

---

1. YSoft SafeQ is the main product of the Y Soft Company, which simplifies and secures print management.
2. Netcat [67] is a simple but powerful Unix utility to write and read data across network.

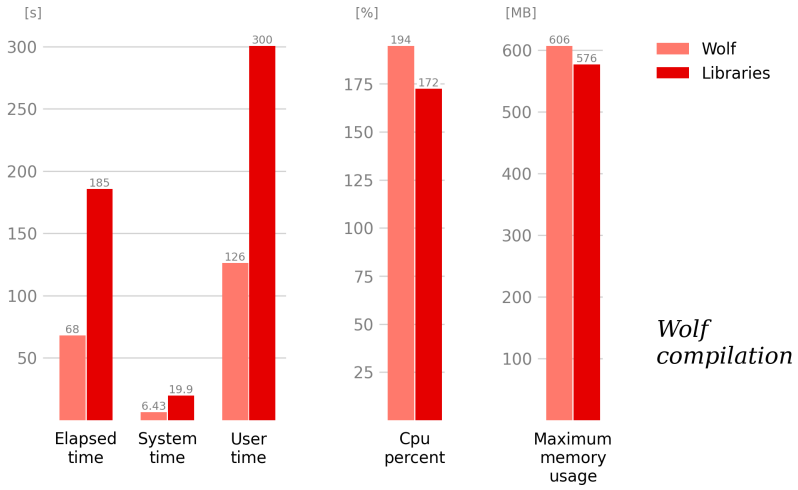

Figure 4.1: Wolf compilation

The benchmarks can be divided into two categories, a general and a specific one. The general benchmarks focus on comparing Logstash and Wolf without any configuration specific for Y Soft, while the other ones compare both tools with the configuration of the collector and the parser.

4.1.1 General benchmarks

The first figure (4.1) shows Wolf library compilation time without any comparison to Logstash since only Wolf needs to be compiled. Note that this is a compilation of Wolf library, not the processing pipeline. Also, the library compilation contains phase of dependencies download, so the total elapsed time depends on the speed of internet connection. But it is useful to know that the compilation of Wolf library with all dependencies takes about four to five minutes.

The second figure (4.2) then shows disk space needed to compile a pipeline. The tool's size columns refer to the size of Logstash and Wolf, but without any dependencies, which would be Java for Logstash and CMake and C++ compiler for Wolf. The Linux docker images which do contain all dependencies show that in total the size is roughly the same. The Windows docker image has unfortunately about 10 GB in size, most of which is the Windows docker image and Visual Studio C++ compiler. However, the biggest difference worth mentioning is

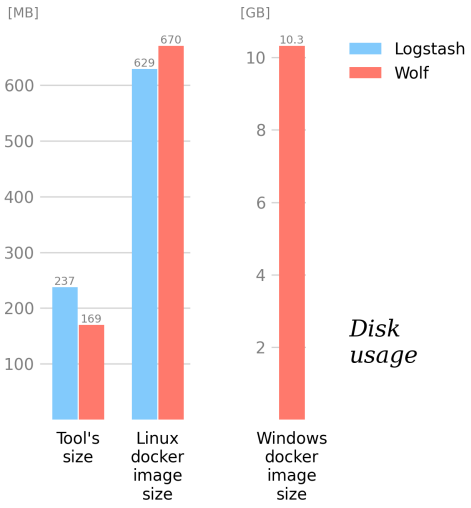

Figure 4.2: Disk usage

that Logstash requires all this disk size to run the processing pipeline, however, Wolf compilation produces a binary of about few megabytes, which can be run independently.

Benchmarking of compilation time follows (figure 4.3). It's important to note that for purposes of all these comparisons, Logstash's time to start up is treated as a compilation time and is removed from all subsequent benchmarks. That is because Wolf requires close to none resources to start, in comparison Logstash takes 20 seconds while fully utilizing both CPUs before the pipeline is started, which would skew all the subsequent results and make the comparison much more difficult to evaluate. To compile an empty configuration, Wolf is twice as fast and four times more CPU efficient, requiring 45% more system memory. However, such use-case is far from a real-world usage, comparison of the collectors' configuration can be found in the next subsection.

The next four benchmarks run with an empty configuration, data is received via TCP input and immediately outputted via another TCP connection, without any processing in between. Also, for the first three benchmarks, where there is no buffering enabled, both Logstash and Wolf are compared to a Netcat.

First benchmark (figure 4.4) shows resource consumption when nothing is being processed. All tools run for 10 minutes to ensure as

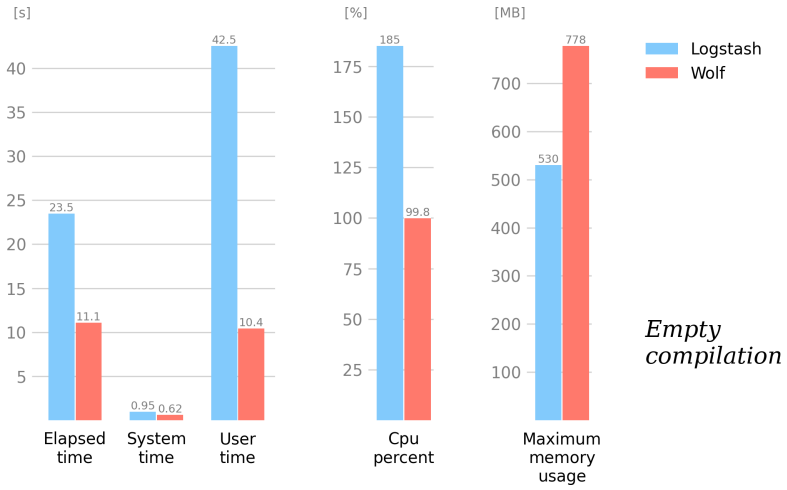

Figure 4.3: Empty compilation

precise result as possible. Both Wolf and Netcat use close to none CPU or system memory, while Logstash needs about 500 MB of system memory and 1.5% of CPU time. That shows that running Logstash even with minimal configuration and no processed data consumes a noticeable amount of system resources while running Wolf does not.

The next benchmark (figure 4.5) compares tools while processing a trickle of events, 1,000 per second. It runs for 3 minutes as all subsequent benchmarks. This benchmark is good for comparison of CPU usage efficiency since all tools process the same amount of data. And while Wolf is four times less CPU efficient in comparison to highly optimized Netcat, it is still 80 times more efficient than Logstash, promising good results when testing with non-empty configuration.

In comparison to the previous benchmark, the next one (figure 4.6) uses eight Netcat processes to write as much data as possible again for three minutes. That takes away some system resources, but the bottleneck is always tested tool's performance, not shortage of system resources. Logstash's elapsed time is a bit longer, since all tools exit when their internal buffers are empty, and that takes some time. While the gap in CPU efficiency closed a bit, Logstash seems to handle the higher load better from this point of view, Wolf still performs much better in both the number of messages processed per second and CPU

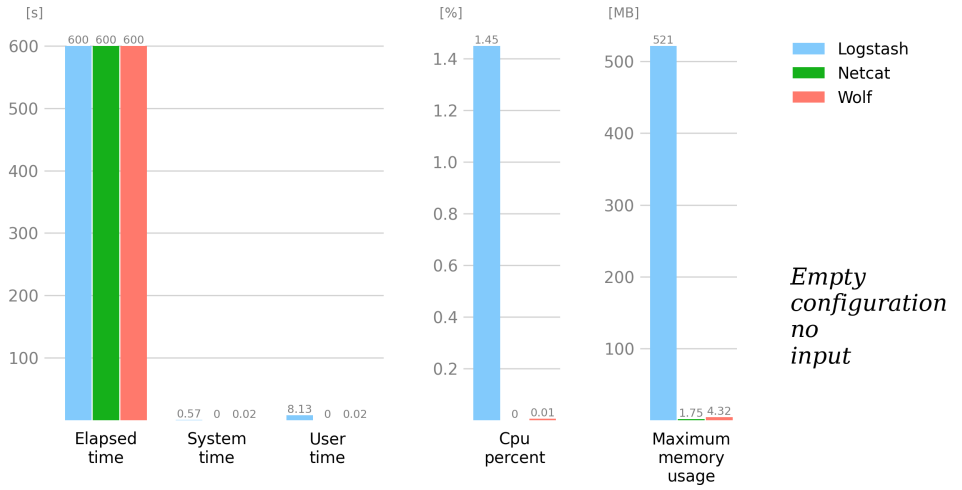

Figure 4.4: Empty configuration no input

efficiency. Wolf also has comparable throughput to Netcat, being able to utilize multiple threads.

The last benchmark (figure 4.7) focuses on the performance of persistent queue, therefore Netcat is not included this time since it does not provide this functionality. Both tools are subjected to full load for three minutes as in the previous benchmark, however, they can only output the data after those three minutes. That forces them to store the data on disk, resulting in one to two gigabytes of used storage. The benchmark is ended, when all received data is outputted.

The first result is that although Wolf's queue size is larger, it also processed more messages. To compare, the average length of a processed event is 477 characters, therefore Logstash stores extra data with no compression and additional metadata. Wolf stores extra data as well, but uses gzip compression before writing data to disk. Therefore, the resulting event size is 36 times smaller. However, it is still far from the 130 times smaller size achieved when compressing the whole input file using standard gzip compression settings.

Secondly, because of used compression, Wolf's CPU efficiency is now only 10 times better than Logstash's one. Logstash also outputted received data much faster, spending less time decompressing it. However, Wolf was able to process 45 times more messages than Logstash, while requiring half the disk size per message.

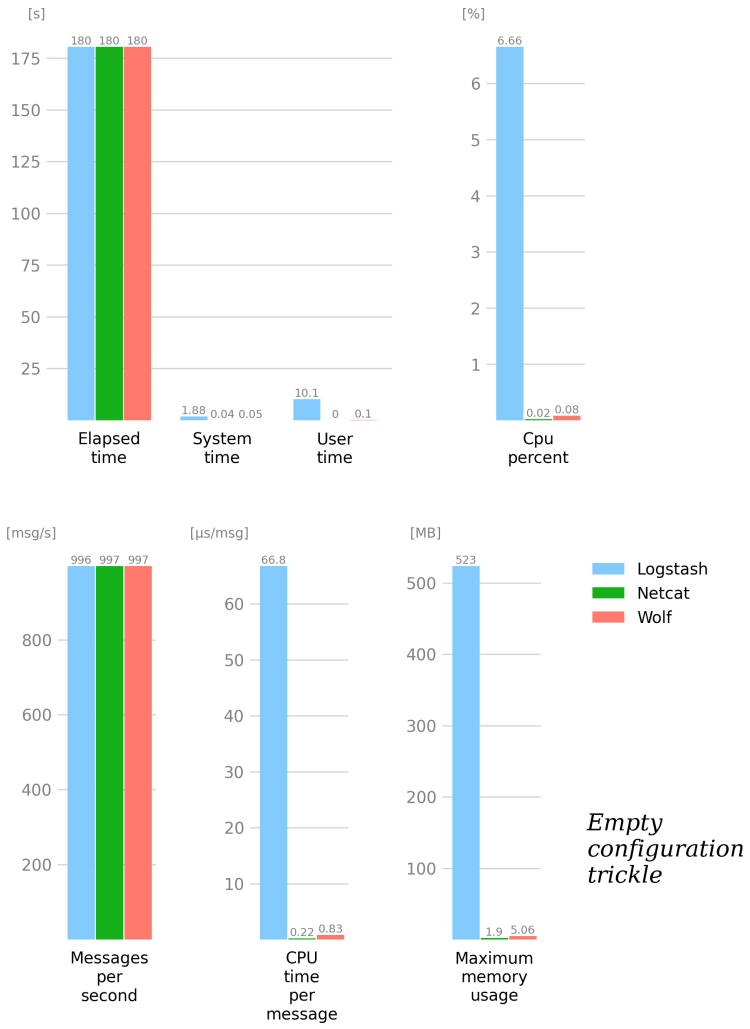

Figure 4.5: Empty configuration trickle

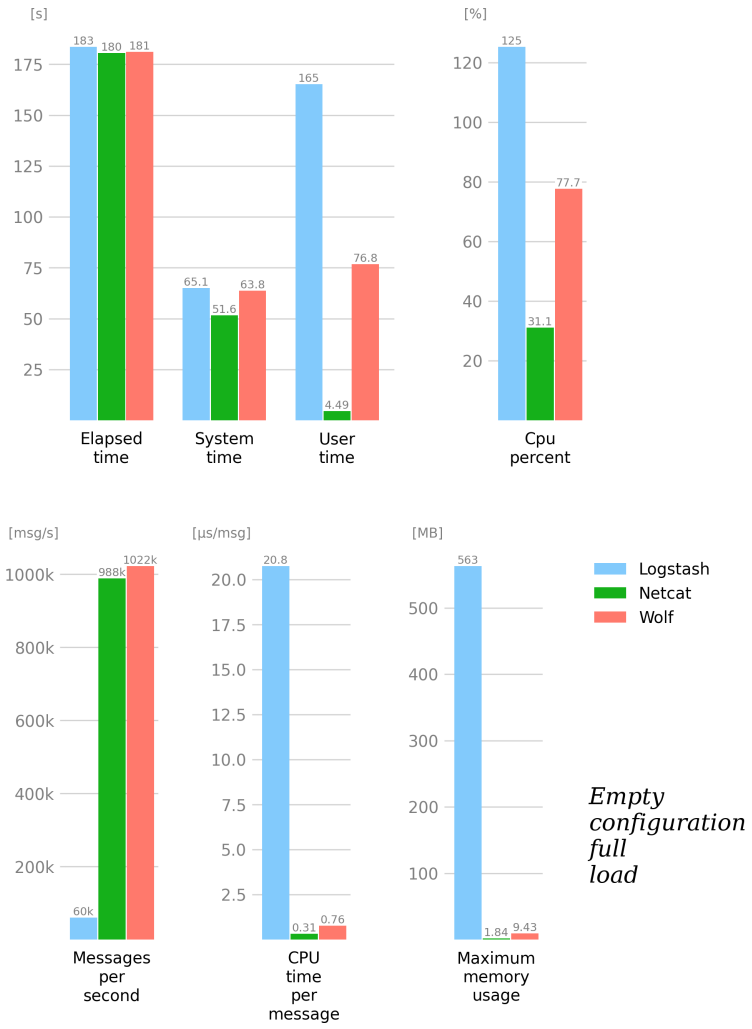

Figure 4.6: Empty configuration full load

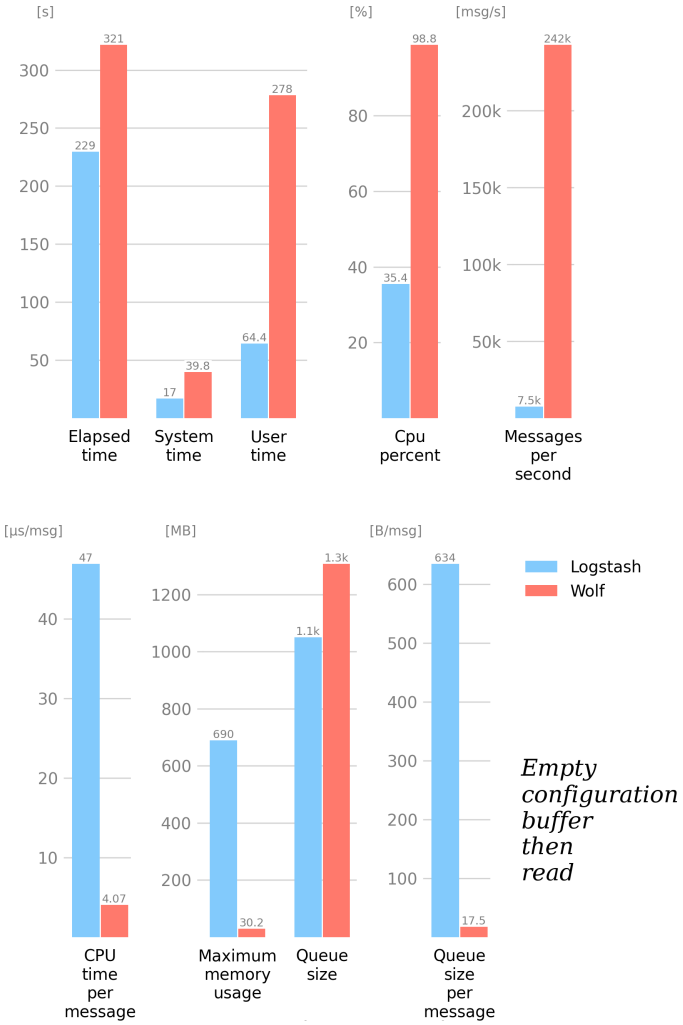

Figure 4.7: Empty configuration buffer then read

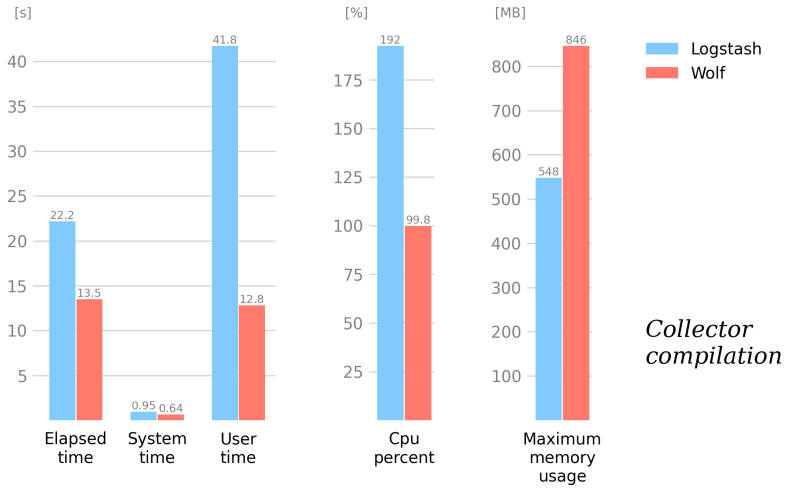

Figure 4.8: Collector compilation

4.1.2 Specific benchmarks

Benchmarks in this section compare the specific configurations used within Y Soft monitoring pipeline. The configurations are not exactly the same, firstly, some changes were made to enable measurement of processed data. Secondly, the configurations used by Y Soft have evolved in the meantime, Wolf already provides functionality which wouldn't be possible to achieve with Logstash. But the overall changes were kept to minimum, all of the main processing functionality remained intact and most importantly, the used configurations of Wolf and Logstash do provide the same functionality.

Firstly, for completeness, there's a comparison of the collector's compilation (figure 4.8). The result does not differ significantly from compilation of empty pipeline (figure 4.3), therefore overview of the parser's compilation is skipped entirely. However, it is worth reminding, that compilation of Logstash stands for the time before it starts up and is present each time Logstash is run but excluded from further benchmarks. For more information, check the description of compilation of the empty pipeline in the previous subsection.

The second benchmark (figure 4.9), again for completeness, is the collector being run for 10 minutes without any data. The result is very similar when running the same test with empty configuration (fig-

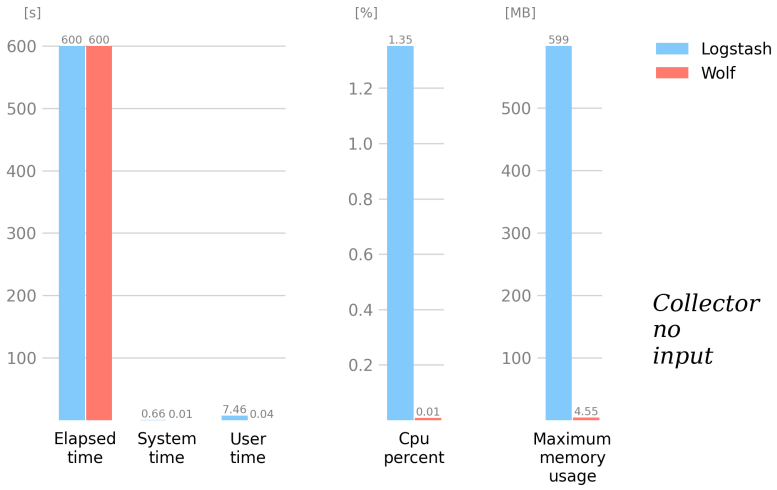

Figure 4.9: Collector no input

ure 4.4), therefore it verifies that adding configuration to the pipeline does not increase consumption of system resources when standing by.

The next benchmark (figure 4.10) shows what is close to real usage of the collector, processing a steady stream of incoming data, 1,000 messages per second. The requirements for replacing Logstash were: increase CPU efficiency, decrease system memory consumption and decrease disk footprint, all of that by an order of magnitude. The results show that the CPU consumption was decreased 12.8 times and memory usage by two orders. Lastly, Wolf's executable has 2 MB while Logstash with Java totals over 400 MB.

Last collector benchmark (figure 4.11) compares the behaviour of both tools while under maximum load for three minutes, with persistent buffer enabled. The result shows that Wolf was able to keep up with the load, while Logstash needed to buffer the incoming load, taking an extra 140 seconds to process buffered data when the load ended. Logstash processed 5 times fewer messages in total and 8.7 times fewer per second, while still being more than 8 times less CPU efficient.

For the parser, compilation and no input benchmarks are skipped, since they do not contain any new information. So the first parser's benchmark (figure 4.12) compares both tools under a trickle of data,

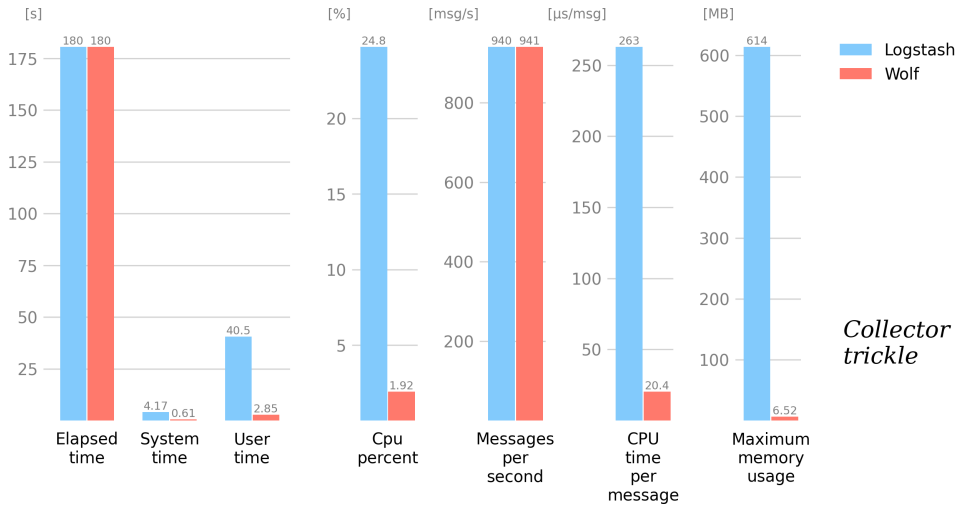

Figure 4.10: Collector trickle

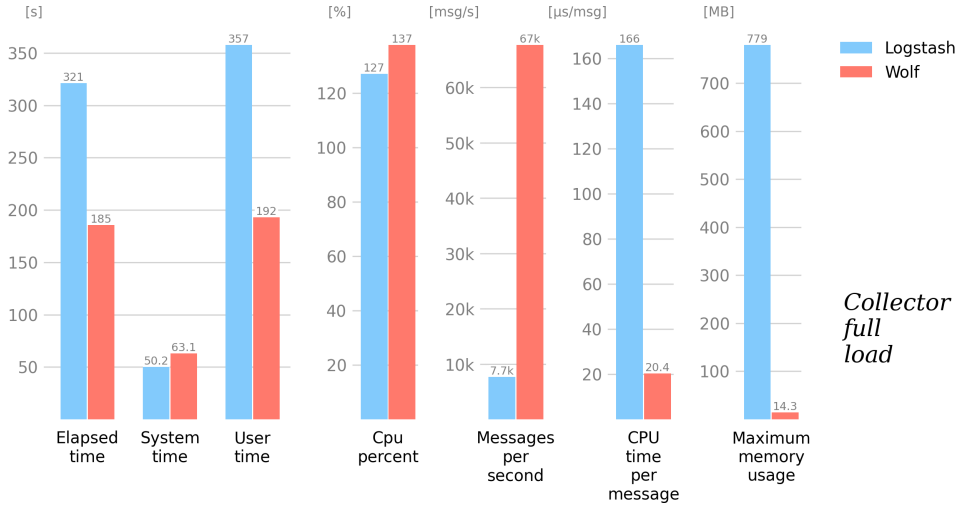

Figure 4.11: Collector full load

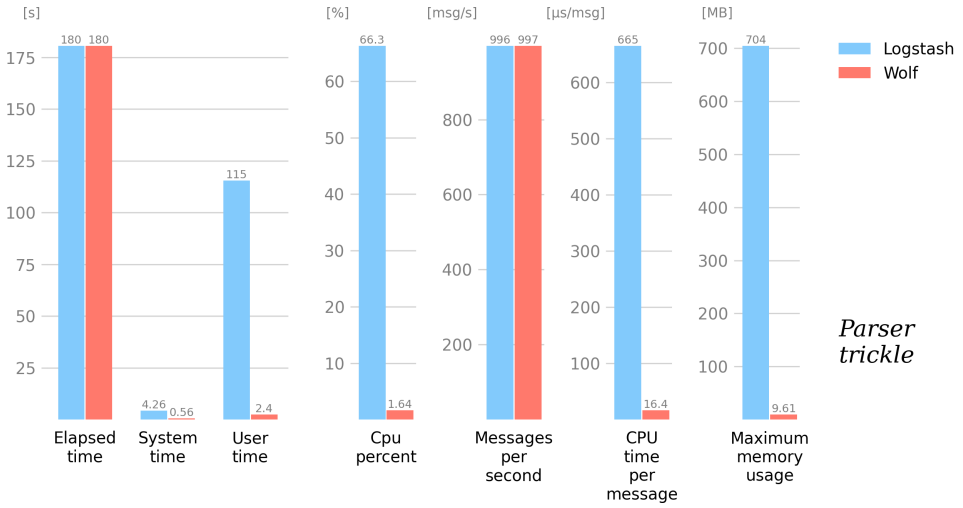

Figure 4.12: Parser trickle

showing that Wolf is 40 times more CPU efficient when processing the data.

However, the requirement for replacing Logstash was achieving an order of magnitude higher throughput, which shows the last benchmark (figure 4.13). To be exact, Wolf was able to process 18 times more messages per second, which is mainly caused by the efficient regex parsing.

4.1.3 Summary

Wolf proved to be a significant improvement in comparison with Logstash in every measured aspect, easily satisfying specified requirements for replacement. Most noticeably both the system memory and disk usage were lowered by two orders of magnitude.

4.2 Comparison on a real environment

Lastly, both Logstash and Wolf were deployed as was their intended use-case to monitor a performance test of YSoft SafeQ. Part of the monitoring stack is also a collection of system metrics, which can be used to evaluate the performance of the monitoring itself. That will

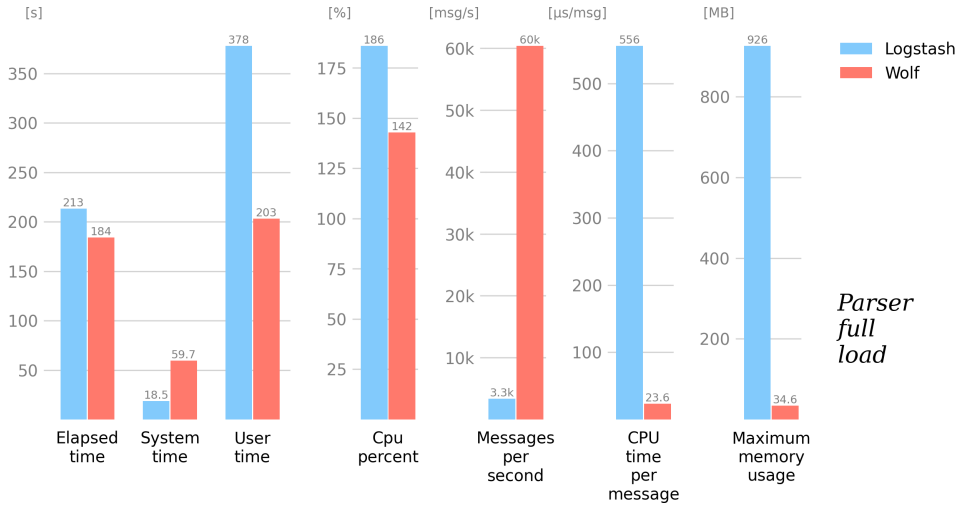

Figure 4.13: Parser full load

not only give us a direct comparison on a real-world environment, but it will also verify Wolf's functionality.

The test was run on a single-server environment, with a load of 60 print jobs per minute, which corresponds to 4,200 logs per minute. Also, approximately 1000 metrics per minute were collected. Presented values (figures 4.14, 4.15, 4.17 and 4.17) are averages over 24 hours of data, collected during the performance test. `calf_parser` and `java` processes are collectors, while `little_stack_parser` and `java#1` are parsers.

The most notable difference in comparison to the benchmarks at the beginning of this chapter is that parser achieves much better CPU efficiency in comparison to the collector. That is because collector filters logs marked as DEBUG and TRACE, although collector outputs to parser the 2,100 logs per minute, it actually receives over 20,000 of logs, most of which are filtered out.

All other differences to the benchmarks (as Logstash's higher system memory usage) are caused by a different operating system since benchmarks were run on a Linux OS and the performance test above on a Windows one, because that's the preferred system to run YSoft SafeQ on.

To summarize the results, Wolf achieved 5.3 times and 19.5 times better CPU efficiency for the collector's and parser's use-case and the

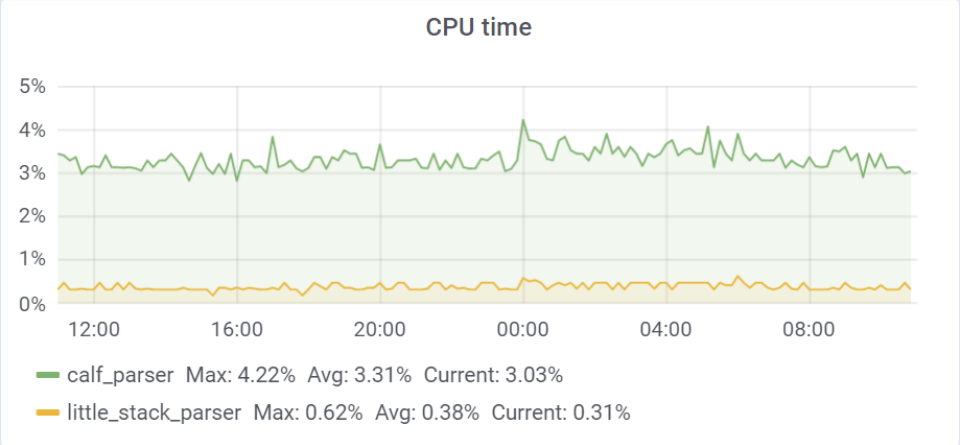

Figure 4.14: Wolf real-world CPU usage

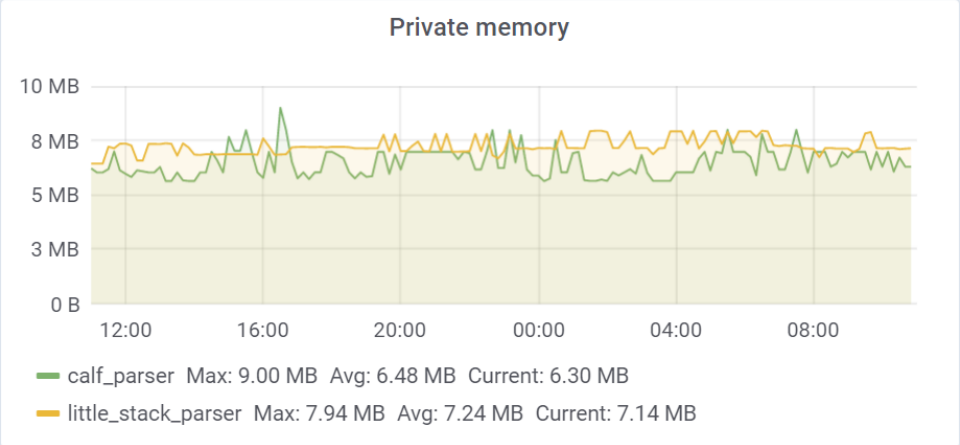

Figure 4.15: Wolf real-world system memory usage

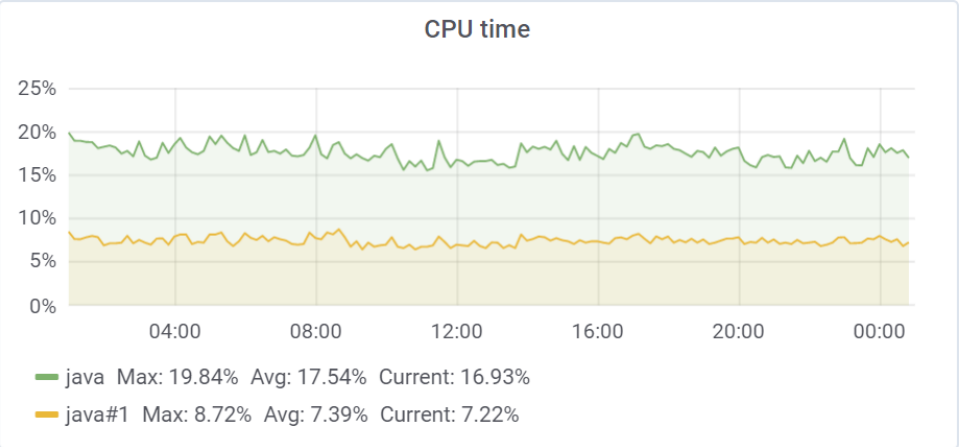

Figure 4.16: Logstash real-world CPU usage

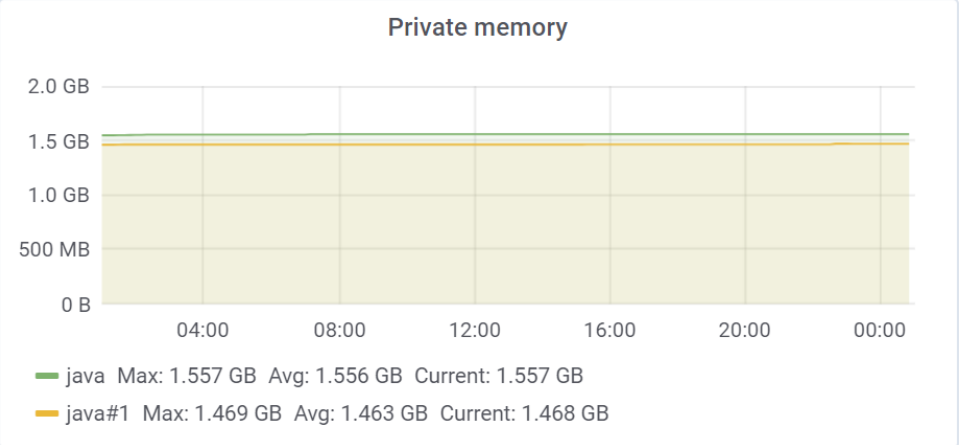

Figure 4.17: Logstash real-world system memory usage

system memory consumption was well over 200 times lower for both cases.

Lastly, it is worth noting that Wolf's CPU consumption could be further lowered by changing the format in which logs are sent from the collector to the parser since for example the timestamp format is strictly set by Logstash. That was not yet done since the format was kept the same so the Wolf's and Logstash's version of both the collector and the parser could be interchangeably deployed. Although Logstash is no longer used within the monitoring pipeline, changing the format takes time since there are many different environments connected to Y Soft's internal monitoring server and both formats will have to be supported during the transition period.

4.3 Comparing usage

This section focuses on a comparison of both tools from various points of view, from the development of the processing pipeline, usage within the Y Soft CI pipeline or user experience.

4.3.1 Pipeline development

It is certainly faster and easier to start developing a Logstash processing pipeline since you only need to download Logstash and the configuration is a plain text file. However, it takes time to get used to its custom syntax, moreover, some constructs are very unintuitive[3]. Which is made even worse by Logstash's long startup time, since that is the only way to verify whether the configuration is correct.

In comparison, Wolf is certainly more difficult to start with. Github repository has to be cloned or a sample project can be created and compiled by a docker image, but to develop a pipeline an integrated development environment would be recommended to set up. However, that will provide the ability to verify correct syntax before the pipeline has to be even compiled, as well as many other useful features such as code navigation and automatic formatting. And especially for

3. For example, to check whether a JSON event contains a field is done by evaluating whether the field contains empty string `("" in [field])` and even that fails when the field contains a numeric value.

more complicated pipelines, being able to split the configuration into multiple files/plugins does the development significantly easier and faster in the long run.

4.3.2 Usage within continuous integration

Although Logstash doesn't have to be compiled ahead, it was still being manipulated by CI servers when building the whole monitoring installation package. The biggest issue came with its many nested subdirectories, the longest relative path is 216 characters. Therefore it was not possible to extract Logstash inside of the build directory because the total path would exceed the 260 characters limit by Windows operating system. That complicated the installation script, because not only the Logstash had to be extracted during the installation, but it severely limited where the final package could be installed, again not to exceed the path length limit.

On the other hand, Wolf needs to be compiled. Fortunately, Docker images make building of Wolf very simple, however, only Linux CI servers support docker build in Y Soft. For the Windows one, Visual Studio compiler had to be installed to all CI servers, which took most of the setup time, because even native Wolf build consists of only two commands to execute.

4.3.3 User experience

Even though neither Logstash or Wolf are directly used by a user of the monitoring solution, replacing Logstash by Wolf made user experience of the installation process much better. Firstly, the client-side installation package is over 10 times smaller, which makes any manipulation with it (as downloading or extracting it) much faster. Secondly, not only because of the time to extract Logstash, but also because of its startup time, it takes two minutes since the installation for the first data to appear in the monitoring server. In comparison, it takes 15 seconds when Wolf is used instead of Logstash, which is very useful for verifying that the installation was successful.

5 Conclusion

This thesis aimed to replace Logstash's usage within Y Soft monitoring pipeline. Since no tool satisfying all set requirements to replace one of the Logstash's use-cases was found, a custom solution was implemented, named Wolf.

Through designed benchmarks and direct comparison on a real environment, Wolf proved to be by an order of magnitude more CPU efficient, alternatively having an order of magnitude more throughput. Furthermore, its system memory usage and disk usage is lower by two orders of magnitude, therefore well satisfying all specified requirements.

Wolf was successfully used to fully replace Logstash in Y Soft monitoring solution, which is not only widely used within its RnD to monitor tens of testing environments, but is currently on a path to be used as a product for customers. Which is as well possible thanks to Wolf, since it made the monitoring solution truly lightweight and performant.

While Wolf will be further developed and maintained by Y Soft, the core library containing all generic plugins stays open-source and free to use, available on Github [58]. Although it does not provide such a wide variety of inputs and output plugins as Logstash, additional ones can be implemented and easily added. However, Wolf is already a very performant and lightweight solution for transforming events transported using TCP connections or Apache Kafka brokers.

Bibliography

1. LOMIČ, Ondřej. Log management in distributed environment [online]. 2017 [visited on 2020-07-07]. Available from: https://is.muni.cz/th/ks1no/.
2. Logstash: Centralize, transform & stash your data [online]. Elasticsearch, 2020 [visited on 2020-07-07]. Available from: https://www.elastic.co/logstash/.
3. ELK Stack: What is the ELK Stack? [online]. 2020 [visited on 2020-07-07]. Available from: https://www.elastic.co/what-is/elk-stack/.
4. Elasticsearch [online]. Elasticsearch, 2020 [visited on 2020-07-07]. Available from: https://www.elastic.co/about/.
5. Elasticsearch: The heart of the free and open Elastic Stack [online]. Elasticsearch, 2020 [visited on 2020-07-07]. Available from: https://www.elastic.co/elasticsearch/.
6. Lucene [online]. The Apache Software Foundation, 2020 [visited on 2020-07-07]. Available from: https://lucene.apache.org/.
7. MACHINE LEARNING: Spot what you might otherwise miss [online]. 2020 [visited on 2020-07-07]. Available from: https://www.elastic.co/what-is/elasticsearch-machine-learning/.
8. Kibana: Your window into the Elastic Stack [online]. Elasticsearch, 2020 [visited on 2020-07-07]. Available from: https://www.elastic.co/kibana.
9. JRuby: The Ruby Programming Language [online]. JRuby, 2020 [visited on 2020-07-07]. Available from: https://www.jruby.org/.
10. The Computer Language Benchmark Game: Java [online]. 2020 [visited on 2020-03-10]. Available from: https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/java.html.

11. The Computer Language Benchmark Game: JRuby [online]. 2020 [visited on 2020-03-10]. Available from: https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/jruby.html.

12. The Computer Language Benchmark Game: Ruby [online]. 2020 [visited on 2020-03-10]. Available from: https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/yarv.html.

13. Logstash Reference [online]. 2020 [visited on 2020-07-07]. Available from: https://www.elastic.co/guide/en/logstash/current/index.html.

14. COX, Russ. Regular Expression Matching Can Be Simple And Fast: but is slow in Java, Perl, PHP, Python, Ruby, ... [online]. 2007 [visited on 2020-07-07]. Available from: https://swtch.com/~rsc/regexp/regexp1.html.

15. Syslog-ng: all Ruby programs [online]. 2020 [visited on 2020-07-07]. Available from: https://www.syslog-ng.com/.

16. Heron [online]. 2020 [visited on 2020-07-07]. Available from: https://heron.incubator.apache.org/.

17. Wallaroo [online]. 2020 [visited on 2020-07-07]. Available from: https://www.wallaroolabs.com/.

18. rsyslog [online]. 2020 [visited on 2020-07-07]. Available from: https://www.rsyslog.com/.

19. Hindsight [online]. 2020 [visited on 2020-07-07]. Available from: http://mozilla-services.github.io/hindsight/.

20. Benthos [online]. 2020 [visited on 2020-07-07]. Available from: https://www.benthos.dev/.

21. Onyx [online]. 2020 [visited on 2020-07-07]. Available from: http://www.onyxplatform.org/.

22. Monyx [online]. 2020 [visited on 2020-07-07]. Available from: https://monix.io/.

23. Heka [online]. 2020 [visited on 2020-07-07]. Available from: https://hekad.readthedocs.io/en/v0.10.0/.

24. S4 [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/apache/incubator-retired-s4.
25. SPQR [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/ottogroup/SPQR.
26. Scribe [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/facebookarchive/scribe.
27. Tigon [online]. 2020 [visited on 2020-07-07]. Available from: https://docs.cask.co/tigon/current/en/index.html.
28. Hailstorm [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/hailstorm-hs/hailstorm.
29. Teknek [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/edwardcapriolo/teknek-core.
30. Concord [online]. 2020 [visited on 2020-07-07]. Available from: http://concord.io/.
31. StreamBox [online]. 2020 [visited on 2020-07-07]. Available from: https://engineering.purdue.edu/~xzl/xsel/p/streambox/.
32. Saber [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/lsds/Saber.
33. Swave [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/sirthias/swave.
34. Bistro streams [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/asavinov/bistro.
35. Zookeeper [online]. 2020 [visited on 2020-07-07]. Available from: https://zookeeper.apache.org/.
36. Apex [online]. 2020 [visited on 2020-07-07]. Available from: http://apex.apache.org/docs.html.
37. Flink [online]. 2020 [visited on 2020-07-07]. Available from: https://flink.apache.org/.
38. Gearpump [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/apache/incubator-retired-gearpump.

39. Samza [online]. 2020 [visited on 2020-07-07]. Available from: http://samza.apache.org/.

40. Spark Streaming [online]. 2020 [visited on 2020-07-07]. Available from: https://spark.apache.org/.

41. Storm [online]. 2020 [visited on 2020-07-07]. Available from: http://storm.apache.org/releases/current/index.html.

42. Kafka Streams [online]. 2020 [visited on 2020-07-07]. Available from: https://kafka.apache.org/.

43. Akka Streams [online]. 2020 [visited on 2020-07-07]. Available from: https://akka.io/.

44. Hazelcast Jet [online]. 2020 [visited on 2020-07-07]. Available from: https://jet-start.sh/.

45. FS2 [online]. 2020 [visited on 2020-07-07]. Available from: https://fs2.io/index.html.

46. Flume [online]. 2020 [visited on 2020-07-07]. Available from: https://cwiki.apache.org/confluence/display/FLUME.

47. Fluentd [online]. 2020 [visited on 2020-07-07]. Available from: https://www.fluentd.org/.

48. LogAgent [online]. 2020 [visited on 2020-07-07]. Available from: https://sematext.com/logagent/.

49. Flowgger [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/awslabs/flowgger.

50. RE2 [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/google/re2.

51. PERI, Noni. Fluentd vs Logstash: A Comparison of Log Collectors [online]. 2020 [visited on 2020-07-07]. Available from: https://logz.io/blog/fluentd-logstash/.

52. Kubernetes Logging: Comparing Fluentd vs. Logstash [online]. 2020 [visited on 2020-07-07]. Available from: https://platform9.com/blog/kubernetes-logging-comparing-fluentd-vs-logstash/.

53. GARG, Abhimanyu. Logstash vs Fluentd: Which one is better! [online]. 2020 [visited on 2020-07-07]. Available from: https://medium.com/techmanyu/logstash-vs-fluentd-which-one-is-better-adaaba45021b.

54. JEFFS, Ashley. Write a Benthos Plugin: I made it difficult for our job security [online]. 2019 [visited on 2020-07-07]. Available from: http://www.jeffail.uk/post/write-a-benthos-plugin/.

55. Trill [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/microsoft/Trill.

56. LightSaber [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/lsds/LightSaber.

57. CMake [online]. 2020 [visited on 2020-07-07]. Available from: https://cmake.org/.

58. Wolf [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/lamanchy/wolf.

59. Wolf's benchmarks [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/lamanchy/benchmark_wolf.

60. Native JSON Benchmark [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/miloyip/nativejson-benchmark.

61. RapidJSON [online]. 2020 [visited on 2020-07-07]. Available from: https://rapidjson.org/.

62. TaoJSON [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/taocpp/json.

63. STXXL [online]. 2020 [visited on 2020-07-07]. Available from: https://stxxl.org/.

64. InfluxDB [online]. 2020 [visited on 2020-07-07]. Available from: https://www.influxdata.com/products/influxdb-overview/.

65. NSSM [online]. 2020 [visited on 2020-07-07]. Available from: https://nssm.cc/.

66. Logstash and Wolf comparison benchmarks [online]. 2020 [visited on 2020-07-07]. Available from: https://github.com/lamanchy/logstash_wolf_comparison.

67. Netcat [online]. 2020 [visited on 2020-07-07]. Available from: https://nc110.sourceforge.io/.

68. Azure: Virtual machines [online]. 2020 [visited on 2020-07-07]. Available from: https://www.azure.cn/en-us/pricing/details/virtual-machines/.

A Example of used Logstash configuration

Parser's configuration was selected to showcase a few examples of obscurity and inefficiencies of Logstash's configuration syntax. But it is worth noting that parser's configuration requires quite complex processing, Logstash's configuration is much more usable for simpler use-cases.

The configuration

```
... 6 lines skipped
```

Heartbeat input is used to generate custom events, which periodically trigger flushing of a specific parts of the processing pipeline and are deleted afterwards.

```
heartbeat {
  interval => 20
  type => "heartbeat"
}
... 7 lines skipped
```

The construct `if !("" in [logId])` stands for "if the JSON event does not have a `logId` field" and it does not work if the `logId` field contains an integer, for that case there is a different construct.

Also notice that each next grok plugin is more and more nested, which is the only way to try matching multiple regexes and mark the processed event with a specific field on success.

```
if !("" in [logId]) {
  grok {
    add_field => { logId => "spoolerJobDataSaved" }
    match => { message => "^Saving data file to job store
        finished successfully, jobId=\[%{UUID:JobGuid}]" }
  }
  if !("" in [logId]) {
    grok {
      add_field => { logId => "spoolerJobAnalysisEnd" }
```

```
      match => { message => "^Job analysis complete, jobId=\[%{
          UUID:JobGuid}]" }
    }
    ... 398 lines skipped
  }
}
... 306 lines skipped
```

Code below is used to sort incoming events by their timestamp, the sorting and flushing is triggered by the heartbeat event. But more importantly, the ruby snippet was written by trial and error, because the official documentation is highly insufficient and most tips and tricks were found on various internet forums.

```
ruby {
  init => "@events = Array.new"
  code => '
    event.set("current_time", Time.now.to_f)
    if event.get("type") != "heartbeat"
      @events.push(event.clone)
    else
      @events.sort! { |eventA, eventB| eventB.get("@timestamp").
          to_f <=> eventA.get("@timestamp").to_f }

      current_time = Time.now.to_f

      while not @events.empty? and current_time - @events.last.
          get("current_time") > 60
        new_event_block.call(@events.pop)
      end
    end
    event.cancel
  '
}
... 61 lines skipped
```

Final piece of configuration showcases joining metric events into one big chunk suitable for one HTTP transfer, metrics are marked for deletion, aggregated, then deleted, and newly generated events created by aggregate plugin continue in the processing pipeline.

```
mutate {
  replace => {
    "[@metadata][delete]" => "yes"
  }
}
aggregate {
  code => '
    map["message"] ||= ""
    map["group"] ||= event.get("group")
    map["message"] += event.get("message") + "\n"
  '
  timeout => 10
  task_id => "%{[type]}-%{[group]}"
  timeout_code => '
    event.set("[@metadata][output]", "influx")
    event.set("[type]", "metrics")
    event.set("[@metadata][delete]", "no")
  '
  push_map_as_event_on_timeout => true
}
if [@metadata][delete] == "yes" {
  drop {
    id => "drop_filter"
  }
}
... 14 lines skipped
```

These examples show that for complex processing, Logstash's configuration is often unintuitive, hard to read or maintain, and therefore hardly usable for such use-case.

B Detailed description of persistent queue

The persistent buffer is an extension of memory only buffer, which was more closely described already directly in the thesis. To quickly remind, the main design principle was having two queues, one for input and one for output, when output (back) queue was empty and input (front) queue contained some events, the queues were switched.

The same principle was used when designing the persistent buffer. When front queue is full and persistent buffering is enabled, a atomic variable swappable[1] is set to false and the memory only queue is extended by persistent buffer. When the persistent buffer is fully emptied, the queue switches itself back to memory only mode, not to compromise performance.

The schema describing persistent buffer[figure] shows two extra processing queues and persistent buffer. The processing queues (with their buffer) are used to serialize/deserialize JSON events, so they can be written onto a disk. These operations are done independently, so the queue can receive events into the front queue, serialize events, deserialize them and output them, all at the same time.

The output thread, when the persistent buffer is empty, tries to lock both front processing queue lock and front queue lock, when successful, the buffer is switched back to memory only mode, when unsuccessful, it unlocks persistent buffer and waits for incoming data.

1. Swappable is set to true whether the front and the back queue can be direclty swapped, in other words, whether there are no data in the persistent or processing buffers.

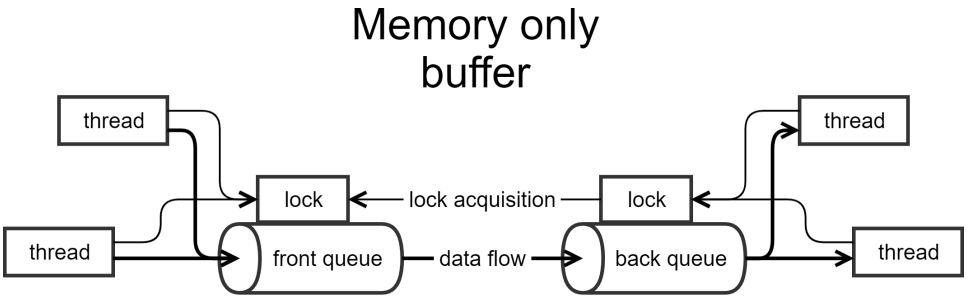

Figure B.1: Schema of memory only buffer

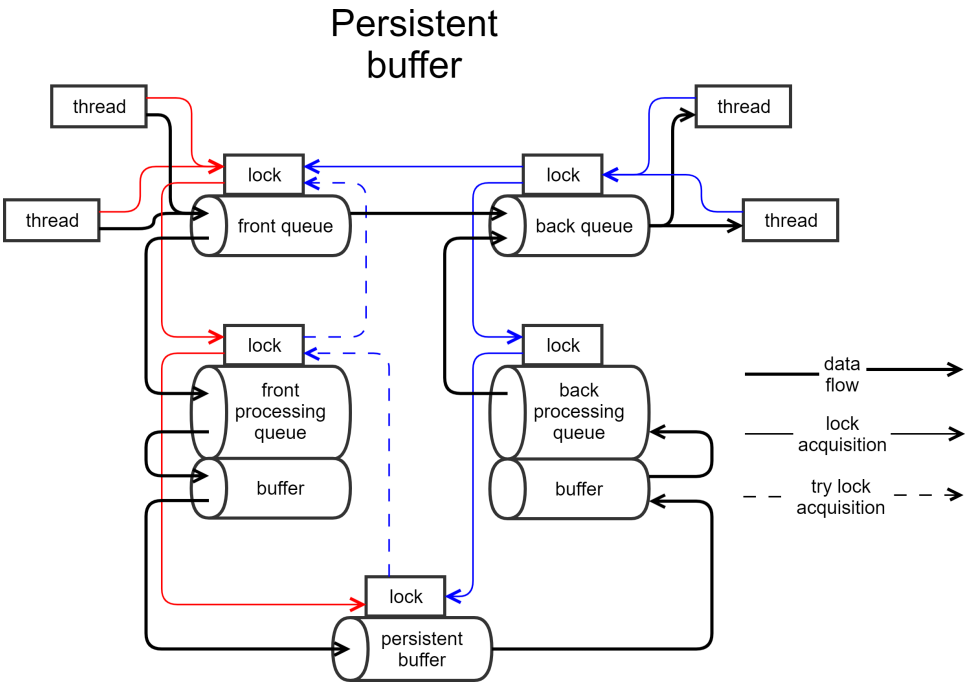

Figure B.2: Schema of persistent buffer

Pseudo codes for both input and output thread follow.

Input thread push pseudo code

```
1  Lock the front queue lock
2  Put the event into front queue
3  If the front queue is not full
       or the pipeline cannot be persistent:
4    Unlock the front queue lock
5    Return
6  Set swappable to false                    queue is now in persistent mode.
7  Lock the front processing queue lock
8  Swap the front queue and the front processing queue
9  Unlock the front queue lock               so the front queue can be used,
                                             while this batch of data is serialized
10 Serialize and compress events from the front processing queue
11 Lock the persistent queue lock
12 Move serialized data into persistent queue
13 Unlock the persistent queue lock and the front processing queue
       lock
```

Output thread try_pop pseudo code

```
1  Lock back queue lock
2  If back queue is not empty:
3    Get event for processing from back queue
4    Unlock back queue lock
5    Process event and return
6  If queue is swappable:          first check for premise, that it could be swappable,
                                   since reading atomic variable is faster than locking
7    Lock front queue lock
8    If queue is swappable:                  this time it the result is a fact
9      If front queue is not empty:
10       Swap front queue and back queue
11     Unlock front queue lock and back queue lock
```

```
12      Return                                  events were added into back queue,
                                                next time the try_pop will have events to process
13    Unlock front queue lock                   swappable was changed
                                                before lock was acquired
14  Lock the back processing queue lock         queue is not swappable
15  Swap back queue
        and back processing queue               if there were events in back processing queue,
                                                they were moved into back queue for processing
16  Unlock back queue lock                      so the events can be processed,
                                                while another batch will be deserialized
17  Lock the persistent queue lock
18  If persistent queue is empty:               premise, that swappable
                                                could be set back to true
19    Unlock the back processing queue
20    Try to lock both front processing queue lock and front queue
          lock
21    If successful:
22      Set swappable to true
23    Unlock locked locks and return
24  Load data from persistent buffer
25  Unlock persistent buffer lock
26  Decompress and deserialize data into back processing queue
27  Unlock back processing queue              events are prepared in back processing queue,
                                              next time the back queue will be empty,
                                              the events will be moved into the back queue
                                              by another execution of this method
```

C Attachments

`wolf.zip`	Wolf library
`logstash_wolf_comparison.zip`	Comparison benchmarks of Logstash and Wolf
`benchmark_wolf.zip`	Wolf configurations for comparison benchmarks