Evaluation of push/pull based load balancing in a distributed logging environment

Utvärdering av lastbalanseringsmetoder i en distribuerad loggmiljö

ROBIN DUDA
GUSTAF NILSTADIUS
Evaluation of push/pull based load balancing in a distributed logging environment

Utvärdering av lastbalanseringsmetoder i en distribuerad loggmiljö

ROBIN DUDA
GUSTAF NILSTADIUS
Abstract

This report compares the characteristics of push/pull load balancing techniques used in the context of a logging system. The logging system is expected to handle a large volume of events. The load balancing techniques are evaluated with focus on throughput during high load. The testing scenarios includes the use of a traditional load balancer (push-based) and the use of messaging queues (pull-based and indirectly context aware) in its place. The ultimate goal of the report is to determine the feasibility of using a messaging queue rather than a traditional load balancer in a distributed logging system. Tests were conducted measuring the throughput of multiple setups with different load balancers. The conclusion of this report is that both messaging queues and load balancing are equally feasible in a logging context.

Keywords

Logging system, Message queue, Load balancer
Sammanfattning


Nyckelord

Logghantering, Meddelandequeue, Lastbalanserare
### Terms and abbreviations

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>IP</td>
<td>Internet protocol</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>RR</td>
<td>Round Robin</td>
</tr>
<tr>
<td>WRR</td>
<td>Weighted Round Robin</td>
</tr>
<tr>
<td>LC</td>
<td>Least Connection</td>
</tr>
<tr>
<td>LL</td>
<td>Least Loaded</td>
</tr>
</tbody>
</table>
# Table of contents

1. Introduction ............................................................................................................. 1
   1.1 Background ............................................................................................................. 1
   1.2 Purpose ................................................................................................................... 1
   1.3 Goals ....................................................................................................................... 1
   1.4 Delimitations ......................................................................................................... 4

2. Theory and background .......................................................................................... 5
   2.1 Load balancing ...................................................................................................... 5
   2.2 Logging .................................................................................................................. 9

3. Process .................................................................................................................... 11
   3.1 Method .................................................................................................................. 11
   3.2 Implementation ................................................................................................... 11
   3.3 Evaluation ............................................................................................................ 13

4. Results .................................................................................................................... 19
   4.1 Baseline ................................................................................................................ 19
   4.2 Load balancing with FileBeat ............................................................................. 22
   4.3 Load balancing with Pen ..................................................................................... 24
   4.4 Message queue .................................................................................................... 24
   4.5 Comparison data ................................................................................................. 27

5. Analysis and Discussion ........................................................................................ 33
   5.1 Analysis ................................................................................................................ 33
   5.2 Discussion ........................................................................................................... 34

6. Conclusions ........................................................................................................... 37

References ..................................................................................................................... 39

Appendix 1 ..................................................................................................................... 41
Appendix 2 .................................................................................................................... 43
Appendix 3 .................................................................................................................... 45
Appendix 4 .................................................................................................................... 47
1. Introduction

1.1 Background

The hosting company Basefarm provides hosting and maintenance for mission-critical applications provided by their customers. Their customers have high demands when it comes to availability and security. These applications and the hosts they run on generates logging data. It is deemed desirable that their customers have access to this logging data. The logging data provided to the customers requires a gateway that enables secure access, enforcing permissions and enables the graphical representation of the data. The system providing these features has to be scalable as Basefarm hosts a large number of applications and growth is expected. This system lays the foundation for this report which investigates how the load balancing from the logging sources to the logic (processing) layer should be handled. The definitive goal is to maximize the system’s potential for utilization, scalability and reliability.

1.2 Purpose

The purpose of the report is to evaluate existing methods of load balancing and to identify areas where improvements could increase the performance and overall stability of a system. The analysis is performed on a system developed for Basefarm, a distributed system for collecting, aggregating and visualizing logging data. In the system, logging data is gathered and forwarded by log shippers (logging agents) on the application hosts. The applications receiving logs from the shippers must be load balanced as there are many shippers generating large amounts of logging data. The implementation of load balancing between the shippers and the log processors is the target of the analysis. The report is conducted as load balancers is not considered to provide enough reliability and performance compared to message queues. The root cause of this issue is that load balancing often ignores the state of the receiver (context such as load) while also having difficulties with spiking loads (no max workload threshold on clients or buffering). The load balancing in the specified system is required to handle failures and workload spikes gracefully as logging data should never be lost (security).

1.3 Goals

This project was divided into three parts, the pilot study, implementation and an analytic part. The goals of each part of the project is presented in this chapter.

1.3.1 Pilot study

The goal of the pilot study was to analyze the conditions specified in the specification from the taskmaster to detect existing products that may be available. As a means of finding what has already been done and which may be utilized during the development of the application. Its goal is also to provide further understanding of the system developed as the specification was further analyzed. It served to iron out any uncertainties and provide a virtual prototype. The virtual prototype was examined and through its result the feasibility of the project was decided.
**Recommended combinations**

A study of technology stacks used for logging purposes, including the collection, storage and visualization of logging data. The analysis focused on the ease of integration between components in the stack and also matched their properties against the specification.

**Components**

Individual components in the technology stacks examined to research whether any of the components in the studied stacks could be replaced with one better matching the specification.

**Storing log data**

Researched methods for storing logging data requiring high performance and replication of data. The data stored must be effectively indexed as large amounts of data is stored and queried.

**Presenting log data**

The analysis focused on ways to graphically represent logging data as required by the specification. The product with the set of features best matching the specification was chosen.

**Logging agent**

The analysis included the matching of log shippers to the specification, determining the best matching log shipper for the purpose. The requirements was partly based on the supported output formats, more supported formats increases the number of load balancers for analysis.

**Load balancing**

Means of load balancing to be analyzed in this report determined. The criterions for the load balancing methods provided the grounds for selecting competitive and interesting load balancing methods for the analysis.

**Maintainable software**

As the prerequisite of the report is the application developed for Basefarm, a study on how maintainable software is developed was conducted. The aim of the study was to improve the maintenance of the application.

1.3.2 Implementation

The goal of the implementation phase was to create a distributed, scalable application to handle logging data that meets Basefarm´s specification. The purpose of the
application is to serve as a foundation for the analytic part where the system is examined and analyzed in order to determine the most feasible method for load balancing in such a system. The implementation phase relies on the pilot study as it defines and concludes the technologies that should be taken advantage of.

The goals for the implementation,

1. **Distributed loghost**
   a. Implementation of load balancing algorithm.
   b. Storing persistent log data.
   c. Handling search queries.
   d. Ensuring high availability.
   e. Hardware requirements measured.

2. **Log agent**
   a. Reading logs (Syslog, File, Windows Event Log)
   b. Identity
      1. Group
      2. Subgroup
      3. Host identification
   c. Sending logs (Syslog, Message queue, Internet protocol)
   3. User interface for loghost
   a. Search queries
      1. Handling search queries
      2. Presenting search results
   b. Diagrams for representing log data
   c. Customizable user defined dashboards.
   d. Live log data feeds.

3. **Users**
   a. Authentication
      1. Username and password authentication with LDAP
      2. Two-factor authentication.
   b. User types
   c. User groups
   d. Users can be assigned log sources/applications
   e. Permissions
   f. Log data is read only

1.3.3 Analysis

The analytic part of the report answers the topic of the report through an analysis of the test results gathered. The analysis includes the following key issues,

- System performance utilizing the different load balancing mechanics defined in the pilot study.
- Hardware requirements analysis defining the hardware required based on number of log events.
The analysis on the system's performance includes the following key aspects,

- The overall performance measured in throughput.
- The reliability of the system under high load.
  - How high load affects nodes
  - The chance of messages being lost
  - Time required to scale out/up

These aspects are measured using the following methods of distributing workloads,

- An out-of-the-box load balancer.
- A messaging queue.

1.4 Delimitations

This report does not cover network based load balancing nor is the final security analysis included. Network based load balancing is deemed out of scope as it depends on the network architecture deployed. The security analysis is not included as it is performed by Basefarm. Load balancing algorithms used with the load balancer is limited to Round-Robin.
2. Theory and background

This chapter covers the background for the study and previous studies conducted in the same area. This chapter is necessary to get an understanding of the purpose of this project.

2.1 Load balancing

This chapter covers the background of load balancing and existing studies in the same area. This chapter is divided into, background that explains the technical background of load balancing, algorithms that explain some of the different algorithms, introduction to pull-based load balancing, comparison of push-based algorithms and redundancy as a by-product of load balancing.

2.1.1 Background

The number of Internet connected devices constantly increases and methods for distributing the generated requests to multiple servers is therefore in high demand. When the generated load for a single server gets too large, the server becomes unresponsive. To handle the ever increasing load the application must scale out, more servers must be deployed and the load distributed among them. A load balancer is a tool used to distribute traffic in order to handle more requests. The traffic can be balanced in different scales from small local area networks all the way up to global wide area networks covering multiple continents. The most basic type of load balancing is to create duplicate instances of a server where each of the instances can be addressed. The client has the responsibility to choose which of the server instances to use from a list. This method has obvious drawbacks as it requires the user to have the knowledge necessary to choose which server to address. A more versatile solution is load balancing with a centralized distribution architecture. The use of such architecture allows for greater transparency, as stated by [1], “Successful load-balancing approaches must be transparent to users, making a distributed system appear as a single host to the outside world “. Transparency implies that the users of the service has no knowledge that load balancing occurs.

Load balancing architectures can be divided into four categories [2]. These categories are client based, DNS-based, server based and dispatcher based. Neither of the four categories requires a specific algorithm to be used, the categorization is only based on architectural differences.

Client based

Like the previously mentioned server mirroring model, the client based load balancing allows the client to select the server instance to send the request to. The clients can be more or less coordinated to ensure load balancing. A benefit of the client based load balancing is that there is no server overhead. The drawbacks of such systems are their limited applicability. [1,3]

DNS-based

DNS-based load balancing utilizes DNS servers for load balancing. When a client performs a DNS lookup the DNS server returns an IP address according to the load balancing algorithm. The benefits of a DNS based approach is that it is already capable
of handling large loads and does not requires additional servers to be deployed. The drawbacks are the lack of control due to the DNS protocol and the lack of fine-grained balancing compared to dispatcher based or server based load balancing. [1,3]

Dispatcher based

Dispatcher based load balancing acts as a proxy between the client and server. The dispatcher receives a packet from the client and forwards the packet according to the load balancing algorithm to the appropriate server as shown in figure 2a. The server replies directly to the client. The drawbacks of a dispatcher based load balancer is that the dispatcher becomes the bottleneck and overhead when headers are added to network packets. The benefits of a dispatcher based load balancer is that it allows for fine tuning and full control of the load balancing since all the request passes through the dispatcher. [1,3]

![Dispatcher based load balancing scheme](image)

Server based

The server based load balancing approach allows the servers to redirect/forward clients. As illustrated in figure 2b, the client first needs to lookup the IP address of the server from a DNS-server. In the first step the conventional DNS server can be substituted with a DNS-based load balancer can be implemented to further aid in load balancing. After the client has acquired the IP address of the server a request can be sent. If the server is considered busy according to the load balancing algorithm, the server will redirect or forward the connection to another server. The pros of a forwarding method is that it is completely transparent to the client and has lower latency times compared to the redirect method. The redirection method does not add as much server overhead as the forwarding approach. The forwarding approach introduces less overhead in the system as a whole. The server based load balancing allows for fine-grained control over the balancing. Due to the distributed architecture it is able to scale well without a central bottleneck. [1,3]
2.1.2 Algorithms

The algorithm is responsible for selecting which server receives the request and may be applicable to all the categories of load balancer types mentioned in 2.2.1. Load balancing algorithms are divided into two groups. These are static- and dynamic algorithms. The static algorithms do not take into account dynamic parameters, for example that different types of requests may vary in processing time. The static algorithms distribute number of jobs evenly with support for weight factors. [1, 4]

Common static algorithms include:

**Random**

The Random algorithm utilizes a random number generator to select the distribution of tasks to the servers. When a request is processed, a random generator is used to generate a number from 1 to N where N is the number of servers. The request is later redirected to the server represented by the generated number. The efficiency of the random algorithm is dependent on the uniformity of the random number generation. [4]

**Round Robin**

The Round Robin (RR) algorithm distributes the tasks among the servers in a merry go around-like fashion. In an environment with X number of servers, request 1 is handled by server 1, request 2 is handled by server 2. This goes around as request N is allocated to server N Mod X. This approach ensures that every server has received the same amount of jobs every time the number of requests is a multiple of the number of servers. [2]
Weighted Round Robin

Weighted Round Robin (WRR) is an enhanced version of RR. WRR assigns a weight to every server, the weights are an integer representing the processing power or other aspects of the server. When WRR is running it creates a scheduling much like RR. However, in WRR one server can occur multiple times on the same scheduling round depending on its weight. [2, 4]

Dynamic algorithms in contrast to static algorithms takes into account the load of a server, the load can be multiple aspects such as number of active connections or CPU load. In a perfect world the algorithm would have real-time access to these aspects, in reality the algorithm can only make assumptions with the information at hand. Following is three commonly used dynamic algorithms:

Least-Connection

Least connection (LC) bases its decision on the number of connections that a server has open. When a server is assigned a job, the load balancer increments the number of active connections for that server. When a connection ends or when a connection timeout occurs the number of active connections is decremented. LC provides support for weight like WRR. The weight represents the server’s capability to cope with jobs and is taken into account during load balancing. If two servers has the same number of connections and the same weight, the server with the lowest ID will be chosen. [5, 6]

Least Loaded Server

Least Loaded Server (LL) has similarities with LC algorithm. Instead of taking account of the number of active connections, the least loaded algorithm takes the server workload into account. This algorithm is hard to put in practice as it is highly dependent on the measurements of the load and requires real-time propagation. [2]

XMitByte

XMitByte takes into account the amount of network traffic to and from a server. If a server has a lower amount of network traffic, the server is considered to have a lower workload and is assigned more jobs. [2]

2.1.3 Push based load balancing

Traditional load balancing is based on the push method. Push-based means that a server has an obligation to complete jobs it is assigned, the job is pushed onto the server. [7]

2.1.4 Pull based load balancing

When using pull based load balancing, the server decides when it can accept more jobs, this is achieved with a queuing system. When a server considers itself to have the resources to take on a new job it notifies the load balancer and gets placed in queue to get new jobs. When a new job is received by the load balancer, a server that is in the message queue will receive the job. Pull-based algorithms can minimize the amount
of unwanted network traffic, since the clients only request information they want and are able to process. [2, 7]

2.1.5 Pull-based versus push-based load balancing

A study by Arild Berggren [2] concluded that the overhead in a pull-based system is significantly higher and that a RR algorithm will largely perform better than the pull-based algorithm. It was however also concluded that under some circumstances the pull-based load balancing did perform better compared to RR. Future suggested work from Arild Berggren is to implement the pull based algorithm in a lower level language to minimize the overhead.

2.1.6 Comparison of push-based algorithms

A study conducted by Amjad Mahmood and Irfan Rashid [4] compared the performance of four dispatcher-, push-based load balancing algorithms. The algorithms compared was Round Robin, Random, Least Load and Least Connections. The results of the testing indicated that the random algorithm performed poorest even though it did perform good in some tests. The other static algorithm in the test, RR also performed very poorly compared to the dynamic load balancing algorithms. The dynamic algorithm LL performed best in all the tests. The LL algorithm ensured that the load was balanced among the servers. Based on this it is concluded that dynamic algorithms are the best alternative.

2.1.8 Redundancy

A benefit of adding dynamic load balancing is the added redundancy. If a system goes down a dynamic load balancer will no longer assign jobs to that server [2]. If all servers have a 98 per cent uptime, this will give the servers a probability of failure of \( p = 0.02 \). If there are three servers available, the expected probability of downtime equals \( p = 0.000008 \).

2.2 Logging

Applications generates metrics which may be formatted to model a logging event. A logging event describes an event that has occurred within a system initiated by the system itself or by an external system. With the constant increase in distributed systems and the requirements for security, the importance of logging has increased.

2.2.1 Purpose

The reasons behind logging application or system data are many. Logs could provide the information needed to pursue an instigator of a data theft, while also providing the means of preventing a future incident.

Logging data could provide the information needed to improve an existing system, by identifying performance bottlenecks and exposing irregularities. In the distributed systems of today performance bottlenecks are spread over multiple hosts and harder to manage.

The log collector could perform data analysis on the logs, to generate alerts and to detect deviations from usual patterns. An example of a deviation is a user logging on
the company network through VPN every weekday. The analyzer (post-tuning) has learned the pattern and recognizes it as safe, suddenly the user connects from another country outside of office hours. A proactive approach may be used to block the user access, or a reactive to generate an alert for the system admin. The importance of logging data cannot be overstated. [8,9]

2.2.2 Formats

There are many existing logging templates to choose from. The most important field in any logging event is the timestamp. Without the timestamp it would be impossible to determine whether the data breach occurred yesterday or tomorrow. There is also a great juridical weight on the timestamp and its accuracy.


The default format of the Apache web server includes a timestamp, event identification which is set to “core:error” in this sample, process id, client IP address and a message as a string containing the reason and cause for the generation of the log event.
3. Process

This chapter defines the methods used in the implementation and analysis of the implementation. Following the methods defined, the result of the report is reproducible.

3.1 Method

The methods chosen for the report are the implementation and experimental models. The reasoning behind this was that literary studies could not provide the environmental parameters sought after and the fact that a new system must be composed to fulfil the specification from Basefarm. The experimental approach allowed for testing the performance and reliability first-hand on the actual implementation used.

3.2 Implementation

This chapter presents the system implemented and used for evaluation and analysis. The test data was gathered from this system. The scope of the analysis is therefore narrowed to the context of the implementation and cannot be considered for other systems. If other systems are to be examined, then the methods defined must be followed in order to be comparable with the results presented in this report.

![Figure 3a, Log system architecture. The testing focuses on the load balancer/message queue component. Using a simulated host to generate logging data the tested component will forward the events to the log processors, where the output will be analyzed.](image)

The system in figure 3a is a result of the pilot study which analyzed existing solutions for logging systems. The components concluded the best components for the system’s purpose and the specification from Basefarm were chosen.

3.2.1 Host

The hosts in the system is the source of all logging data. The logs may be generated from an application running on the hosts or the host system itself. The formats for these logs include Windows Event Logs, Syslog and file based logging. A log shipper (logging agent) runs on these hosts to collect the emitted log data and to forward it to the load balancing solution. The log shipper used on the hosts was Filebeat from Elastic. [11]
3.2.2 Load Balancer

The load balancer handles the receipt of logging data from the log shippers. As the log processors might perform extensive processing on each event they must be load balanced. The load balancer in the system is the target for the analysis. The load balancer “pen” from the “Ubuntu 16.04 LTS” and the included load balancer in FileBeat was used, pen claims to have an algorithm superior to the round robin algorithm. [12]

3.2.1 Message Queue

Load balancing using Apache Kafka messaging queue is an alternate solution for load balancing. Messaging queues are recommended by Elastic as they provide reliability in the form of replayability when a receiver node goes down. A message queue also provides the ability to buffer events, as the processing layer might be saturated by processing or by waiting for the storage layer (indexing). The buffering reduces the risk of losing events and increases system stability. The analysis includes cases where the message queue increases the system's reliability over a load balancer as well as its performance drawbacks. [13, 14]

3.2.2 Log processor

The log processor is responsible for receiving events from the load balancer or messaging queue. The processor applies logic to the events received and forwards them to the persistence layer, where they are indexed. The utilization of the processor is interesting to this report as it is a means of measuring the utilization of nodes using the chosen load balancing technique. The log processor in use is Logstash from Elastic. [15]

3.2.3 Log storage

The logging storage is responsible for persistent storage of logging data and the ability to query them efficiently. These requirements demand that the solution has the ability to store and index huge amounts of data. In order for the analysis to be successful it is vital that this layer does not impair the throughput of the processors. To achieve a high indexing rate, the persistence solution uses sharding to distribute data in a cluster of storage nodes, which also provides transparent load balancing. The solution in use is ElasticSearch from Elastic. [16]

3.2.4 Dashboard

The dashboard has no effect on the outcome of the analysis and is only a component included to render the system usable by a person. This requirement is set by the taskmaster Basefarm and is not a direct goal of the report and as such it will not be further analyzed. The dashboard in use is Kibana from Elastic. [17]

3.2.5 Limitations

As the report focused on a limited set of components the results gained in the analysis is not empirical for any such combination of components. This concludes that no general analysis is performed on the subject of load balancers versus messaging queues.
in a logging environment. Rather the results of the report are mostly based on the individual implementations. Factors such as the actual implementation differences in different messaging queues and their configuration denies the generality of the results. However, if the implementation specifics and configuration diversities can be overlooked, the results may be increasingly seen as deterministic in the case of load balancers versus messaging queues. In order to improve the results for the general case and reduce the impact of implementation differences, the analysis focuses on scenarios where the characteristics of each load distribution technique most affects the results. This includes scenarios testing the reliability under high load and the performance in low latency networks.

3.3 Evaluation

Defines how the system is evaluated, describing evaluation scenarios and their parameters and the expected results.

3.3.1 Model

The evaluation model focuses on the reliability and performance as the system is put under heavy load. Heavy load in the system is expected and the amount of logging events passing through the system may vary during the day. It is therefore important to analyze how the load distribution techniques handles these specific cases. [2]

The load distribution techniques evaluated are the traditional load balancer and a messaging queue. The performance and reliability of the system will be measured in the following ways,

- The chance of messages being lost as message rate increases.
- The capacity of the load balancer measured in throughput, events/s.
- The workload balance between load processors.

The model favors reliability and stability during high loads, the speed at which the events arrive at the final destination is not considered as timestamps are used.

3.3.2 Expectations

The existing report mentioned in the model analyzes the two methods of distributing load, a load balancer and a message queue. The report concludes that the message queue introduces additional overhead compared to the load balancer. This results in the load balancer providing more performance measured in events per second. This report is not expected to disprove this, rather to focus on measuring under more specific conditions. These conditions include heavy load, where a message queue is likely to buffer messages and distribute the load more evenly. Not forcing events on the client nodes is anticipated to reduce the risk of losing logging data. The existing study also proved that the pull-based properties of a messaging queue works well with inhomogeneous systems, as the client nodes experience even utilization. This analysis does not consider heterogeneous versus homogenous systems as it is deemed already covered by an existing study [2].

3.4 Test environment

Describes the system setup used during the tests to allow the results to be reproduced and verified.
3.4.1 System

Software and their versions used in the measurements,

- “ElasticSearch 5.0.0-alpha1” [18]
- “LogStash 5.0.0-alpha1” [19]
- “Apache Kafka 0.9.0.0” [20]
- “Pen_0.18.0-l_i386” [12]
- “FileBeat 5.0.0-alpha1” [21]
- “Java 8 update 91” [22]
- “Ubuntu server 16.04 LTS” [23]
- “Ubuntu 14.04 LTS” [24]
- “Oracle VirtualBox 4.3.36” [25]

Any network setup where the clients and masters are on the same LAN and the network provides enough bandwidth to not bottleneck the test scenarios. The conclusions and the results of the tests are also dependent on the fact that logging systems often are LAN-based. Therefore, the impact of latency on the efficiency of messaging queues is minified.

Hardware of the host system, the test was conducted on a Lenovo Thinkpad t530 with the following specifications,

- Intel i7-3630QM CPU
- 8GB
- Intel® SSD 520 Series 180GB

3.4.2 Setup

The tests were conducted in a virtual environment on the host machine. There were six virtual machines on the host machine. The first machine “ubuntu-1” had the following specifications,

- “FileBeat 5.0.0-alpha1” [21]
- “Apache Kafka 0.9.0.0” [20]
- “Java 8 update 91” [22]
- “Ubuntu server 16.04 LTS” [23]
- 1536MB RAM
- 2x CPU cores, 100% Execution cap.

Ubuntu-1 was running FileBeat and a log generator developed in Java, see appendix 1. This virtual machine is represented by the hosts in figure 3a and had the role of log source. In the tests where the built-in load balancer in FileBeat and Apache Kafka was tested this machine also acts as the load balancer entity in figure 3a.

Ubuntu-2 had the specifications of ubuntu-1 with the addition of Pen load balancer. This virtual machine was in the tests with pen acting as the load balancer.
Ubuntu-3, ubuntu-4, ubuntu-5 and ubuntu-6 had the following specifications,

- “ElasticSearch 5.0.0-alpha1” [18]
- “LogStash 5.0.0-alpha1” [19]
- “Java 8 update 91” [22]
- “Ubuntu server 16.04 LTS” [23]
- 1024MB RAM
- 1x CPU, 32% Execution cap.

These virtual machines had the role of log processors, as illustrated in figure 3a. The execution cap was deemed necessary to ensure the capacity of the log processor to be the weakest link, hence making sure the system could be tested at maximum load.

The FileBeat configuration is available in appendix 2.

The Logstash configuration is available in appendix 3.

3.4.3 Tests conducted

The testing was done in six stages, in every stage tests were conducted for the following scenarios,

- 50 events/s and total 10,000 events
- 100 events/s and total of 10,000 events
- 500 events/s and total of 50,000 events
- 1000 events/s and a total of 20,000 events
- 1000 events/s and total of 100,000 events

The above events/s are the target speed, the actual speed is determined by the precision of the Java libraries used and the power of the host system. Between the test runs the system was configured to stay idle for ten minutes to make sure that no events were left for processing in the buffers.

No balancing, baseline

The first stage of testing conducted a test without a load balancing or message queue. For this test ubuntu-1 and ubuntu-3 were used. Ubuntu-1 as the source of logs and ubuntu-3 as the log processor with Logstash. The purpose of this test was to establish the overhead induced with either load distribution method and to calculate the theoretical throughput if no overhead was present.

Load balancing with FileBeat, two instances of log processors

The second stage of testing used the load balancing algorithm present in FileBeat 5.0.0. In this test two Logstash instances were used and the load was balanced between the two processors. Ubuntu-1 was the log source and ran FileBeat with load balancing. Ubuntu-3 and ubuntu-4 were receivers running Logstash.

Load balancing with Pen, two instances of log processors

The third stage tested the Pen load balancer using the ubuntu-1 machine as the source of the logs, ubuntu-3 and ubuntu-4 in the same manner as the second stage. Ubuntu-2 was used as a dedicated load balancer. This machine ran an instance of Pen. FileBeat
shipped the log events to Pen which forwarded events to one of the two instances of Logstash on ubuntu-3 or ubuntu-4.

Message queue with Apache Kafka, two instances of log processors

The fourth stage tested the system using the Apache Kafka messaging queue and two instances of Logstash. In this stage, like in stage two, ubuntu-1, ubuntu-3 and ubuntu-4 were used. FileBeat on ubuntu-1 shipped the log events to the message queue. After shipping the events to the message queue, FileBeat were no longer in control of the messages and the instances of Logstash had the responsibility of collecting the messages from the message queue.

Load balancing with FileBeat, four instances of log processors

The fifth stage was a continuation of the second stage. Instead of two instances of log processors on ubuntu-3 and ubuntu-4, this stage required all four of the log processor machines and load balancing between them using the load balancing algorithm present in FileBeat.

Message queue with Apache Kafka, four instances of log processors

The sixth stage was a continuation of the fourth stage. Like in stage five, the test was now conducted with four instances of Logstash. The Apache Kafka messaging queue was used to distribute the load between them.

3.4.4 Measuring

Passive measuring methods may be used as the log data stored is marked with timestamps. This is recognized as the most accurate method as it does not interfere with the performance of the system.

The following properties are measured,

- Amount of packets generated at the source.
- Time of generation for every event.
- Amount of packets stored at the destination.
- Time of processing at destination for every event.
- Portion of packets being lost.
- The utilization of processing power on client nodes.

The results of the testing provide the baseline required to estimate the hardware requirements.

3.4.5 Logging events

All logging tests were performed using the same logging template. The log template is a bit more verbose than the sample in chapter 2.2 from Apache Tomcat. It is deemed to be an accurate depiction of an average logging event, utilizing the JSON format and a relatively small size at 395 characters.
The logs were generated using the program listed in Appendix 1.

The primary field used in the logging message is the “@timestamp” field. This field is generated by Logstash when it receives a logging event. After the tests are complete the timestamps of the received events was parsed using a small Java program shown in Appendix 4.

The purpose of the parsing is to determine how many packet are received each second. The program creates “buckets” in a binary tree that increments for each event parsed that was received in that second. When all events are parsed, the tree is traversed inorder to get the counts of all events. The first and last times of the message set are recorded and used to determine the duration of the test. An array is then initialized with the length set to the timespan of the test, the keys of the counts are iterated and their values inserted into the array. Upon completion the cardinal position in the array represents the number of seconds from the first received event and its value the number of events received in that second.

The program writes the result of the parse to a comma separated value file (csv), from which the graphs and tables were created.
4. Results

This chapter covers the results from the tests in chapter 3. The results are achieved by calculating the number of events that was processed during a timeframe of one second at the receiving Logstash instances. This is possible as each event passing through the system is marked with a timestamp at the Logstash instances. The measurements contain the number of events processed each second as a marker for the methods load distribution capabilities. The graphs display the number of events processed each second at the Logstash instances as the average event throughput over time. The total number of events transmitted during the tests is included to detect if the average throughput declines when handling a sustainable stream of events over longer periods. The number of events also allows the results to be comparable to other tests using the total running time.

4.1 Baseline

For the results to be comparable a baseline must be established. A test scenario was therefore performed to measure the system’s performance without any load balancing. The baseline system utilizes a single Logstash host and a single instance for generating the logs.

![Event Rate: Single](image)

*Figure 4a, processing 10,000 events generated at 50 events per second.*

Figure 4a illustrates the average throughput over time when 50 events are generated per second and a total of 10,000 events. Some flickering can be noted up until half the runtime. The test system has no issues dealing with such a low rate of events.
Figure 4b, processing 10,000 events generated at 100 events per second.

Figure 4b shows the result from the test scenario where 10,000 events were generated at the speed of 100 events per second. The throughput is stable throughout the test.

Figure 4c, processing 50,000 events generated at 500 events per second and the moving average over 15 seconds.

In the test illustrated in figure 4c, the number of events is increased to 50,000 and are generated at 500 events per second, the red dots indicate the moving average over 15 seconds. Initially the throughput is even at 500 events/s, roughly 15 seconds into the test the throughput becomes uneven. The reason for the uneven throughput is that Logstash has detected a high input rate and resorts to buffering events in buckets of 2048 messages. When the buffer is full the messages are written as a batch operation to its output. This behavior is implemented in Logstash as a means to improve performance when event rates are high. Publishing processed events to ElasticSearch for example would require 2048 TCP connections to be established and each connection would transfer a single event using a HTTP request when bulking is disabled. With bulking enabled Logstash establishes a single connection and streams the 2048 events over the same connection in a single HTTP request. This behavior saves CPU time and allows more events to be processed. This behavior is visible in later figures using the load balancer. [26]
Figure 4d, processing 20,000 events generated at 1000 events per second. The graph shows the output in events/s and the moving average over 10 seconds.

Figure 4d illustrates the result from a smaller amount of events, 20,000 at the increased rate of 1000 events per second. The purple line shows the moving average over 10 seconds. The average throughput is 198 events/s.

Figure 4e, processing 100,000 events generated at 1000 events per second. The red line shows the moving average over 15 seconds.

In figure 4e the same behavior is noticed as in figure 4c, the average throughput is 198 events/s. The test has a total running time of slightly above 500 seconds when 1000 events per second are generated and a total number of events set to 100,000. The red line shows the moving average over 15 seconds, the average throughput in this test is 190 events per second.
4.2 Load balancing with FileBeat

These chapter contains test results from tests done using FileBeat with its built-in load balancer to the Logstash hosts. In these tests two Logstash hosts are deployed and connected to FileBeat.

![Graph](image)

*Figure 4f, processing 10,000 events generated at 50 events per second.*

The results from the first test using the built-in load balancer is shown in figure 4f. The event emitter was set to generate 50 events per second for a total of 10,000 events. The test results show no sign of instability or buffering as expected at this low rate of events. The cause of the drop in performance at 150 seconds into the tests are unknown.

![Graph](image)

*Figure 4g, processing 10,000 events generated at 100 events per second.*

The second test shown in figure 4g shows no change in throughput when the generator was set to double the rate of messages to 100, the total number of events generated are set to 10,000.
Figure 4h, processing 50,000 events generated at 500 events per second. The moving average is shown as the blue line measured over a period of 10 seconds.

The third test increases the amount of generated packets up to 500 per seconds, additionally the total number of events was set to 50,000 in order to extend the running time. The throughput of the system is initially stable at 500 events per second, later in the tests the throughput becomes uneven. The reason for this is that Logstash has detected a high input rate and resorts to buffering as described under figure 4c. Events are placed in the buffer with a size of 2048 messages before writing them to the configured output. This avoids the overhead which is introduced when Logstash otherwise must establish a connection for each message, this of course is only applicable to stateful outputs. The average throughput in the test matches the input and no bottleneck is present. The average throughput is close to 500 events per second as indicated by the moving average over 10 seconds, in the figure shown as the blue line. [26]

Figure 4i, processing 20,000 events generated at 1000 events per second, the graph shows the moving average with a period of 5 seconds.

The data in figure 4i is normalized into 5-second buffers to better represent the average throughput in the system. The average throughput in this test-run is 714 events per second when the generators are set to generate 1000 events per second and a total number of events of 20,000.
Figure 4j, processing 100,000 events generated at 1000 events per second. The blue line indicates the moving average with a period of 8 seconds.

The result in figure 4j includes the running time of the test which is 140 seconds. In that time 100,000 events are transmitted through the system at an average of 826 events/s. It is not obvious that there is a bottleneck in the system when looking at this graph as there seems to be idle time. Inspecting the average throughput reveals that the system cannot handle this many packets. The visible spikes in throughput is still caused by the buffering present in Logstash and is not equal to idle time. The spikes merely indicate that Logstash resorts to buffering to maximize its output. The blue line indicates the moving average over an 8 second period.

4.3 Load balancing with Pen

The Pen load balancer does not support load balancing over stateful connections and therefore its results has been discarded. The tests performed resulted in only one Logstash instance being utilized.

4.4 Message queue

This chapter contains the results of the tests conducted in stage four. Utilizing two log processors and the Apache Kafka message queue.
The first test of the message queue in figure 4k shows no signs of irregularities as expected. The variance in throughput is notably lower than that of the same test in the load balancer.

The second test shown in figure 4l using one hundred events per second at a total of 10,000 events looks very stable.

The third test at 500 event/s is also very stable at a throughput of nearly 500. The reason for this is that there is a spike in the beginning of the test and that the output of the logging generator is less accurate. The dip in throughput in the middle of figure 4m could be discarded. This test had a runtime of 120 seconds and if the system had issues handling the events it would be visible in more than a single point.
The generation rate was doubled from the last test up to 1000 events/s. Figure 4n shows a steady throughput of around 833 events/s with the increased input. It seems that the system cannot handle the packets any faster using this configuration.

Figure 4o shows another measurement at the same packet rate and an increased number of total packets as the last test in figure 4m. It seems that the system is able to handle the high load during the runtime of the test, which is 120 seconds. The average in throughput in this test is the same as in figure 4n, 833 events per second.
4.5 Comparison data

This chapter contains measurements for comparison between the different systems using running time of the tests and the average throughput.

In this chapter the term “theoretical” throughput will be used. The theoretical throughput is based on the baseline, e.g. the actual throughput for a single instance of a log processor measured in the testing environment. The baseline was measured at 100.000 events and a rate of 1000 events/s, its throughput was measured to 198 events/s. The theoretical throughput is calculated at O(n) (linear), i.e. the theoretical throughput for a scenario with four log processing instances is four times the baseline.

<table>
<thead>
<tr>
<th>Test number</th>
<th>Total events</th>
<th>Generated events/s</th>
<th>Log generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.000</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>100.000</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>100.000</td>
<td>1000</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 4p, parameters for the tests in figure 4q

Figure 4p shows the parameters used in the test data visible in figure 4q. Two tests used the standard high number of events of 100.000 and one at the lower 20.000 as a reference point.

<table>
<thead>
<tr>
<th>Component</th>
<th>Test 1 Throughput events/s</th>
<th>Test 2 Throughput events/s</th>
<th>Test 3 Throughput events/s</th>
<th>Average throughput events/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>396</td>
</tr>
<tr>
<td>Message Q</td>
<td>833</td>
<td>952</td>
<td>833</td>
<td>872</td>
</tr>
<tr>
<td>Balancer</td>
<td>714</td>
<td>869</td>
<td>689</td>
<td>757</td>
</tr>
<tr>
<td>Message Q*</td>
<td>833</td>
<td>909</td>
<td>826</td>
<td>856</td>
</tr>
<tr>
<td>Balancer*</td>
<td>592</td>
<td>444</td>
<td>572</td>
<td>536</td>
</tr>
</tbody>
</table>

Figure 4q, test results measured in average throughput using the parameters in figure 4q. (*) uses four receiving Logstash instances increased from two, to check if the maximum throughput could be increased further.

When the message queue and the load balancer was configured to use 4 Logstash instances instead of 2, a decrease in throughput was noticed. This is expected as it should take longer to distribute events to more hosts and the fact that the input rate is set to the same as before. The reason for not increasing the number of generated events when adding additional instances is that the system is already at max load. The table also reveals that test 3 has a significantly lower throughput, the number of log
generators in test 3 was set to 4 instead of 2 as in the previous test 1 and test 2. The message queue and load balancer tests (*) uses four receiving Logstash instances instead of the previously used two. This was done to detect if additional generators would affect the throughput.

![Figure 4r](image)

*Figure 4r, Time taken to process 10,000 events using two Logstash instances for the message queue and load balancer. Events are generated at 100 events per second at the source.*

At low rates of events the data in figure 4r shows that the message queue appears to achieve slightly slower throughputs when 10,000 events are sent at a rate of 100 events per second. This is not the case as a more accurate measurement of the actual event rate reveals that the message queue is being fed fewer messages. Measurements at higher event rates are more trustworthy as the small differences in generated number of packets have less impact on the results.

![Figure 4s](image)

*Figure 4s, Average throughput processing 20,000 events using two Logstash instances when generating input at 1000 events per second.*

In figure 4s a comparison of the different load balancing techniques is shown. A constant input of 1000 events/s for a total of 20,000 events is applied and the throughput of each system measured. The theoretical throughput is set to two times the baseline, as two Logstash hosts are used. Even with a theoretical overhead of zero the theoretical throughput is marginally lower than the other systems. Data from test 1 was selected as it shows results that are in line with results in test 3 according to the table in figure 4q. Test 1 also has simpler parameters than that of test 3.
Figure 4t, distribution of 10,000 events between two Logstash instances using the load balancer. Events are generated at 100 events per second. Figure 4t shows the comparison between two instances of Logstash placed on a timeline during the test where 10,000 events were generated at 100 events/s. The y-axis represents the proportion of events that is processed that second by each of the two Logstash instances. There is an even distribution of the events in the graph as expected when using load balancing at such low rates of events.

Figure 4u, distribution of 100,000 events between two Logstash instances using the load balancer. Events are generated at 1000 events per second. Figure 4u illustrates the comparison between two instances of Logstash placed on a timeline during the test where 100,000 events are generated at 1000 events/s. The y-axis represents the proportion of events that is processed that second by each of the two Logstash instances. There is an initial imbalance of the workload, after which it seems to distribute the load evenly. When observing this graph, it must be taken into account that messages are not processed as they are arrived, rather when the buffers fills. Therefore, an even distribution would show every other line to be of the opposing color, as is nearly the case in the graph.
Figure 4v, distribution of 10,000 events between two Logstash instances using the messaging queue. Events are generated at 100 events per second.

Event distribution using the message queue in figure 4v shows that events are very evenly distributed. This is an attractive attribute of the messaging queue and expected as each Logstash instance determines when they need more events. When the work is more evenly distributed, the utilization of the system increases. The drawback of this method is the overhead induced by the latency from the destination to the queue. In this test case the latency was lower than a millisecond, as the hosts runs on the same machine.

Figure 4w, distribution of 100,000 events between two Logstash instances using the messaging queue. Events are generated at 1000 events per second.

When the number of events is increased tenfold, the distribution seen in figure 4w is still very fair. No starvation or instability is present when the message queue is used. Fair workload distribution leads to a system with higher utilization.
The graph in figure 4x shows the distribution in the system when the number of Logstash instances are increased to 4. Fair distribution is observed. The gaps in the graph occurs as the Logstash systems does not have their clocks synchronized. The results were observed when generating 10,000 events at a rate of 100 events per second.

When the number of events emitted are increased, the load balancer is distributing load as if it were using Round-Robin. Each Logstash instance processes the messages when the buffer of 2048 messages is filled as described previously, the results of the graph is therefore hard to interpret. Some form of load balancing is definitely occurring.
Event distribution using the messaging queue is fair, even when the number of Logstash instances are increased to four. Figure 4z shows that events are continuously processed by Logstash and not processed only when the buffer is full. Whether this is a benefit or disadvantage is not obvious. As the results in figure 4q states that the messaging queue achieves a higher throughput it may be concluded an advantage. This may be highly dependent on the configured Logstash output, as some outputs might benefit largely from chunked writing. It is also possible that the Logstash instances could in theory use buffered writing for larger messages even when a message queue is deployed. Results are from a test where 10,000 events were generated at 100 events/s.

The messaging queue proves its fairness as the number of events is again increased tenfold. The figure 4z shows that all Logstash instances are continuously writing to their outputs. It is expected that if one of the Logstash instances had their CPU cap increased, it would be capable of processing more events than the other instances. When using a load balancer, this is only the case if the load balancer is configured using weights. The messaging queue appears to be very good at utilizing the Logstash instances and remaining stable through high throughputs.
5. Analysis and Discussion

In this chapter the results from chapter 4 is analyzed and compared to the expectations. In chapter 5.1 “Analysis” the result is analyzed and compared to the expectations. In chapter 5.2 “Discussion” the analysis is examined and discussed.

5.1 Analysis

It can be noticed that in the tests of the baseline that Logstash is not able to handle the increased load as the events are nearing the inbound speed of 500 events/s. This was expected as the Logstash log processor utilizes a backoff timer and refuses new connections if the load is too heavy and the buffering behavior is triggered. The buffering behavior seeks to maximize the throughput as it avoids the creation of a new connection and request for each message. Instead a connection is established when the connection is full and in that connection the whole buffer is transferred as a single request. This effect is visible in figure 4c and 4d. This saves both CPU time and increases network throughput.

When using two instances of Logstash the throughput increases as expected.

The flickering in throughput shown in figure 4c baseline tests, is not existent in the same test for the message queue in figure 4m when a message queue is used. The pull based message queue does not show the same signs of flickering and is stable with the exception of a spike halfway through testing. Further increasing the rate of inbound events to 1000 events/s and decreasing the total number of events to 20,000, the message queue is still stable and shows an even distribution. The message queue does not choke the Logstash instances which leads to a more stable throughput throughout the tests. The average for the message queue is higher compared to the average for the load balancing algorithm in the same test. Both load balancing and message queue is significantly higher compared to the theoretical throughput for the same scenario, as illustrated in figure 4s.

Further increasing the total number of events to 100,000 the load balancing algorithm receives an increase in average throughput. The average throughput is increased to 779 events/s measured over two test runs. This is to be compared to the average throughput of two test runs for the same scenario with the message queue at 892 events/s. The averages are computed from figure 4q.

The pull based message queue manages to achieve more stability compared to the load balancing algorithm since the Logstash instances are never choked. An important factor when measuring the efficiency of the message queue is the latency. As the network latency between the virtual Logstash machines were very low as they were running on the same machine, the major drawback of the message queue and pull based load balancing in general was minimized. The effect of the message queue was that it prevented Logstash from going into buffering mode, if this is a desirable trait is unclear. The results suggest that it has a positive effect as the average throughput of the message queue is higher.

The Pen load balancer does not support load balancing over stateful connections and therefore its results has been discarded. The tests performed resulted in only one Logstash instance being utilized.

Analyzing the result from the balancing of events between two instances of log processors in the case of FileBeat, it can be determined that FileBeat uses a Round
Robin-like load balancing algorithm. First choking one log processor before continuing to choke the next log processor with a short transit period in a RR like manner, as visible in figure 4t and 4u.

Comparing this to the results from the message queue illustrated in figure 4v and 4w where the message queue succeeds in an almost perfect balancing of events over the entire timeline.

Increasing the number of Logstash instances to four, the FileBeat load balancing algorithm shows the same characteristics as indicated while testing with two Logstash instances. FileBeat is shown to put the load on one log processor after the other.

As expected the message queue still exhibits a very even load distribution when the number of Logstash instances are increased to 4. This is true for test cases at both high and low loads as illustrated in figure 4z and figure 4aa.

Examining the results from testing with four log processor instances and message queue, the results is noticeable similar to the test with two log processor instances. The load distribution is even, as expected with four equally powerful log processors, the load balance is done continuously and no single log processor is choked.

All the tests were executed in virtual environments on the same machines while CPU usage was capped to a third of a single core. The throughput of the test results could therefore be multiplied by three to calculate real-world performance, only taking into account CPU usage of the hardware on which the tests ran. The number of events the system could handle is expected to achieve an even higher output as both the generation and distribution is not being performed on the same machine. In reality the hardware on which the tests were executed on has 4 cores. Not taking into account any bottlenecks resident in the load distribution or persistence layer, the throughput rates could be multiplied once again with 2 (not 4 as each test were executed using 2 instances) to calculate real-world performance.

Calculations of the possible throughput does not consider any bottlenecks present other than the CPU of the Logstash instances. Using the same hardware as in the tests to receive logs, if fed with a message queue the system could theoretically achieve 5232 events/s or using the built-in load balancer 4542 events/s in throughput. The sample throughputs are calculated by multiplying the averages for two Logstash instances in figure 4q and multiplying by the CPU cap to simulate a fully utilized core, it is then multiplied once more to simulate a four-core processor.

This is barely sufficient to handle the 4000 events/s specified by Fredrik Svantes [27]. As such it is deemed that either solution is viable, however the message queue or another load balancer is recommended as the built-in load balancer in FileBeat does not provide decoupling from the load distribution method.

5.2 Discussion

As mentioned in chapter 3.3.2, a previous study by Arild Berggren at University of Oslo [2] stated that the context of the testing the pull based load balancing added significant overhead and did not perform better compared to Round Robin and random load balancing in the context stated in Arild Berggrens report.
The main differences in testing environments are the following,

- A web server context was used, which requires responses from the servers.
- The events transmitted are larger (1.2KB vs 395 characters)
- Response delays are simulated to mimic dynamic web pages.
- Response times are measured and not throughput.

Each of these factors contributes to the different results seen in this report. Time taken for a logging event to be processed for long term storage is not as important as a user waiting for a webpage to load. Logging events are also in general smaller than that of the average web page.

This report focused on the load balancing in the specific context of logging with the ELK stack using FileBeat as the log shipper and Logstash as the log processor. With this report it has been proved that in the context, a message queue is slightly advantageous to load balancing. Thereby contradicting the study by Arild Berggen [2] in the specific context stated in this report.

The cause of message queue performing better in this context is due to the choking of the Logstash instances when using push based load balancing and that very fast polling rates are achieved on the internal network. [28]

The tests executed were only based on throughput and no tests on availability were performed. A message queue is expected to deliver higher availability as it is time-decoupled while a load balancer is not. This means that if all Logstash nodes would go down, no messages would be lost and the processing would resume when they start up again.

The possibility to utilize both methods to maximize throughput is discouraged. It increases the configuration complexity and is also proven to not actually improve the performance of a system. [28]

No measured events were dropped or duplicated during testing. The report fails to conclude a result on this point as all tests were using reliable protocols.

Choosing the most effective method of distributing load, pull-based or push-based leads to reduced cost. The message queue provides a more even load distribution, which increases the utilization of client nodes. The message queue has limited throughput when latency increases, i.e. the poll rate is decreased. The polling rate in the message queue system is the main contributing factor of its decline in throughput in distributed systems. [28] Therefore in order to select the most effective method the expected poll rate must be taken into account. The most economical solution is also the most environmentally friendly, as both aspects target to increase efficiency by improving utilization.

Whether a push-based or pull-based means of load distribution is most ethical is hard to determine. The pull-based method seems more ethical from the receiving point of view, as choking may be prevented.
6. Conclusions

In the case of message queues versus load balancers there is no clear winner performance-wise. The results have shown that the message queue is able to achieve a slightly higher total throughput. It is important to factor in the latency of the network which affects the polling rate of each solution. In the testing environment the network latency was lower than 1ms which favors the messaging queue, as the message queue suffers in performance at higher polling rates.

In conclusion, the selected method for load balancing in the context of a logging system should not be based solely on its throughput. Other important factors are time-decoupling which only the messaging queue would provide and also a more even distribution of the workload. If a load balancer is selected it is important to use a dedicated server and not use host-based load balancing as done in the tests. Host-based load balancing does not provide space-decoupling as the clients themselves directly communicate with the end nodes, this leads to scalability issues as configuration complexity increases.

Future studies in the area should be conducted on the reliability between the two methods using a non-reliable protocol. Network-level load balancing was not included in this study and could be interesting to study in order to handle an extremely large amount of logging events, a study covering network load balancing may include load balancing with software defined networking.
References


https://www.elastic.co/products/logstash

[16] Elasticsearch | Search & Analyze data in real time, Elasticsearch 2016-04-28
https://www.elastic.co/products/elasticsearch

https://www.elastic.co/products/kibana

[18] Downloads | Elasticsearch, Elasticsearch 2016-04-28
https://www.elastic.co/downloads/elasticsearch

https://www.elastic.co/downloads/logstash

http://kafka.apache.org/downloads.html

https://www.elastic.co/downloads/beats/filebeat

[22] Java download for all operating systems, Oracle 2016-04-28
https://www.java.com/sv/download/manual.jsp

[23] Download Ubuntu Server, Canonical Ltd 2016-04-28
http://www.ubuntu.com/download/server

[24] Ubuntu 14.04.4 LTS (Trusty Tahr), Canonical Ltd 2016-04-28
http://releases.ubuntu.com/14.04/

https://www.virtualbox.org/wiki/Download_Old_Builds_4_3

[26] Logstash output | Filebeat Reference [1.2], Elasticsearch BV, 2016-05-24


[28] Pull versus Push Mechanism in Large Distributed Networks: Closed Form Results, Wouter Minnebo and Benny van Houdt, 2016-05-25
Appendix 1

LogGenerator.java

```java
import java.io.IOException;
import java.util.Timer;
import java.util.TimerTask;
import java.util.logging.*;

public class LogGenerator extends TimerTask{
    private static Logger logger;
    private static int counter = 0;
    private static int totalEvents = 0;
    private static Timer timer;
    private static long startTime;

    public static void main(String[] args) {
        System.out.println("This is log generator, generating random logs, max 1000 events/s");
        if (args.length >= 1 && args[0].contains("-h")){
            System.out.println("Usage, *filename* *events/s* *number of events (optional)*. \nUse -h to display this message");
        } else {
            new LogGenerator(args);
        }
    }

    public LogGenerator(String[] args){
        if (args.length == 3){
            totalEvents = new Integer(args[2]);
        }
        logger = Logger.getLogger("Log_generator");
        FileHandler fileHandler;
        try {
            fileHandler = new FileHandler(args[0]);
            SimpleFormatter formatter = new SimpleFormatter();
            fileHandler.setFormatter(formatter);
            logger.setUseParentHandlers(false);
            logger.addHandler(fileHandler);
        } catch (IOException e) {
            e.printStackTrace();
        }
        timer = new Timer();
        startTime = System.nanoTime();
        timer.schedule(this, 0, (1000)/new Integer(args[1]));
    }

    @Override
    public void run() {
        logger.info("This is log number " + ++counter);
        if (totalEvents == counter && totalEvents != 0){
            timer.cancel();
            System.out.println("Average events/s: " + (float) counter/((float) System.nanoTime() - (float) startTime)/1000000000));
            System.exit(0);
        }
    }
}
```
Appendix 2

Template filebeat configuration

# Filebeat Configuration Template#
...

### Filebeat Configuration

```yaml
filebeat:
  prospectors:
    #Omitted
  paths:
    - /home/kakor/log.log
  input_type: log

### Output

```
Appendix 3

Logstash pipeline

```ruby
input {
    beats {
        port => 5044
        ssl => false
    }
}

filter {
    grok {
        match => { "message" => "%{TIMESTAMP_ISO8601:creation} INFO LogGenerator run This is log number %{NUMBER:logcount}" }
    }
}

output {
    file {
        path => "/home/kakor/logstash-out.json"
    }
}
```
Appendix 4

JsonParseToCsv.java

```java
import io.vertx.core.json.JsonObject;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.TreeMap;

/**
 * Aggregates logging events based on second-intervals and returns count per second.
 */
class JsonParseToCsv {
    private static final String ISO_8601 = "yyyy-MM-dd'T'HH:mm:ss.SSSXXX";
    private TreeMap<Integer, Integer> count = new TreeMap<>();
    private int end = 0;
    private int begin = Integer.MAX_VALUE;

    JsonParseToCsv input(ArrayList<JsonObject> lines) throws ParseException {
        SimpleDateFormat format = new SimpleDateFormat(ISO_8601);
        for (JsonObject line : lines) {
            int time = (int) Math.floor(format.parse(line.getString("@timestamp"))
                    .getTime() / 1000);
            count.put(time, (count.get(time) == null) ? 1 : count.get(time) + 1);
            if (time > end) end = time;
            if (time < begin) begin = time;
        }
        return this;
    }

    void output(String path) throws IOException {
        Path file = Paths.get(path);
        String data = "";
        for (int count : parse()) {
            data += count + "n";
        }
        Files.write(file, data.getBytes());
    }

    int[] parse() {
        int[] counts = new int[end - begin + 1];
        for (Integer time : count.keySet()) {
            counts[time - begin] = count.get(time);
        }
        return counts;
    }
}
```