Log analysis and classification of CERN control systems

September 2014

Author:
Zahari Dimitrov Kassabov

Supervisor:
Manuel Gonzalez Berges

CERN openlab Summer Student Report 2014

Project Specification

Partner/Group	EN-ICE - Siemens/ETM
Title	Log Analysis of WinCC OA Projects
Description	CERN is running several large WinCC OA distributed systems, each of them counting ~100-150 computers and several thousands of processes. The log files currently produced by these processes are only looked at very seldom to debug problems. The project will consist of introducing a modern log analysis tool to analyze these log files in real time and inform the operators/experts of possible problems. The tools will also be used to extract statistics and produce reports about the control systems. The selected tool will be Elastic Search with Kibana.
Supervisor	Manuel Gonzalez Berges
Student	Zahari Dimitrov Kassabov

Acknowledgments

I would like to thank Manuel Gonzalez for offering good and honest advice, as well as encouraging me to work on my idea, Daniel Davis, and Fernando Varela for providing the necessary tools for my project and Paul Burkimsher for the very well structured course on WinCC OA. I would also thank to all the EN-ICE group for being kind enough to patiently answer my questions and occasionally walk with me to the server lab to see that the power was down again in the middle of the coffee break.

Abstract

This project has consisted in examining different alternatives for monitoring and analyzing the log messages produced by the control system software using at CERN. Its main accomplishments have been a performance measurement of the Logstash tool under both realistic and extreme load scenarios, the development of a flexible and powerful performance testing tool, and the proposal of a system that automatically determines the importance that a given log message might have for an operator based on a Bayesian filter.

Contents

1 Introduction

1.1 Overview

The complexity of the systems currently in place at CERN causes them to output too many log messages for a human operator to process. Therefore automatic tools are needed to extract useful information from the contents of the log messages. On the other hand, a log monitoring infrastructure cannot have a significant performance impact on the running control systems where the priority must always be that sufficient computing resources are allocated for it to run properly. For that, it is necessary to assess the performance of the log monitoring stack before it is implemented in production.

Chapter 2 is concerned with the performance analysis of the *ELK Stack* (see Section 1.3) and in particular, the most performance critical part of it, Logstash. It describes in detail the tools developed to accomplish that goal in the Section 2.2.

Once the messages are collected, an interesting improvement would be to try to determine in an automatic way which of them actually reveal critical information about the state of the system. Chapter 3 describes a proposal to implement such an automatic relevance determination system based on the same principle as many spam filters.

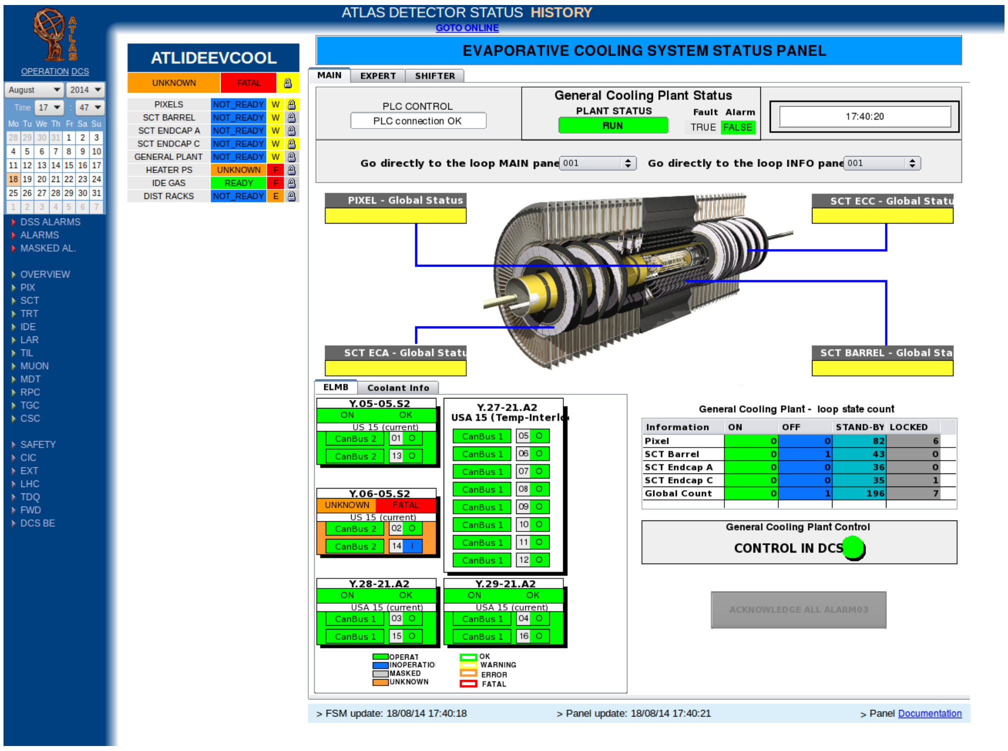
1.2 WinCC OA

Many control systems at CERN are based on the software WinCC OA (previously known as PVSS) developed by ETM PROFESSIONAL CONTROL GMBH. These systems include the four LHC experiments, as well as other experiments a and infrastructure like the electrical system.

WinCC OA has been successfully used at CERN and its use is increasing among different experiments. On top of the capabilities that this software provides, there is a notable effort at CERN to provide developer tools and guidelines to aid the implementation of control systems. These include the JCOP[1] and UNICOS[2] frameworks. An ongoing effort within the CERN Industrial Controls & Engineering Group is to provide tools that help the operators find abnormal situations with their systems by monitoring different aspects of them. One of them is the log message monitoring with which this project is concerned.

1.3 The ELK stack

The ELK stack is the proposed solution to parse, store and visualize the log messages. It is composed by a parser (*Logstash*), a key-value store (*Elasticsearch*) and a web interface



ATLIDEEVCOOL

	UNKNOWN	FATAL	
PIXELS	NOT_READY	W	
SCT BARREL	NOT_READY	W	
SCT ENDCAP A	NOT_READY	W	
SCT ENDCAP C	NOT_READY	W	
GENERAL PLANT	NOT_READY	W	
HEATER PS	UNKNOWN	F	
IDE GAS	READY	F	
DIST RACKS	NOT_READY	E	

PLC CONTROL: PLC connection OK

General Cooling Plant Status — PLANT STATUS: RUN; Fault: TRUE; Alarm: FALSE

17:40:20

Go directly to the loop MAIN pane 001 Go directly to the loop INFO pane 001

Y.05-05.S2: ON OK; US 15 (current); CanBus 2 01 O; CanBus 2 13 O

Y.06-05.S2: UNKNOWN FATAL; US 15 (current); CanBus 2 02 O; CanBus 2 14 I

Y.27-21.A2 USA 15 (Temp-Interlo: CanBus 1 05 O; CanBus 1 06 O; CanBus 1 07 O; CanBus 1 08 O; CanBus 1 09 O; CanBus 1 10 O; CanBus 1 11 O; CanBus 1 12 O

Y.28-21.A2: ON OK; USA 15 (current); CanBus 1 03 O; CanBus 1 15 O

Y.29-21.A2: ON OK; USA 15 (current); CanBus 1 04 O; CanBus 1 16 O

🟩 OPERAT, 🟦 INOPERATIO, ⬜ MASKED, 🟧 UNKNOWN; OK, WARNING, ERROR, FATAL

General Cooling Plant - loop state count

Information	ON	OFF	STAND-BY	LOCKED
Pixel	0	0	82	6
SCT Barrel	0	1	43	0
SCT Endcap A	0	0	36	0
SCT Endcap C	0	0	35	1
Global Count	0	1	196	7

General Cooling Plant Control: CONTROL IN DCS

> FSM update: 18/08/14 17:40:18 > Panel update: 18/08/14 17:40:21 > Panel Documentation

Figure 1.2.1: A modern WinCC OA web interface that controls the ATLAS cooling system

(*Kibana*) The process is as follows:

Logstash Reads the relevant log files, finds the new log messages and parses them. In this step it assigns values to different fields based on the contents of the message and other contextual information. It then sends the resulting key-value structure to *Elasticsearch*.

Elasticsearch Indexes the processed log message it receives and allows to retrieve them according to different criteria and also compute multiple statistics.

Kibana Allows the visualization of statistics related to the log messages.

The Figure 1.3.1 shows an example of the Kibana interface for the CERN electricity control systems (`psen1` and `psen2`). This gives an idea on how the log monitoring interface would look like upon successful implementation of the project.

The main focus of this project has been evaluating the performance of Logstash, which is the most critical part of the stack as it has to run on the same machine as the control system (other options are discussed in).

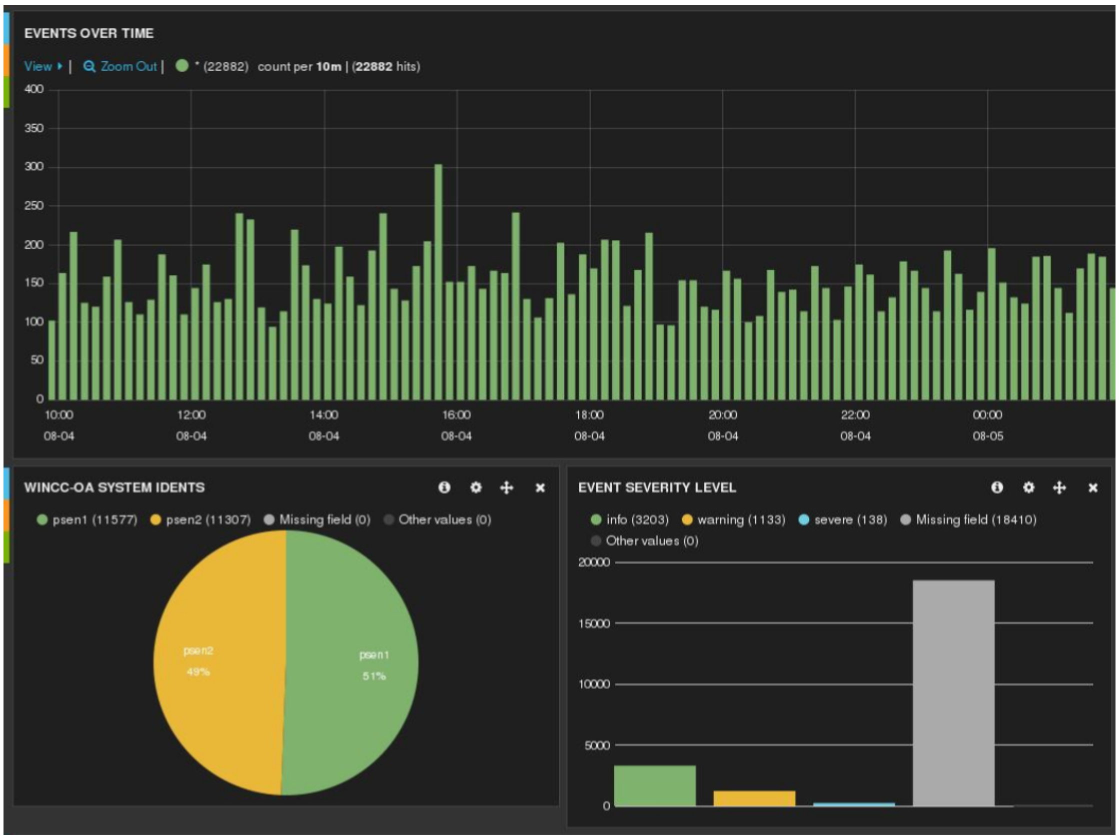


Figure 1.3.1: An example of a resulting Kibana panel showing statistics such as the number of events over time, the origin of the log messages by system and the total number of messages by severity level.

1.4 Code

The code of this project can be found in the following repository:

https://svn.cern.ch/reps/en-ice-svn/trunk/tools/JCOP/Projects/Framework/Software/Tools/fwCentralLogging/LogAnalysis/Performance-Analysis/

In the discussion of the technical parts of the project in the next chapters, it will be assumed that the reader has obtained a copy of this code and is able to follow the instructions to run it.

2 Measurement of the performance of Logstash

2.1 Setting up Logstash to monitor WinCC OA log messages

Logstash can be installed via rpm packages on the CERN Linux systems. It can be started as a system service with options predefined in configuration files as a normal program. The first option is more useful for deployment and the later is more useful for evaluating the program, which is the objective. In particular, this allows to specify the configuration file as a command line option. The necessary configuration file was already provided. The Logstash configuration file uses a custom JSON-like language in which the following has to be specified:

Inputs Specifies which log files will Logstash monitor. For a WinCC OA this includes the `log` folder of the project.

Filter Specifies how is the content of the log message parsed into a key-value data structure. This is done by specifying regular expressions to be matched against the messages. Other information such as the system name is also included in the data structure. The main difficulty is the parsing of multi-line log messages, since given a log line there is no indication that the next line is going to belong to the same message.

Output Indicates where is the resulting data structure going to be sent. For this project, this is an Elasticsearch instance running on a dedicated host.

The Logstash configuration file does not allow to access environment variables or any other way of defining variables such as the paths of the WinCC OA projects or the system names externally, so that they must be hard-coded in the file. Therefore a mechanism to generate a Logstash configuration file on deployment should be considered.

2.2 Performance analysis tools

In this project a number of tools to measure the performance of the Logstash process were developed in a way in which they should be easily adaptable and extensible to monitor other processes and to entirely different tasks. A first iteration of this toolset relied on the manual set up of the different components with the help of WinCC OA graphical user interfaces. This process was then automated by the introduction of a testing tool which allowed for a great reduction in the number of steps required to run the tests, making the procedure considerably less tedious and error prone.

2.2.1 The WinCC OA project

The repository of the code has a folder structure compatible with that of a WinCC OA project. It is capable of generating error and debug messages at the specified rate, as well as peaks of error messages generated without delay.

Some components are only required to run the less advanced GUI-based tests. These include all the panels, the DataPoints and DataPointElements saved in the `dplist` folder (since the automatic test tool does not use DataPoints) and the majority of the scripts. The *required* components are then:

- The `msg` folder containing the error message strings.
- The `cont_arguments.ctl`, `peak_arguments.ctl`, and `debug_arguments.ctl` files in the `scripts` folder which generate the log messages based on command line arguments.
- The `measure_eff_argparse.py` script that is used to measure the resource consumption of a given process.
- The `logstash_conf` folder containing configuration files for Logstash. These must be edited so that the paths defined in them match those of the project (see 2.1)

These files can be copied in an empty project.

2.2.2 The performance measurement script

The `measure_eff_argparse.py` script can be used to measure the performance (resident memory and CPU percentage) of a given process. It is completely general and can be used to measure the performance of any running process.

It is based on the Python `psutil` library (with a version greater than 2). It has a rich command line that allows for optional arguments (including `--help`).

The usage of the script is:

```
measure_eff_argparse.py [-h] [-d MEAS_DELAY] [-a] [-p KILL_THRESHOLD]
                               [-t KILL_TIME]
                               pid file_name
```

where the last two (pid and file_name) are the only required options. The complete description of the arguments is:

Positional arguments

pid PID of processes to monitor or match against `prgrep -of` (the oldest running process with the given match in its command line)

file_name Filename (without extension) of the `csv` generated when terminating the process.

Optional arguments:

-h, –help Show this help message and exit

-d MEAS_DELAY, –meas-delay MEAS_DELAY Delay between measurements

-a, –autokill Terminate measurement when the process uses less than KILL_THRESHOLD of CPU.

-p KILL_THRESHOLD, –kill-threshold KILL_THRESHOLD CPU percent level to autokill if the option is enabled.

-t KILL_TIME, –kill-time KILL_TIME Seconds below the threshold needed to activate autokill.

2.2.3 Running Logstash performance tests with testool.py

2.2.3.1 Introduction

The test tool aims to automatize the procedure and allow for an easy replication of the rests under different environments.

The test are configured via a `.ini` configuration file whose options should be self-explicative. Then the test is run with a simple command line:

```
./testtool.py my_config_file.ini
```

The results can be recovered with the command:

```
./fetch_results.py
```

This will download the results and plot them and save the plots.

Also, the tests can be stopped at any time with:

```
fab stopnow
```

2.2.3.2 Dependencies

All the dependencies both in client and server are Python libraries. The recommended way to manage them is though the miniconda installer program. The advantages of this approach is that it manages properly the binary dependencies, does not require to compile the packages, allows to use different sandboxed environments (including for external binaries), can be manages without root privileges. The disadvantage is that not all the packages are available with the `conda` installer, and must be installed with `pip`. For the test tool to work, the default Python binary must be replaced by the `miniconda` one which is done by default on installation by adding the `miniconda` binaries to the PATH.

Client dependencies

testool.py After investigating several alternatives, Fabric was found to be the only that allowed for easily sending `ssh` commands and understanding the output. The disadvantage is that it requires a *daemonizer* program on the server side (see 2.2.3.2).

- Fabric: Installable with

  ```
  pip install fabric
  ```

  If using `miniconda`, `pip` should be installed first with (`conda install pip`).

- `dateutil` (`conda install dateutil`)

fetch_results.py

- Fabric
- Numpy
- Pandas
- Matplotlib

The easiest way to get these is by installing the whole scientific Python stack:

```
conda install anaconda
```

Or alternatively, they can be installed one by one with the `conda` tool.

Server dependencies

Performance measurement (*measure_eff_argparse.py*)

- Psutil (version 2). Installable with

  ```
  conda install psutil
  ```

Testing tool

Fabric does not get along really well with detached programs, and some daemon software needs to be installed on the server. The chosen one is:

- zdaemon (`pip install zdaemon`)

Installing offline In a server without Internet connection, download the necessary packages (conda packages are in `miniconda/pkgs`) and their dependencies and install using

```
conda install --use-index-cache <package.tar.bz2>
```

and

```
pip install <package.tar.gz>
```

for pip packages.

The packages that need to be installed on the server side are found in the `testool/python_packages` folder. It can be uploaded to the server and then:

```
conda install --use-index-cache setuptools-3.6-py27_0.tar.bz2 --yes
conda install --use-index-cache pip-1.5.6-py27_0.tar.bz2 --yes
conda install --use-index-cache dateutil-2.1-py27_2.tar.bz2 --yes
pip install ZConfig-3.0.4.tar.gz
pip install zdaemon-4.0.0.tar.gz
```

Ready miniconda A ready to use miniconda installation is prepared for the server (can be found in `testool/`). Just extract it in a server directory and update the PATH variable, by for example adding this line to `.bashrc`

```
export PATH="/path/to/minoconda/bin:$PATH"
```

2.2.3.3 Set Up

Once the Python dependencies are met (see 2.2.3.2), the test tool assumes you have the GenerateErrors project running in your server (and is the only one). Also it would be convenient to have a host entry in the (local) `~/.ssh/config` file describing the host you want to access.

2.2.3.4 The configuration file

This is an example configuration file:

```
[general]
test_name = TestTest
#Host should be configured in .ssh/config
host = tmachine
end = 13/8/2014 16:00
generate_errors_path = /home/unicryo/PVSS_projects/GenerateErrors
#Must be a file in {Project folder}/logstash_conf
logstash_binary = /opt/logstash/bin/logstash
logstash_config_file = logstash-config-unicryo.cfg
#Take one measurement per minute
monitor_options = --meas-delay 60
```

```
#Continous manager
[cont.default]
#rate is messages per second
rate = 0.5

#Peak manager
[peak.short_bursts]
rate = 0.001
number = 30000

#Another peak manager
[peak.regular_bursts]
rate = 0.01
number = 300
```

test_name Should be an unique string for every test. It is used in the file names and the title of the plots.

host Is a string to access the server by the ssh protocol that could.

end_date When the test should end (as seen for the local machine clock).

generate_errors_path Is the (remote) path to the WinCCOA project with the additional files of the repository (specifically the monitor script and the Logstash configurations).

logstash_binary The path where logstash is.

logstash_config_file The configuration to use for logtash (must be a file in `{Project folder}/logstash_conf`)

monitor_options Options to be passed to the *measure_eff_argparse.py* script (see `measure_eff_argparse` `--help` for details)

[cont.*] Starts a continuous rate error generator (*cont_arguments.ctl*) with the rate (errors per second) given in the `rate` option.

[debug.*] Starts a continuous rate debug message generator (*debug_arguments.ctl*) with the rate (errors per second) given in the `rate` option.

[peak.*] Starts a peak error generator (*peak_arguments.ctl*) with the rate (peaks per second) given in the `rate` option and the number of erros per peak given in the `number` option.

Any number of continuous and peak managers can be started.

2.2.3.5 The code

The code should be understandable and easy enough to modify to eg monitor other process. A few key points are:

- The running tests are kept track of in the *.running.csv* file. Only one test per host can run at the same time.
- The program will try to remove all the WinCCOA logs before starting a new test.
- The `run_detached` function is responsible for starting the `zdaemon` process. It returns the standard output of the command. The pid of the detached command can be obtained with the `parse_pid` function. If the `kill_on_end` flag is set (yes by default), the process will be stopped on the `end_date`.
- The program will generate a *killfile* on the remote machine which will sleep until the end date and the stop the managers, logstash and the monitoring script. Another *killnowfile* can be executed with `fab stopnow` to stop the tests at any time (if executed directly on the remote server the *.running.csv* file has to be amended after recovering the data).
- The `monitor_decorator` decorator can couple to the output of a run_detached process and will start the monitoring script to that process. Currently it is only applied to Logstash.

2.3 Results of Logstash performance tests

2.3.1 Set Up

The latest tests were performed on a machine that is alike those used in production: `cs-ccr-pvss2`. There is a single system, *GenerateErrors* that generates log messages in a way that mimics a typical running system (specifically, the PSEN1 system). The produced error rates are:

- A continuous rate of 0.5 errors per second.
- Peaks of 300 errors every 100 seconds.
- Peaks of 30000 errors every 10000 seconds.

The processed log messages are sent to an Elasticsearch instance running in `pcitco192`.

A testing tool was developed to allow running tests in different configurations. This allows to easy repeat the test with different machines and configurations. See here for more details.

2.3.2 Results

Here are presented the results for the first 17 hours of a still running test with the configuration specified in 2.3.1 (figures 2.3.1 and 2.3.2). Similar tests have showed analogous results. The mean CPU usage is 4.27% (with peaks to up to 200% for a small amount of time that are averaged out in the graph). The memory increased until it was stabilized at 720 Mb, and until it jumped to 751 Mb. The garbage collector is expected to kick in before memory usage reaches 1 Gb and free some memory.

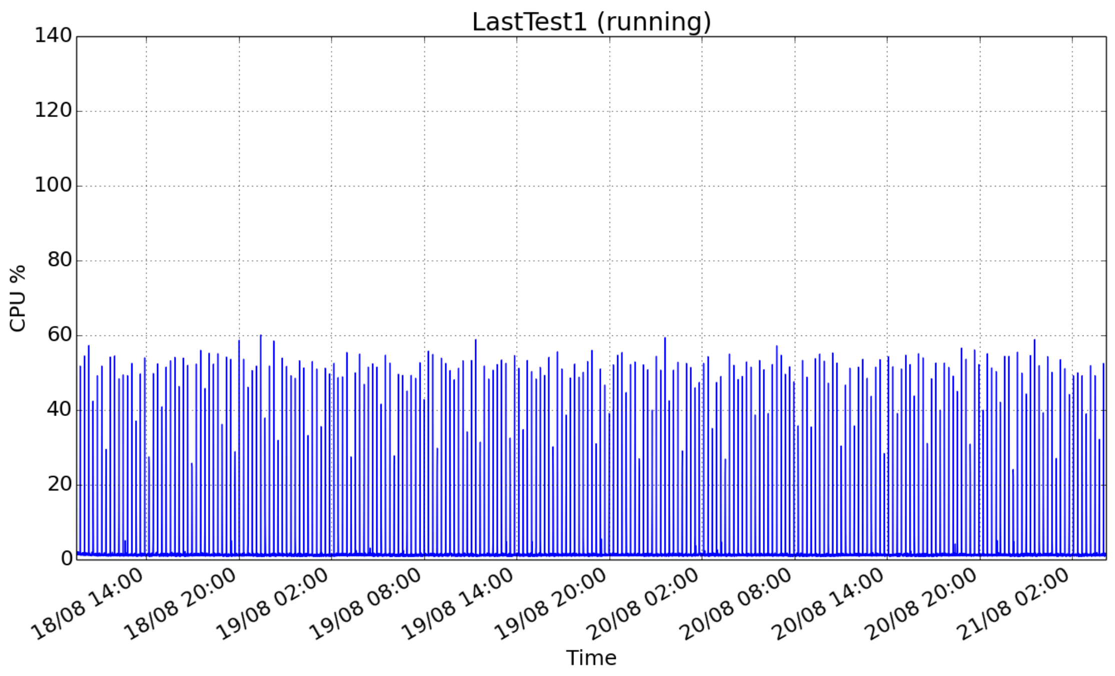


Figure 2.3.1: CPU usage

2.3.3 Different configurations

The test and discussions have concluded that the best way to run Logstash is having one instance per machine, and then sending the messages to Elasticsearch. A number of different configurations were mentioned:

Several Logstash processes It was tested if several Logstash processes on the same machine (running more two WinCCOA systems) would perform better. However, since the Logstash has high initial memory costs (of starting the JVM and JRuby), this is not the case. Also, the JVM processes do not share memory with each other. The CPU usage was also worse in this configuration.

NFS mount It was suggested to have several log folders mounted on a central machine trough NFS. However it was argued that this would generate too much network

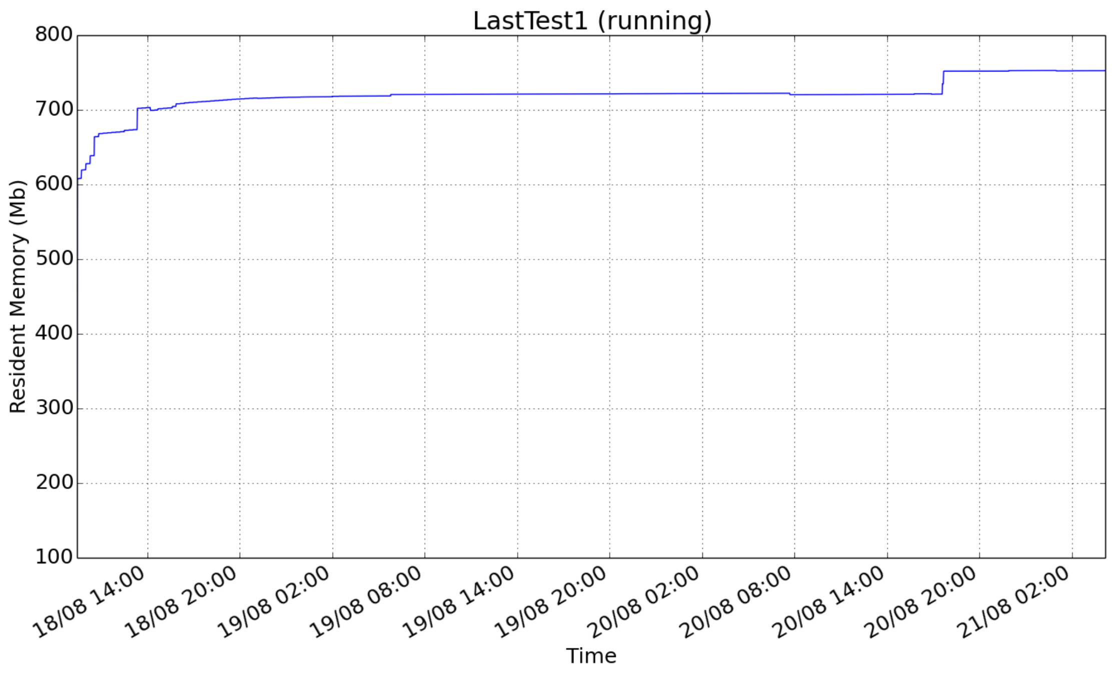


Figure 2.3.2: Memory usage

traffic for the possible performance gains in memory and CPU, and it was not tested.

Offloaded parsing A Logstash configuration was tried in which only the minimum message processing was done in the running system. This showed little CPU gains (order of 20% of the total usage) and did not influence the memory usage.

2.3.4 Other considerations with Logstash

2.3.4.1 Problems with WinCC OA log rotation

It was shown that Logstash loses log messages when the WinCC OA log file is renamed to with the `.bak` extension. It also loses the backup file when the original file is renamed a second time. This could be solved by making clever use of the postLogFileHandler[1] feature of WinCC OA.

Alternatively, making use of the standard Linux `logrotate` mechanism could constitute a more robust solution and also more compatible with Logstash. This however would not work for Windows systems.

[1]https://j2eeps.cern.ch/wikis/display/EN/howCanIPreserveMyPvssIILogfile

2.3.4.2 Flush bug

The multiline feature keeps the last few log messages to be sent to Elasticsearch. See this ticket.

2.3.5 Limiting resource usage

2.3.5.1 CPU

CPU usage can be effectively limited by setting the *niceness* of the Logstash process to a high value. This will allow it to run at full speed when the resources are free while not stealing CPU time from more critic processes.

2.3.5.2 Memory

The initial amount of memory that the Java process allocates can be controlled with the `-Xms` option. Not setting this option will make the JVM heuristics decide the initial amount of memory. Experiments show that regardless of the initial memory allocated, the Logstash process will expand until it consumes around 700 Mb if allowed.

The maximum amount of memory (after which the process crashes) can be set with the `-Xmx` option.

These options can be passed to Logstash by setting the JAVA_OPTS environment variable before starting it.

2.3.6 Conclusions

Logstash consumes more resources than what one would expect for its functionality. This is due to the many layers of abstraction it is built on (JRuby over a Java Virtual Machine). However these resources amount for a small percentage of those available on a production system running WinCC OA systems. Specifically, the Logstash process consumes on average around 0.7 % of the total memory and 0.08 % of the CPU. Also, these systems run typically at loads that are far from their maximum capacity.

Logstash is a tool in very active development, and it did seem easy to find and ask for information when needed. It is also somewhat immature and a number of bugs are to be expected (see 2.3.4) .

The requirements for developing from scratch a custom specialized tool that would work instead of Logstash were also assessed[2]. This would involve significant costs of development which would not pay off without the real need to optimize the process. Of the off-the-shelf alternatives to Logstash, Heka looks promising. It should give better performance, but it is newer and less tested than Logstash. It would also require to rewrite the parser in the Go language.

The easiest and most obvious configuration of Logstash (one running instance per system) should work well for a typical system.

[2]https://espace2013.cern.ch/en-dep-ice-scd-col/_layouts/15/WopiFrame.aspx?sourcedoc=/en-dep-ice-scd-col/Shared%20Documents/Projects/JCOP%20Framework/Components/Central%20Logging/Hypothetical%20Logstash%20substitute.docx&action=default

3 Automatic relevance determination of log messages

3.1 Introduction

In this chapter we propose a system able to determine automatically if a given log message is *important* or not. The requirements for such system are:

Automatic While the system should use the knowledge of human operators to tell it what constitutes an *important* message, it should be able to *learn* from that information and classify the new incoming messages by itself.

Real time Typically the existence of an abnormal situation in a control system requires an immediate action and therefore the classifier should be able to asses the importance of a log message immediately after it is produced.

Resource-cheap The classification process should not demand a big fraction of the computing resources of the systems it runs in.

This solution should in no way be considered tested or production ready, but rather a proof of concept worth to build upon. In Section 3.4 some fundamental limitations are described, along with proposals to circumvent them.

3.2 Naive Bayes classification

A *naive Bayes classifier* is a simple probabilistic that can satisfy the requirements outlined in Section 3.1. It has several additional features that make it well suited for the task:

- It is simple both to understand conceptually and to implement.
- It has proven to work reasonably well for similar problems (namely, many spam filters are based on this approach).
- It is efficient for large datasets and a number of feature, and can be sequentially updated (*online training*) without needing to access past data (except possibly to compute some features).

3.2.1 Messages to features

Given the message ℓ, calculate an array of *features:*

$$\ell \rightarrow (x_1, x_2, x_3, ..., x_n)$$

Each feature can be a different data type. Some examples are:

- Categorical data: $x_i \in \{\text{'INFO'}, \text{'WARNING'}, \text{'SEVERE'}, \text{'CRITICAL'}\}$
- Numerical data: $x_i = \#\text{of similar messages}$
- Text, x_i is the actual content of the log message.

Out of these, there are many examples and implementations to treat text and numerical data (see for example the scikit-learn Python library [3]), however no implementation was found to treat categorical data that was flexible enough and able to satisfy the requirements in Section 3.1. Therefore a suitable implementation was developed as will be discussed in Section 3.3. In the following we consider that the possible values of x_i correspond to some finite and possibly unknown a priori set.

It is possible to convert numerical to categorical data by binning it (assign different labels to intervals in the allowed range of the variable), or if that is not desired, probability estimates of different naive Bayes methods can be combined together by simple multiplication thanks to the *statistical independence* assumption.

3.2.2 Statistical independence

Statistical independence is the main assumption of the naive Bayes model, and the one that is most often completely wrong. It states that all the features are uncorrelated and the fact that the feature x_1 has the value α_1 does not affect the probability of any other feature, for example x_2 having a particular value α_2. In this case we can write:

$$P(x_1 = \alpha_2, x_2 = \alpha_2, ...) \underset{\text{independence}}{\longrightarrow} P(x_1 = \alpha_1)P(x_2 = \alpha_2)$$

With this assumption joint probabilities turn into multiplications of individual probabilities.

We estimate the probability of some feature having a particular value by simply dividing the number of occurrences of that value between the total number of occurrences of the feature:

$$P(x_1 = \alpha_2, x_2 = \alpha_2, ...) = \frac{\#(x_1 = \alpha_1)}{\#x_1}\frac{\#(x_2 = \alpha_2)}{\#x_2}$$

Note that we do not assume that the occurrences of x_1 are the same than those of x_2. This allows to consider missing features (for example debug log messages do not have the same fields as error log messages) and richer initialization options, as discussed in 3.3.2.1.

Now for each set of values $\alpha = (\alpha_1, \alpha_2, ...\alpha_n)$ feature array $x = (x_1, x_2, ..., x_n)$ we want to predict the target variable y:

$$y \in \{\text{'important','not important'}\}$$

We compute the *posterior probability* $P(y = important|x = \alpha)$ as a function of *prior, likelihood* and *evidence* which we evaluate as:

Prior $P(y)$ Counts for important over total counts.

Likelihood $P(x|y)$ Counts for x over importants

Evidence $P(x)$ Counts for x over total counts

The Bayes theorem states that the posterior probability is then:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} \tag{3.2.1}$$

and since we know how to estimate each of the variables, we can obtain a result for it.

3.3 Implementation

3.3.1 Discarded technologies

Some technologies were evaluated to determine if they could avoid the need to implement the categorical classifier from scratch, but it was determined that this would lead to a more complicated and less maintainable design:

Elasticsearch scripts Since the operations that are required are very simple, it was studied if they could be implemented using the Elasticsearch script API. However this was found to be very constrained and underdocumented at present. For example, the Python scripting only supports Jython 2.5 and scripts have to be run as `eval` statements. While it would be possible to implement the classifier despite these limitations, the difficulty and future maintenance costs do not seem to pay off.

Existing implementations The existing naive Bayes implementations (in particular scikits-learn) were found to not be convenient for categorical features and online training: The input is usually a fixed size numeric vector and categories are mapped into tuples of numbers. Adding dynamically new possible values to the categories requires the input vector to be resized, which is absolutely impractical.

3.3.2 Classifier

A categorical classifier was implemented in a way that is well suited for one-by-one online training and to be smart about new unseen inputs.

3.3.2.1 Classifier initialization

It is possible to set "*blind priors*" to enable the production of sensible outputs even before the operators have started to manually sort the results in order to train the classifier: The probabilities of $P(x = \alpha)$ are estimated as:

$$P(x = \alpha) = \frac{\#(x = \alpha) + w_{x=\alpha} * prior_{x=\alpha}}{\#x + w_{x=\alpha}}$$

Where $prior_{x=\alpha}$ an estimation of a probability that a vector containing this value is important and $w_{x=\alpha}$ is a measure of how confident we are in this estimation. The bigger it is the more messages must be amended for the classifier estimation to change. When this is substituted in the Bayes formula (3.2.1) it will produce the appropriate result.

This feature allows to write code like this to initialize the severity feature:

```
blind_priors = {'DEBUG':0.5, 'INFO':0.5,
                'WARNING':0.7, 'SEVERE':0.9, 'FATAL: 0.95'}
blind_weight = 100
severity_f = Feature('severity', blind_priors, blind_weight)
```

3.3.2.2 Valid input

The classifier is implemented in a way that allows for flexible valid input which can be sorted out properly with the right initialization options. In particular all of the following are valid input:

- Vector with missing features.
- Vector with new unseen features.
- Vector with new unseen value for a feature.

The implementation has the option to set default probabilities for unknown values, in a way similar to 3.3.2.1.

3.3.2.3 Code

The code (`bayesfilter/category_classifier.py`) contains documentation of the most important functions as well as some basic tests. The classifier objects can also be *pickled* (saved as python objects) which is a basic way of storing the classification data.

3.3.3 Computed features

A proof of concept implementation of the classifier has been developed (file `bayesfilter/es_classifier.py`). It uses a library of convenience functions to interact with the Elasticsearch database `bayesfilter/`estools.py. The features that are considered are:

- Host name.

- System name.
- Severity.
- Similar messages in the last five minutes.
- Identifier of the message (for errors).

3.3.4 Client-Server architecture

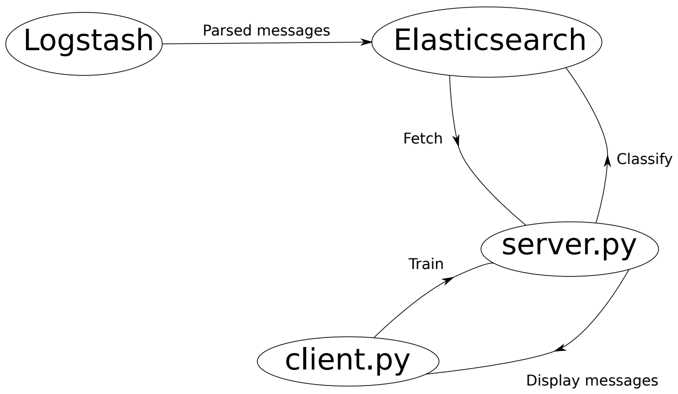


Figure 3.3.1: Representation of the relation between the components of the system.

The executable code of the sample is implemented as client and a server based on the `tornado`[1] library.

- Server:
  - Polls ElasticSearch to find unclassified messages.
  - Classifies the messages.
  - Accepts user input to train the classifier.
- Client:
  - Receives new messages from server.
  - Classifies the messages manually to train the classifier

Client and server communicate using the Websockets protocol. This is a bidirectional communication standard supported by the major browsers. It should allow to easily implement the client code along the Kibana interface.

[1]`http://www.tornadoweb.org/`

3.4 Future work

3.4.1 Intelligent weighting of messages

Using all the incoming messages to automatically train the classifier would lead to unstable behavior: A message repeating many times would dominate in all the probability estimates for its features.

For example, if a system is stuck in a loop that throws a message with severity 'SEVERE' and these are determined to be not important, then if these were automatically fed to the classifier, the probability of the 'SEVERE' trait corresponding to an important message would approach zero over time. This is a known problem for spam filters, and a strategy to address it is called tf-idf[4], which stands for *term frequency-inverse document frequency.* The idea is that the weight of features values that occur mire frequently is decreased compared to the less frequent ones.

Several decisions are needed in order to implement this, such as whether to apply the weight reduction to *similar* messages (one weight per message) or to same category values (different weight for each feature in the message, based on counts). In the first case, a precise definition of *similar messages* is also required.

3.4.2 Computing other features

More informative features can be imagined than those outlined in 3.3.3.

3.4.2.1 Origin of the messages

An important shortcoming of the WinCC OA logs is the impossibility to determine in which script or panel was the message originated. This could be resolved by standardizing the message strings in a way that they provide this information.

3.4.2.2 Incorporating text

Another addition could be using standard text classification techniques to incorporate the raw text of the messages into the classification algorithm.

3.4.2.3 Pattern recognition techniques

Another group is working on the problem of determining automatically which situations are important for an human operator using pattern recognition techniques. This has a key advantage over the approach proposed here in that it is able to exploit the correlations between different events, which are obviously very important in determining the importance of a given state.

The naive Bayes approach is blind to the correlations between the features it trains by construction. However this can be bypassed by computing features that are themselves function of the global state of the system rather than only of the contents of the log message. The feature “Similar messages in the last five minutes” that is computed in the sample implementation is an example of this idea.

Therefore one can imagine that patterns that are determined to correspond to important situations are mapped into features of the log messages, whose relevance is finally computed using the naive Bayes approach, which also takes into account the opinion (and the evolution thereof) of the human operators in charge of the systems.

Some care must be taken in the performance hit caused by computing these features if the real time classification capability is to be preserved.

3.4.3 Implementation thoughts

While the implementation of this idea is still a long way ahead, some ideas were briefly discussed during the development of this proof of concept sample:

- Placing the server between Logstash and Elasticsearch (see 3.3.1) would give the classification earlier and reduce database operations, since it would no longer be necessary to retrieve the messages and update them with the classification information.
- The performance (speed, memory, CPU) should be assessed for typical systems compared to Elasticsearch. While the classifier concept is simple in terms of implementation and computation, calculating some features might impact the performance.
- The architecture allows for multiple processes performing classification and communicating with clients simultaneously. In this case some database technology should be used to manage the classifier state and allow concurrent I/O operations.

Bibliography

[1] CERN EN-ICE Group : Industrial Controls & Engineering. JCOP framework. `https://j2eeps.cern.ch/wikis/display/EN/JCOP+Framework`.

[2] CERN EN-ICE Group : Industrial Controls & Engineering. UNICOS. `https://j2eeps.cern.ch/wikis/display/EN/UNICOS`.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[4] Stephen Robertson. Understanding inverse document frequency: on theoretical arguments for idf. *Journal of Documentation*, 60(5):503–520, 2004.