BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SUPPORT FOR DYNAMIC CONFIG RELOAD INSIDE RSYSLOG

PODPORA DYNAMICKÉ ZMĚNY KONFIGURACE V RÁMCI RSYSLOG

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR ATTILA LAKATOS
AUTOR PRÁCE

SUPERVISOR doc. Mgr. ADAM ROGALEWICZ, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Master's Thesis Specification

25154

Student:	**Lakatos Attila, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Information Systems and Databases
Title:	**Support for Dynamic Config Reload Inside Rsyslog**
Category:	Operating Systems

Assignment:

1. Study the Rsyslog project (the rocket-fast system for log processing). Get acquainted with its main components.
2. Propose a way of analyzing the difference between a configuration of a running Rsyslog instance and a configuration specified in a newly provided configuration files.
3. Propose a way of changing internal structures of a running Rsyslog instance without compromising the ability to process incoming data.
4. Design and implement an option in Rsyslog, which allows users to dynamically reload configuration for core components without a full restart and with special attention to memory utilization.
5. Evaluate the solution and propose possible improvements. Discuss the implementation complexity, usefulness and limitations of proposed improvements.

Recommended literature:

- Github repository of Rsyslog project: https://github.com/rsyslog/rsyslog
- Rsyslog homepage: https://www.rsyslog.com/

Requirements for the semestral defence:

- Items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rogalewicz Adam, doc. Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 18, 2022
Approval date:	November 3, 2021

Abstract

Logs are one of the most valuable assets when it comes to IT system management and monitoring. As they record every action that took place on a machine, logs provide the insight system administrators need to spot issues that might impact performance, compliance, and security. For this reason, the rsyslog software utility can be used as it offers the ability to accept inputs from a wide variety of sources, transform them, and output the results to diverse destinations by a set of rules. One shortcoming the software currently has is that it needs to be restarted in order to modify the rule set. The author of this master's thesis points out what types of problems a user might encounter during this period of time, such as messages entering the system are lost, TCP/UDP based connections are disturbed, even if no changes are made. The goal of this thesis is to design and implement an option, which allows users to dynamically reload configuration for core components without the need of a full restart. The improvements aim to address problems raised by the research, as well as increase performance by reusing already existing resources.

Abstrakt

Logy sú jedným z najcennejších aktív, pokiaľ ide o správu IT systému a monitoring. Keďže zaznamenávajú každú činnosť, ktorá sa uskutočnila na stroji, logy poskytujú prehľad správcovi systému aby vedel zistiť pôvod problémov, ktoré môžu ovplyvniť výkon, súlad a bezpečnosť. Z tohto dôvodu je možné softvérový nástroj rsyslog použiť, keďže ponúka možnosť prijímať vstupy zo širokej škály zdrojov, transformovať ich a odosielať výsledky rôznym destináciám na základe súboru pravidiel. Jedným z nedostatkov tohto softvéru v súčasnosti je to, že ho je potrebné reštartovať, aby akceptoval aktualizované zmeny v pravidlách. Autor tejto diplomovej práce poukazuje na to, s akými typmi problémov sa môže stretnúť užívateľ počas reštartu nástroja. Medzi najkritickejšie patria strata správ vstupujúcich do systému a narušenie TCP/UDP spojenia, aj keď neboli vykonané žiadne zmeny v pravidlách. Cieľom diplomovej práce je navrhnúť a implementovať riešenie, ktoré umožňuje používateľom dynamicky znovu načítať konfiguráciu základných komponentov bez potreby úplného reštartu. Navrhované zmeny sú zamerané aj na riešenie problémov, ktoré boli odhalené počas vývoja ako aj na zvýšenie výkonu opätovným použitím už existujúcich zdrojov.

Keywords

logging, rsyslog, TLS, tcp, udp, log processing, journal

Kľúčové slová

logovanie, rsyslog, TLS, tcp, udp, spracovanie logov, žurnál

Reference

LAKATOS, Attila. *Support for dynamic config reload inside rsyslog.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Adam Rogalewicz, Ph.D.

Rozšírený abstrakt

V dnešnej dobe je nevyhnutné aby každý systémový správca sledoval logy a mal tak povedomie o tom, čo sa deje na ním spravovaných zariadeniach a mohol tak na prípadné incidenty včas reagovať, v lepšom prípade im predísť a eliminovať bezpečnostné riziká. Preto potrebuje zabezpečiť zhromažďovanie, analýzu a uchovávanie logov. To umožňujú logovacie nástroje.

Príkladom takého nástroja, ktorý implementuje protokol Syslog, je projekt Rsyslog. Jedná sa o software s otvoreným zdrojovým kódom (tzv. open source software). Okrem iného, ponúka možnosť prijímať vstupy zo širokej škály zdrojov, transformovať ich a odosielať výsledky rôznym destináciám na základe súboru pravidiel. Je to populárny nástroj používaný ako predvolený logovací démon na operačných systémoch Fedora Linux, CentOS Linux a Red Hat Enterprise Linux. Tento produkt je vyvíjený v nezanedbatelnej miere v spoločnosti Red Hat.

Jedným z cieľov jeho dizajnu je podporovať obrovské množstvo správ za sekundu. Nevýhodou tohto softvéru v súčasnosti je to, že ho je potrebné reštartovať, aby akceptoval aktualizované zmeny v pravidlách. V praxi, všetky nami spravené zariadenia zbierajú logy lokálne a odosielajú ich na centrálny rsyslog server. Ak sa rozhodneme pridať nové pravidlá do konfigurácie, bude potrebné vypnúť rsyslog a znova ho zapnúť, aby sa zmeny udiali. V tomto okamžiku sú všetky spojenia narušené a správy vstupujúce do systému sa stratia.

Cieľom tejto práce je navrhnúť a implementovať riešenie, ktoré umožňuje používateľom dynamicky znovu načítať konfiguráciu základných komponentov bez potreby úplného reštartu. Táto práca je riešená v spolupráci spoločnosťou Red Hat a komunitou rsyslog.

Jedným z dôvodov vedúcich k tejto práce boli časté požiadavky užívateľov, ktorí nechcú prichádzať o logy počas reštartu nástroja. V dnešnej dobe väčšina logovacích systémov je schopná dynamicky znovunačítať konfiguráciu. Taktiež sa objavila potreba oživenie projektu a získanie viac prispievateľov. Je dôležité si uvedomiť, že ostatné nástroje implementujúci Syslog protokol sú drahé komerčné produkty.

Správna implementácia navrhovanej funkcie si vyžiadala potrebu úpravy veľkého množstva existujúceho kódu. Autor tejto práce prispel k open source projektu rsyslog a neustále zlepšoval jeho kód na základe spätných vazeb poskytovaných komunitou. Navrhnuté zlepšenia, ku ktorým došlo počas autorovej práce by mohli byť prínosom pre celý projekt rsyslog, ako aj pre spoločnosti balíčkujúci rsyslog, vrátane Red Hat, Inc.

Počas implementácie bola vykonaná rozsiahla zmena v logike rsyslogu. Niektoré hlavné komponenty boli definované ako samostatné prvky, pri čom spolu súviseli. Vloženie týchto komponentov do konfiguračného objektu umožnilo jednoduchšiu manipálicu hlavným konfiguračným súborom.

Okrem zamerania sa len na hlavnú tému, doposial neznáme chyby a defekty boli objavené a následne opravené. Veľké úsilie bolo vynaložené na refaktorovanie niektorých častí kódu, ktoré boli vytvorené a udržiavané počas rokov niekoľkými desiatkami prispievateľov, z ktorých každý mierne používal odlišný štýl kódovania.

Nová funkcionalita umožnuje dynamicky znovu načítať konfiguráciu základných komponentov bez potreby úplného reštartu. Okrem toho je možné zmeniť vnútorné štruktúry rsyslogu bez toho, aby bola ohrozená schopnosť spracovanie prichádzajúcich dát u nezmenených komponentoch. Počas práce boli implementované len najdôležitejšie testy, aby čiastočne pokryli novú funkcionalitu programu. Nakoniec, viacero nápadov bolo predstavené

ako potenciálny plán do budúcnosti. Vzhľadom k dosiahnutým výsledkom a možnostiam budúceho rozšírenia by práca mohla byť užitočná aj v praxi.

Support for dynamic config reload inside rsyslog

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of doc. Mgr. Rogalewicz Adam Ph.D. The supplementary information was provided by Ing. Vít Mojžíš and Rainer Gerhards. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Attila Lakatos
May 10, 2022

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Doc. Mgr. Rogalewicz Adam Ph.D. and Ing. Vít Mojžíš for the continuous support of my master's thesis. I also want to express my gratitude to Rainer Gerhards, the official rsyslog project maintainer for his professional technical help, valuable advice and suggestions that helped me to complete this work. Furthermore, I would like to thank my family and friends for supporting me through the whole duration of my academic pursuits.

Contents

1 Introduction **2**

2 Introduction to logging on RHEL **4**
- 2.1 Log files 4
- 2.2 System auditing 5
- 2.3 Systemd journal 6

3 Rocket-fast system for log processing **7**
- 3.1 Syslog 7
- 3.2 Design 9
- 3.3 Configuration 10

4 Implementing dynamic configuration reload option **24**
- 4.1 Motivation 24
- 4.2 Refactoring 25
- 4.3 Using x-macros 31
- 4.4 Invocation of dynamic configuration reload 33
- 4.5 Reloading configuration directives 35
- 4.6 Bugfixing 41

5 Testing and evaluation **43**
- 5.1 Github actions 43
- 5.2 Custom tests 47
- 5.3 Limitations 49
- 5.4 Proposed enhancements 50

6 Conclusion **51**

Bibliography **52**

A Rsyslog Upstream Work **55**

B Contents of the Attached Media **57**

Chapter 1

Introduction

Ever since humankind developed the ability to write, much of our progress has been made thanks to recording and reusing saved data. In the past, notes on the production and gathering of resources, the exact number of sold goods and other crucial personal information, were created and stored by hand. Due to this documentation method, important data was exposed to being lost, stolen, or mishandled. Nowadays, nearly every important event is logged somewhere in some form. So having a fast and reliable log management system is now more important than ever before. Unlike other commonly used applications, such as web browsers, text editors or antivirus software – log management systems are not part of the list that a typical user gives much thought to. Big percentage of users are not even aware of what logs are, how they operate and what purpose they serve. Notwithstanding, log management tools belong to the most significant parts of the IT industry. This fact is absolutely acceptable until there is someone in the background who takes care of managing logs, e.g. a system administrator. Taking into consideration that technology changes and evolves quickly every day, we need the proper tools to gather information regarding problems that occurred in our system.

Users of Red Hat Enterprise Linux distributions are entitled to use the rocket-fast log processing system for managing their logs [25]. It offers the ability to accept inputs from a wide variety of sources, transform them, and output the results to diverse destinations by a set of rules specified inside configuration files. While rsyslog evolved into a kind of swiss army knife of logging, it lacks one particular feature when comparing with other well-known logging applications. Currently, it lacks the ability to automatically reload the core configuration without a complete restart. Above all, this causes poor performance, disturbance of TCP based connections between servers and clients (even if no changes are made), and loss of messages entering rsyslog while it's being restarted. The thesis aims to implement a dynamic configuration reload option for rsyslog. The proposed feature is targeted towards open-source project repository[1], so other users can benefit from it.

Chapter 2 provides a basic introduction to logging systems on Red Hat Enterprise Linux (RHEL) distributions and explains the reasons why it's recommended to use them. Next, properties, features, pros and cons of such systems are described. This chapter ends with an overview of existing logging approaches that are used widely in the production environment. In Chapter 3, the rocket-fast system for log processing, rsyslog is deeply introduced with special attention to its configuration file. In this part of the thesis, the reader will get acquainted with the core elements that can modify the behavior of an rsyslog running

[1]https://github.com/rsyslog/rsyslog

instance. Afterwards, the author of the thesis points out why it is necessary to implement a dynamic configuration reload option inside the mentioned logging system. Designed and proposed changes are discussed in Chapter 4, along with implementation details for individual parts. Finally, achieved results and possible future development ideas are discussed in Chapter 5.

Chapter 2

Introduction to logging on RHEL

The aim of this chapter is to give the reader an insight into the complex process of logging. Such knowledge is inevitable in order to understand the necessity of maintaining and improving existing logging services. The chapter begins with a brief description of logs, with emphasis on their importance. Furthermore, we thoroughly explain how the process of logging and monitoring is accomplished on RHEL distributions.

2.1 Log files

Every single computer system, regardless of the operating system, has a mechanism that records nearly all the performed activities. This kind of information is called *log* in information technology. Log files are usually text files that contain messages about the system, including the kernel, services, and applications running on it [25]. Logs show us what happened behind the scenes and when it happened, so that if something should go wrong with our systems, we have a detailed record of every action prior to the anomaly. Usually, an average user is barely interested in such a list of events, however it plays a fundamental role in the life of a system maintainer if something extraordinary happens.

Originally, logs were used primarily for troubleshooting problems, but now serve for many different purposes, such as recording user activities, tracking authentication attempts, crashes of systems, system resource exhaustion, and other security events that may indicate possible attacks, hardware malfunction, or other issues. Due to an increasing number of threats against networks and systems, the number of security logs also increases. Log files make it easier for both developers and system administrators to get insights and identify the root cause of issues with applications and infrastructure [17]. Apart from that, they can also reveal security breaches, or identify weaknesses that might affect compliance of a system with common criteria[1].

Log management provides insight into the status and compliance of our systems and applications. Without it, we would be stumbling around in the dark hoping to pinpoint sources of bugs, unexpected behavior, performance issues, and other similar unwelcomed problems. We would be forced to manually inspect multiple log files while trying to troubleshoot production issues. This is painfully impractical, slow, fault-prone, expensive, and unscalable [2]. Fortunately, enumerated drawbacks do not apply to users of RHEL. For such purpose, the *sosreport* command line tool is used, which collects configuration details,

[1]an international set of specifications and guidelines designed to evaluate information security products and systems [1]

system information and diagnostic information from a system, including logs generated by different logging systems. Generally speaking, the output of a *sosreport* is the starting point for Red Hat engineers when troubleshooting a service request [27]. The following sections will introduce various logging daemons utilized on RHEL.

2.2 System auditing

The Linux Audit system is a native feature to the linux kernel that collects certain types of system activity to facilitate incident investigation [26]. According to a predefined rule set, Audit records all kinds of information about the events that occur on our system. In general, they capture who performed an activity, what activity was executed, and how the system responded. In order to avoid any misunderstanding, it's important to mention that Audit does not provide extra security to our system; rather, it is used to track down violations of security policies occurring on our system [26]. It was designed to be integrated with SELinux[2] and other parts of the kernel [5] that can prevent discovered violations.

The following list encapsulates some of the most important use cases:

- **Recording security related actions** - keeps an eye on failed authentication attempts.
- **Keeping track of system calls** - generates log entries in case a certain system call is invoked.
- **Watching file access** - registers whether a file or directory has been accessed, modified, executed, or its attributes have been changed.
- **Searching for events** - provides the ability to filter out logs based on their properties
- **Creating summary reports** - suspicious activities can undergo further analysis by using automatically generated reports

Regarding its architecture, the Audit system consists of two parts, the user-space application and utilities, and kernel-side system call processing. The kernel component receives system calls from various applications. If no *exclude* filter is associated with a certain system call, then one of the following filters is applied [26]:

- *user* - events originating in user space before being relayed to the audit daemon
- *task* - events related to creating new processes
- *fstype* - events associated with a certain filesystem
- *exit* - events that take place on system call exit

Afterwards, based on the configured Audit rule set, each system call records is sent to the Audit daemon[3] for further processing. Figure 3.4 illustrates this process. It is important to note that the *auditd* daemon offers an ability to dynamically reload its configuration via SIGHUP signal [31], which is the ultimate goal of the thesis for rsyslog.

[2]stands for Security Enhanced Linux, which is an access control system that is built into the Linux kernel
[3]a long-running background process that answers requests for services

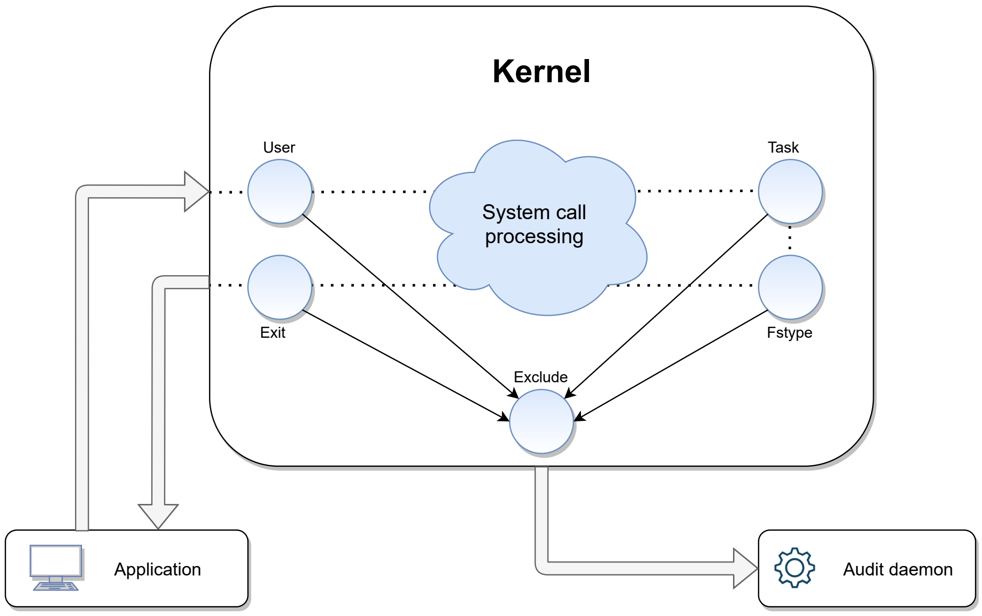


Figure 2.1: Audit system architecture

2.3 Systemd journal

Another possibility to control log files is through the *journald* daemon – a core component of *systemd*[4]. The Journal was not developed to replace other logging daemons, such as *rsyslog*; rather, it was introduced to solve problems associated with traditional logging [25]. Nevertheless, the *journald* daemon is capable of capturing both Operating System and application level messages in one place. Logged data is not stored in a large text file. It consists of multiple binary files, also-called journals, that are maintained by the daemon itself, thus can not be opened by a traditional text editor, making logs more secure. In general, the native journal file format is indexed and structured, so lookups are much faster than with plain text files. Furthermore, structured data means that messages do not only consist of a fixed list of fields but makes it possible for applications to define their own format for the message. By default, above-mentioned files are saved in a small ring-buffer or in memory, which is enough to show recent journal history. However, it is highly customizable and it supports persistent storage as well. At the same time, *journald* does not include a well-defined remote logging implementation. Instead, it relies on existing syslog implementations to relay messages to a central log host [12].

As mentioned earlier, journals are not intended to be opened by traditional text viewers; instead, the *journalctl* command line tool comes in handy when we need to query these files. This program utilizes the native journal format to give extremely fast access to log entries filtered by certain conditions, such as values of PID[5], UID[6], service name and many other fields[16]. With this in mind, we are capable of viewing the last few log entries of a service, as the journal gives fast access to its data. To sum up, the log format of *journald* makes it less difficult for programs to retrieve only the information they want to know.

[4]a system and service manager for Linux operating systems
[5]abbreviation for process identification number
[6]abbreviation for user identifier

Chapter 3

Rocket-fast system for log processing

The aim of this chapter is to present the rocket-fast system for log processing(rsyslog) with special attention to its configuration file. The chapter starts with a brief introduction to the syslog protocol, which is the absolute bedrock for rsyslog. Afterwards, detailed description for each basic structure is presented. Such knowledge is necessary in order to be able to contribute to the project.

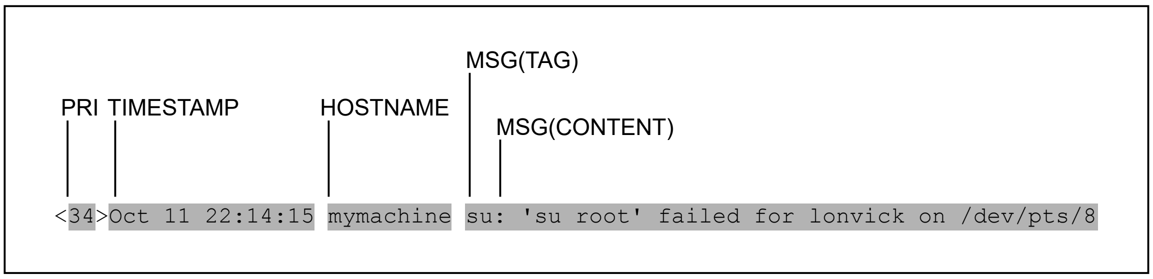
3.1 Syslog

Syslog is a logging format and protocol created in the 1980s, and has since gained popularity in Unix-like systems – including BSD Unix, Linux and macOS – as well as networks devices, such as printers, switches, firewalls and routers. These events are typically logged locally where they can be reviewed and analyzed by the appropriate person, typically a system administrator. However, monitoring huge amounts of logs over a vast number of devices could be time consuming and impractical. This can be addressed by forwarding those specific logs to a centralized server. In other words, it provides a way to record logs using a united format, receive syslog messages from other sources, as well as forward them to diverse destinations. It sounds astonishing, at least in theory - using a predefined format to represent all kinds of log messages and transmit them through various systems [19]. In most cases, syslog messages can be defined in two major formats:

- The BSD syslog Protocol(RFC3164) [3]
- The Syslog Protocol(RFC5424) [22]

The BSD syslog Protocol

The first document that strove to standardize the syslog protocol was RFC 3164 [3], which dates back to 2001. Although RFC suggests it's a standard, it was more like a collection of what was found and used in the wild at the time, rather than a clear set of specifications that implementations will adhere to [18]. As a result, there are numerous variations of it. With this in mind, applications implementing the BSD syslog protocol might not be compatible with each other. That said, most logs respecting the RFC 3164 might have the following structure:



```
<34>Oct 11 22:14:15 mymachine su: 'su root' failed for lonvick on /dev/pts/8
```

Figure 3.1: The anatomy of an RFC 3164 format syslog message [3]

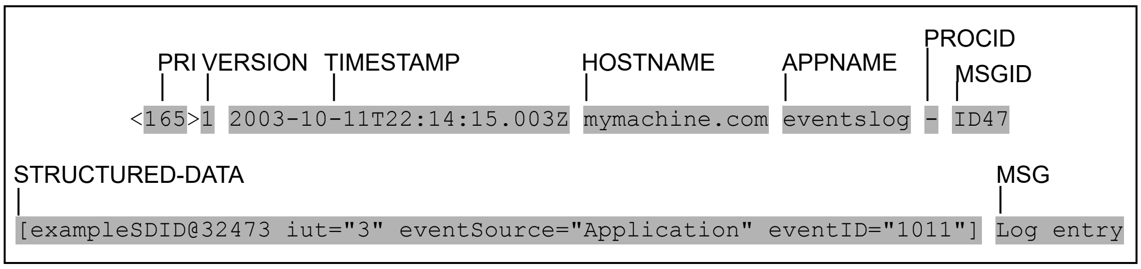
Following is a description of each field:

- PRI - represents the facility number multiplied by 8, to which severity is added
- TIMESTAMP - the local time in the format of “Mmm dd hh:mm:ss”
- HOSTNAME - hostname, IPv4 or IPv6 address of the sender
- MSG consists of two parts:
  - TAG - the process or program that generated the message
  - CONTENT - details of the message

While the BSD syslog protocol was a good starting point towards standardization, it clearly needed to be reevaluated. First of all, the *timestamp* attribute misses the year, time-zone and doesn’t have sub-second information. Moreover, the beginning of section 4 states that “*The payload of any IP packet that has a UDP destination port of 514 MUST be treated as a syslog message.*” [3]. This can be interpreted as messages sent to a UDP destination port of 514 are not strictly required to have any special format, so junk messages are also treated as valid syslog messages. To sum up RFC 3164, it was meant to describe what has been seen in practice and provide some useful information, but does not precisely specify anything.

The Syslog Protocol

The need for a complete specification had arisen because standardization efforts for reliable and secure syslog extensions suffered from absence of clear message format and transport-independent definition. Without such a document, every application needs to define its own syslog packet format and transport mechanism, which could cause compatibility issues over time. RFC 5424 [22] came up in 2009 to deal with the problems caused by RFC 3164. It’s an actual standard that applications and libraries can adhere to. Figure 3.2 shows an example of a valid syslog message using RFC 5424.



```
<165>1 2003-10-11T22:14:15.003Z mymachine.com eventslog - ID47
[exampleSDID@32473 iut="3" eventSource="Application" eventID="1011"] Log entry
```

Figure 3.2: The anatomy of an RFC 5424 format syslog message [22]

Timestamp field has been formalized according to RFC 3339[1], and it does not lack year and timezone information. Next, additional attributes are introduced, including *application name* and *process ID*. The new format also supports UTF8 and other encodings, and it's easier to extend due to a version number. Moreover, it provides a mechanism to express information in a well-defined, easily parsable and interpretable data format. In other words, *structured-data* field may express metadata about an application-specific syslog message.

3.2 Design

Rsyslog is an open-source software utility used on Unix-like computer systems, which is capable of accepting inputs from a wide variety of sources, transform them, and output results to diverse destinations [20]. It implements the basic syslog protocol, and among others, extends it with additional features, such as:

- support for modular design
- rich filtering capabilities
- logging directly into various databases
- Reliable Event Logging Protocol(RELP)
- ability to forward logs via TCP protocol
- increased security via TLS/SSL protocols
- ISO 8601 timestamps with millisecond precision and time zone information
- support for RFC 5424, RFC 5425[2], RFC 5426[3] and many other improvements

One of its design goals is to support a huge amount of messages per second. Rsyslog was optimized for performance by means of massive multi-threading, but that does not necessarily mean that each configuration, and even each use case, can actually benefit from it. When configured properly, it is able to deliver over one million messages per second to local destinations. Figure 3.3 encapsulates various modules rsyslog can communicate with by either gathering logs from various sources or forwarding messages to diverse destinations.

[1]https://datatracker.ietf.org/doc/html/rfc3339
[2]https://datatracker.ietf.org/doc/html/rfc5425
[3]https://datatracker.ietf.org/doc/html/rfc5426

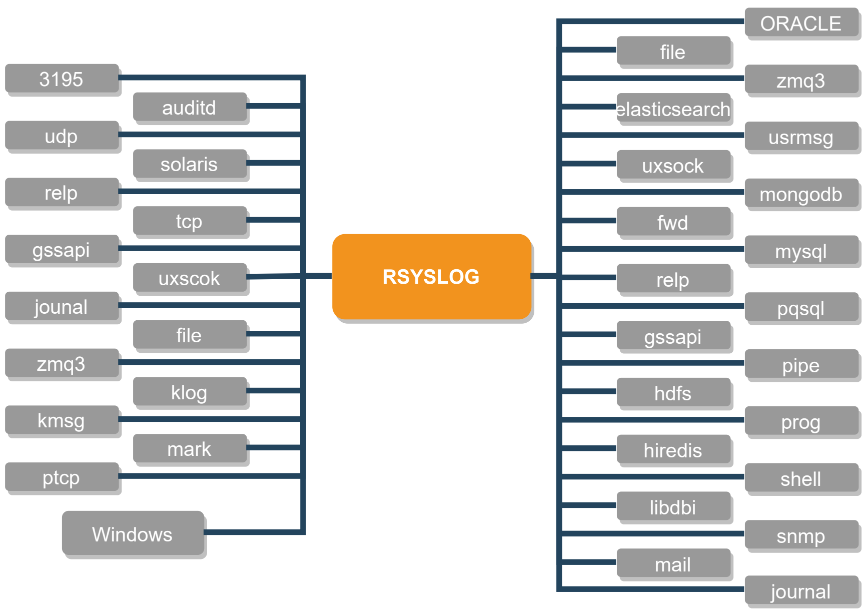


Figure 3.3: Modular design[20]

Rsyslog uses an object-oriented paradigm, however it is written in C programming language. On the other hand, rsyslog is also modular – its core communicates with loadable plugins, which may implement message parsing, input, output or other functionalities. With this approach, a complex product can be broken down into smaller parts that are independently designed and later integrated. From a technical point of view, it can be achieved via macros, more precisely through the token-pasting operator. Detailed explanation on how the aforementioned operator is utilized can be found in the implementation part of the thesis. Each plugin must be put into some well-defined class, for instance parser plugins are responsible for parsing the content of a message, input plugins gather messages for further processing and output plugins forward messages. For each class, a specific interface exists, which must be implemented by the appropriate developer responsible for the plugin. Some parts of the interface do not have to be explicitly implemented, in that case the default behavior is used. Such a phenomenon can be associated with class hierarchy and inheritance. Nevertheless, most of the core functionality is provided in the shape of class-like modules. When a plugin attempts to access the functionality, it first acquires access to the class's public interface. Afterwards it's capable of managing objects by calling appropriate member functions. In rsyslog terminology, objects are actually C structures, while member functions are basically pointers to functions [23].

3.3 Configuration

We can think of rsyslog as a framework with some basic processing that is fixed in the way data flows, but is highly customizable in the details of this message flow. During configuration, this customization is done by defining and customizing the rsyslog core elements. Upon startup, rsyslog reads its main configuration from the */etc/rsyslog.conf* file by default. Mentioned file may contain references to include other configuration files. For historical reasons, rsyslog supports three different types of configuration statements concurrently:

- **Syslogd** - rsyslogd is based on the standard syslogd project. While it contains many additional features, a great effort has been made through the years to keep the configuration file as compatible as possible. This is exceptionally advantageous when migrating from syslogd.
- **Legacy rsyslog** - set of statements that begin with a `$` character. They influence both local and global configuration parameters. As the name implies, such components are not recommended to be used but are part of the project to ensure backwards compatibility.
- **RainerScript** - easily readable and practical scripting language that was designed to be used for more complex cases.

From syntax point of view, listings 3.1 and 3.2 demonstrate the same example but written in different formats.

```
$FileCreateMode 0644
$FileOwner bob
$FileGroup users
$DirOwner bob
$DirGroup users
*.* /var/log/mylog.txt
```

Listing 3.1: Legacy(old) format

```
action(type="omfile"
  file="/var/log/mylog.txt"
  fileCreateMode="0644"
  fileOwner="bob" fileGroup="users"
  dirOwner="bob" dirGroup="users"
)
```

Listing 3.2: RainerScript(new) format

At first glance, it might not be obvious that legacy format lacks the ability to have multiple parameters on the same line, moreover it's not exactly clear which parameters are part of the output. The definition of multiple actions may be unintentionally intermixed, which may result in needlessly complex configuration files. *RainerScript* format addressed this issue by encapsulating object data between parentheses, so configuration files became more human-friendly and less buggy.

Input modules

As the name implies, input modules are used to gather messages from various sources, including container logs, standard text files, messages via http protocol, kernel logs and many more. The full list of sources is shown on the left side of figure 3.3. They are generally defined via `input` configuration objects. Without input, no processing happens at all, because no message enters the system. Inputs are treated as objects that have different parameters:

- **generally available parameters** for all inputs regardless of their purpose. The most notorious one is the `type` field, which is a string identifying the input module.
- **input-specific parameters** are specific to only a certain type of input. Every single input module has a comprehensive documentation provided by the community.

In order to give a comprehensive overview of how inputs work in rsyslog, we will demonstrate how a centralized server gathers inputs from various sources via `TCP` protocol. For such a use case, the `imtcp` module is recommended[4] to be utilized because it provides

[4]`imptcp` module is an alternative to be used but lacks the ability to provide encryption

the ability to receive syslog messages via TCP. Among other enhancements, encryption is natively provided by the appropriate network stream driver, which is vital to keep the confidential content of syslog messages secure. The following example, listing 3.3 reveals how straightforward it is to collect messages addressed to port 514 via TCP.

```
module(
   load="imtcp" # module name
   maxSessions="500" # the maximum number of sessions supported
   maxListeners="10" # the maximum number of listeners(server ports) supported
   notifyOnConnectionClose="on" # to emit a message if the remote peer closes connection
)
input(type="imtcp" port="514")
```

Listing 3.3: Gathering inputs on a centralized server

Output modules

As the name implies, output modules are used to transmit messages to a vast number of different targets, including standard text files, systemd journal, various databases, TCP/UDP servers and many more. The full list of targets is shown on the right side of figure 3.3. They are generally defined via `action` configuration objects. The action object has highly customizable parameters:

- **generally available** - action-specific parameters that are accessible for all outputs regardless of their purpose
- **action queue** - queue-specific parameters that apply to all parameters as well, however these are not action-specific
- **action-specific** - parameters that are specific to only a certain type of actions

To give a demonstration of how actions are defined in configuration files, let's assume that we wish to forward system logs to a centralized rsyslog server via `TCP` protocol. For such a use case, the `omfwd` module is recommended to be utilized because it provides the ability to forward traditional messages via `TCP`. Encryption can also be turned on by selecting the appropriate network stream driver, which helps to protect private information from third parties. The example, shown in listing 3.4, is an rsyslog client that forwards every single message to an rsyslog server.

```
module(load="omfwd")
action(
   type="omfwd"
   target="10.0.138.187" # name or IP address of the rsyslog centralized server that shall
    receive messages
   port="514" # numerical value of port to use when connecting to target
   protocol="tcp" # type of protocol to use for forwarding
)
```

Listing 3.4: Forwarding logs to a centralized server

Parser modules

Unfortunately, in practice, many syslog messages transmitted over the wire are malformed. As it was mentioned in section 3.1, this might have unpleasant consequences, above all compatibility issues between applications implementing the syslog protocol. This problem can be avoided by using the appropriate parser modules. The purpose of a message parser is to convert message data into rsyslog's internal format in a lossless manner. On a high-level overview, message parsing takes place after messages are received through inputs and right before the application-level processing[5] [21].

It's critical to be aware of the fact that rsyslog parsers do not operate at the transport layer; rather, as their name suggests, they operate on raw messages. Part of the RFC 5424 specification is a layered architecture for the syslog protocol [22]:

- **syslog content** - the management information contained in a syslog message
- **syslog application** - application-specific content, syntax, routing and storage of syslog messages
- **syslog transport** - puts messages on the wire and takes them off the wire, such as UDP, TCP

The purpose of the transport layer is to specify how a stream of messages is assembled at the sender side and how it's later disassembled into smaller chunks at the receiver side. From a networking point of view, this is called *framing*. With the help of framing, all kinds of message content can be properly transferred. On the other hand, for instance, if by chance a sender splits a message into two parts and encapsulates into two frames, the message parser will not be able to undo that. This is the reason why message parsers are not intended to be used for fixing a malformed framing. Regrettably, many implementations used in practice violate the standard, which sometimes makes it extremely difficult to interpret the content of a message or to process messages from various sources by the same rules.

Parsers should be grouped together to provide more flexibility, resulting in so-called parser chains. Each parser chain consists of one or more message parsers. The position inside the list of parsers can be interpreted as a priority, so parsers being on the left side of the chain take precedence over their right neighbors. In other words, it is recommended to place more specific parsers at the beginning of the chain and similarly, general parsers should be the last ones.

For demonstration purposes, let's extend listing 3.3 by introducing `rsyslog.rfc5424` and `rsyslog.rfc3164` message parsers, which were briefly explained in section 3.1.

```
module(
   load="imtcp" # Module name
   maxSessions="500" # The maximum number of sessions supported
   maxListeners="10" # The maximum number of listeners(server ports) supported.
   notifyOnConnectionClose="on" # To emit a message if the remote peer closes a connection.
)
ruleset(name="ruleset" parser=["rsyslog.rfc5424", "rsyslog.rfc3164"]) {
        action(type="omfile" file="/var/log/messages")
}
input(type="imtcp" port="514" ruleset="ruleset")
```

Listing 3.5: Collected inputs are parsed through either RFC 5424 or RFC 3164

[5]for example, write data to a database or forward to other destinations

To summarize, message parsers provide the ability to preprocess malformed messages, parse them according to their semantics and generate valid internal message structures from it.

Rulesets

An rsyslog configuration file is made up of rules (some documents tend to call them selectors). Each rule consists of a filter and one or more actions to be executed when that particular filter evaluates to true. A filter may be:

- a simple syslog priority-based filter, for instance *mail.info* or
- a complex expression, similar to those used in programming languages

Depending on the defined action type, actions do something with a message, e.g. they might forward it to remote hosts, write it to a file or database. Inside a ruleset, messages are processed in the order of appearance in the configuration file until either no more rules are left or an explicit stop command is found. Every single input must bind to a ruleset, otherwise the default is used. Binding to a ruleset means that a specific input will only deliver collected logs to that particular ruleset.

Rulesets are useful when a standard system that logs its local messages to the usual files under /var/log/ directory wants to log messages received from remote hosts to separate files. Listing 3.6 demonstrates the process of bounding an input to a ruleset.

```
# define a ruleset containing a single action
ruleset(name="remote514"){
    action(type="omfile" file="/var/log/remote514.log")
}

# collect logs on port 514 via tcp protocol
module(load="imtcp)
input(type="imtcp" port="514" ruleset="remote514")
```

Listing 3.6: Usage of rulesets

The example above does not cover how complex the ruleset directive actually is. Among other things, it may contain conditional statements, foreach loops, variables of different types, expressions, both direct and indirect calls to other rulesets, regular expressions and many more. Basically, it can be thought of as a restricted programming language.

Properties

One common problem that users run into when using properties is the fact that the different types of variables were added to rsyslog at different times, and as a result there are different ways they are named [6]. In rsyslog terminology, data items are called *properties*[6] which are distinguished by their origin. Their main purpose is to ease our work when writing rsyslog configuration files. From a syntactic point of view, properties are used either in templates or conditional statements, and can be grouped by the following three different aspects:

[6]variables

- **message properties** - these are extracted by rsyslog parsers from the original raw message. The most important properties belonging to this category are source/destination IP addresses, syslog tag, program name and priority but the full list is enormous.
- **system properties** - are provided by the rsyslog core engine but unrelated to the original message. Such properties might encompass information about the time when the message was created at the original sender or when the message was actually handed over from the operating system to rsyslog's buffers.
- **user-defined variables** - most recent versions of rsyslog allow us to define our own variables in the config file in addition to the ones created by the system itself. Variable customization is considered to be an aid for template generation and modification.

Templates

Templates have been mentioned multiple times, however no explanation was given regarding what exactly this term means or what role it plays in rsyslog's configuration file. They allow to specify any kind of format a user might want to utilize. Every single output has a corresponding template, even though it's not visible to the outside world. At first glance, some might be curious about what makes them different from message parsers. Message parsers take the received raw message and create the internal structure out of it. Afterwards, templates are used to specify a clearly defined format to modify the log message that we send out to any destination.

Some might be confused about how this actually works when no template was specified at all in previous examples. By default, rsyslog uses a few builtin templates that are compatible with the original syslogd formats. In case of omitting the template field from an output action, the hardcoded ones are used.

Templates are defined via `template()` statement. It's worth to mention that it is a static element, thus all templates are defined when the configuration file is read by rsyslog. Each template has a parameter `name`, which identifies the template name, and a parameter `type`, which represents the template and can be set to one of the following: list, subtree, string, plugin.

The **list template** is generated from a list of `constant` and `property` statements. As the name states, `constant` elements represent constant values and `property` elements represent properties. To grasp the idea, let's consider the following:

```
template(name="tpl1" type="list") {
  constant(value="Syslog MSG is: '")
  property(name="msg")
  constant(value="', ")
  property(name="timereported" dateFormat="rfc3339" caseConversion="lower")
  constant(value="\n")
}
```

Listing 3.7: List template example

In comparison with the legacy template statement, **string templates** are relatively similar, with only a couple of improvements in terms of syntax and performance. They have a mandatory parameter `string`, in which the template to be applied is constructed.

This type of template is made up of constant text and replacement variables[7]. String-based templates are recommended to be used for simple use cases, particularly if no complex manipulation of properties is required. For example, listing 3.8 shows a string template containing basic log entries, such as time, hostname, syslogtag and message content without the last LF[8].

```
template(name="tpl2" type="string"
  string="%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag% %msg:::drop-last-lf%\n"
)
```

Listing 3.8: String template example

The next type we will cover is the **plugin template**, which is generated by a plugin. In contrast to the previous template types, its format is hardcoded and does not provide any ways to influence it. While this sounds as a huge drawback, it provides superior performance, and is often used for that reason.

```
template(name="tpl3" type="plugin"
  # the old style default log file format with low-precision timestamps
  plugin="RSYSLOG_TraditionalFileFormat"
)
```

Listing 3.9: Plugin template example

The **subtree template** is responsible for creating output structures based on a CEE[9] subtree. Such a template comes in handy for outputs that represent hierarchical structures.

Filter conditions

Filter conditions were designed to enable the logical selection of property records, using simple statements in various conditions. Currently, rsyslog offers three different types of filtering statements, including:

- severity and facility based selectors
- property-based filters
- expression-based filters

First of all, **selectors** are the traditional way of filtering messages. They are still part of rsyslog, because they provide high performance message selection and backwards compatibility with traditional syslog configuration files. Selectors are recommended to be used for filtering messages based on facility and priority. The selector itself is made up of a facility and a priority, separated by a comma. The example in listing 3.10 is part of the default configuration file used in Fedora Linux 35 distributions[10].

[7] rsyslog variables might be altered by property replacers
[8] Line Feed
[9] it's a JSON/XML structured log format more powerful and easy to use than logging and parsing custom string formats
[10] https://src.fedoraproject.org/rpms/rsyslog/blob/f35/f/rsyslog.conf#_47

```
# Log anything of level info or higher(*.info), except for private authentication
# messages(authpriv.none), mail(mail.none) logs and automated cron messages(cron.none)
*.info;mail.none;authpriv.none;cron.none    /var/log/messages

# The authpriv file has restricted access
authpriv.*                                  /var/log/secure

# Log all the mail messages in one place
mail.*                                     -/var/log/maillog

# Log cron job messages
cron.*                                      /var/log/cron
```

Listing 3.10: Severity and facility based selectors

The **property-based filters** provide rich filtering capabilities based on message properties, like source, destination or even message content. This filter is capable of checking a particular property against a specified value via specified compare operation. The following listing, 3.11 demonstrates how we can use it in our configurations. For simplicity, we omit message destinations.

```
# if source IP is equal to 95.102.107.212
:fromhost-ip, isequal, "95.102.107.212"
# if the content of message starts with IMPORTANT
:msg, startswith, "IMPORTANT"
```

Listing 3.11: Property-based filters

The **expressions-based filters** were designed to process arbitrary complex expressions, such as boolean, arithmetic and string operations. These filters use rsyslog's own scripting language called RainerScript to build complex filters [20]. From syntax point of view, expression-based filters are indicated by the keyword `if` at the start of a new line. The `then` keyword separates the expression from the action. Optionally, we are given the opportunity to employ the `else` keyword to specify what action is to be performed in case the condition is not met. In listing 3.12, logs created by a program called `sudo` containing error messages are written into a file.

```
# if EXPRESSION then ACTION else ACTION
if $programname == 'sudo' and $msg contains 'error' then /var/log/sudo-errors.log
```

Listing 3.12: Expression-based filters

Network stream drivers

Traditional syslog is a clear-text protocol. That means anyone with a sniffer tool can have a peek at our data. In some environments, this is no problem at all. In others, it is a huge setback, probably even preventing deployment of syslog solutions. Thankfully, there are easy ways to encrypt syslog communication, and this is the reason why TLS encryption plays a crucial role in rsyslog.

Transport Layer Security (TLS) [11] is a crucial part of cybersecurity protocols for organizations of any size, including logging systems such as rsyslog. TLS is designed to secure data against hackers and to ensure that sensitive information such as passwords and credit card numbers stay safe.

In order to build an encrypted syslog channel, we need to use the appropriate netstream drivers on both the server and client sides. Each side also requires a digital certificate. Such a certificate enables SSL operation, as it provides crypto keys being used to secure the connection. Along with the digital certificate, a root CA[11] is also required to secure syslog communication over the wire [24]. For the sake of clarifying how TLS encryption works in rsyslog, already demonstrated examples 3.3 and 3.4 will be extended with TLS protected transport via the OpenSSL library.

```
global(
    DefaultNetstreamDriverCAFile="/etc/rsyslogd.d/ca-cert.pem"
    DefaultNetstreamDriverCertFile="/etc/rsyslogd.d/server-cert.pem"
    DefaultNetstreamDriverKeyFile="/etc/rsyslogd.d/server-key.pem"
)
module(
    load="imtcp"
    permittedPeer="client-hostname"
    protocol="tcp"
    streamDriver.Mode="1"
    streamDriver.Name="openssl"
)
input(type="imtcp" port="514")
```

Listing 3.13: Securely gather inputs on a centralized server

```
global(
    DefaultNetstreamDriverCAFile="/etc/rsyslogd.d/ca-cert.pem"
    DefaultNetstreamDriverCertFile="/etc/rsyslogd.d/client-cert.pem"
    DefaultNetstreamDriverKeyFile="/etc/rsyslogd.d/client-key.pem"
)

module(load="omfwd")
action(
    type="omfwd"
    target="centralized-server-IP"
    port="514"
    protocol="tcp"
    streamDriver="openssl"
    streamDriverMode="1"
    streamDriverAuthMode="x509/name"
    streamDriverPermittedPeers="server-hostname"
)
```

Listing 3.14: Securely forward logs to a centralized server

Extending existing configuration with TLS support is not demanding at all. Note that there are many different authentication modes available in rsyslog. In the example above,

[11] A Root CA is a Certificate Authority that owns one or more trusted roots.

the `x509/name` was put into use, which stands for certificate validation and subject name authentication as described in IETF's TLS transport mapping for rsyslog[12].

Queues

If there is a single object absolutely vital to understand the way rsyslog works, this object is undoubtedly the queue. The reason for why it has not been explained earlier is the pre-requisites required for understanding such a concept. With the help of queues, rsyslog is able to process multiple messages simultaneously and to apply several actions to each of them at the same time [28]. The figure below describes the data flow inside rsyslog:

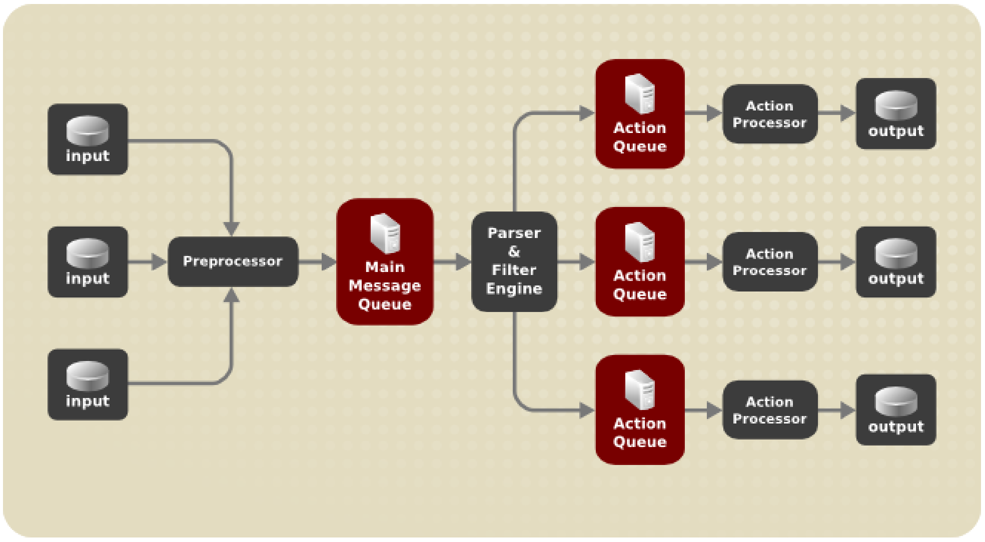


Figure 3.4: Message flow in rsyslog [28]

Once rsyslog receives a message, it forwards this message to the preprocessor and places it into the *main message queue*. Depending on the system traffic, messages wait there to be decoupled, and furthermore passed to the *parser and filter engine*. Next step is about applying the rules defined in the configuration file. Based on predefined rules, the *filtering engine* decides which actions will be carried out. Note that each action has its own action queue. Messages are forwarded to n action queues, with n being the number of actions that the message in question needs to be processed by. In general, a message will be in at least two queues during its lifetime – main message queue and one action queue. Bear in mind that at this point, multiple actions can be carried out at once on each message. To sum up, queues provide two main advantages that lead to increased performance of message processing:

- decoupling of producers(inputs) and consumers(outputs) in the structure of rsyslog
- support for parallelization of actions performed on messages

According to the location where messages are stored, various types of queues can be distinguished: *direct*, *in-memory*, *disk* and *disk-assisted in-memory queues* [28].

A **direct queue** does not buffer any of the main queue elements but rather immediately passes the element directly from the producer to the consumer. At first look, this might

[12]https://datatracker.ietf.org/doc/html/draft-ietf-syslog-transport-tls-12

sound strange, but there is a good reason behind this. Direct mode queues allow the use of queues generically, even in places where queuing is not wanted. In general, we do want to buffer database writes or forwarding actions but not simple local file writes, thus it always depends on the actual use case.

On the contrary, **disk queues** store messages strictly on a hard drive and do not buffer anything in memory. In other words, this type of queuing is the most reliable solution of all, but by far the slowest mode. This mode can be used to prevent the loss of highly important data at all costs.

With an **in-memory disk queue**, the enqueued data elements are held in memory. While this type of queue is extremely fast, queued messages are lost if the machine is shut down or power cycled. In-memory queues are capable of holding data for a theoretically infinite amount of time, for instance when the destination server is offline. At the time of writing this thesis, rsyslog implements two in-memory queue modes, linked list and fixed array. Both are quite similar from the user experience point of view, but utilize different algorithms. A fixed array queue uses a fixed, pre-allocated array that holds pointers to individual queue elements. It offers the best run time performance when there is a small amount of elements waiting to be processed. On the other hand, linked list queues ensure that all elements are dynamically allocated when there is a need for that. This type of in-memory queue is well-suited for queues where only occasionally a large number of elements need to be queued.

Finally, both disk and in-memory queues have their advantages and rsyslog lets us combine them in **disk-assisted in-memory queues**. In an ideal situation, they are capable of storing messages in memory, so they do not even touch the disk. However, if there is a need for it, an unlimited number of messages can be buffered and data can be persisted after complete shutdown. Using disk-assisted memory queues is not simply about writing everything to disk once the in-memory queue is full. Instead, a much more practical algorithm was put into use, which involves an upper and lower limit regarding the queue size. Once the number of queued items reaches the upper limit, the queue will begin to write data elements to disk until it reaches the pre-configured lower limit. The purpose of this approach is to hold enqueued elements in-memory for performance tuning but enable persistent storage for occasional message burst to ensure reliability.

In order to demonstrate the usefulness of queues, let's suppose the task is to forward log messages from the system to a remote server, and ensure that no message is lost in case of a server outage. We can do it by extending an already introduced example 3.4 with a disk-assisted in-memory queue. To do so, we need to configure the following:

```
module(load="omfwd")
action(
    type="omfwd" target="10.0.138.187" port="514" protocol="tcp"
    queue.type="LinkedList"
    queue.filename="/var/log/actions-queue"
    queue.saveonshutdown="on"
    Action.resumeRetryCount="-1"
)
```

Listing 3.15: Enqueue data items when forwarding to a remote server

Where:

- `queue.type` specifies the type of queue being used, e.g. Linked list in-memory queue,

- queue.filename enables disk-assisted queue functionality,
- queue.saveonshutdown specifies if data should be saved at shutdown
- action.resumeRetryCount option prevents are rsyslog from message loss when remote server is offline.

Dynamic stats

As mentioned in section 2.1, log management provides insight into the health and compliance of our systems, and without such a feature we would be stumbling around in the dark hoping to find the root cause of performance issues. In order to achieve that, rsyslog produces runtime stats, which allow users to study service health, performance and bottlenecks of their system. In rsyslog terminology, *dynamic stats* are responsible for providing metrics during log monitoring, and can be adjusted from the simplest to the most complex use cases. Unlike previous core elements, dynamic stats configuration requires a two part setup. Firstly, we need to define a bucket and set the appropriate properties that control behavior of the bucket. In the second step, the property to be monitored must be defined alongside with additional settings, such as specifying a way to react to non-zero error codes [20]. For the sake of demonstration, let's assume that a centralized rsyslog server needs to count the number of received messages from each rsyslog client.

```
dyn_stats(name="message_per_host"
    resettable="on" # counters should be reset when they are reported
    maxCardinality="3000" # maximum quantity of unique names to track
    unusedMetricLife="600") # do not occupy resources after 600 seconds of being idle

set $.inc = dyn_inc("msg_per_host", $hostname);
if ($.inc != 0) then {
    # handle non-zero return codes
}
```

Listing 3.16: Dynamic stats configuration

Lookup tables

Lookup tables are a powerful construct to obtain information based on the message content [20]. A subset of message properties can be used as an index into a lookup table that is capable of returning a value. For instance, the *fromhost-IP* message property can be used as an index, alongside with the value representing the remote office it is located in. While this functionality might be emulated using already implemented rsyslog features, in terms of performance, lookup tables are much more effective and practical. Similarly to dynamic stats, lookup table configuration requires multiple steps to be executed. Firstly, we need to define a *lookup_table* object and set the appropriate properties that control behavior of the table alongside with its initialization. In the second step, lookups are put into use to search for certain values indexed by message properties. Optionally, a third step can be included to reload lookup tables asynchronously whenever a change is made into the external json database file [20]. The example below, 3.17 is a sample of how an IP address to office building mapping might look like:

```
# Step 1
lookup_table(
    name="ip_to_office_building" # name of the lookup table
    reloadOnHUP="on" # reload table when receiving HUP signal
    file="/var/lib/mapping.json" # path to external json database file
)
# Step 2
set $.building = lookup("ip_to_office_building", $fromhost-IP);
if ($.building == "unknown") then {
    # Handle if not found
}
# Step 3 (optional)
if ($.do_reload == "y") then {
    reload_lookup_table("ip_to_office_building", "unknown")
}
```

Listing 3.17: Lookup table configuration

Timezones

Since different parts of Earth enter and exit daylight at different times, we need different time zones in real life, as well as in rsyslog. As the title implies, `timezone` objects describe timezones. Actually, these objects are used by message parser modules to convert timestamps from raw strings to internal variables in rsyslog. Each timezone is identified by an ID, which must contain the zone name as reported within the raw timestamp. It might happen that different vendors use various, sometimes non-standard names, so it is crucial to use the IDs that messages entering our system contain.

In most of the cases, timezones are put into use whenever an rsyslog client and server are in different timezones, and we would like to force the rsyslog server to record log messages using the same time as the rsyslog client.

```
timezone(id="UTC" offset="+00:00")
timezone(id="CET" offset="+01:00")
timezone(id="CEST" offset="+02:00")
timezone(id="METDST" offset="+02:00") # duplicate to support different formats
timezone(id="EST" offset="-05:00")
timezone(id="EDT" offset="-04:00")
timezone(id="PST" offset="-08:00")
timezone(id="PDT" offset="-07:00")
```

Listing 3.18: Timezone configuration

Output channels

Output channels were a new concept introduced in the very early versions of rsyslog and were not removed due to backwards compatibility reasons. The idea of an output channel definition is that it provides an umbrella for any type of output that the user might want to use. In theory, an output channel encompasses everything that is needed for an output action. In practice, the current implementation consists of the following:

- filename

- maximum file size
- command to be called when specified maximum file size is reached
- optional template

```
module(load="builtin:omfile")
# outchannel name,file-name,max-size,action-on-max-size
$outchannel log_rotation,'/var/log/outchannel.log', 50000,./rotate-logs.sh'
:omfile:$log_rotation
```

Listing 3.19: Outchannel configuration

One drawback the directive currently has is that it lacks guarantee for backwards compatibility. Nowadays, output channels are barely used in production environments but are still part of the main configuration a user might want to set. With this in mind, there might be no need to implement a dynamic configuration option for such a feature, but technical discussion needs to be opened upstream to determine if there is a need for it.

Chapter 4

Implementing dynamic configuration reload option

The aim of this chapter is to design and implement the ability to be able to dynamically reload the rsyslog configuration file. The chapter starts with a brief motivation on why such a feature makes the program more secure and reliable. The remaining part is dedicated to implementation specific details.

4.1 Motivation

As was pointed out in Chapter 2, other logging solutions used on RHEL distribution, such as *journald* and *auditd* are capable of automatically reloading their core configuration without the need of a full restart. Every once in a while, the main configuration must be updated due to changes in the system. For instance, our workplace might be extended with additional machines that we need to monitor, so there is a need to alter existing configuration files. Right now, the only possible solution to apply newly introduced rules is to shut down the running instance of rsyslog and start it from scratch. Although the amount of time needed for the program to start functioning as a logging daemon is relatively small, it is enough to break the most important requirement of logging solutions – lossless message delivery. Another known issue with not being able to reload configuration files at runtime is that TCP/IP based connections need to be disconnected, even if no changes are made. Such a use case can be seen in figure 4.1. At first, the centralized rsyslog server collects logs from various clients through the TCP/IP protocol via port 514. However, at some point we might need to extend the configuration file to accept inputs via another port as well. In order to apply the changes made to the core configuration, we would need to restart the running instance of rsyslog. With this approach, all the existing connections need to be stopped and later reinitialized. Moreover, the situation gets more complicated when encryption is used and TLS tunnels are opened.

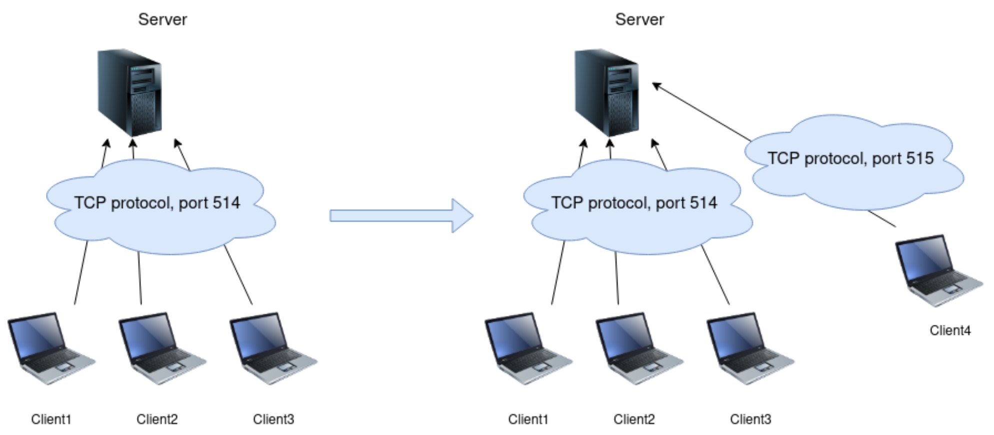


Figure 4.1: Extending the configuration

Whilst the concept of dynamically changing configuration directives seems pretty straightforward, it is not so when it comes to an application with complex configuration, such as rsyslog. Even before trying to think about proposing a way to change internal data during runtime, it is necessary to get acquainted with the code itself. Taking into consideration that chapter 3 introduced configuration directives from system administrator point of view, this chapter will approach them with emphasis on developer perspective.

4.2 Refactoring

At this point, code of the rsyslog project is not ready to take in the dynamic configuration option. This fact has been discovered while getting to know the project and finding out how it works.

Loaded and currently running configuration

By looking at the code, global variables with suspicious names, such as `loadConf` and `runConf` were discovered in multiple hundreds of source files. After requesting additional information regarding this topic from the github maintainer, it's clear that they were originally introduced to facilitate a later move to dynamic config loads. The former was intended for candidate config (config to be loaded), the latter was designed for the currently running config. Both variables are pointers to a complex structure and more importantly point to the exact same location, which arouses suspicion that over the years the use of these pointers has become unclear. One of the early tasks to deliver is to investigate their use, and correct it.

Providing a fix to the problem above does not simply consist of replacing one pointer to another. However, proper investigations need to be carried out to make sure that functions, which contain these variables are called either before or after configuration is loaded into internal variables. Firstly, the rate limiting functionality needed to be adjusted accordingly.

The rsyslog ratelimiter

Rate limitations on logging solutions are in place to prevent logging from using excessive levels of system resources. To log an event, it might need to be written to disk which uses system resources. If there are too many of these events coming in that need to be recorded to disk they can overwhelm a system and cause more important services to respond slowly or fail. For this reason it is generally not recommended to completely disable rate limiting, but to tweak it as required. At the same time we do not want to drop important messages that may be required to generate a critical alert, so a balance needs to be found. This was covered via a pull request[1].

Locating function references

As the fix for clarifying the pointers in rate limiting has been accepted and merged upstream, other parts of the project needed to undergo similar changes[2] as well. The overall approach was to explicitly use either the config that's being loaded or the actually running one, depending on when the function in question is being called. It's important to note that in some cases it was extremely hard to know which one to choose, because of the way rsyslog is implemented. It is common to see lots of different macros defined for function declaration and definition. Such methodology probably improves code readability for experienced developers, but that can not be said for fresh contributors. Such a phenomenon was encountered while looking for the exact location where the `rsconf_t` configuration data structure is constructed. In normal circumstances, references for such elements can be easily found through the *ctags* programming tool.

The *ctags* tool generates an index(tag) file of names found in source and header files of the C programming language [15]. Functions, variables, macros and so on may be indexed. These tags allow definitions to be quickly and easily located by our text editor. However, that was not the case when searching for references regarding the `rsconfConstruct()` function, especially when a token-pasting operator is used.

The ## preprocessing operator performs token pasting. When a macro is expanded, the two tokens on either side of each ## operator are combined into a single token, which then replaces the ## and the two original tokens in the macro expansion. Usually both will be identifiers, or one will be an identifier and the other a preprocessing number. When pasted, they make a longer identifier [9]. For the sake of demonstration, let's analyze what the `rsconfConstruct()` function looks like.

```
BEGINobjConstruct(rsconf)
    cnfSetDefaults(pThis);
ENDobjConstruct(rsconf)
```

Listing 4.1: Constructor for rsconf_t struct

Parameter type of the newly created function will be formed from the value passed to the first parameter of the `BEGINobjConstruct` macro and the `_t` suffix. Definition for the above mentioned macros can be seen in listing 4.2.

[1]https://github.com/rsyslog/rsyslog/pull/4743
[2]https://github.com/rsyslog/rsyslog/pull/4754

```
#define BEGINobjConstruct(obj)                                          \
  static rsRetVal obj##Initialize(obj##_t __attribute__((unused)) *pThis) \
  {                                                                     \
    DEFiRet;

#define ENDobjConstruct(obj)                                            \
    RETiRet;                                                            \
  }                                                                     \
  rsRetVal obj##Construct(obj##_t **ppThis);                            \
  rsRetVal obj##Construct(obj##_t **ppThis)                             \
  {                                                                     \
    DEFiRet;                                                            \
    obj##_t *pThis;                                                     \
    assert(ppThis != NULL);                                             \
                                                                        \
    if((pThis = (obj##_t *)calloc(1, sizeof(obj##_t))) == NULL) {       \
      ABORT_FINALIZE(RS_RET_OUT_OF_MEMORY);                             \
    }                                                                   \
    objConstructSetObjInfo(pThis);                                      \
                                                                        \
    obj##Initialize(pThis);                                             \
  finalize_it:                                                          \
    OBJCONSTRUCT_CHECK_SUCCESS_AND_CLEANUP                              \
    RETiRet;                                                            \
  }
```

Listing 4.2: Macro containing token-pasting operator

After successful expansion, requested data structure will be dynamically allocated. Moreover, additional functions defined between *BEGINobjConstruct* and *ENDobjConstruct* macros will be called. Whilst the final code will be much cleaner and consistent, text editors will suffer from not being able to find all references a function currently has.

Integrating global parameters into the config

The very first step towards supporting dynamic configuration reload was through the *global* directive. In rsyslog, a global directive is a configuration variable with global scope, meaning it affects the behavior of the entire program. Among other things, it is most often used to:

- set the working directory that rsyslog uses for work files, where both temporary and permanent files will be written, e.g. queues saved to disk
- set the calling process's file mode creation mask
- set default netstream driver certificates
- abort when configuration is incomplete
- set environment variables
- set default parameters for queues, actions, rulesets, parsers
- and many more

From implementation point of view, global directives were declared as global variables, meaning they were not part of the `rsconf_t` configuration structure. Although it does not affect current functionality of the project, it would have serious drawbacks if we tried to dynamically reload a configuration. In that case, global variables would be overwritten with new values and there would be no way to compare new values with the old ones. Every single global directive has its own getter and setter, so the task is to locate places where aforementioned functions are called and investigate which configuration should they be working with – currently being loaded, actually running or both depending on the actual use case.

The idea behind the proposed changes[3] was to extend getters with an additional parameter specifying the config in question. The setter was explicitly set to just use the new config, because at this point there was absolutely no reason to modify it sometime later. At first, the initial pull request contained implementation of two global directives for demonstration purpose, as the author of the thesis was not sure if the proposed changes would be accepted by the maintainer. It was important to highlight that suggested modifications would change the interface structure. Thus, more information was requested regarding possible changes in the interface, whether it's feasible to do it or it's a bad approach that should not be considered at all. Fortunately, suggested changes were accepted[4] and any questions were answered in a very short amount of time, so implementation of remaining global directives could begin.

During debugging of the implemented changes, multiple problems arose including:

- broken testing environment[5]
- some global directives are used even before the main configuration structure exists or after it has been destroyed, which is illustrated in figure 4.2

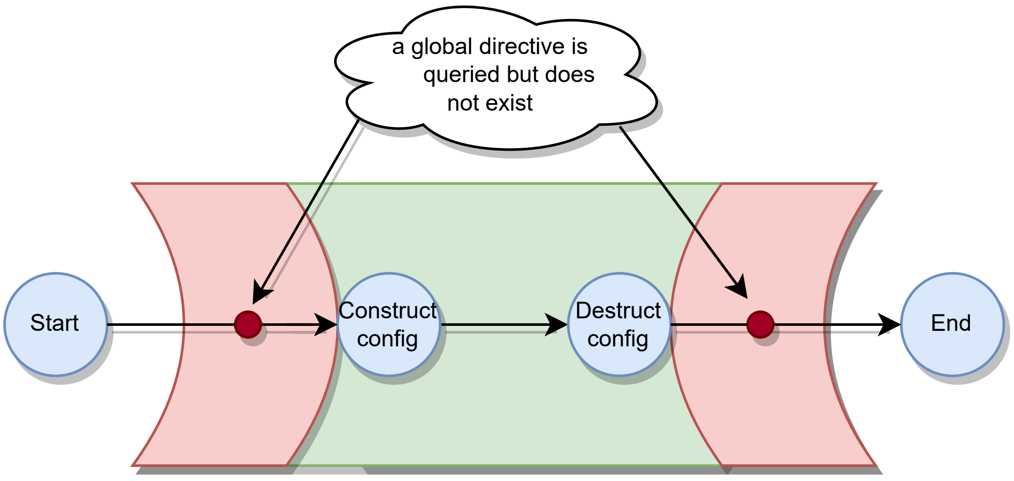


Figure 4.2: Lifespan of rsyslog from configuration structure point of view

It's important to mention that some global parameters can be customized through the *global()* interface in the configuration file, as well as via regular command line arguments. For instance, in order to instruct rsyslog to use only *IPv4* protocol we can pick one of the following approaches:

[3]https://github.com/rsyslog/rsyslog/pull/4760
[4]https://github.com/rsyslog/rsyslog/pull/4760#issuecomment-1001991769
[5]https://github.com/rsyslog/rsyslog/pull/4760#issuecomment-1001991981

- legacy - start rsyslog with *“-4”* command line argument
- recommended - include “global(net.ipprotocol=’ipv4-only’)” inside the main config file

Using the legacy style was still permitted, although it was not recommended by the official documentation because it makes configurations less clean. From code point of view, multiple comments pointed out that it had been marked as deprecated over the years and would not be part of the rsyslog project for all eternity. We can see an example of such a comment in figure 4.3.

```
switch((char)ch) {
case '4':
	fprintf (stderr, "rsyslogd: the -4 command line option will go away "
		"soon.\nPlease use the global(net.ipprotocol=\"ipv4-only\") "
		"configuration parameter instead.\n");
	  glbl.SetDefPFFamily(PF_INET);
	break;
```

Figure 4.3: Obsoleted command line arguments are pointed out in the source code

Therefore, the topic of ‘whether it’s feasible to completely remove it from the source code’ became the subject of a discussion on upstream[6], started by the author of the thesis. After some deep technical discussion, the proposed changes were accepted by the maintainer.

Unfortunately, the pull request introduced a regression[7], where segmentation fault could be reproduced if certain environment variables are not set. The issue[8] was detected by the rsyslog community and resolved[9] via follow-up commits.

Integrating timezones into the config

Moving onto the next task, the *timezone* directive, which was briefly introduced in subsection 3.3, needed to undergo similar changes as well. Unfortunately, the *timezone* was not part of the main configuration structure, but was implemented as a standalone component. In addition to addressing[10] the aforementioned task, several other improvements were proposed, including:

- splitting timezone specific functions and data types into a separate file for better readability
- simplifying conditional statements

[6] https://github.com/rsyslog/rsyslog/pull/4760#issuecomment-1002990805
[7] the term is used when a feature that has worked before stops working
[8] https://github.com/rsyslog/rsyslog/issues/4810
[9] https://github.com/rsyslog/rsyslog/pull/4812
[10] https://github.com/rsyslog/rsyslog/pull/4764

Integrating action specific attributes into the config

As mentioned in 3.3, action directives are responsible for outputting enqueued logs to diverse destinations. Without them, no message leaves the main message queue at all. Rsyslog keeps track of all actions defined in the configuration regardless of their current status. The status of an action indicates whether the directive in question is defined properly(without syntax error), has all required parameters set, and is not suspended. This is extremely useful in cases where the configuration contains syntax errors associated with a particular action, so the program will not abort but continue running without processing the incorrect action. With this in mind, two valuable aspects need to be underlined, including the overall number of actions and the currently available ones. Whilst the latter is part of the main configuration structure, the former is certainly not. The fix was successfully delivered via a separate pull request[11].

Integrating the parser directive into the config

Parsers were thoroughly explained in section 3.3. From an implementation point of view, every single rule set including the default one has a list of possible parsers. If no parser is specified at all, the defaults are put into use. By the time of writing, two default parsers were defined, namely the *RFC 5424* and the *RFC 3164*. Both of them were already introduced in section 3.1. With that said, rsyslog keeps track of two separate lists of parsers, namely:

- the **default** parser list - hardcoded parsers defined in the source code
- the **user defined** parser list - parsers defined in the configuration file by the user

Similarly to the *timezone* directive, the *parser* was also excluded from the configuration structure, even though it plays a crucial role in the way that messages are translated from raw format into the internal representation. Both lists needed to be moved to the `rsconf_t` configuration structure. Unfortunately, the change affected the *parser* interface as well, because functions had to be extended with an additional parameter specifying the configuration they operate on. However, such an approach is only acceptable if the interface number is incremented. This statement was confirmed[12] by the maintainer of the project.

Comparison of configuration directives

In section 3.3, we briefly explained one of rsyslog's most complex and powerful directives, the queue. From a developer perspective, constructing a queue is a multilevel procedure depending on whether the directive in question is used as the main message queue, an action queue or a ruleset queue.

Figure 4.4 illustrates high level steps to construct a queue. The process starts with resource allocation, then continues with assigning default values to each parameter. Next, parameters specified in the configuration are applied either via legacy[13] or rainerscript[14] syntax. Steps listed so far take place during configuration load, more specifically the new configuration is being loaded into internal variables and checked for syntax errors. This is the right place for comparing an already running queue with a newly constructed one.

[11] https://github.com/rsyslog/rsyslog/pull/4800
[12] https://github.com/rsyslog/rsyslog/pull/4760#issuecomment-1001991769
[13] old syntax
[14] new syntax

If they are equal in terms of their content, we grab that particular queue and relocate it to the new config. This is a simple but effective approach towards enabling support for dynamic configuration reload, however it has one drawback.

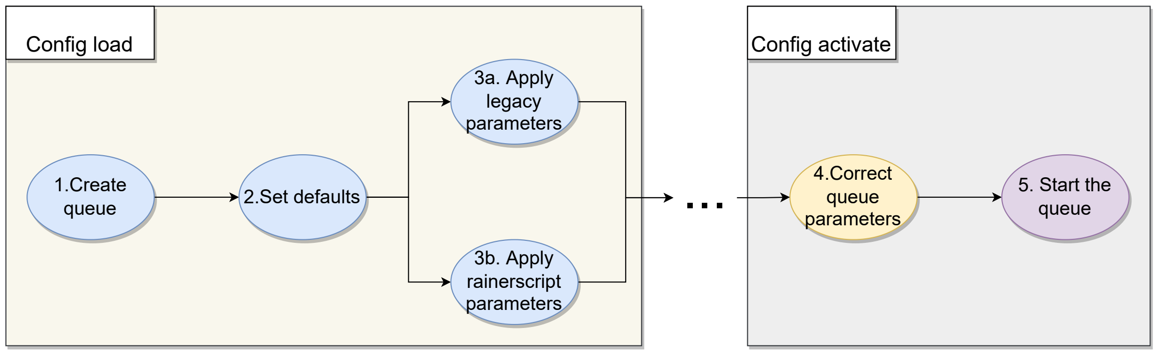

Config load
1.Create queue
2.Set defaults
3a. Apply legacy parameters
3b. Apply rainerscript parameters
...
Config activate
4.Correct queue parameters
5. Start the queue

Figure 4.4: Process of queue construction

Currently, there is an extra step associated with finalizing queue parameters. Users of rsyslog have the possibility to alter the behavior of queues through parameters. However, if some values are set incorrectly, then rsyslog will not abort but instead will try to adjust those specific attributes just to avoid unexpected behavior. One such example could be a queue instructed to have a maximum size of 1 kilobyte, which is considered to be really low.

Unfortunately, correction of invalid parameters takes place after we compared the queues, which is incorrect. This is why there was a need to do some refactoring here as well, so the 4th step of queue construction was appended to the left side of figure 4.4 via a new pull request[15]. One last thing that should be taken into consideration is the process of comparing two configuration directives, e.g. queues. The C programming language does not provide built-in functions for comparing the content of two complex structures. However, that will be the main topic of section 4.3.

4.3 Using x-macros

From implementation point of view, a configuration directive is always defined as a complex C structure that holds information of various types of different and sometimes complicated parameters. In extreme situations, it might possess more than 40 different parameters. It needs to be highlighted due to the process of comparing two configuration directives, for instance, queues. Unfortunately, the C programming language **does not provide a builtin function for comparing the content of two complex structures**. Consequently, we need to explicitly compare every single existing field. In terms of code quality, the solution is neither practical nor clean.

[15]https://github.com/rsyslog/rsyslog/pull/4791

```c
int
queuesEqual(qqueue_t *pOld, qqueue_t *pNew)
{
return (
   (strcmp(pOld->pszSpoolDir, pNew->pszSpoolDir) == 0) &&
   (pOld->iMaxQueueSize == pNew->iMaxQueueSize) &&
   (pOld->iDeqBatchSize == pNew->iDeqBatchSize) &&
   (pOld->iMinDeqBatchSize == pNew->iMinDeqBatchSize) &&
   (pOld->toMinDeqBatchSize == pNew->toMinDeqBatchSize) &&
   (pOld->sizeOnDiskMax == pNew->sizeOnDiskMax) &&
   (pOld->iHighWtrMrk == pNew->iHighWtrMrk) &&
   (pOld->iLowWtrMrk == pNew->iLowWtrMrk) &&
   (pOld->iFullDlyMrk == pNew->iFullDlyMrk) &&
   /* ~30 parameters skipped for better readability */
   (pOld->iLightDlyMrk == pNew->iLightDlyMrk))
}
```

Listing 4.3: Queue comparison function

```c
typedef struct
qqueue_t
{
    char *pszSpoolDir;
    int iMaxQueueSize;
    int iDeqBatchSize;
    int iMinDeqBatchSize;
    int toMinDeqBatchSize
    ;
    int sizeOnDiskMax;
    int iHighWtrMrk;
    int iLowWtrMrk;
    int iFullDlyMrk;
    /* ... */
    int iLightDlyMrk;
} qqueue_t;
```

Listing 4.4: Queue struct

If we were to introduce a new parameter inside the configuration directive(listing 4.4), which is considered to be a regular task, then we would need to update the appropriate comparison(listing 4.3) function as well. If by accident a developer forgets to update aforementioned function, potential bugs could be introduced into the project.

Fortunately, there is a way to avoid such problems by introducing *x-macros* in the project. X-macros are a powerful coding technique that makes extensive use of the C programming language pre-processor. This technique has the capability to eliminate several classes of common bugs [30]. In other words, it can create a self-maintaining and inter-dependent piece of code. When the change of one part of a program leads to a change in another part, then the code is said to be inter-dependent. An x-macro application consists of two parts:

- the definition of the list's elements - listing 4.5
- the expansion of the list to generate fragments of statements or declarations - listing 4.6

```c
#define QUEUE_PARAMETERS      \
    X_STRING(pszSpoolDir)     \
    X_INT(iMaxQueueSize)      \
    X_INT(iDeqBatchSize)      \
    X_INT(iMinDeqBatchSize)   \
    X_INT(toMinDeqBatchSize)  \
    X_INT(sizeOnDiskMax)      \
    X_INT(iHighWtrMrk)        \
    X_INT(iLowWtrMrk)         \
    X_INT(iFullDlyMrk)        \
    X_INT(iLightDlyMrk)
```

Listing 4.5: The definition of the list's elements

```c
typedef struct
qqueue_t {
    #define X_INT(NAME) int NAME;
    #define X_STRING(NAME) char *NAME;
    QUEUE_PARAMETERS
    #undef X_INT
    #undef X_STRING
} qqueue_t;

static int queuesEqual(qqueue_t *pOld, qqueue_t *pNew)
{
    #define X_INT(NAME) (pOld->NAME == pNew->NAME) &&
    #define X_STRING(NAME) (strcmp(pOld->NAME, pNew->NAME) == 0) &&
        return QUEUE_PARAMETERS 1;
    #undef X_INT
    #undef X_STRING
}
```

Listing 4.6: The expansion of the list's elements

The list is defined by a macro named `QUEUE_PARAMETERS`, which generates no code by itself, but merely consists of a sequence of invocations of a macro (usually having prefix "X_") with the elements' data. Each expansion of `QUEUE_PARAMETERS` is preceded by a definition of `X_INT` and `X_STRING` with the syntax for a list element. The invocation of `QUEUE_PARAMETERS` expands `X_*` for each element in the list.

The outcome of the macros defined above will have the same result as listings 4.3 and 4.4 extended with one key advantage – inter-dependent piece of code. If by chance the `qqueue_t` data structure is extended with additional attribute, the comparison function will be automatically adjusted as well due to macro expansion. Whilst x-macros help to create self-maintaining and inter-dependent code, the output might become less readable and complex to understand, especially for newcomer developers. With this in mind, suggested changes will be up to the community and primary maintainer of rsyslog to decide whether it is feasible to introduce it in the project.

4.4 Invocation of dynamic configuration reload

Dynamic configuration is the ability to change the behavior and functionality of a running system without requiring application restarts. An ideal dynamic configuration system enables administrators to view and update configurations easily, and delivers configuration updates to the program efficiently and reliably. It should empower them with tools to reduce the risk associated with changing existing systems. The section will introduce and evaluate different approaches on how the invocation of configuration reload could look like.

Command line interface The first way to achieve desired results it to utilize the command line interface. Rsyslog is a daemon program that runs as a background process, rather than being under the direct control of an interactive user. We could introduce a new binary executable file, which would have only one purpose. Once executed, it shall instruct the running instance of rsyslog to perform dynamic configuration reload. This could be accomplished by setting a flag to true, which is periodically monitored by rsyslog. Whilst the solution is realizable, it definitely has some drawbacks:

- problematic when multiple instances of rsyslog are running at the same time
- communication interface between proposed binary and running rsyslog must be introduced

Signal handlers In the C Standard Library, signal processing defines how a program handles various signals while it executes [7]. A signal can report some exceptional behavior within the program (such as division by zero), or a signal can report some asynchronous event outside the program(such as dynamic configuration reload). It's important to mention that rsyslog is already capable of handling numerous signals, such as *SIGINT* for terminating the program, *SIGUSR1* for toggling debugging mode and most importantly *SIGHUP* for closing all open files. Originally, the *SIGHUP* signal was designed to notify the process of a serial line drop(hangup). In modern systems, many daemons will reload their configuration files and reopen their log files instead of exiting when receiving this signal. Such behavior was already pointed out when system auditing and systemd journal were explained in sections 2.2 and 2.3.

Regular file monitoring File system event monitoring is essential for many types of programs ranging from file managers to security tools. The linux kernel supports *inotify*, which allows a monitoring program to open a single file descriptor and watch one or more files or directories for a specified set of events, such as open, close, move, delete, create or change attributes [29]. It can be used to automatically update directory views, **reload configuration files**, log changes, backup, synchronize, and upload. The idea is to monitor all the configuration files of rsyslog, and dynamically reload them whenever at least one of the following criteria is met:

- a watched file is modified
- a new file is created within a watched directory
- a watched file is deleted
- metadata, e.g. timestamps or permissions are changed on a watched file/directory

However, we have to take into consideration that a regular user might have many small configuration files. This often leads to the need to edit a couple of files to complete a task. In that case, monitoring the config file set and reloading on change may lead to an incomplete or non-working config. Moreover, if by chance all the monitored files are changed, it will force rsyslog to reload all of them after each file change.

After some discussion with the primary maintainer, the signal handling approach was chosen from the abovementioned candidates. It was implemented as part of a bigger pull request[16]. Unfortunately, the *SIGHUP* signal was already taken and used for different purposes. In order to preserve backwards compatibility, there was a need to introduce a new global variable *hup.reload.config*. By default, it is turned off and the old behavior applies, but it can be enabled in a pretty straightforward way as shown in listing 4.7.

[16] https://github.com/rsyslog/rsyslog/pull/4783

```
global(
    hup.reload.config="on"
)
```

Listing 4.7: Usage of the new global variable

4.5 Reloading configuration directives

The very first step towards reloading rsyslog directives is making sure that already implemented functions used during loading and activation of configuration are reusable. For instance, every single module has attributes indicating whether the module in question has already been processed, so no duplicate is allowed. This is why we need to reset such variables.

From implementation point of view, modules are objects that provide member functions. In rsyslog terminology, objects are actually C structs, while member functions can be though of as pointers to functions. Each module has a unified interface with pointers to disjoint functions. Among many other things, modules must define:

- constructors and destructors
- a way to parse module specific parameters
- a way of creating new instances of the module in question
- reaction to signals
- additional functions depending on the type of module

The number of currently available rsyslog modules is huge and every single one of them has a primary maintainer. The aim of the thesis is not to implement support for all available modules; rather provide it for the most commonly used ones and inspire other developers to implement it for the module they maintain.

The *module* interface needed to be extended with an additional member function, `reloadCnf()`. This will instruct a module to dynamically reload its parameters, make a delta between currently running and newly loaded config, adjust running instances as per changes and many other things. Obviously, this needs to be implemented for modules that want to benefit from the suggested feature.

The imtcp input module

The imtcp input module was chosen as the first to undergo the suggested changes. In general, it is considered to be the most well known one in the configuration as it provides the ability to receive syslog messages via TCP. Moreover, it is extremely fast as all instances of the imtcp module run in a separate thread. Whilst this is one of the key advantages of rsyslog, it certainly made the implementation[17] and debugging phase more difficult. For the sake of demonstration, listing 4.8 provides two instances of the imtcp module.

[17]https://github.com/rsyslog/rsyslog/pull/4783

```
module(load="imtcp"
    StreamDriver.Name="ossl" NotifyOnConnectionClose="off" # module parameters
)
input(type="imtcp"
    Port="514" RateLimit.Burst="10000" RateLimit.Interval="5" # instance parameters
)
input(type="imtcp"
    Port="515" RateLimit.Burst="10000" RateLimit.Interval="5" # instance parameters
)
```

Listing 4.8: Sample imtcp configuration with 2 instances

In terms of implementing the `reloadCnf()` function, all input modules will proceed according to algorithm 4.5 with slight differences. The pseudo-code consists of three phases, where each of them serves one purpose. In phase 1, we search for changes made to the module configuration structure. If at least one of the module parameters changed, we can not use old instances due to parameter inheritance from modules towards instances, thus there is a need to reload all of them. Phase 2 plays a crucial role in the algorithm and delivers one of the key tasks of the thesis. If an already running instance(from old config) is also present[18] in the config being loaded, that particular instance will be relocated from the old config to the new one. In case of the imtcp module, it means that an unchanged server instance listening to a certain port will continue collecting logs without a break. Finally, phase 3 is responsible for the removal of already running instances, which are not part of the newly loaded config.

[18]was not modified/removed

Algorithm 1 Module config reload pseudo-code

```
procedure RELOADCNF()
    ▷ Phase 1
    if not modulesEqual(loadModConf, runModConf) then
        for instance in runModConf.instances() do
            destruct(instance)
        end for
        goto end
    end if

    ▷ Phase 2
    for loadInstance in loadModConf.instances() do
        for runInstance in runModConf.instances() do
            if instancesEqual(loadInstance, runInstance) then
                destruct(loadInstance)
                loadModConf.addInstance(runInstance)
                break
            end if
        end for
    end for

    ▷ Phase 3
    for runInstance in runModConf.instances() do
        found ← 0
        for loadInstance in loadModConf.instances() do
            if instancesEqual(loadInstance, runInstance) then
                found ← 1
                break
            end if
        end for
        if not found then
            destruct(runInstance)
        end if
    end for
end:
    return void
end procedure
```

One thing that is certainly different for all modules is the way of destructing their instances. Proper deinitialization of removed instances during configuration reload required deep investigation on how the tcp server was implemented. Unfortunately, the imtcp module hadn't had implemented functionality for stopping just a single imtcp instance. It was either all or none of them. This was addressed[19] by extending the tcp server interface with the `SetTerminateInput()` function. Until this point, dynamically allocated resources were freed only at program termination. However, this had to be completely changed as some instances might be instructed to be deleted during configuration reload. Conse-

[19]https://github.com/rsyslog/rsyslog/commit/ce9080480ec0a6b0cae4a33361ce4f845b134df2

quently, the process of deallocating instance resources had to be invoked whenever a thread was killed. The inspiration[20] came from other source files of rsyslog, where the desired functionality was achieved through thread-cancellation clean-up handlers, more precisely `pthread_cleanup_push()` and `pthread_cleanup_pop()`. A clean-up handler is a function that is automatically executed in the following circumstances [13]:

- when a thread is canceled
- when a thread terminates by calling `pthread_exit()`
- when a thread calls `pthread_cleanup_pop()` with a nonzero execute argument

The ruleset and output modules

Ruleset is certainly the most complex configuration directive when it comes to the amount of possibilities it offers. Comparing rulesets of one configuration with another requires deep knowledge on the fields it may contain. As basis, the 4.5 algorithm was utilized and extended with additional functionality. The comparison functions were implemented[21] recursively due to the fact that a ruleset might even contain other rulesets as well. From the perspective of a developer, rulesets might contain the following elements:

- variable creation and removal
- output actions
- direct or indirect call to other rulesets
- conditions
- foreach loops
- prifilts - traditional filter strings
- propfilts - property filters

All the aforementioned elements needed to have compare functions implemented. The hardest challenge was caused by the condition element as it encapsulates nearly everything a programming language offers – function calls, arithmetic operators, logical operators, relational operators, variables and many more.

Next, each output module that wants to benefit from dynamic configuration reload feature must implement both `instancesEqual()` and `modulesEqual()` member functions via the *modules* interface. The former is used for comparing the content of two **action specific instance parameters**, and the latter serves for checking **action specific module parameters**. Besides that, we still need to take into consideration **general action parameters** and **action specific queue parameters** that were explained in section 3.3. Fortunately, these parameters are generally available and apply to all actions, meaning that specific output modules do not have to implement them separately. For the sake of demonstration, listing 4.9 clarifies the difference between aforementioned parameters.

[20] https://github.com/rsyslog/rsyslog/pull/4783/commits/d0bf0f93c1a575c21f05baa595d1c65ad6a889ad
[21] https://github.com/Cropi/rsyslog/commit/874124622262e82cd360cb2f841496e55d4a5625

```
# action specific omfwd module parameters
module(load="omfwd" template="RSYSLOG_TraditionalForwardFormat")

action(
    # action specific omfwd instance parameters
    type="omfwd" target="10.0.138.187" port="514"

    # action specific queue parameters
    queue.type="Disk" queue.maxfilesize="1m" queue.filename="queuename"

    # general action parameters
    action.errorfile="/var/log/error.log" action.repeatedMsgContainsOriginalMsg="off"
)
```

Listing 4.9: Action parameters

Parser modules

As we've seen in 3.3, parsers are usually chained together to provide more flexibility. In rsyslog, chains are implemented as linked lists, which makes iteration over parsers smooth. Similarly to output modules, parsers need to implement the `instancesEqual()` member function. However, the `modulesEqual()` can be omitted as parsers don't have such a thing at disposal. Proposed changes[22] are based on the code written in previous pull requests.

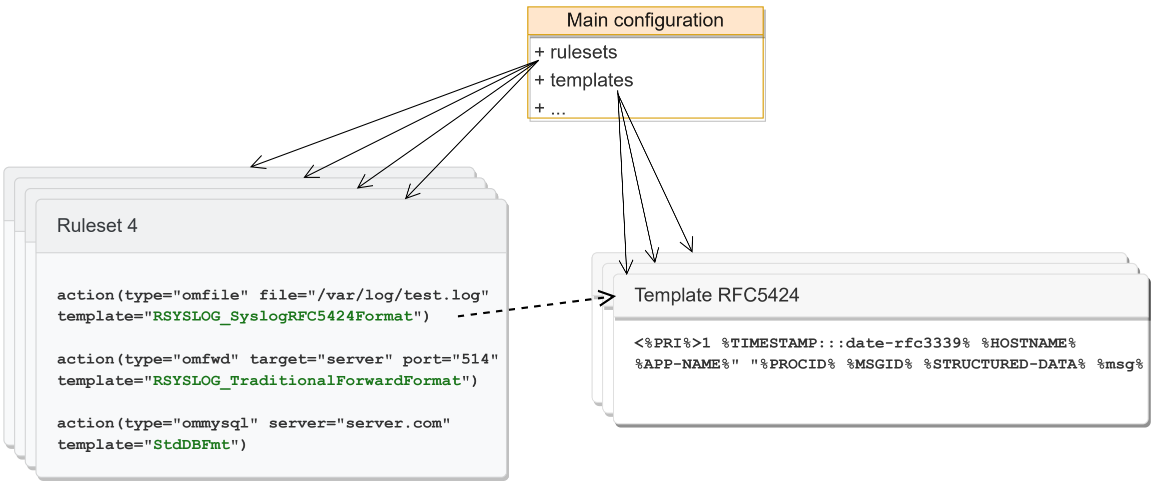
There are lots of different parser modules available in the official rsyslog repository. The commit above does not include dynamic configuration reload option for all of them. It needs to be explicitly implemented and most importantly requires basic knowledge of the parser in question. Above all, one of the default parsers in rsyslog, rfc3164 is part of the delivered changes as it was thoroughly explained in section 3.1.

Templates

As it was mentioned in 3.3, templates are a key feature of rsyslog that allow to specify any kind of output format a user might want. By default, rsyslog has many built-in templates that are not visible to the user base. These are used as a fallback mechanism whenever a template is required but is not specified in the configuration. From implementation point of view, templates are defined as standalone objects that must be bounded to existing actions.

In order to be able to dynamically reload the template directive, we need to analyze how templates are bound to specific actions. The intention of figure 4.5 is to illustrate the high level implementation of the connection between templates and actions. In order to maintain readability, many other fields of the configuration not relevant to templates are omitted. During configuration load, templates are dynamically created and put into a singe linked list. After the process of template creation is finished, rulesets need to be created, which may probably contain actions. As soon as an action is created, a user defined template shall be associated with it. Otherwise, one of the default templates is put into use depending on the output module in question. In the C programming language, associations are carried out via pointers.

[22]https://github.com/Cropi/rsyslog/commit/3ec1cb9c907718cbe3b90818385396e98b8a12dc



```
action(type="omfile" file="/var/log/test.log"
template="RSYSLOG_SyslogRFC5424Format")

action(type="omfwd" target="server" port="514"
template="RSYSLOG_TraditionalForwardFormat")

action(type="ommysql" server="server.com"
template="StdDBFmt")
```

```
<%PRI%>1 %TIMESTAMP:::date-rfc3339% %HOSTNAME%
%APP-NAME%" "%PROCID% %MSGID% %STRUCTURED-DATA% %msg%
```

Figure 4.5: Connection of actions and templates

Dynamic configuration reload support for the template directive was successfully implemented[23], which consisted of the following steps:

1. create a comparison function capable of obtaining information whether two template instances are equal in content
2. reload the *templates* linked list, while preserving unmodified ones
3. actualize template pointers of actions inside *rulesets*

Globals and timezones

Support for the *globals*[24] and *timezones*[25] directives were partially delivered and deeply described in the refactoring section of this chapter. However, some global parameters required special care, especially those that are not allowed to be reconfigured during dynamic configuration reload. For instance, the *privdrop.user.name* specifies name of the user rsyslog should run under after startup. Among other jobs, it sets the effective user ID of the calling process. This allows a set-user-ID (other than root) program to drop all of its user privileges, do some unprivileged work, and then reengage the original effective user ID in a secure manner [14]. Consequently, modification of security related global parameters is prohibited[26].

Dynamic Stats, Percentile Stats and Lookup Tables

Section 3.3 briefly explained why gathering statistics of our systems is useful. In rsyslog, statistics provide us information regarding service health, performance and bottlenecks of the system we are collecting logs from. These can be later analyzed and help us to narrow down performance issues. Taking into consideration that a typical rsyslog user does not want to reset already gathered statistics when a configuration reload is invoked,

[23]https://github.com/Cropi/rsyslog/commit/c801309670501aa74b40d731d0248c93e3dbf092
[24]https://github.com/rsyslog/rsyslog/pull/4760
[25]https://github.com/rsyslog/rsyslog/pull/4764
[26]https://github.com/Cropi/rsyslog/commit/e370eddd4648954a9fb96ff876707dadc056c5de

there is clearly a need to introduce the dynamic configuration reload option for aforementioned directives. Although, it will probably be customizable in a future version of rsyslog. The initial version for dynamic stats[27], percentile stats[28] and lookup tables[29] has already been implemented and can be found in the forked repository of rsyslog. All the mentioned directives needed to undergo similar changes as the *template* directive.

4.6 Bugfixing

While developing the new feature of rsyslog and experimenting with it, multiple bugs were discovered. These specific defects were already part of the project even before the author of the thesis started to contribute to the project. Each of them was reported and later fixed on upstream.

Use of dangerous functions

During reviewing the code base, some minor issues have been identified including use of problematic functions. The *rsyslog-omsnmp* module was using the `inet_addr()` function to convert the Internet host address from IPv4 numbers-and-dots notation into binary data in network byte order, which is then returned [8]. However, if the input is invalid, `INADDR_NONE` (usually -1) is returned. Use of this function was problematic because -1 is interpreted as a valid address (255.255.255.255). The final decision[30] was to avoid its use in favor of `inet_aton()`, which provides a cleaner way to indicate error return. More precisely, the `inet_aton()` stores the address in the structure provided and returns 1 if the input is valid, so error code is not part of the converted binary data.

Memory leaks in outchannels

A memory leak reduces the performance of the computer by reducing the amount of available memory. Eventually, if too much of the available memory may become allocated, some applications may slow down or fail. Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate. Due to these facts, proper memory management is required inside rsyslog. During creation of code snippets and documenting configuration directives in subsection 3.3, memory leaks were discovered inside the *output channel* directive. The bug was reproducible on every single operating system using the `valgrind` tool. The root cause lied in not calling the *outchannel* destructor after rsyslog is terminated. This has been reported and fixed via a separate pull request[31].

Client side certificate checking

While implementing the *imptcp* module specific changes described in 4.5, a minor bug has been discovered in the architecture of the tcp server. When the server accepted a new connection, some parameter initializations were left out. Due to this bug, client certificates were not checked on the server side. This played a crucial role in the security of transporting

[27] https://github.com/Cropi/rsyslog/commit/ef540d917308baca77665c79c15fcf3ededc0bdc
[28] https://github.com/Cropi/rsyslog/commit/61b9ef75095bbfa3e417f041b46cbd31492d128b
[29] https://github.com/Cropi/rsyslog/commit/7fd161eb1cd9664374893f4f9b237be549c980ee
[30] https://github.com/rsyslog/rsyslog/pull/4729
[31] https://github.com/rsyslog/rsyslog/pull/4745

logs from one host to another. Finally, it was addressed by a minor but still important pull request[32].

Compilation error

In the recent released version of rsyslog, an internal compilation error started to occur not due to a logic error, but rather due to a syntax error in the code itself. The issue[33] was reproducible if the program was compiled with the *–enable-omfile-hardened* configuration option. The appropriate structure definitions were missing due to not having a particular header file included. In order to address this, a new pull request[34] had been created.

Proper release of file descriptor resources

The *imfile* module provides the ability to convert any standard text file into a syslog message. The file is read line-by-line and any line read is passed to rsyslog's rule engine. The module is very rich in terms of parameters. One of the optional parameters is called the *freshStartTail*, which is used to tell rsyslog to seek to the end of input files (discard old logs) at its first start, and process only new log messages. However, some resources were not released when the aforementioned option was defined in the configuration. Consequently, the CWE-772 was present in the code. The term CWE-772 refers to a weakness of a program, where the software does not release a resource after its effective lifetime has ended, i.e., after the resource is no longer needed [4]. Proper memory management is exceptionally important for programs which by definition never terminate, e.g. daemons. The issue was deeply investigated and resolved by the author of the thesis via a pull reqeuest[35].

Defects reported by coverity scan

Coverity identifies critical software quality defects and security vulnerabilities in code as it's written. After executing coverity scan using the Red Hat Coverity Scan Scheduler[36], issues were found that had to be investigated. Rsyslog and its related dependencies were tested, namely the liblognorm and the librelp libraries. The former provides the capability to normalize log messages inside rsyslog, and the latter is used to receive syslog messages via the reliable *RELP* protocol. Whilst the coverity scan tool can detect potential bugs not visible to the naked human eye, it is quite sensitive for false positives. Approximately 30% of the reported problems were marked as real issues. Finally, identified defects were solved for both librelp[37] and liblognorm[38] components.

[32] https://github.com/rsyslog/rsyslog/pull/4671
[33] https://github.com/rsyslog/rsyslog/issues/4827
[34] https://github.com/rsyslog/rsyslog/pull/4832
[35] https://github.com/rsyslog/rsyslog/pull/4640
[36] https://cov01.lab.eng.brq.redhat.com/covscanhub/
[37] https://github.com/rsyslog/librelp/pull/238
[38] https://github.com/rsyslog/liblognorm/pull/357

Chapter 5

Testing and evaluation

In order to thoroughly evaluate the implemented feature, there are certain requirements which need to be done. The first one is to ensure that no regression is introduced at all. Obviously, there is no guarantee that a new version of rsyslog will not contain any bugs but with proper tools and procedures the amount can be reduced.

Regression testing can be thought of as a type of software testing to confirm that a recent program or code change has not adversely affected existing features. Generally, it is nothing but a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine. Moreover, it ensures that the old code still works once the latest code changes are done. In case of rsyslog, build creation and testing is done through Github Actions. The mentioned platform played a crucial role in the testing phrase, because huge part of the implementation consisted of rewriting existing code.

5.1 Github actions

Github Actions is a continuous integration and continuous delivery (CI/CD) platform that allows us to automate our build, test, and deployment pipeline [10]. It is fully configurable for Github to react to certain events automatically according to our preferences. With this in mind, we can create workflows that build and test every pull request to our repository, or deploy merged pull requests to production. One of the key advantages of using such a tool is that it is fully integrated into Github and therefore no external site is required at all. Github Actions is made up of several tightly coupled components. Figure 5.1 visualizes the workflow of the Fedora Linux 34 environment that is triggered by a specific event.

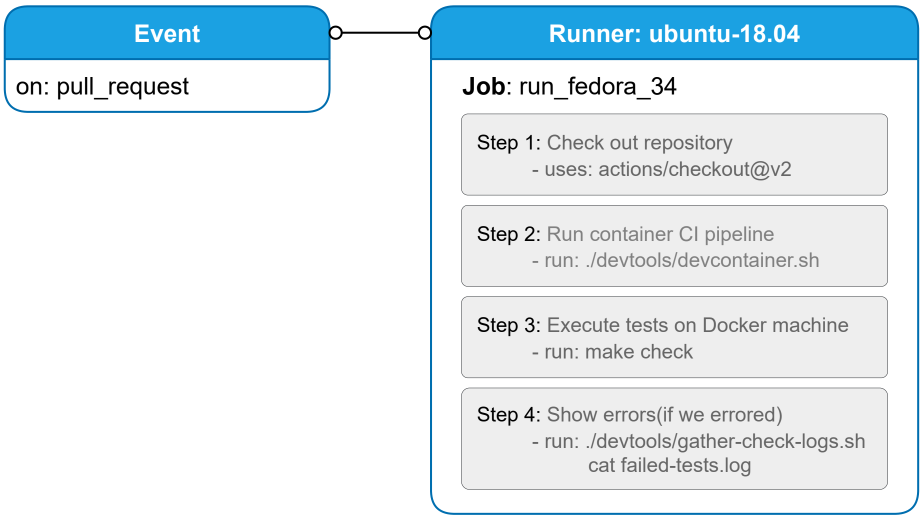

Event
on: pull_request
Runner: ubuntu-18.04
Job: run_fedora_34
Step 1: Check out repository
- uses: actions/checkout@v2
Step 2: Run container CI pipeline
- run: ./devtools/devcontainer.sh
Step 3: Execute tests on Docker machine
- run: make check
Step 4: Show errors(if we errored)
- run: ./devtools/gather-check-logs.sh
cat failed-tests.log

Figure 5.1: Visualizing the Fedora34 workflow file

Workflows A workflow can be thought of as a configurable automated process that will run one or more jobs. We can define as many workflows as it is needed for the project. Each workflow is defined by a YAML file located in our repository and will run when triggered by an event. The rsyslog community puts a great emphasis on to be able to use the program on different operating systems with various versions of compilers. Each operating system has a custom workflow defined for catching build errors.

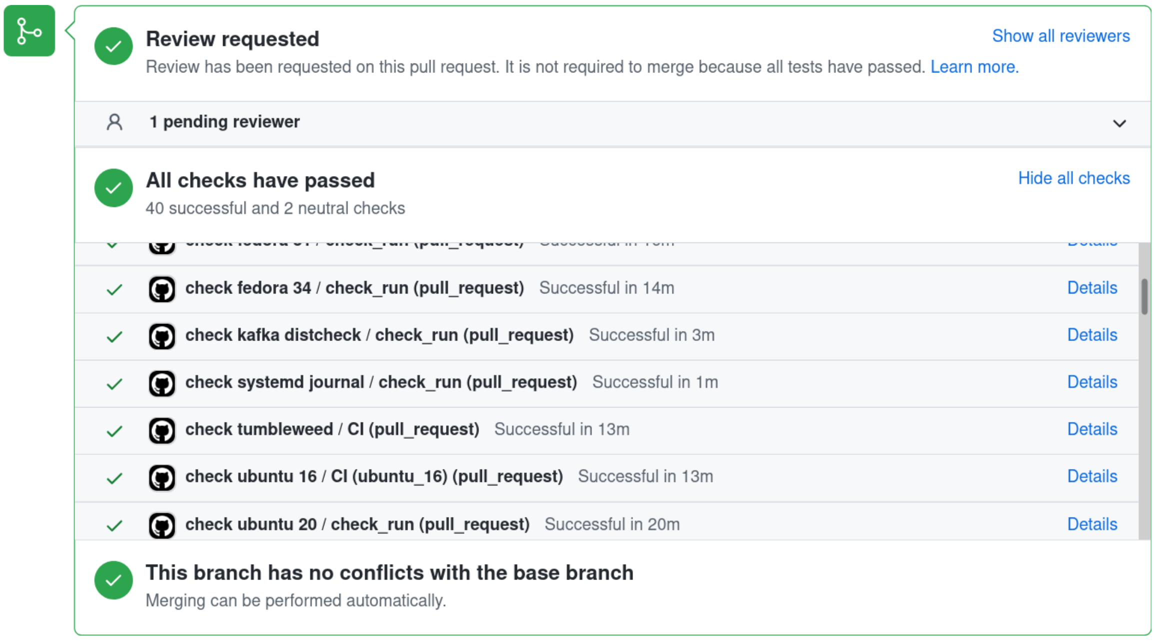
Events These are specific activities that trigger the execution of a workflow. Such an activity can originate from Github when someone creates a pull request, opens an issue or pushes a commit to a repository. In rsyslog, it can be triggered by either pushing to the master branch or creating a pull request.

Jobs A job consists of a set of steps inside a workflow that execute on the same runner. A step can be though of as a shell script or another action that will run.

Actions Actions are the smallest portable building block of a workflow and can be combined as steps to create a job.

Runners It is a machine with the GitHub Actions application already installed, whose function is to wait for the work to be available and then be able to execute the actions and report the progress and the results. By default, Github provides Ubuntu Linux, Microsoft Windows, and macOS but rsyslog uses only the first one. Within Ubuntu Linux, a docker container is used to build and test rsyslog on many other operating systems.

Each pull request is associated with a list of checks that are queued and later executed once the PR is created. Such scenario can be observed in figure 5.2. If by chance a check fails, meaning a potential regression is about to be introduced then merging is automatically blocked until the bug is not fixed.


Review requested Show all reviewers
Review has been requested on this pull request. It is not required to merge because all tests have passed. Learn more.

1 pending reviewer

All checks have passed Hide all checks
40 successful and 2 neutral checks

✓ **check fedora 34 / check_run (pull_request)**	Successful in 14m	Details
✓ **check kafka distcheck / check_run (pull_request)**	Successful in 3m	Details
✓ **check systemd journal / check_run (pull_request)**	Successful in 1m	Details
✓ **check tumbleweed / CI (pull_request)**	Successful in 13m	Details
✓ **check ubuntu 16 / CI (ubuntu_16) (pull_request)**	Successful in 13m	Details
✓ **check ubuntu 20 / check_run (pull_request)**	Successful in 20m	Details

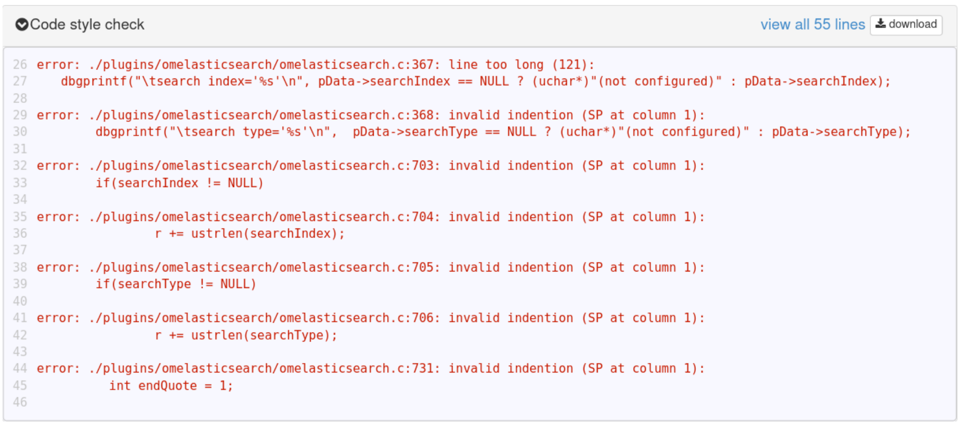
This branch has no conflicts with the base branch
Merging can be performed automatically.

Figure 5.2: Executed workflows on PR creation

Testing the final implementation did not only consist of comparing observed results with expected output. Many tools were used to adhere to compliance standards, including the *rsyslog code style check*, the *LGTM* analysis tool, the *Codefactor* automated code review, *PR structure validation* and the *clang static analyzer*.

Code style check

Code style can be boiled down to anything that is a stylistic choice in the code that has no impact on the behavior of the code. It is the key to code consistency. Once a code style is decided with a team everyone should follow it. This is not about preference but about writing clean and consistent code. With this in mind, rsyslog has its own code style rules defined in a configuration file that is checked against each pull request. Usually, the most common use case is invalid indention, for instance using spaces instead of tabs and vice versa. An example of how such an error is reported can be seen in figure 5.3.


Code style check view all 55 lines download

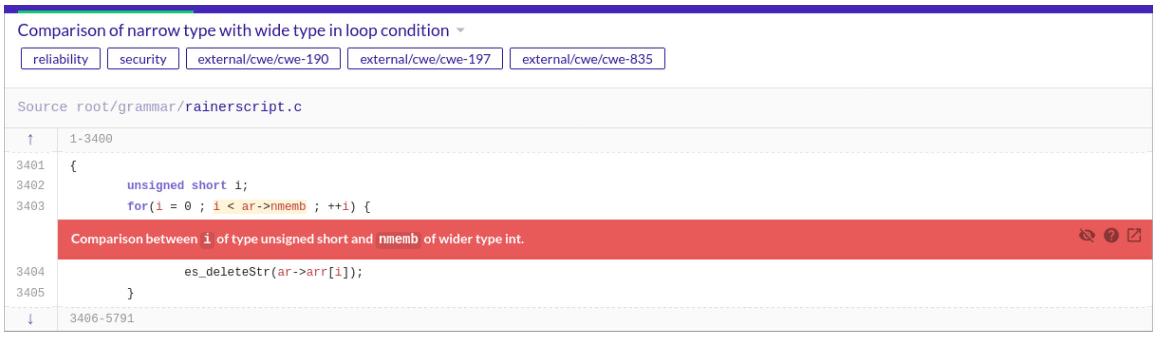
```
error: ./plugins/omelasticsearch/omelasticsearch.c:367: line too long (121):
   dbgprintf("\tsearch index='%s'\n", pData->searchIndex == NULL ? (uchar*)"(not configured)" : pData->searchIndex);

error: ./plugins/omelasticsearch/omelasticsearch.c:368: invalid indention (SP at column 1):
        dbgprintf("\tsearch type='%s'\n",  pData->searchType == NULL ? (uchar*)"(not configured)" : pData->searchType);

error: ./plugins/omelasticsearch/omelasticsearch.c:703: invalid indention (SP at column 1):
        if(searchIndex != NULL)

error: ./plugins/omelasticsearch/omelasticsearch.c:704: invalid indention (SP at column 1):
                r += ustrlen(searchIndex);

error: ./plugins/omelasticsearch/omelasticsearch.c:705: invalid indention (SP at column 1):
        if(searchType != NULL)

error: ./plugins/omelasticsearch/omelasticsearch.c:706: invalid indention (SP at column 1):
                r += ustrlen(searchType);

error: ./plugins/omelasticsearch/omelasticsearch.c:731: invalid indention (SP at column 1):
          int endQuote = 1;
```

Figure 5.3: Example of code style check fail

LGTM analysis tool

LGTM is a continuous security analysis tool for finding zero-days[1] and preventing critical vulnerabilities. It helps to prevent bugs from ever appearing in the project by automatically catching them in the review process before they get merged. Besides that, it is capable of analyzing the entire history of a project, so we can see how the alerts have changed over time, and which specific events or commits had the biggest impact on our code quality. The tool is reliable and barely shows any false positives.

According to the rsyslog history analyzed by LGTM, the most common vulnerability is considered to be the *comparison of narrow type with wide type in loop condition.* For instance, comparison between unsigned short and integer.


Comparison of narrow type with wide type in loop condition

reliability | security | external/cwe/cwe-190 | external/cwe/cwe-197 | external/cwe/cwe-835

Source root/grammar/rainerscript.c

```
1-3400
{
        unsigned short i;
        for(i = 0 ; i < ar->nmemb ; ++i) {
```

Comparison between i of type unsigned short and nmemb of wider type int.

```
                es_deleteStr(ar->arr[i]);
        }
3406-5791
```

Figure 5.4: Example of a vulnerability detected by LGTM

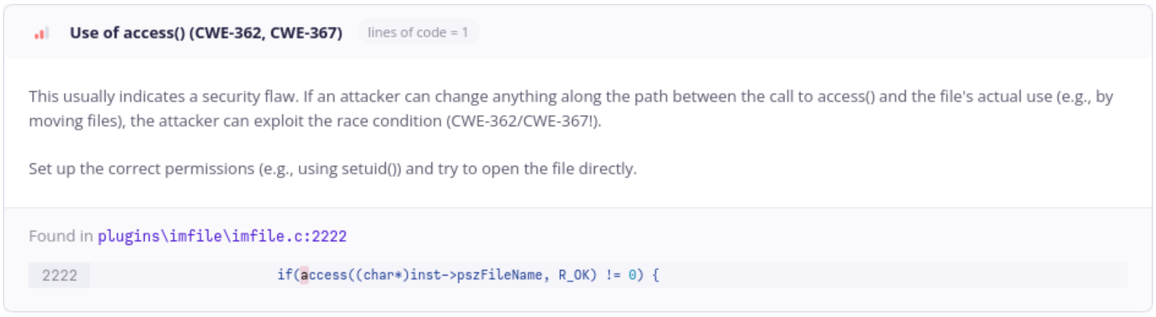
Codefactor

Codefactor provides real-time actionable feedback about potential quality issues as soon as they occur. Identifies hot-spots the most impactful list for refactoring. With the help of this tool, we can get a glance of code quality for the whole project, recent commits and the most problematic files.

An awesome feature of the tool is that it creates a list of chunks of rsyslog code that might be victims of some Common Weakness Enumerations(CWE). CWE is a community-developed list of common software and hardware weakness types that have security ramifications. Weaknesses can be thought of as flaws, bugs, faults or other errors in software or hardware implementation that if left unaddressed could result in systems being vulnerable to attack [4].

The tool is able to identify problematic chunks of code, detect how many lines are affected, briefly describe what the CWE is about and how an attacker could make use of the bug in question. Figure 5.5 illustrates how an attacker could exploit a race condition inside rsyslog. One of the drawbacks the tools currently has that it may contain false positives, so the output must be manually checked by the developer.

[1]a software vulnerability unknown to those who should be interested in its mitigation


Use of access() (CWE-362, CWE-367) lines of code = 1

This usually indicates a security flaw. If an attacker can change anything along the path between the call to access() and the file's actual use (e.g., by moving files), the attacker can exploit the race condition (CWE-362/CWE-367!).

Set up the correct permissions (e.g., using setuid()) and try to open the file directly.

Found in plugins\imfile\imfile.c:2222

```
2222                    if(access((char*)inst->pszFileName, R_OK) != 0) {
```

Figure 5.5: Example of a potential CWE detected by Codefactor

PR structure validation

The goal of this check is to ensure that git history will remain clean and straightforward. Developers should not combine multiple features or bug fixes into a single pull request. Generally speaking, this is needed to make the review easier, thus the maintainer can merge it more quickly. The commit message should describe the implemented changes as briefly as possible. Putting all the information into a Github PR instead of the commit message is considered to be a bad approach. Whilst Github may go away in the future, the history of git remains eternal and is readily available when a developer needs it[2].

5.2 Custom tests

The entire test bench is built upon the *imdiag* input module. With the help of this module, we are capable of querying rsyslog regarding internal data. This approach allows us to check during runtime whether the main message queue contains any data elements or it's empty. This comes in handy especially, when some test needs to compare observed output with expected data. Before that happens, all messages need to be successfully processed by the running instance of rsyslog. Obviously, we could use a timer, for instance a 5 seconds window would probably solve the problem, but it's far less practical than knowing when exactly a main message queue is drained. Moreover, if the specified time period is exceeded the behavior would be unpredictable.

Due to the fact that aforementioned module is a loadable plugin and each input plugin that wants to benefit from the dynamic configuration reload option must slightly extend its implementation. The module in question is only used for testing purpose, thus it has absolutely no documentation at all. Consequently, changing internal parts of such module is quite challenging. Firstly, the task required deep investigation on the module and was resolved via a commit[3]. Besides that, the test bench tool was also extended with a new functionality that is able to send the appropriate signal(SIGHUP) to the running instance of rsyslog.

Above mentioned actions made it possible to implement test cases specifically for the dynamic configuration reload option. The purpose of this test suite was to ensure that new changes introduced to the project will not break the proposed feature. It needs to be highlighted that each test case may cover multiple configuration directives, as some of them

[2]https://rainer.gerhards.net/2019/03/howto-great-pull-request.html
[3]https://github.com/Cropi/rsyslog/commit/77698e33529c1f5f7630307512de20f5d234e74e

highly depend on each other. For example, a test must contain at least one input for collecting logs, otherwise no message enters the system at all.

In rsyslog, a special library is included at the beginning of each test to provide commonly used functionalities. Practically speaking, it is a complex shell script customizable via environment variables. It is mainly used to:

- create a configuration file
- start the rsyslogd logging daemon
- search for a free port number
- shut down the running instance of rsyslog when the main queue is empty
- compare observed output with expected results and many more

Testing is a critical part of software development - and Shell, which is already part of Unix can help to do it quickly and easily. With that said, tests were implemented[4] as shell scripts with emphasis on simplicity. A strong requirement of any test is that it must be clear and concise as not only the author of the test case may execute them. Listing 5.1 demonstrates how the *imtcp* input module is tested.

```
#!/bin/bash
. ${srcdir:=.}/diag.sh init
generate_conf
add_conf '
global(
    # enable dynamic configuration reload via HUP
    hup.reload.config="on"
)
module(load="../plugins/imtcp/.libs/imtcp")
input(type="imtcp" port="0" listenPortFileName="'$RSYSLOG_DYNNAME'.tcpflood_port")

template(name="outfmt" type="string" string="%msg:F,58:2%\n")
:msg, contains, "msgnum:" action(type="omfile" template="outfmt"
                        file="'$RSYSLOG_OUT_LOG'")
'
startup
add_conf '
input(type="imtcp" port="0" listenPortFileName="'$RSYSLOG_DYNNAME'.tcpflood_port2")
'
reloadConfig 1
assign_tcpflood_port2 "${RSYSLOG_DYNNAME}.tcpflood_port2"
tcpflood -p$TCPFLOOD_PORT2 -m5000
shutdown_when_empty
wait_shutdown
seq_check 0 4999
exit_test
```

Listing 5.1: An imtcp input module test case

As it was mentioned before the script starts with environment initialization, which is followed by configuration generation. In the configuration above, only a single imtcp input

[4] https://github.com/Cropi/rsyslog/commit/da2de935f9b833a79ca344ac677565d8a0fce5a2

instance is collecting logs through the TCP/IP protocol via port 514. Although, it is later extended with an additional listener that is capable of collecting logs on port 515. As soon as the HUP signal is delivered to the rsyslog instance, configuration is dynamically reloaded, meaning that the service is able to collect logs via port 515. Next, the *tcpflood* C program is used to generate and forward 5000 unique messages through the mentioned connection channel. Although rsyslog is extremely fast, message processing does not happen in a single instant. This is the right place where the *imdiag* module is utilized, which is able to stall the script until the main message queue is drained. Finally, observed output and expected results are compared with zero difference strategy.

5.3 Limitations

During the implementation phase a great effort has been made to merge all changes into the official github repository. However, approval of certain pull requests by the github maintainer took much more time(6+ weeks) than initially planned, thus some changes were introduced only to the forked repository.

One of the limitations of the feature is that some global parameters are not permitted to be reconfigured during configuration reload. For instance, the *privdrop.user.name*, which is used to set the effective user ID of the calling process other than the current ID. If by chance this parameter was reconfigured during configuration reload, behavior of the program would be unpredictable.

In order to collect logs more quickly via the TCP/IP protocol on a certain port, each input instance of the *imtcp* module must run in a separate thread. Upon activating the imtcp config, a new thread is created that will be responsible for the creation of all input instance threads. Whilst this approach seems to be astonishing, it has one particular downside when dynamic configuration reload is invoked. If that happens and a new input instance is present in the config, 2 other threads will be spawned even though only 1 new input has been added. To sum up, each time the imtcp module is reloaded, n new threads will be spawned, where n is the total amount of new input instances + 1. The disadvantage comes from the design of input instance handling in rsyslog. In order to address that, big part of the rsyslog core should be changed, which would probably introduce regressions by time. Linux has a setting for maximum threads per process, which specifies the maximum number of simultaneous executions that the process can handle. However, it is theoretically impossible to reach that number, moreover most configurations specify just a few imtcp input instances.

Drawbacks of more complex features are usually discovered after a new version of rsyslog is officially released. In general, it takes a while until package maintainers of certain operating systems decide to rebase[5] rsyslog to the latests version on the OS in question. A regular user does not work directly with the version of rsyslog that is publicly available on Github, because compilation of such program requires knowledge of dependencies and configuration options that can be turned on during build time.

[5]synchronizing a branch with the originating branch by merging all new changes in the latter to the former.

5.4 Proposed enhancements

Even though rsyslog has been developed for many years, it is not a finished product that does not need any future plans at all. The primary goal of this section is to suggest possible improvements for the implemented feature.

Support for other plugins

As it was mentioned before, the main goal of the thesis was to support dynamic configuration reload option for the main directives. However, rsyslog has many loadable plugins that are used and developed by the community. All plugins that want to benefit from the proposed feature must implement support for comparing instance parameters as well as module parameters. Furthermore, input modules must undergo additional changes as they are not part of the *ruleset* and differ in terms of implementation. Generally, each input instance is executed in a separate thread. When the configuration reload functionality is invoked, we need to take into consideration that a certain input instance might get deleted. Among other things, proper deallocation of an input instance involves thread termination as well. Any developer who wants to contribute to this feature of the project shall follow:

- input module support - example implementation of the *imtcp*[6] module
- any other module support - example implementation of the *omfile*[7] module

Configuration file monitoring

Configuration file monitoring was briefly explained in 4.4, where pros and cons of this approach were listed. Practically speaking, it was one of the possible approaches on how the invocation of configuration reload could look like. The purpose of this method was to use watch descriptors to monitor rsyslog related configuration files and dynamically reload the main directives whenever one of the files being watched is modified. Finally, it turned out to be a relatively complex and not urgently required approach to take.

Completely reload runtime statistics

Although dynamic reload support for runtime statistics produced by rsyslog has already been introduced in 4.5, one possible drawback has been pointed out. When a configuration is reloaded, statistics gathering continues on as if nothing extraordinary happened. Whilst this approach is feasible for most users, others might want to reset those specific statistics. This could be solved by introducing a new configuration option, which picks from one of the two approaches. However, the default value must not be changed as we need to respect backward compatibility.

[6] `https://github.com/rsyslog/rsyslog/commit/13ba8610c19969cd86413072b94f5044f18282c9`
[7] `https://github.com/Cropi/rsyslog/commit/b408c17fd60e72a431108817dabd37736c02fbe1`

Chapter 6

Conclusion

This work described the problem of manually reloading the main rsyslog configuration file. The rsyslog project was described in detail with emphasis on configuration directives, including its implementation. Multiple types of problems were pointed out, which can happen during manually reloading the config, including message loss and connection disturbance. A goal to extend rsyslog with dynamic configuration reload option was presented. Based on the review, numerous code changes were proposed.

The author contributed to the rsyslog open source project and continuously improved his code based on code reviews and suggestions provided by the community and continuous integration tools. Proper implementation of the proposed feature required a need to modify big amount of existing code. The changes needed to be reviewed and later merged into the default branch of the project by the maintainer. Unfortunately, the review process was slow, taking more than 6 weeks in some cases. The implementation could continue on without merging the suggested changes, however pull requests have no meaning to be created when a proposed not yet merged patch is needed for that particular change. Due to this fact, some changes were introduced only to the forked version of the repository.

Many improvements that were made during author's work on this thesis could be beneficial for the entire rsyslog project, as well as for companies shipping rsyslog, including Red Hat, Inc. Besides focusing only on the main topic, multiple bugs were discovered during studying and experimenting with the project, which were reported and fixed by the author. A big effort has been made to refactor some parts of the code, which were created and maintained throughout the years by several dozens of contributors each using slightly different coding style.

The improvements enabled to dynamically reload configuration for core components without the need of a full restart. Moreover, internal structures of rsyslog can be changed without compromising the ability to process incoming data on unchanged components. The test suite has also been extended to cover the new feature. Finally, multiple ideas were introduced as future plans.

Bibliography

[1] *What is Common Criteria?* [online]. Forcepoint, october 2021 [cit. 2021-10-02]. Available at: https://www.forcepoint.com/cyber-edu/common-criteria.

[2] Antonio Messina, I. and Giovanni Giacalone. *Log monitoring and analysis with rsyslog and Splunk* [online]. 2015 [cit. 2021-10-01]. Available at: https://www.researchgate.net/publication/301536905_Log_monitoring_and_analysis_with_rsyslog_and_Splunk.

[3] Chris Lonvick. *The BSD syslog Protocol - RFC3164* [online]. RFC Editor, 2001 [cit. 2021-10-17]. Available at: https://datatracker.ietf.org/doc/html/rfc3164.

[4] Corporation, T. M. *Common weakness enumeration: A Community-developed List of Software and Hardware Weakness Types* [online]. 2021 [cit. 2022-02-29]. Available at: https://cwe.mitre.org/index.html.

[5] David Kernel. *Understanding the Linux Kernel Auditing System* [online]. SANS, september 2018 [cit. 2021-10-16]. Available at: https://www.sans.org/white-papers/38605/.

[6] David Lang. *Log Filtering with Rsyslog* [online]. 2013 [cit. 2021-10-24]. Available at: https://www.usenix.org/system/files/login/articles/06_lang-online.pdf.

[7] Free Software Foundation, I. *Signal handling* [online]. 2013 [cit. 2022-02-14]. Available at: https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html.

[8] Free Software Foundation, I. *Host Address Functions* [online]. 2022 [cit. 2022-03-05]. Available at: https://www.gnu.org/software/libc/manual/html_node/Host-Address-Functions.html.

[9] Free Software Foundation, Inc.. *The C Preprocessor: The token pasting operator* [online]. 2022 [cit. 2022-02-01]. Available at: https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html.

[10] GitHub, I. *GitHub actions: Using workflows* [online]. 2022 [cit. 2022-02-24]. Available at: https://docs.github.com/en/actions.

[11] Jay Thakkar. *What Is TLS Encryption and Why You Need It* [online]. 2020 [cit. 2021-11-04]. Available at: https://www.venafi.com/blog/what-tls-encryption-and-why-you-need-it.

[12] Jorgen Schäfer. *Why Journald?* [online]. Loggly, january 2016 [cit. 2021-10-15]. Available at: https://www.loggly.com/blog/why-journald/.

[13] Kerrisk, M. *The Linux Programming Interface: pthread_cleanup_push(3) — Linux manual page* [online]. 2021 [cit. 2022-02-21]. Available at: https://man7.org/linux/man-pages/man3/pthread_cleanup_push.3.html.

[14] Kerrisk, M. *The Linux Programming Interface: setuid(3) — Linux manual page* [online]. 2021 [cit. 2022-03-05]. Available at: https://man7.org/linux/man-pages/man2/setuid.2.html.

[15] Michael Kerisk. *Ctags(1p) — Linux manual page* [online]. 2021 [cit. 2022-03-05]. Available at: https://man7.org/linux/man-pages/man1/ctags.1p.html.

[16] Michael Kerisk. *Journalctl(1) — Linux manual page* [online]. 2021 [cit. 2021-10-15]. Available at: https://man7.org/linux/man-pages/man1/journalctl.1.html.

[17] Prasanta Kumar Sahoo, D. and Dr. Gunamani Jena. *Syslog a Promising Solution to Log Management* [online]. 2012 [cit. 2021-10-02]. Available at: https://www.researchgate.net/publication/332780312_Syslog_a_Promising_Solution_to_Log_Management.

[18] Radu Gheorghe. *Syslog parsing* [online]. 2008 [cit. 2021-10-17]. Available at: https://www.rsyslog.com/doc/master/whitepapers/syslog_parsing.html.

[19] Radu Gheorghe. *What is Syslog: Daemons, Message Formats and Protocols* [online]. 2017 [cit. 2021-10-17]. Available at: https://sematext.com/blog/what-is-syslog-daemons-message-formats-and-protocols/.

[20] Rainer Gerhards. *The rocket-fast Syslog Server* [online]. 2008-2022 [cit. 2021-10-19]. Available at: https://www.rsyslog.com/.

[21] Rainer Gerhards. *Message parsers in rsyslog* [online]. 2009 [cit. 2021-10-22]. Available at: https://www.rsyslog.com/doc/master/concepts/messageparser.html#message-parsers-in-rsyslog.

[22] Rainer Gerhards. *The Syslog Protocol - RFC5424* [online]. RFC Editor, 2009 [cit. 2021-10-17]. Available at: https://datatracker.ietf.org/doc/html/rfc5424.

[23] Rainer Gerhards. *Rsyslog: going up from 40K messages per second to 250K* [online]. 2010 [cit. 2021-10-19]. Available at: https://www.researchgate.net/publication/332780312_Syslog_a_Promising_Solution_to_Log_Management.

[24] Rainer Gerhards. *Encrypting Syslog Traffic with TLS(SSL)* [online]. 2021 [cit. 2021-11-14]. Available at: https://www.rsyslog.com/doc/master/tutorials/tls.html.

[25] Red Hat, Inc. Viewing and managing log files. *System administrators guide* [online]. Red Hat, august 2021 [cit. 2021-10-01]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system_administrators_guide/ch-viewing_and_managing_log_files.

[26] Red Hat, Inc. *System auditing* [online]. Red Hat, august 2021 [cit. 2021-10-14]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/security_guide/chap-system_auditing.

[27] Red Hat, Inc. *What is an sosreport and how to create one in Red Hat Enterprise Linux?* [online]. Knowledgebase, october 2021 [cit. 2021-10-13]. Available at: https://access.redhat.com/solutions/3592#sosreport.

[28] Red Hat Inc,. *Working with Queues in Rsyslog* [online]. 2021 [cit. 2021-11-17]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system_administrators_guide/ch-viewing_and_managing_log_files#s1-working_with_queues_in_rsyslog.

[29] Shields, I. *Monitor Linux file system events with inotify* [online]. 2010 [cit. 2022-02-20]. Available at: https://developer.ibm.com/tutorials/l-inotify/.

[30] Staff, E. *Reduce C-language coding errors with X macros* [online]. 2013 [cit. 2022-02-11]. Available at: https://www.embedded.com/reduce-c-language-coding-errors-with-x-macros-part-1/.

[31] Steve Grubb. *Auditd(8) — Linux manual page* [online]. 2021 [cit. 2021-10-16]. Available at: https://man7.org/linux/man-pages/man8/auditd.8.html.

Appendix A

Rsyslog Upstream Work

1. https://github.com/rsyslog/rsyslog/pull/4743 - Adjust ratelimiting to dynamic config reload
2. https://github.com/rsyslog/rsyslog/pull/4729 - Replace problematic functions
3. https://github.com/rsyslog/rsyslog/pull/4671 - Minor fix when accepting new connection
4. https://github.com/rsyslog/rsyslog/pull/4745 - Outchannels memory leak fix
5. https://github.com/rsyslog/rsyslog/pull/4754 - Introduce placeholder for the new config
6. https://github.com/rsyslog/rsyslog/pull/4760 - Make global parameters part of the main config, so we can analyze the difference between two config files
7. https://github.com/rsyslog/rsyslog/pull/4764 - Make timezone specific variables part of the config, so timezone directives can be reloaded
8. https://github.com/rsyslog/rsyslog/pull/4783 - Introduce dynamic configuration reload option and implement it for the imtcp input module
9. https://github.com/rsyslog/rsyslog/pull/4791 - Make the main message queue part of the runtime config
10. https://github.com/rsyslog/rsyslog/pull/4800 - Provide a way to be able to distinguish between loaded and active actions
11. https://github.com/rsyslog/rsyslog/pull/4812 - Fix regression introduced by PR4760
12. https://github.com/rsyslog/rsyslog/pull/4832 - Resolve compilation error
13. https://github.com/rsyslog/rsyslog/pull/4831 - Make parsers part of the main config
14. https://github.com/rsyslog/rsyslog/pull/4640 - Release resources when fd is not needed anymore

15. https://github.com/rsyslog/librelp/pull/238 - Fix defects reported by covscan analysis on the librelp dependency package
16. https://github.com/rsyslog/liblognorm/pull/357 - Fix defects reported by covscan analysis on the liblognorm dependency package
17. https://github.com/Cropi/rsyslog/commit/a9ff4 - Reload imtcp instances when global directive changes
18. https://github.com/Cropi/rsyslog/commit/b408c - Reload rulesets and implement output module reload support
19. https://github.com/Cropi/rsyslog/commit/3ec1c - Compare parser lists and implement it for rfc3164
20. https://github.com/Cropi/rsyslog/commit/77698 - Adjust test bench tool to work with dynamic config reload
21. https://github.com/Cropi/rsyslog/commit/c8013 - Reload templates
22. https://github.com/Cropi/rsyslog/commit/e370ed - Prohibit security related global parameter changes in globals
23. https://github.com/Cropi/rsyslog/commit/ef540d - Reload dynamic stats
24. https://github.com/Cropi/rsyslog/commit/61b9e - Reload percentile stats
25. https://github.com/Cropi/rsyslog/commit/7fd16 - Reload lookup tables
26. https://github.com/Cropi/rsyslog/commit/47050 - Rework template handling
27. https://github.com/Cropi/rsyslog/commit/db884d - Move the parser directive to the main config
28. https://github.com/Cropi/rsyslog/commit/0ecfc - Reload parsers

Appendix B

Contents of the Attached Media

```
./
├ src - source code for this text in Latex
├ build.sh - Script for installing dependencies and compiling rsyslog on Fedora
├ thesis.pdf - This text in PDF format
├ rsyslog.conf - The main configuration file
└ rsyslog.tar.gz - The rsyslog source code
```