
Master thesis : Distributed Logging Transport for Unreliable and Lossy Networks

Auteur : Scheer, Egon

Promoteur(s) : Leduc, Guy; 12788

Faculté : Faculté des Sciences appliquées

Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"

Année académique : 2021-2022

URI/URL : <http://hdl.handle.net/2268.2/16294>

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

Distributed Logging Transport for Unreliable and Lossy Networks



Egon Scheer

School of Engineering and Computer Science
University of Liège

A thesis submitted for the degree of
*Master of Science in Computer Science with a professional focus on
"Computer systems security"*

Supervised by Prof. G. Leduc & E. Tychon

Academic year 2021-2022

Acknowledgements

First of all, I would like to warmly thank my supervisors Professor Guy Leduc and Emmanuel Tychon who made this work possible. Their guidance, advice, and feedback have helped me tremendously throughout this year. I firmly believe that this project would never have been what it is without their patience and availability.

I express my sincere gratitude to my friends and family for their continuously provided encouragement, and especially Barry Sajid for his proofreading, advice, and unconditional friendship while writing this paper.

I am also thankful to David Lang, Sr Security Engineer at SigFig, for his expertise in the field of high performance logging and his detailed recommendations during our email exchanges. Similarly, I would like to thank Rainer Gerhards, author of the *rsyslog* engine, for his dedication and investment in the open-source syslog community. His extensive research and reading on optimizing a syslog engine has been an indispensable asset for this project.

For all those who helped me during this work, directly or indirectly, thank you.

Liège, 20th August 2022

Egon Scheer

Abstract

Message logging is the tool of choice to stay informed about the health of a machine or application. These messages, called logs, are used for various purposes, including system management, performance optimization, investigation of suspicious activities, and more generally analysis and debugging. Operations that demand a level of reliability at least equivalent to the emphasis placed on them during their use. However, the syslog protocol was originally designed to work exclusively over UDP. Traditional applications, which have not benefited from the a posteriori additions such as TCP, are forced to communicate over a network that is not suitable for them (corrupted or lost messages, reordering, or unreachable server) and over which they have no control. The objective of this work is to develop a resilient syslog relay that will operate downstream of applications, collect their syslog messages and send them to a central syslog server. Several mechanisms such as the use of the TCP protocol and the retention of messages in case of connection loss guarantee reliability. Topics related to message ordering and strategies in case of an overload are also discussed and several approaches are presented to either mitigate or regulate their impact. The implementation, in the form of a prototype, is deployed inside a router running the Cisco IOx environment and features the modern syslog message engine, *rsyslog*. The model is evaluated on the basis of its functionality and performance in a test environment with network quality such as 3G cellular and EDGE. Several configurations are proposed depending on the type of usage involved. Although the solution does not cover all possible and imaginable problems, such as router outages, the evaluations demonstrate the efficiency and scalability of the proposed solution, which can for example easily handle several tens of thousands of messages per second with a very low resource footprint.

Contents

List of Figures	ix
List of Abbreviations	xiii
1 Introduction	1
1.1 Purpose of the Research	1
1.2 Constraints	1
1.3 Methodology	2
2 Background	3
2.1 What is Syslog?	3
2.1.1 Daemons	5
2.1.2 Message formats	6
2.1.3 Protocols	8
2.2 Related work	13
2.2.1 Partially reliable delivery	13
2.2.2 Syslog data	13
3 Prototype	17
3.1 Architecture	17
3.1.1 Hardware constraints	17
3.1.2 Logging appliance	18
3.1.3 Key objectives	18
3.1.4 Protocols and formats used	22
3.1.5 High-level design	23
3.1.6 Message flow	25
3.1.7 Ordering messages	32
3.1.8 Securing log forwarding	34
3.2 Configuration	35
3.2.1 Cisco IOx	36
3.2.2 Docker	39
3.2.3 Rsyslog	41
3.3 Log analysis	47

4	Evaluation	51
4.1	Purpose	51
4.2	Environment setup	52
4.2.1	Testbed specifications	55
4.2.2	Testbed parameters	57
4.3	Functional tests	58
4.3.1	Reliability	58
4.3.2	Store and forward	62
4.3.3	Chronological order	65
4.4	Performance	68
4.4.1	Cellular networks	69
4.4.2	High-speed low-latency network	71
4.4.3	Queue build-up prediction	76
4.4.4	Results	79
5	Conclusion	83
5.1	Challenges faced	83
5.2	Future works	84
5.3	Final words	85
Appendices		
A	Project source code	89
A.1	Overview	89
A.2	Structure	89
	References	91

List of Figures

2.1	Architecture of <code>sysklogd</code> logging utilities[6, p. 178].	4
2.2	An example of an RFC 3164 format syslog message[14].	6
2.3	An example of an RFC 5424 format syslog message[14].	7
2.4	An practical example of a BEEP session with eight channels, allowing eight separate syslog sessions[21].	10
2.5	The content length density distribution[24].	14
2.6	The normal and payload length cumulative distributions[24].	14
2.7	The inter-arrival time density distribution for the whole sample[24].	15
2.8	The exponential and inter-arrival time cumulative distributions[24].	15
3.1	High-level architecture design for reliable syslog transport.	24
3.2	High-level architecture of rsyslog daemon inside IOx.	24
3.3	Generic model of data flow inside rsyslog engine.	25
3.4	Synchronous communication between a producer and a consumer. .	25
3.5	Decoupling system with a queue and optional storage.	25
3.6	Batch's message processing state[31]	26
3.7	How data flow inside the router's IOx application.	28
3.8	Dropping strategy that discards <i>less important</i> messages.	29
3.9	Interval (s) duration after x retries, with $C = 1800, R = 30$	31
3.10	The number of attempts made (y-axis) according to the downtime duration (maximum 3,000 seconds, x-axis).	31
3.11	Applications management on Cisco IOx Local Manager.	36
3.12	Configuration menu of the <i>syslog relay</i> application on Cisco IOx Local Manager.	37
3.13	Data directory of the <i>syslog relay</i> application on Cisco IOx Local Manager, containing three <code>.pem</code> files	38
3.14	Qemu emulation workflow on an amd64 host machine.	40
3.15	The project's Docker filesystem layers. Inspired from [43, p. 63]. . .	40
3.16	Syslog severity levels as defined by RFC 3164.	46
3.17	Loghub datasets details[47].	47
3.18	Number of messages per day based on the <i>Linux</i> dataset.	48
4.1	Testing environment topology.	52

4.2	A 2-stage Markov chain that captures burst behaviour.	53
4.3	The token bucket algorithm which allows or denies messages depending on the levels of traffic required.	54
4.4	WANem software advanced configuration panel.	55
4.5	Relay's messages flowchart, highlighting unreliable areas in <i>bold</i> and the dropping strategy in <i>dotted-line</i>	59
4.6	Central syslog server's console after receiving 30,100 messages. . . .	60
4.7	Message generator's console after sending 30,100 messages (300s timeout).	60
4.8	The relay's internal counters evolution over the 300s test and a rate of 100 messages/s.	61
4.9	Cumulative sum of the relay's input and output counters. Both have a maximum value of 300,100 messages.	61
4.10	Simulation of the dropping strategy for 20 seconds with $r = 5000$, $d = 0.8$, $s = 45600$, $p = 0.5$. We can see very well the staircase effect induced by the dropping strategy (solid line).	63
4.11	The relay's queue size evolution second by second, with the dropping strategy threshold set to 80% of the queue. Same parameters as FIGURE 4.11.	63
4.12	Number of reconnection attempts made during a downtime period of more than 10 hours. Resume interval of 30 seconds with a ceiling of 1800 seconds.	64
4.13	Behaviour of the relay during a catch-up test. 100 messages are sent every second for 60 seconds. After 120 seconds, the central server is switched on.	65
4.14	The <code>loggen</code> script instantiating 10 devices, each in a thread with its own (random) clock.	66
4.15	<code>loggen</code> script closing after sending 2,136 messages in 300 seconds. .	67
4.16	Audit of the two log files populated during the test on the chronological order. Both have the same configuration except for the <i>time requery</i> which is 2 for the first file and 1 for the second.	67
4.17	Two tests with identical configuration except for the <i>time requery</i> which is respectively 2 on the left and 1 on the right. Displays the number of forwarded messages (cumulative sum) that requery 2 has over requery 1 (at equal time). On average, requery 2 is ~ 4.6 messages ahead of requery 1 (a percentage lead of $\sim 0.2\%$).	68
4.20	Small profile internal counters under a 10MB/s burst. On average, 247,422 messages are sent to the server every 10 seconds. No messages were dropped.	72

4.21	Number of <code>recvmsg()</code> OS calls performed vs actual messages received (one thread). Each call returns at least one message. Point (A) denotes the event in which the queue size maximum capacity was reached. Every 10 seconds, an average of 247,421 messages are received and 225,101 calls are made.	73
4.22	<code>logserver</code> script receiving an average of 27,215 messages/s, for a total of 10,886,549 messages.	73
4.23	<code>logburst</code> script which sent 11,570,815 messages in 300 seconds. . .	73
4.24	Every message received by the relay has been transmitted, totalling 10,886,549 messages. It can be observed that the forwarding process induces almost no delay (overlap).	74
4.25	General information about the <code>FastEthernet0/0/1</code> port where the message generator is connected. Image taken during the burst from the router's web interface. The RX buffer is under heavy load (rxload at 221/255 for an input rate at $\sim 35,000$ packets/s) but retains some space.	74
4.26	Rsyslog's threads under load (using 'H' inside <code>top</code> command). . . .	75
4.28	Test performed to gather data on the evolution of the size of the main queue as a function of the number of messages received. Note that the number of messages received (dashed line) is an average over 10 seconds.	77
4.29	A close up view of FIGURE 4.29 , one can clearly see the periods of burstiness followed by moments of stillness. The solid line represents the main queue size containing very indicative episodes of burst absorption with an escalation in size followed by a steep decline to zero.	77
4.30	Quadratic regression curve based on the observed data (maximum values only) from FIGURE 4.29	78
4.31	Quadratic regression curve predicting values for a rate up to 100,000 messages/s. The horizontal lines represent the three resource profiles.	78

List of Abbreviations

IOT	<i>Internet Of Things</i>
IOx	<i>Cisco IOx is an application environment that combines Cisco IOS and the Linux OS for highly secure networking.</i>
ARM	<i>Advanced RISC Machines</i>
MB	<i>MegaByte</i>
MHz	<i>Megahertz</i>
RAM	<i>Random-Access Memory</i>
CPU	<i>Central Processing Unit</i>
BSD	<i>The Berkeley Software Distribution is a discontinued operating system based on Research Unix.</i>
IETF	<i>Internet Engineering Task Force</i>
UDP	<i>User Datagram Protocol</i>
ISO	<i>International Organisation for Standardisation</i>
RFC	<i>Request For Comment</i>
CEE	<i>Common Event Expression</i>
TLS	<i>Transport Layer Security</i>
OSX	<i>OSX is a former name of Apple's operating system macOS.</i>
TCP	<i>Transmission Control Protocol</i>
RELP	<i>Reliable Event Logging Protocol is a syslog protocol that provides reliable delivery of event messages.</i>
PRI	<i>The PRI is a priority value present in syslog messages and is used to derive the facility and severity of the message.</i>
MSG	<i>MSG refers to the part of a syslog message which contains additional information about the process along with the actual message.</i>
IPv4	<i>Internet Protocol version 4</i>
IPv6	<i>Internet Protocol version 6</i>

UTF-8	<i>Unicode Transformation Format-8-bit</i>
DTLS	<i>Datagram Transport Layer Security</i>
IP	<i>Internet Protocol</i>
MTU	<i>Maximum Transmission Unit</i>
BEEP	The <i>Blocks Extensible Exchange Protocol</i> is a framework for creating network application protocols.
ACK	<i>ACKnowledgement</i>
SCTP	<i>Stream Control Transmission Protocol</i>
PR-SCTP	<i>Partially Reliable-Stream Control Transmission Protocol</i>
DDR4	<i>Double Data Rate 4</i>
GPLv3	<i>GNU General Public License version 3</i>
NTP	<i>Network Time Protocol</i>
HTTP	<i>HyperText Transfer Protocol</i>
CIA	<i>Confidentiality, Integrity, and Availability.</i>
SSL	<i>Secure Sockets Layer</i>
HTTPS	<i>Hypertext Transfer Protocol Secure</i>
DHCP	<i>Dynamic Host Configuration Protocol</i>
OS	<i>Operating System</i>
KB	<i>KiloByte</i>
OOM	The <i>Out-Of-Memory Killer</i> is a Linux safeguard that helps the system to remain stable by eliminating processes that consume excessive memory resources.
LAN	<i>Local Area Network</i>
WAN	<i>Wide Area Network</i>
Mbit	<i>Megabit</i>
Gbit	<i>Gigabit</i>
VM	<i>Virtual Machine</i>
USB	<i>Universal Serial Bus</i>
GB	<i>GigaByte</i>
NIC	<i>Network Interface Controller</i>
ICMP	<i>Internet Control Message Protocol</i>

- MTR** *Matt's traceroute* is a cross-platform network diagnostic tool.
- OWD** A *One-Way Delay* measures the amount of time it takes for a packet to travel from source to destination across a network.
- NAS** *Network-attached storage*
- RX** The *RX Buffer* is a shared buffer between the device driver and the network interface card.

1

Introduction

Contents

1.1	Purpose of the Research	1
1.2	Constraints	1
1.3	Methodology	2

1.1 Purpose of the Research

Some IoT networks are composed of edge gateways that are physically on the move, such as in a vehicle or a shipping container. They might get disconnected without warning, without any information as to what is happening when they are disconnected. While most IoT applications are coping with this very well, on the other side, traditional applications won't and rely on *Syslog* protocol to send logging messages to a central logging server. But that server might be out of reach exactly when we need it the most. The purpose of this work is to develop a resilient syslog server that will be running at the edge, collecting syslog messages from applications and send them to a central syslog server. In case of disconnection between the edge device and the central syslog server, syslog messages are stored on the edge device until the connectivity is restored.

1.2 Constraints

The solution must run inside **Cisco I0x** environment, through a packaged application embedding a **Docker** container on a Linux distribution. The router hosting

the environment has a 700MHz Quad-Core ARM CPU and 862MB¹ of RAM which should not be fully consumed. It has no permanent storage, although a hard disk can be installed at a later stage (with the exception of flash memory, but this cannot be manipulated for regular use). In terms of implementation constraints, any proposed solution must be completely transparent to the outside world. Indeed, the IoT devices and the central log server running at the edge must operate as if there were no intermediaries between them. In addition, the solution must be reliable, capable of handling loss of connectivity, and preserve the order of messages as much as possible.

1.3 Methodology

The proposed approach is as follows: I will first discuss the syslog protocol, namely how it was originally designed and for what purpose. I will then explain the different variants that have been developed and the Internet standards that followed.

I will then show the different possible paths to the original problem, where it stands in relation to the current available possibilities, what is feasible, and at what cost. I will address the solution in an iterative way in order for the reader to find at each step a compromise between the deployed service and its complexity. Indeed, the very nature of the syslog protocol invites simplicity. It will therefore be essential to know the added value that sophistication would bring to the problem, in the light of the constraints defined above.

Finally, the solution is tested by comparing several standard profiles and find their best parameters. This performance evaluation will take into account the nature of the network and the capabilities of the IOT devices.

¹The router actually has a total of 4GB of RAM but only 862MB of that memory is allocated for IOx applications (which they must share).

2

Background

Contents

2.1	What is Syslog?	3
2.1.1	Daemons	5
2.1.2	Message formats	6
2.1.3	Protocols	8
2.2	Related work	13
2.2.1	Partially reliable delivery	13
2.2.2	Syslog data	13

2.1 What is Syslog?

In a computer science context, a *log*, or sometimes called an *event*, is defined by the CEE Editorial Board as “[..] a single occurrence within an environment, usually involving an attempted state change. An event usually includes a notion of time, the occurrence, and any details the explicitly pertain to the event or environment that may help explain or understand the event’s causes or effects”[1, p. 86]. Almost all systems and applications generate log files. The advantages of logging are numerous, including monitoring and troubleshooting. This is even more true when dealing with lots of different kinds of message from a plethora of subsystems within each system, because some of them might need administrator’s attention immediately while others will simply need to be stored for future reference. Some will be automatically processed, information extracted, and reported each month. Having the logs centralized can provide a global picture of the health of the infrastructure,

but it can also allow, if necessary, for a sharp breakdown of a particular time frame.

First developed in the 1980s by Eric Allman[2] as part of the Sendmail[3, p. 18] project and integrated into the Berkeley Software Distribution (BSD), *Syslog* eventually became an unofficial standard format used by other, unrelated, programs. First documented by IETF[4] in 2001 as a behaviour observation of the syslog protocol, it has become since March 2009 the standard logging solution on Unix and Linux systems[5].

During its long history, the term syslog has referred to several things, to the point of sometimes being confused in the literature. Indeed, syslog is not a particular service but more like a set of utilities, primarily consisting of a library and a daemon. The syslog library provides interfaces for logging, and the syslog daemon gathers logs and stores them as a file. The architecture is depicted on FIGURE 2.1.

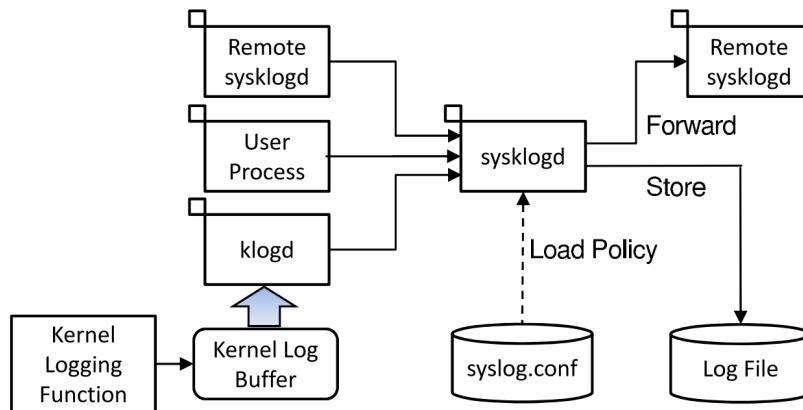


Figure 2.1: Architecture of syslogd logging utilities[6, p. 178].

The syslog library provides functions for user program to send log messages to the syslog daemon[7], such as through a shell command interface[8] or a syslog interface[9]. If the `libc` interface implementation is used, the syslog function sends messages to `/dev/log` with the `send` or `write` system call, and the syslog daemon gathers logs from `/dev/log` with the `read` system call[6, p. 178]. In user space, the daemon[10] listens to a number of domain sockets, mainly `/dev/log` but others can be configured such as the UDP port 514 for messages. It also fetches messages from the kernel log daemon[11], which accumulates its own logs in a ring buffer. It then writes these messages to some files in `/log`, or to named pipes, or sends them to some remote hosts via the syslog protocol on UDP port 514, thanks to rules

configured in `/etc/syslog.conf`[12].

Of course, this architecture only represents the foundations of syslog, namely what it was when it was created. It has since birthed other daemons, such as **rsyslog** or **syslog-ng**, to replace the ageing implementation. There have also been several Request for Comments (RFCs) released over the past forty years that have guided, if not always followed or implemented, syslog's successors. An evolution which has, for example, allowed the addition of a timezone within the message, but also the implementation of the protocol within TLS.

I will therefore devote the remainder of this section to the classification of the different existing implementations, for each syslog utility, by defining the features that are of interest, and relevant, and that follow the initial constraints of the problem.

2.1.1 Daemons

A syslog daemon is a service that offers:

1. A recipient for syslog messages, collected through socket, kernel logs, or port interface.
2. Writing to files, or not, thanks to configurable policies.
3. Message forwarding to the network or other destinations, traditionally via UDP but not only.

The very first syslog daemon, conceptualized and created by Eric Allman in the 1980s and often referred to as **sysklogd**[7], is derived from the stock BSD sources and has been used by most Linux distributions up until 2009¹. It is however still maintained by some operating system such as latest BSDs and OSX. However, most of them are not compliant with the latest RFC releases.

Alternatives to this daemon flourished in the early 2000s, including:

¹For example, Ubuntu distribution used **sysklog** until the release of *Ubuntu 9.10* which introduced **rsyslog** for the very first time, see *manifest* at <http://old-releases.ubuntu.com/releases/>. Debian distribution used it until the release of *Debian GNU/Linux 5.0* which also traded **sysklog** for **rsyslog**, see *package* at <http://archive.debian.org/debian/dists/>. On the other hand, the famous *Alpine Linux* image still ships **sysklog** as default.

- The `syslog-ng` project started in 1998 as an extension to the `sysklogd` current implementation. While it offers most of `rsyslog` features, with the exception of some being premium such as TLS, the few years' lead it has had over its competitor has given it greater maturity and a superior portability.
- The `rsyslog` project which began in 2004 when Rainer Gerhards, the primary author, decided to fork the `sysklogd` daemon and compete with the current in-place alternative, `syslog-ng`. He said, "From day one, I thought about creating something that in the long term can become a default syslogd [...]”[13]. It embeds the basic syslog protocol, with advanced filtering, network outage management, support for the latest RFCs, timestamp with year, millisecond granularity, timezone information, and other features such as TCP, TLS and RELP.
- And others such as NXLog which supports all major operating system: Windows, macOS, IBM, and more.

2.1.2 Message formats

The simplicity of syslog, or at least in its initial implementation, was mainly due to its message format. There are currently two formats defined by the Internet Society: the original BSD format, namely RFC 3164[4], and the new format that provides a foundation that syslog extensions can build on, RFC 5424[5].

RFC 3164

This first Request for Comments, released in August 2001, laid down the first guidelines for a logging protocol. It is, however, guidelines and not a standard². Indeed, as stated at the beginning of the document, "This memo provides information for the Internet community. It does not specify an Internet standard of any kind. [...]”[4]. As a result, there are nowadays several variants of the theoretical format, most of which are of little or no consequence in terms of compatibility.

According to the RFC 3164, a message has the following format:

PRI	TIMESTAMP	HOSTNAME	MSG (TAG)
			MSG (CONTENT)
<34>	Oct 11 22:14:15	mymachine	su: 'su root' failed for lonvick on /dev/pts/8

Figure 2.2: An example of an RFC 3164 format syslog message[14].

²Only a few RFCs are considered standards. RFCs are classified with regard to status within the Internet standardization process (maturity, topic covered, ...): *Informational*, *Experimental*, *Best Current Practice*, *Internet Standard*, *Proposed Standard*, or *Historic*.

Quoting the RFC 3164, “The full format of a syslog message seen on the wire has three discernable parts. The first part is called the **PRI**, the second part is the **HEADER**, and the third part is the **MSG**. The total length of the packet must be 1024 bytes or less. There is no minimum length of the syslog message although sending a syslog packet with no content is worthless and should not be transmitted.”[4].

- **<34>** is the priority number. It represents the *facility number* multiplied by 8 and then summed with the *severity value*. As such, the facility is numerically coded with decimal values ranging from 0 to 23, and from 0 to 7 for the severity possible values. Tables are visible in the RFC document[4].
- **Oct 11 22:14:15** is the syslog timestamp. Notice that the format does not allow for the year, the time-zone, and nor has millisecond precision.
- **mymachine** is the host name but the IPv4 address or the IPv6 address is also accepted.
- **su** is the tag, this is usually the process name and process id between angle brackets and must be of at most 32 characters.
- The remainder is the message content.

RFC 5424

RFC 5424[5] was released in March 2009 to primarily address the problems encountered with the RFC 3164 guidelines. It is, unlike its predecessor, an Internet standards track protocol. According to the authors of the document, it has been written with the original design goals for traditional syslog in mind but with a solid basis that “[the architecture] allows code to be written once for each syslog feature rather than once for each transport.”[5]. Here is an example of such message:

VERSION		HOSTNAME		PROCID	
PRI	TIMESTAMP		APP-NAME	MSGID	
<165>1	2003-10-11T22:14:15.003Z	mymachine.example.com	evntslg	ID47	
[exampleSDID@32473 iut="3" eventSource="Application" eventID="1011"]					BOMAn application event log entry...
STRUCTURED-DATA					MSG

Figure 2.3: An example of an RFC 5424 format syslog message[14].

The new format introduces versioning, an ISO-8601 timestamp[15], the use of dashes to indicate missing features, a message id for fast filtering, a data structure that contains key-value pairs, the encoding of the message’s content in UTF-8, and more. However, this format is proving slow to take hold in the community and in companies,

although it was for example implemented in some versions of the BSDs, and in rsyslog. Moreover, even if the daemons were to accept and comply with the norm then other parties would have to follow and upgrade, which includes, for example: the **GLIBC** logging interface, the relays that do potential filtering or load balancing on it, the collectors if going through a network, the list is endless.

The old format is already well established and most of the new features offered were already possible through the use of **JSON** in the syslog message or custom formatting, as Cisco does, but at the expense of reducing the raw message content capacity.

Cisco approach

As introduced above, one way to avoid the information shortage in an RFC 3164's message is to use its content to redefine a new format. As such, the syslog messages generated by Cisco IOS devices begin with a percent sign (%) and use the following format[16, p. 184]:

```
%FACILITY-SEVERITY-MNEMONIC: Message-text
```

Note that while the severity value is the same as defined in RFC 3164, facility one is Cisco specific and is only relevant within the message string. The mnemonic field is a device-specific code that uniquely identifies the message. One can also add an ISO-8601 timestamp by beginning the message with a special character (*), such as:

```
*Mar 6 22:48:34.452 UTC: %LINEPROTO-5-UPDOWN: Line protocol on
Interface Loopback0, changed state to up
```

The format slightly changes depending on the Cisco system[16, p. 185]: CatOS, Cisco PIX Firewall, etc.

2.1.3 Protocols

One of the fundamental tenets of the syslog protocol and process, if not its flexibility, is its simplicity. From the very beginning and even before it turned into a standard, RFC 3164 already recommended³ the use of the UDP protocol because it was in line with the lightweight approach that syslog advocated for. Indeed, in most cases, no responding acknowledgement of receipt of the event is required or even desired. It is in this regard that syslog has gained its popularity, to the point of having its protocol standardized in 2009 as part of the RFC 5426[17].

But while this logic remained true for many years, the implementation of protocols to address the lack of security that syslog exhibited gradually gave rise to standardized extensions such as the use of TLS[18] or even DTLS[19].

³“Syslog uses the user datagram protocol (UDP) as its underlying transport layer mechanism. The UDP port that has been assigned to syslog is 514.”[4].

UDP

With RFC 3164 which described the syslog protocol as it was observed in existing implementations, i.e. before 2001, the UDP transport was already specified. But as reported in RFC 5426[17], such solution also comes with its drawbacks, “Network administrators and architects should be aware of the significant reliability and security issues of this transport, which stem from the use of UDP.”[17]. These include:

1. Absence of sender authentication and message forgery
2. No confidentiality
3. Replaying messages
4. Unreliable delivery, due to...
 - (a) No mechanism for lost datagrams
 - (b) Message corruption
 - (c) Congestion control
 - (d) Out of order
5. No message prioritization based on severity
6. Denial of service

As for the protocol specification, RFC 5426 only handles one syslog message at a time but it can be truncated. The maximum size supported is that of UDP, namely 65535 bytes minus the UDP and IP headers. However, to mitigate packet fragmentation⁴ at the IP layer, UDP messages should be constrained by the Internet maximum transmission unit (MTU) of 1500 bytes minus the UDP header. RFC 3164 defines a protocol requirement of 1024 bytes, which practically all senders enforce (either by dropping longer messages or splitting them into multiple conforming messages).

⁴Fragmentation is to be avoided if possible, namely because it waste resources at both sender and receiver end, adds protocol overhead, and any fragment be lost/corrupted the whole datagrams will need to be resent.

BEEP

In November 2001 the Internet Society published a standard, RFC 3195[20], that describes how to realize the syslog protocol when **reliable delivery** is selected as a required service. RFC 3195 offers multiplexing capabilities and multiple channels to exchange on using the BEEP protocol framework. It allows for connection-oriented, asynchronous interactions. Within BEEP, features such as message aggregation, authentication, privacy, and reliability through retransmission are provided.

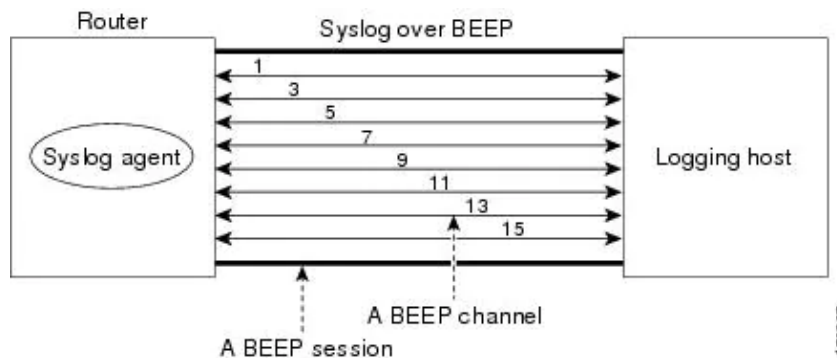


Figure 2.4: An practical example of a BEEP session with eight channels, allowing eight separate syslog sessions[21].

BEEP as a transport protocol for syslog messages provides a configurable channel as a separate session to the same host. For example, the design approach used by Cisco and depicted on **FIGURE 2.4** is to have as many severity levels as channels to allow fined-tuned flow control: channel priorities, (relative) buffer allocations, and so on.

However, due to the lack of demand, RFC 3195 has received little to no attention in practice. The main functionalities that BEEP would have brought to syslog and which is not possible with a plain TCP, i.e. a negotiation model to define the requirements⁵ and multiple sessions to a single logging host, failed to seduce. This is mostly due to the fact that RFC 3195 has considerable overhead and lacks compatibility for updated IETF syslog standards such as missing RFC 3339 timestamps and references to the outdated RFC 3164 (which has since been published as RFC 5424, but was not named at that time).

⁵Requirements can negotiate privacy, authenticity, but also transport protocol, and much more.

TCP

As introduced in the SECTION 2.1.3, there are a variety of reasons why plain UDP may not be the ideal choice for log messages. While the RFC 5424 describes the format of a syslog message, it does not specify any transport layer protocol. It is only in April 2012 and as part of the RFC 6587[22] that the IETF describes what has been observed with legacy syslog over TCP. Indeed, as stated in the document, it is nothing more than a description and not a standard: “There have been many implementations and deployments of legacy syslog over TCP for many years. That protocol has evolved without being standardised and has proven to be quite interoperable in practice. This memo describes how TCP has been used as a transport for syslog messages.”.

Despite several advantages such as error recovery and acknowledgement mechanism to name a few, plain TCP syslog is *not a reliable solution*, or at least should not be used if a **highly reliable** and tamper-proof logging system is required. Indeed, one of the problem is the missing application level acknowledgment: the transport level ACK is for use by TCP and it shall have no bearing on the application layer. For example, TCP `send()` API returns success⁶ but actually buffers the message locally, with an upper limit on the buffer size set to the negotiated congestion window, and as such in the worst-case scenario there is no way for the application layer to know if it succeeded or not until the keep-alive fires (which is of two hours for Windows and Linux distributions), moment at which the application layer will most likely have forgotten about the *sent* messages. The following are other examples of problems that can occur after a call to `send()`: the receiver crashes before receiving the packet, there is no guarantee that the application layer has received the bytes, the network could be unreachable and cause a timeout after a long period, it could receive some of the segments correctly and then encounter one of the errors listed above, so on and so forth. These are rare, but not impossible, events.

TLS

Transport Layer Security (TLS) has been standardized in March 2009 in the RFC 5425[18] to provide a secure connection for the transport of syslog messages with, if required, sender’s authentication. Please note that while it offers hop-by-hop security, it does not offer end-to-end security, nor does it authenticate the

⁶The success indicates that the data has been successfully copied into the TCP socket’s outgoing-data-buffer.

communication (just the last sender). As such, if one uses a relay to transport messages then they may have been originated from a malicious system, which placed invalid hostnames and/or other content into it.

DTLS

The Datagram Transport Layer Security (DTLS) provides secure transport for syslog messages in cases where a connectionless transport, such as UDP, is desired. It is described in the RFC 6012[19]. DTLS features:

1. Confidentiality.
2. Integrity checking on a hop- by-hop basis.
3. Server or mutual authentication.
4. Cookie exchange to prevent DoS attacks.
5. A sequence number to counter replay attacks.

Likewise, the security considerations in SECTION 2.1.3 also apply here. Moreover, such transport allows for messages to be lost or removed by an attacker without the knowledge of the receiver. *Please note that DTLS does not require (or provide) reliability nor does it deliver data in-order.*

REPL

As introduced in the SECTION 2.1.3, application layer reliability cannot be achieved with plain TCP. Reliable Event Logging Protocol (RELP), developed in 2008 as part of `rsyslog` toolkit, expands the syslog protocol's capability to allow reliable delivery of event messages. While it uses TCP for message transmission, it employs a backchannel to relay information about messages processed by the receiver to the sender. As a result, even if the connection is aborted, RELP will always be aware of which messages have been successfully received.

Although RELP is not IETF defined, RFC 3195 syslog served as inspiration for it. Sender and receiver negotiate session settings, such as the supported commands or application level window size, during the first connection. Logs are transmitted as commands, and after a command has been processed by the recipient, it is acknowledged. Both the sender and the receiver can end a session. When a session aborts, RELP records the transaction numbers for each command and negotiates

which messages must be delivered again when the session is re-established.

As of now, the protocol has only been implemented in a few libraries like `rsyslog` or `logstash`, which considerably reduces the choices for the central syslog server.

2.2 Related work

2.2.1 Partially reliable delivery

As put forward in the SECTION 2.1.3 and defended by Tsunoda et al[23], using TCP to send syslog messages is only reliable while connected to the central server. The authors also point out the drawbacks of TCP's transmission control which does not differentiate messages importance, thus enabling for example timeliness retransmission for important logs. They propose an alternative version of TCP which bounds the number of acknowledgments required to the log severity. Similarly, Rajiullah et al.[24] suggest and assess the usage of PR-SCTP, an existing partial reliability modification of the SCTP transport protocol, to provide priority to syslog messages while balancing timeliness and reliability. This approach aligns with the RFC 5424[5, p. 27] that states that when syslog messages need to be discarded, message-prioritisation based on the severity values should be taken into account.

Although these approaches are not the current focus of the project, they highlight the importance of compromise and show that "one size fits all" does not always work.

2.2.2 Syslog data

In M. Rajiullah et al. work[24], called *Syslog Performance: Data Modelling and Transport*, the authors modelled syslog data using real traces from an operational network. The model was used as input in their performance evaluation. Their approach is a great asset to the project as it provides a methodology for extracting patterns from syslog messages. As a result, it is possible to generate messages that follow the derived distributions. The dataset used comes from a syslog server located in the Computer Science Department network at Karlstad University, in Sweden. It was a two-year sampling which started back in late 2008. The authors have not officially stated the format of the messages, but it most likely followed the RFC3164 format.

Message length distribution

The authors first determined the distribution of syslog message's length. Most captured messages were missing the priority and header, and as such only the message's payload was used to compute the distribution.

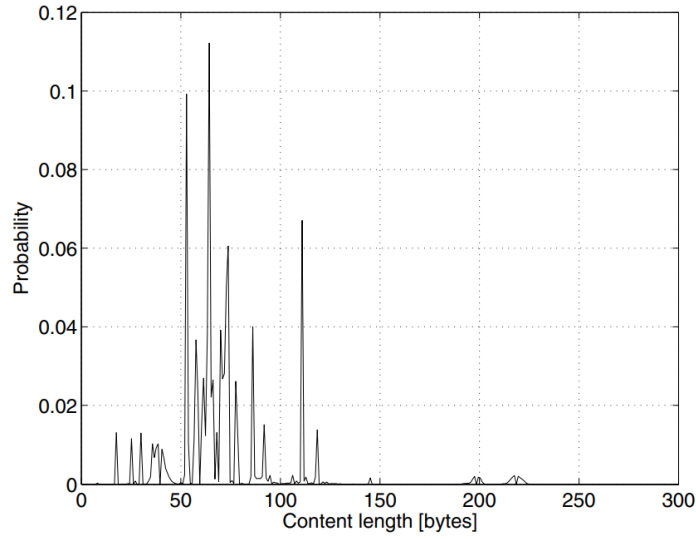


Figure 2.5: The content length density distribution[24].

The average header length and priority were empirically calculated through a one-day capture. The authors concluded that the behaviour of **FIGURE 2.5** followed the normal distribution of $\mathcal{N}(71.5, 30.2^2)$. After adding the priority and header part of 13.5 bytes, the message length distribution was $\mathcal{N}(85, 30.2^2)$. The **FIGURE 2.6** below depicts their cumulative distribution and exposes their fitness.

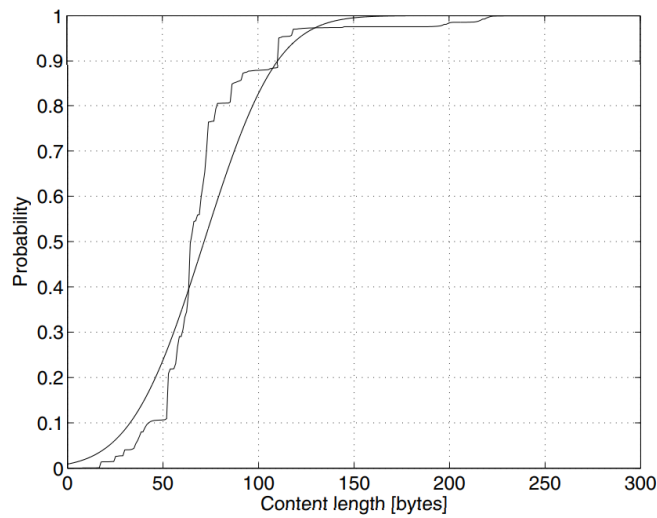


Figure 2.6: The normal and payload length cumulative distributions[24].

Inter-arrival time distribution

The inter-arrival time distribution, which describes the time between successive messages entering the server, was determined using the same method. The one-day sample was also analysed to ensure consistency. The distribution is shown on FIGURE 2.7.

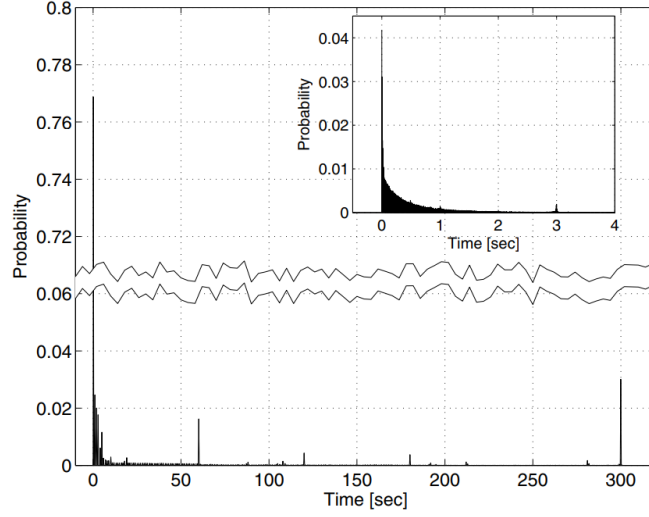


Figure 2.7: The inter-arrival time density distribution for the whole sample[24].

Based on the plot pattern, the authors concluded that the inter-arrival distribution could be modelled using an exponential distribution of parameter $\lambda = 1.361^{-1}$. Their cumulative distributions were plotted to illustrate their fitness (as seen in FIGURE 2.8).

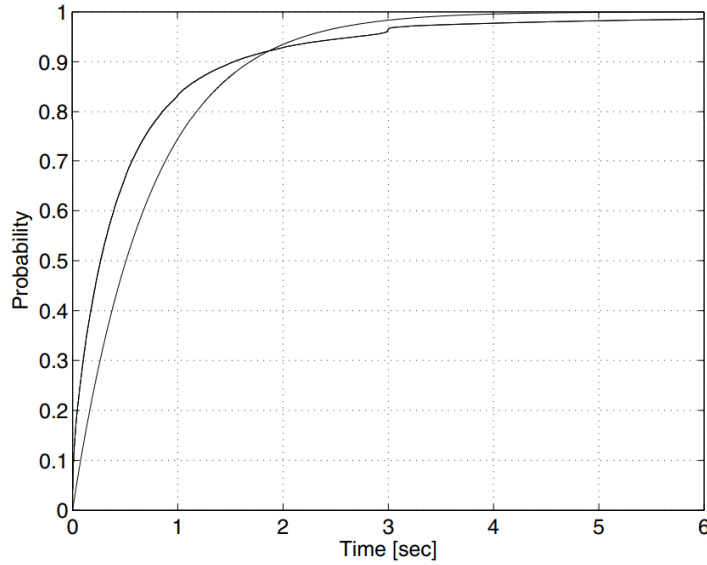


Figure 2.8: The exponential and inter-arrival time cumulative distributions[24].

Important message distribution

Finally, they defined *important* messages as the one with severity values from 0 to 3 (emergency, alert, critical, and error) and found out that 7.6% of their sample were important messages.

Message generator

Based on these results, the authors were able to develop a syslog message generating application, which they exploited to fuel their experimental assessment.

It is important to mitigate such targeted and environment-specific results. It is only representative of a particular situation and may introduce bias if used in the wrong context. However, these results highlight a trend in syslog messages and prove that they do follow distributions and can therefore, if given the proper dataset, be generated automatically.

3

Prototype

Contents

3.1	Architecture	17
3.1.1	Hardware constraints	17
3.1.2	Logging appliance	18
3.1.3	Key objectives	18
3.1.4	Protocols and formats used	22
3.1.5	High-level design	23
3.1.6	Message flow	25
3.1.7	Ordering messages	32
3.1.8	Securing log forwarding	34
3.2	Configuration	35
3.2.1	Cisco IOx	36
3.2.2	Docker	39
3.2.3	Rsyslog	41
3.3	Log analysis	47

3.1 Architecture

3.1.1 Hardware constraints

As stated in the SECTION 1.2, the solution must be deployed as a Docker image for it to be installed in the application environment Cisco IOx. The target IoT platform which will host the container, the router IR1101, runs under the ARM 64-bit (aarch64) architecture family and with minimalist capabilities[25] that must be shared between all IOx applications: 1255 CPU units, 862MB of DDR4 memory, and

701MB of flash storage memory. Although a storage method is natively proposed it is not intended to be the de facto tool as any write to persistent storage causes flash wear on the device. Very frequent operations may quickly degrade flash, rendering the device non-functional. Moreover, it is crucial to constraint the resources that an application can use because they are limited and shared with others already in place. It is hence important to define an architecture that is capable of operating with a minimal footprint in terms of RAM and computation units, but also capable of working without having to rely on a storage area (accepting the loss of reliability that this entails).

3.1.2 Logging appliance

SECTION 3.1.1 introduced an important aspect of the hardware limitations of the gateway, and as such, calls for its intelligent use. The Docker container is therefore based on what the market has best to offer in terms of distribution compactness: Alpine Linux.

Inside the container operates the **rsyslog** daemon, whose main job is to relay messages. It implements the basic syslog protocol and extends it with filtering, queued operations, different input/output modules, TCP for transport, and all this through a flexible configuration system and external plugins. The major motivation for the adoption of rsyslog, and not for one of the alternatives mentioned in SECTION 2.1.1 or even for the creation of an engine from scratch, is largely due to its maturity. The daemon is packaged on most architectures, including ARM, and is available as part of Alpine Linux packages[26]. Moreover, it is licensed under GPLv3[27] which means that while the rsyslog as a whole cannot be used in a commercial product, the rsyslog runtime can. It is an open-source solution with over 20 years of exposure and practical use. It supports a wide range of message formats (even those that deviate from standards), every conceivable type of input and output, and in general offers features that will allow the project to scale with future needs: more reliability, compression, redundancy, and more.

3.1.3 Key objectives

Before the design of the architecture can be discussed, the key objectives of the solution must first be defined with consideration of the environment in which it will be deployed (and the load it will be under). It is not expected to have a system that can receive a massive amount of messages with outstanding performance, but rather a solution that is tailored to the reality of its users: IoT devices on the edge of the network operating on old technologies.

Transparency

IoT devices on the edge of the network must operate as if the gateway was not there. Their sole purpose is to deliver their messages in the legacy syslog (RFC 3164 et al.) format using UDP. The gateway silently processes them and relays them to a centralized syslog server, with a *suited* protocol and standardized formatting (RFC 5424). *Moreover, IoT devices that do not need network access beyond the local network can be insulated from the outside for extra security.*

Reliability

To ensure reliability, it is first necessary to identify all the points of failure. To lose a syslog message sent from an IoT device, assuming there is a relay and a central server, at least one of these four points must fail:

1. During the message's transportation
 - A. from the device to the relay
 - B. from the relay to the central syslog server
2. During the handling of the message
 - A. inside the gateway
 - B. inside the central syslog server

Please note that while confidentiality, integrity, and availability of data is not the main concern in the architecture, it is still discussed here and, if possible without spending much cost, enforced.

From the device to the relay (1.A), if we assume that both are on a sufficiently guarded private network then the setup can be considered reasonably secure by simply using UDP protocol. Moreover, supposing that the IoT devices and the relay are close-by, then a best-effort network is viable and will not impact much of the reliability. It is in any case acceptable because IoTs are, in this context, considered black boxes and cannot be expected nor requested to provide more than what they can, namely legacy UDP syslog.

Inside the gateway itself (2.A), there is room for failure and loss of messages as the gateway capacity is finite and if reached, due to whatever reasons, choices will have to be made about which messages to keep and which to discard. Policies and rules,

in the case of such events, need to be installed and discussed. The issue of message persistence is discussed in the section *Store and forward* (3.1.3).

During the relaying of the message (1.B) from the gateway to the central syslog server the distance can be great and the loss of connectivity is not negligible. Due to the nature of the network being unreliable and lossy, no assumptions can be made on the authenticity of the message and the author, the proof of arrival, its confidentiality, etc., and as such UDP is not an option. TCP should be enforced and if possible, enhanced with TLS as the price to pay for the security gain is minimal (SECTION 3.1.8). Moreover, in case of connectivity loss a mechanism should be set up to store logs during downtime.

Finally the central syslog server (2.B) can also be a point of failure, as explained in the SECTION 2.1.3, due to the very nature of TCP and the lack of an application layer acknowledgment. Indeed, the TCP send buffer of the relay can contain at any time from around 25 to 850 messages¹. If the relay receives the error status, the application has no clue of which messages were lost and which made it. If such reliability is needed, a protocol such as RELP should be used between the relay and the central syslog server.

Store and forward

As explained in the *Enterprise Integration Patterns* book by Gregor Hohpe[29, p. 124], guaranteed delivery cannot be achieved without a mechanism to store the message and to retry delivery until the receiver becomes available. This is called *store and forward*.

In case of a connectivity loss between the gateway and the central syslog server, a store and forward mechanism should be installed. But whose job is it to keep the messages, and what happens if at some points there is no one to fulfil it? While the flash memory is not an option, messages can be queued directly on the DRAM of the gateway, up until a limit. A threshold can also be set to smoothen messages burst by discarding queued logs below a specified severity.

If available, messages can be stored on a hard disk or even directly queued in it. Messages will remain even if the system crashes. However, persistence increases

¹Supposing a syslog message size of 150 bytes and a write buffer ranging from 4KB to 128KB, as stated in the TCP manual[28].

reliability, but at the expense of performance. Additionally, the local disk drive might not be built to keep this much data under burst-traffic circumstances, which are more likely to occur during network outages due to the nature of the environment². This remains a solution, if not the only one, which guarantees a high level of reliability in the event of a lengthy network outage, an unreachable server, but also if the router crashes³.

Finally, as noted by G. Hoppe[29, p. 125], reliability in computer systems is never a 100% value and the cost to move from 99.9% to 99.99% is more-likely not worth the investment.

Chronological order

By default rsyslog uses multiple threads to process messages, each claiming a batch of messages through a lock mechanism, which means that thread₁ will probably be still processing its batch $\{m_0..m_{99}\}$ when thread₂ sends its first messages of its batch $\{m_{100}..m_{199}\}$. Furthermore, if a connection failure occurs between the remote syslog server and a relay there is no mechanism on the server-side to ensure that when the connection is re-established old messages will be piped in chronological order.

Trying to guarantee in-order delivery is probably bound to fail at some point, but that doesn't mean that the order itself cannot be guaranteed at all. It is indeed always possible to index server logs based on the timestamp, supposing that the expected message is missing, it will *eventually* be inserted in the correct⁴ time slot. Server pipeline such as **logstash** proposes such capabilities.

However, to ensure the accuracy of the time-stamping of messages, several mechanisms are necessary:

1. The gateway must have set a proper timezone and be bound to a pool zone through the NTP protocol⁵. This provides highly accurate time and ensures that all relays are synchronized, and thus time can be safely use as a means of comparison.

²If the central server cannot be reached, e.g. due to a loss of cellular connection, it is likely that IOTs will generate a large amount of logs due to one or more of their services being down.

³Provided that the messages are already on the disk or that the router shutdowns *gracefully*. Please note that messages that are being processed before or after the queue will still be lost.

⁴Correct from the point of view of the relay, not necessarily the IOT devices.

⁵An NTP update still propagates inside a Docker container as it uses the same clock as the host, and it cannot change it.

2. Using the RFC 5424 message format, the timestamp field is a formalized timestamp derived from RFC 3339[15] and includes the year and milliseconds with respect to the timezone.
3. When relaying IoT devices' messages, outgoing timestamps should be those of the gateway internal clock, e.g. at the time of reception. It is not wise to use the timestamps of the devices directly, considering that they are in constant movement and therefore their timezones can change over time, the IoT might not have a clock synchronization and will most likely provide a wrong timestamp. Moreover, legacy syslog messages sent from those devices lack of time precision (milliseconds and timezone is missing). It is nevertheless worth adding it as an extra piece of information in case of conflict or to reconstruct the timeline of messages from the same device, appended for example inside the `structured-data`⁶ proposed by RFC 5424.

3.1.4 Protocols and formats used

At this point, it is beneficial to discuss the protocols and formats that will be deployed. Indeed, SECTION 2.1.3 and 2.1.2 show the wide range of flavours that currently exist for syslog. Although some reasoning over which protocols and formats to use has already been formulated in SECTION 3.1.3, it is interesting to compare them within the practical constraints and feasibility of the project.

First, regarding the protocols, as explained in SECTION 3.1.3 and 3.1.3, IOT devices on the edge of the network are considered as black boxes and will only provide legacy logs in UDP. The flexibility remains in the transport of the message between the relay and the central syslog server. Due to the lossy nature of the network, most likely a cellular one, it is not possible to guarantee the transmission of messages. Since this requirement is fundamental, UDP is not acceptable (and so is DTLS). TCP on the other hand provides reliability, congestion management, and ordered data transmission. Since the router will not have to handle hundreds of thousands of logs per second[30, p. 331], this protocol is suitable. The RELP protocol could have been a solution, but the absence of standardization and the scarcity of implementation in other syslog products greatly limits the possibilities on the central server side. Similarly, BEEP could have been a candidate but as discussed in SECTION 2.1.3, the protocol has considerable overhead and lacks compatibility

⁶The structured data is a named list of key-value pairs for easy parsing and searching, storing for example meta-information about the syslog message or application-specific information.

with recent RFCs. Finally, not to decouple the reasoning, the discussion on why TCP has been enhanced with TLS is available in **SECTION 3.1.8**.

Concerning the syslog formats, the input messages sent from the IOTs are in a format that cannot be changed, namely RFC 3164⁷. On the output side, in order to manage ISO timestamps and key-value structure, the forwarded message follows the RFC 5424 template. This format is also the only one to provide the host in its header. Indeed the RFC 3164 does not specify one and it was therefore customary[30, p. 130] for the receiver to take the host from the socket, which in a NATed environment can only hold the mangled NAT address. The socket approach is even more flawed here since the host is the relay (the router) and not the message's instigator.

3.1.5 High-level design

Now that we have formalized the objectives and requirements of the project, we can design the architecture of the solution. One can reasonably divide this architecture into two parts:

1. The *old style* section, which still uses syslog over UDP, mainly the local private network composed of IoT devices.
2. The *enterprise* portion, i.e. where traffic may be passing through the internet, will use the new syslog architecture based on TLS.

The gateway will benefit from clock synchronization via the NTP protocol and, if present, will be able to take advantage of a storage system to retain logs. The architecture design is depicted on **FIGURE 3.1**.

⁷Syslog has been around since the 1980s and has long acted as the de facto logging standard without any authoritative written specification. Released in 2001, the RFC 3164 regroups the best behaviours regarding 20 years of legacy log formatting, but does not enforce any. As such, vendors who already had deployed their own variations are reluctant to any change. Parsing those logs is a tedious task.

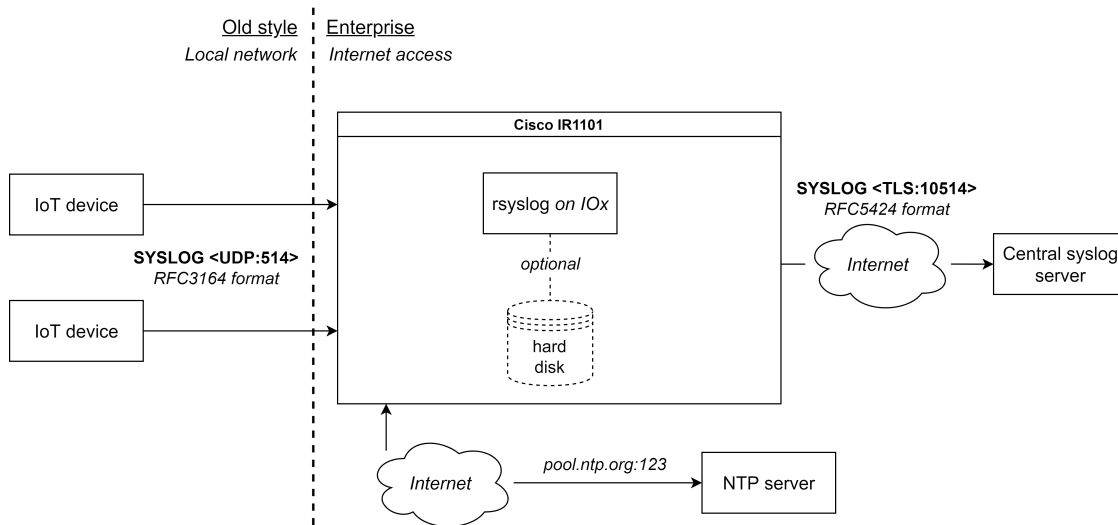


Figure 3.1: High-level architecture design for reliable syslog transport.

Inside the IOx application runs the rsyslog engine which is responsible for the reliable forwarding. As shown in FIGURE 3.2, the core engine receives input from an UDP port, processes it, and outputs it via TLS to a remote system.

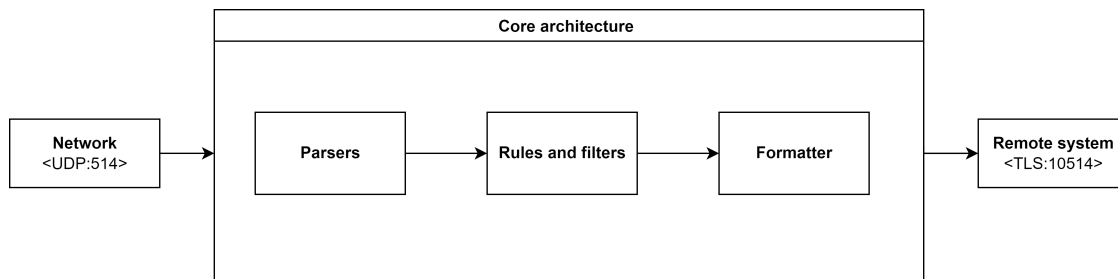


Figure 3.2: High-level architecture of rsyslog daemon inside IOx.

Within this core, syslog messages are first parsed into name-value pairs so that they are no longer handled in string format in further processing. Parsers also populate useful properties such as time-related information. A rule that includes a filter and one or more actions to be taken when the filter evaluates to true is bound to the message. For each action, and using the message's key-value pairs, an output string is created through the formatter which uses a specific template (e.g. the RFC 5424 format). Such architecture has no direct relationship between input and output strings and as such is able to take formats on the ingress and process them in new ways while retaining compatibility.

3.1.6 Message flow

Message flow or how system log or event messages are transported inside the relay from one end to the other is the central process of the project. The architecture of data flow within the rsyslog engine will therefore be thoroughly examined in this section. While SECTION 3.1.5 enlighten the key concepts that articulate the engine, here we will see what components connect them and their roles in the flow.

Rsyslog approach

As depicted on FIGURE 3.3, a typical syslog message passes through multiple stages inside the rsyslog engine. When messages enter the engine through input modules, they are first preprocessed before being added to the main queue, pulled off by worker threads, filtered, and then added to one or more action queues. Those queues also have workers which pull off messages and deliver them to the proper output module.

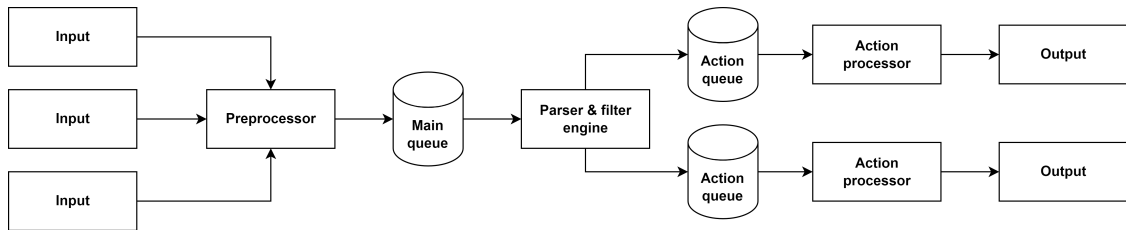


Figure 3.3: Generic model of data flow inside rsyslog engine.

Each input runs on at least one thread depending on the module (e.g. UDP, TCP, local files, ...). The main message queue decouples the input from the rest of the process. In fact, all queues inside rsyslog follow that paradigm. The FIGURE 3.5 illustrates the effectiveness of such mechanism.

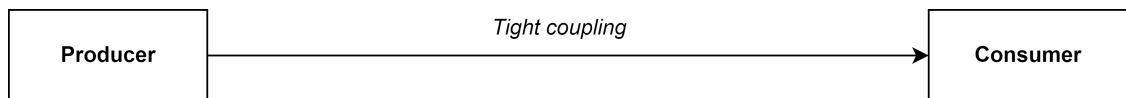


Figure 3.4: Synchronous communication between a producer and a consumer.

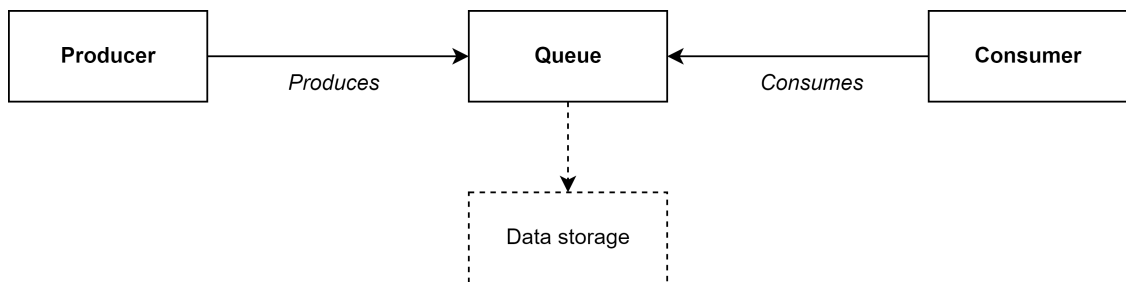


Figure 3.5: Decoupling system with a queue and optional storage.

With tight coupling, as visible on **FIGURE 3.4**, producers are unable to submit their logs if the central server is unavailable; as a result, they must either ignore the message or hold onto it while they wait for the connection to be restored. On the other hand, a decoupled architecture allows messages to be stored in a memory queue and, when possible, ingested by consumer(s). Additionally, messages can be re-enqueued after an outage until the connection is re-established.

A batch is a collection of messages that can be processed. It is the processing unit of rsyslog. Dequeuing several messages at once from a queue and sending them to the lower action layer is done using batches. If a queue has less messages than what is necessary to produce a batch, whatever it currently has is taken and forms the batch. *As a result, a batch might include as little as one message.* The **FIGURE 3.6** showcases the different processing states a message inside a batch can be in.

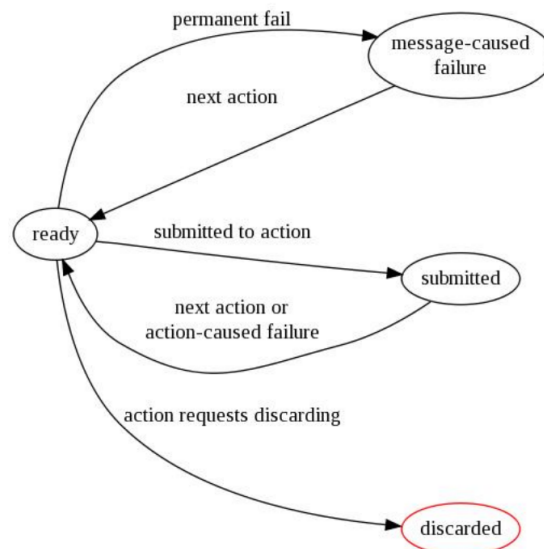


Figure 3.6: Batch’s message processing state[31]

All queues inside rsyslog can be kept in memory, written to disk, or a combination of both. The main queue has a pool of worker threads, based on the configuration and traffic volume, whose job is to pull data from the queue and run it through the parser engine. If the queue is a disk, meaning that it buffers directly in the hard drive, then no memory is used. Messages are stored in chunks, receiving its individual file so that processed data can be deleted and free for other uses. On the other hand, if an in-memory queue is used then enqueued data elements are held in memory. It is also possible to use a mix of both approaches, which allows to take

advantage of the reliability of disk and the responsiveness of RAM: when the high watermark level triggers, messages that would previously go to the memory are written to the disk until the level drops to the low watermark. In case resources are reaching their limit, a dropping strategy can be configured such that when the queue reaches a specific mark, it will begin rejecting⁸ logs with less than a given severity, until it falls below the mark.

The main queue's workers pull messages and run them through the parser to extract syslog message's *properties*⁹ and then through the filter engine which evaluates the message against all *rules*¹⁰, one after the other, until it matches one. The message is then enqueued to each action's queue contained in that rule. As previously stated, such queues also decouple and have their own worker thread(s).

Finally, messages are dequeued from their action queues using their associated worker threads. Those messages are ran through their action processor (e.g. processes the *template*¹¹ to create the plugin calling parameters) and forwarded to the proper output plugin.

This approach is particularly suitable for log processing which requires modular design. Examples of such functionalities offering this modularity include, but are not limited to, the following:

- Possibility to ingest several inputs: from the kernel, systemd log, via TCP, UDP, RELP, text files, HTTP, unix socket, and many others. The same goes for output modules.
- Actions can be grouped and triggered iteratively or based on conditional, depending on particular properties value or simply to switch to a redundant server for example.
- Parser can be chained to handle multiple formats simultaneously (e.g. RFC 3164 and Cisco IOS).

⁸Please note that both freshly incoming and previously queued low-priority messages are discarded.

⁹Properties are data items which are used in *templates* and *conditional statements*. They are mostly variables derived from the original message, but can also be engine related or user-defined.

¹⁰A rule consists of a filter and one or more actions to be performed when the filter evaluates to true. Filtering can be done based on the message's properties. Note that if a message matches with no rule, it is thrown away.

¹¹Templates alter and format the rsyslog output which enable users to define whatever format they may choose.

- Decoupling via message queues allows scaling according to hardware and specifications through configuration adjustments: queue size, number of worker threads, batch size, and several others.

Implementation

In the specific context of log messages forwarding, the data flow can be modelled as shown on FIGURE 3.7.

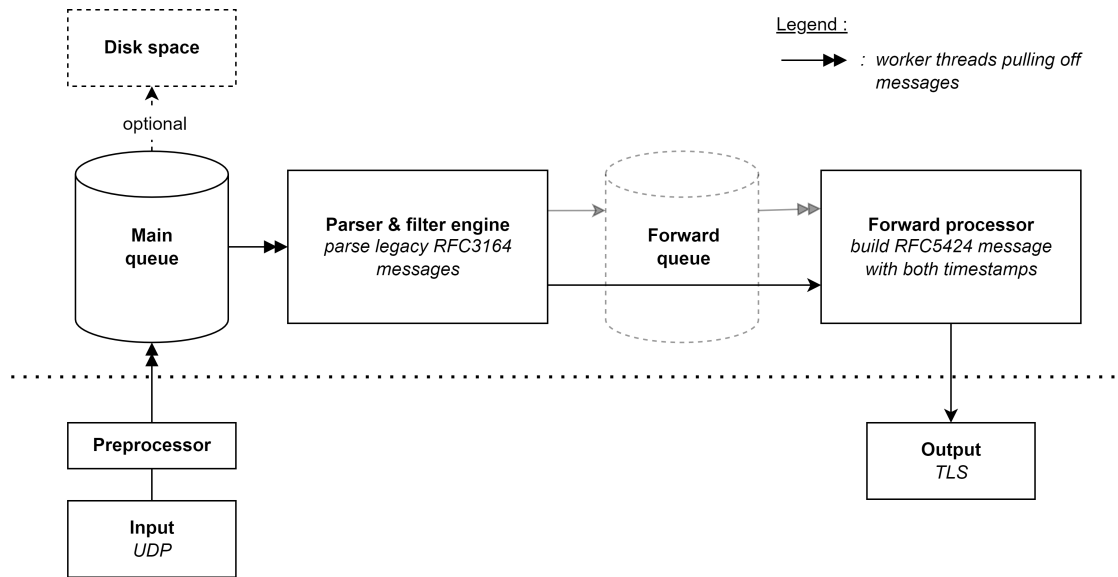


Figure 3.7: How data flow inside the router's IOx application.

The input module listens for UDP syslog messages on a specific port and are extracted by batch (using `recvmsg()` call) thanks to the pool of worker threads bound to that input¹². These messages are first preprocessed to extract key information such as its severity value and then pushed inside the main queue. Dedicated worker threads pull off messages from that queue and run them through the parser which ensures that messages are in the RFC 3164 format before extracting key information as properties. Messages are pulled by batch to minimize lock acquisitions[32, p. 168]. While this strategy does not preserve order, as pointed out in SECTION 3.1.3, it is possible to rearrange them eventually.

A dropping strategy is configured to ensure (partial) reliability when the main queue hits its maximum capacity.

¹²If needed, additional threads (with respect to number of cores) can be added to improve speed and reduce message loss[30, p. 168].

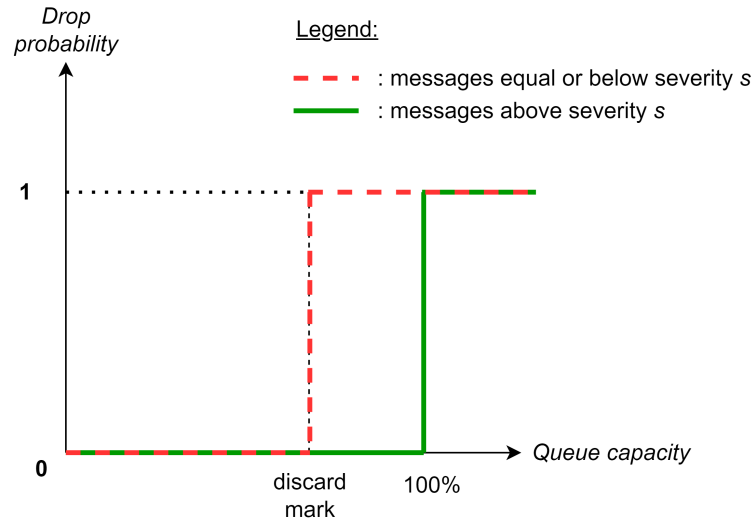


Figure 3.8: Dropping strategy that discards *less important* messages.

The queue stores a *discard mark* level that, if reached, instructs it to dump incoming and already enqueued logs below a specified severity s . This mechanism stops when dropping below the *discard mark* level. Messages with severity s or greater are sheltered until the queue completely fills up with severity s or plus messages, at which point all incoming logs will be discarded until room is made in the queue. *Please note that the severity's value actually decrease as message are more important (0 is emergency and 7 is debug), therefore from a numerical standpoint it is messages with value equal or greater than the specified severity that will be discarded.*

If referring to SECTION 3.1.6, main queue's workers should enqueue parsed messages into the action (called here *forward*) queue. However, in this situation there is no added value in having an extra queue. Indeed, action queues are most useful when there are several of them and they have to perform long tasks (e.g. transmitting logs or writing to a database) in parallel. Without them, the main queue would block¹³ at the slightest wait or failure, until it succeeds or timeouts. In our case, only one action is deployed and thus does not require a queue¹⁴.

Although there is only one queue, it is reasonable to ask where to place it. It was decided to position the queue as close to the UDP input as possible in order to minimize losses. Furthermore, if messages have to be removed from the queue

¹³As there is no decoupling, the main queue's workers are both producer and consumer (i.e. fully synchronised).

¹⁴The *forward* queue is set to "direct" which tells the driver to forward messages to the action processor without queueing.

due to the dropping strategy, they will not have consumed any CPU resources yet. However, those very arguments might be viewed as a disadvantage because when the central server is not accessible the engine remains idle (except for receiving/dropping messages) whereas it could be parsing and filtering messages. Depending on the resource or performance needs, this can be seen as a gain or a loss. Given the limited characteristics of the router, resources are prioritized.

Workers from the main queue, given a batch of parsed messages, passes them through the *forward* action which builds the output string using the RFC 5424 template and a custom structured-data. The following is a valid example of such syslog message:

```
<165>1 2003-10-11T22:14:15.003Z mymachine.example.com evntslog -
ID47 [timereported="Oct 11 22:14:15"] BOMAn application event log
entry...
```

The use of an extra variable, i.e. `timereported`¹⁵, can diminish the message ordering problem¹⁶ for the central syslog server. Indeed, as discussed in the SECTION 3.1.3, several steps in the data flow do not preserve the order: IOTs sending logs through UDP or workers pulling concurrently and in batch to name a few. Unfortunately, IOT devices provide messages with poor timestamps: no year, no timezone, and no millisecond precision. It is hence impossible with that value alone to order messages coming from several devices with desynchronized clocks.

The timestamp set when constructing the output string, called `timegenerated`¹⁷, takes advantage of the high resolution (RFC 3339) and timezone of the router, itself relying on the NTP protocol. This property is initialized when the message object is received by the workers at the UDP input¹⁸.

¹⁵Property that contains the timestamp from the original message. Resolution depends on what was provided in the message (in most cases, only seconds).

¹⁶Programs and subsystems employ log services, as indicated in [33], for recovery, audit trails, and for performance monitoring. Logs become valuable when they can be placed on a timeline, but this is impossible without a mechanism to order them.

¹⁷Property that contains the timestamp when the message was received. Always in high resolution. It has the value "2003-10-11T22:14:15.003Z" on the message's example.

¹⁸The assignation is visible in rsyslog source code's file `plugins/imudp/imudp.c:559`. For performance reasons[32, p. 6], the syscall `time()` is queried only every n messages, presuming that n is small enough such that `time()` would most likely return the same value even if queried for each message (`time()` offers millisecond precision).

Based on these two properties, it is possible to order the messages with some degree of confidence. This ordering process, performed on the central syslog server, is detailed below in SECTION 3.1.7.

Finally, the output module forwards output strings to the central syslog server using TLS (see SECTION 3.1.8 for details). If not reachable, the worker in charge of the message periodically attempts to restore the connection. This period is used to avoid using too many resources for retries. The interval grows linearly with the number of retries x , the *resume interval* R (in seconds), and features an upper limit C :

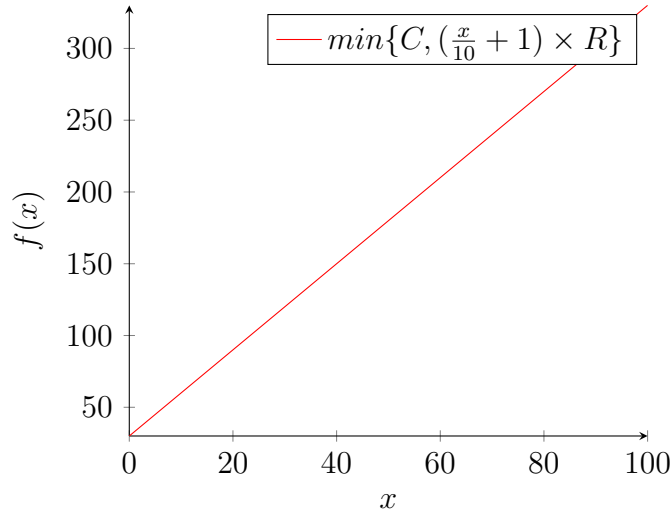


Figure 3.9: Interval (s) duration after x retries, with $C = 1800$, $R = 30$

The bottom graph (FIGURE 3.10) shows all reconnection attempts during a downtime period of maximum 3,000 seconds.

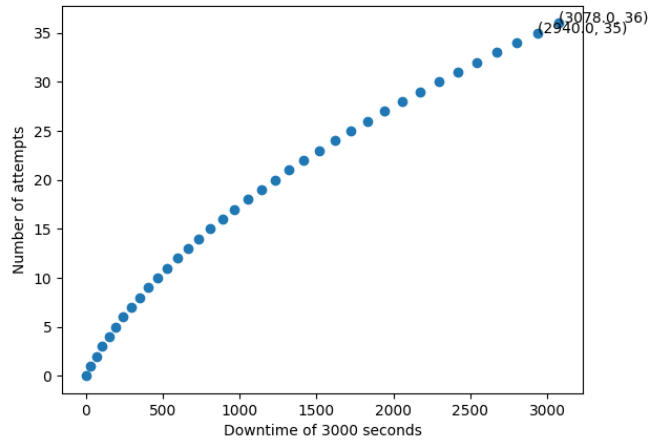


Figure 3.10: The number of attempts made (y-axis) according to the downtime duration (maximum 3,000 seconds, x-axis).

Cisco queuing methods

Someone might reasonably think that the queueing mechanism implemented in the message flow (see SECTION 3.1.6) is nothing more than a less efficient duplicate of the existing queueing system in Cisco routers. A router does, indeed, have an input queue for each interface where incoming packets are stored while they wait to be processed by the *Routing Processor*¹⁹ (RP). The maximum number of packets that may be placed on each input queue is indicated by its size. Additional arriving packets are dropped by the interface after the input queue is full. The maximum queue value can be increased by using the "hold-queue <value> in" command for each interface (between 0 and 4096 packets).

Not only it is more efficient, but it also works even if the server is unavailable since messages are either halted in the router's queue or in the worker threads' batch. However, the issue is that this queue cannot not provide message timestamping as they arrive, nor can it remove them based on their severity. Furthermore, if storing messages on a disk is considered desirable (now or in the future), then it is necessary to go through a rsyslog engine's queue. Finally, although this is not really a reason but more of a user experience concern, it could be argued that requiring manual configuration on the router negates the plug and play feature of IOx applications. On the other hand, if one accepts that modifications can be made to the router, it may be worthwhile investigating and discussing other means of enhancing the router's performance. Examples of performance improvements are given by K. Dooley et al. in their book *Cisco IOS Cookbook*[34, p. 456].

3.1.7 Ordering messages

The idea to use *Lamport's logical clocks*[33] to reliably provide a temporal partial order is tempting. A Lamport logical clock is a numerical counter value maintained by each process that increments before each local event. Considering two events A and B from the same process, Lamport defines the "happened before" relation (\rightarrow) which states that:

$$A \rightarrow B, \text{ if } A \text{ happened before } B; A \rightarrow B \text{ and } B \rightarrow C, \text{ then } A \rightarrow C$$

¹⁹The routing processor performs the route processing and maintain/distribute routing tables.

Lamport also provides means to order events between processes but that requires additional synchronization²⁰ which cannot be achieved here. Nevertheless, this relation still offers ordering for a process. Based on the logical clock definition from above, such value could be the device’s local time and the increment the elapsed time since the last event.

As of now, with the available information and constraints, the ordering strategy (at the central server) works as follows:

1. Ordering between IOTs is performed through the `timegenerated` property. Please note this only provide a *nearly-sorted sequence* due to the many causes for out-of-service delivery (such as multiple paths on the a network, parallel processing, etc.) on the ingress and the non-existing order management in UDP.
2. Ordering between events of an IOT is performed through the `timereported` property.

To further elaborate on point (1.), even if the propagation delay was the only source of disorder between messages, a mechanism would still be needed to calculate it. Since the relay cannot exchange with the transmitter (no round-trip time for example), the only usable measurement is called the *One-Way Delay* (OWD) and depicts the delay that a data packet suffers in the trip from source to destination. If the source and destination clock are synchronized, the measurement is trivial. In the present case, they are not. This problem is well-known in the scientific literature and is one of the most important parameters for evaluating performance measurement in computer networks[36]. However, as stated by L. De Vito et al. in [36], “[...] *OWD measurements are not possible without synchronization, i.e., OWD cannot accurately be obtained because the end system clocks are not synchronized with each other*”. Even if an estimate were possible, it would necessarily be calculated on the basis of the timestamp provided by the sender and the receiver. The accuracy of this delay would therefore be equal to the lowest precision between the sender’s and receiver’s clock, in this case in seconds. Given the initial assumptions in SECTION 3.1.3—that the devices and the relay are close enough for message loss to be negligible in UDP—it is reasonable to assume that this delay will be no more than one second.

²⁰When two events take place in separate processes without exchanging messages (directly or indirectly), then the two processes are concurrent, meaning that the ordering of the two events cannot be determined[35, p. 369].

Hence the contradiction.

An even stronger assumption would be that there is no delay or that it is constant and equal for all devices. This would imply that the messages arrive in order in the relay. As pointed out in SECTION 3.1.3, the order is preserved if there is only one thread that dequeues the UDP reception buffer. The stamping precision, as explained in the SECTION 3.1.6 and in [32, p. 6], is defined by the syscall `time()` and is called once every n messages. The lower the n , the lower the performance. If set to 1 or close to it, then it is reasonable to expect that the `timegenerated` property preserves the global order of messages.

Alternatively, M. Weidlich et al. have proposed[37] to solve the partial order of the logs using a behavioural model. By drawing probabilistic conclusions from the execution of other processes, these behavioural models allow the resolution of incomplete orders. However, as stated by the authors, these methods are inapplicable to logs with high variety and uncertainty. Moreover, the execution time can soar and at times reach several hours for a few tens of thousands of logs.

3.1.8 Securing log forwarding

The study of system and device event log is a frequent duty for a network, systems, or security administrator. This operation might be performed to reconstruct a timeline, to diagnose the state of a network, determine if a system or network intrusion happened, or simply to monitor network devices. Actions that require total trust in the messages. However, it is easy to convince oneself that logs sent over the internet do not satisfy the CIA triad²¹. Indeed, syslog messages can travel over unsecured networks and/or through untrusted intermediaries. The primary threats, depicted in [18], are the following:

- *Masquerading* by means of unauthorized senders forwarding to a legitimate receiver.
- *Modification* through man-in-the-middle attack.
- *Disclosure* by examining the content of syslog messages.

²¹The *CIA triad* stand for Confidentiality, Integrity, and Availability, and provides a model for the development of security policies and procedures.

The age of syslog far precedes the desire to secure the web, considering for example the release of SSL in 1995 and the change to HTTPS for Gmail in 2010[38, p. 82], that's more than a 20 years gap. Since then, the syslog working group has been pushing to catch up[39, p. 6], with as most notable release the RFC 5425[18] which describes the use of Transport Layer Security (TLS) to provide a secure connection for the transport of syslog messages. It counters the above threats through, respectively, confidentiality, integrity-checking, and server/mutual authentication. Please note that it does not provide end-to-end security, and it does not authenticate the message itself (just the last sender, i.e. the relay).

Encrypted channel introduces additional computational costs, most notably the asymmetric encryption and the TLS handshake[38, p. 82], but is realistically manageable as it counts for less than a few percent overhead[38, p. 82][40]. In worst case scenario, considering an average syslog message of 250 bytes, a batch containing a single message, and that the total overhead of the encrypted data is about 40 bytes[41], the overhead is 16%. This situation, however, does not reflect reality because messages are transmitted as a group[30, p. 100]. Furthermore, it makes little difference for the bandwidth if only a few messages must be transmitted. Let us consider a TCP maximum segment size of 1460 bytes, then the overhead is:

$$\frac{40}{1460-40} = 2.81\%$$

As a result, periods of interest such as a burst or recovery are guaranteed to go below that percentage since the MSS is more likely to be met.

Finally, trusted certificates are generated upstream and distributed to the relay and central server manually. As a result, private keys must be adequately safeguarded and inaccessible to third parties.

3.2 Configuration

Now that the different technologies have been presented, the key functionalities determined, and a high-level architecture modelled, this section will discuss the solution implementation, namely the configuration. Indeed, the implementation of the solution mainly concerns the configuration of the rsyslog daemon and the installation of its dependencies, all of which must run inside a deployable application on the Cisco IR1101 router.

3.2.1 Cisco IOx

A gateway fulfils several roles, including managing traffic and the devices that connect to it, but here it is mainly the functionality of the IR1101 router to host IOx applications that interests us. It is indeed through this mechanism that the syslog relay is deployed.

The IOx architecture of IR1101 uses a virtual machine and containers to run applications. The deployment and lifecycle control of the applications are performed via the router's local web management interface, as depicted on **FIGURE 3.11**.

In practical terms, an IOx application is an integrated Docker image in the form of a package, compressed via the `ioxclient` utility by means of a descriptor containing the requirements and metadata about the application: CPU architecture, network interface used, opened ports, resources needed, and more.

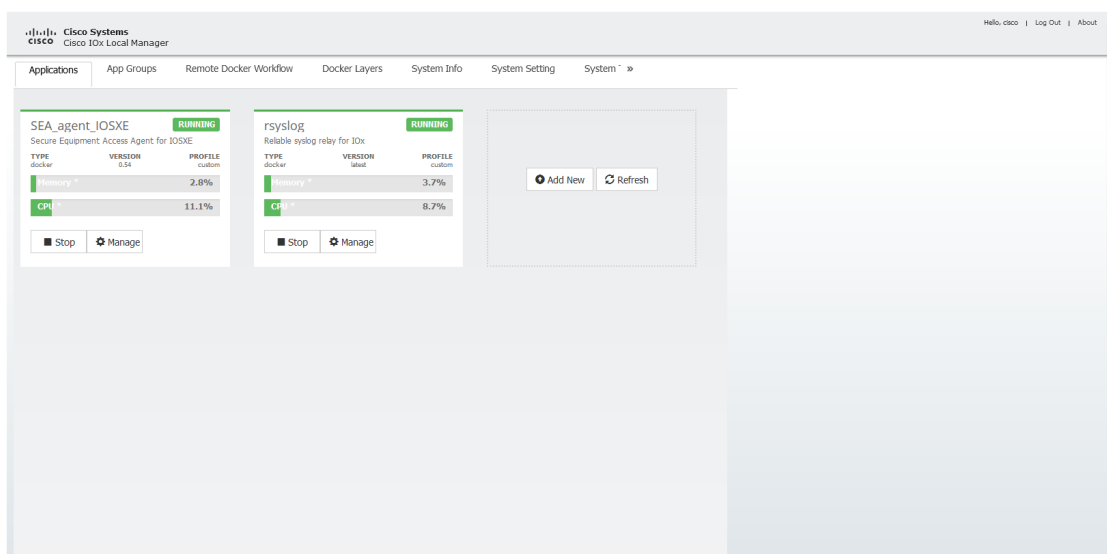


Figure 3.11: Applications management on Cisco IOx Local Manager.

When an application is added to the host system (in a compressed format), its lifecycle can be manipulated. Activating an app reserves the required but not yet used CPU and memory resources at the host, as well as an IP address obtained from an IOx-specific DHCP pool. Starting an application allows it to make use of the allocated resources. One can also stop or delete a container, the former retaining resources and the latter not. It is also possible to update the application, preserve

resources such as the assigned IP address but also the application data.

To provide a seamless user experience during deployment and maintenance, several environment variables have been placed in the rsyslog configuration files and can be manipulated directly from the local manager as shown in FIGURE 3.12. It is possible for example to switch to debug mode, to change the IP address of the central server, and more. However, the modifications will only be effective after the application has been restarted, because the entry point is in fact a shell wrapper that retrieves the configuration, parse it, load the environment variables, and then returns the hand to the rsyslog engine.

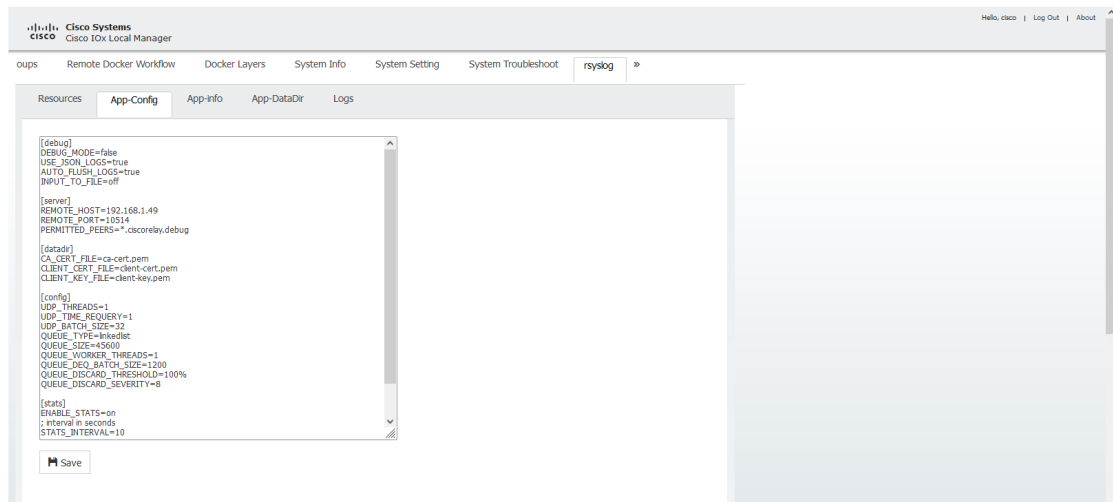


Figure 3.12: Configuration menu of the *syslog relay* application on Cisco IOx Local Manager.

Similarly, the certificates needed to open a TLS session with the central syslog server are to be uploaded to the IOx platform (see FIGURE 3.13). This way, it is possible to update the client keys and/or the authority certificate at any time. Note that the name of the files can be changed from the configuration to match their relative path (is verified by the wrapper).

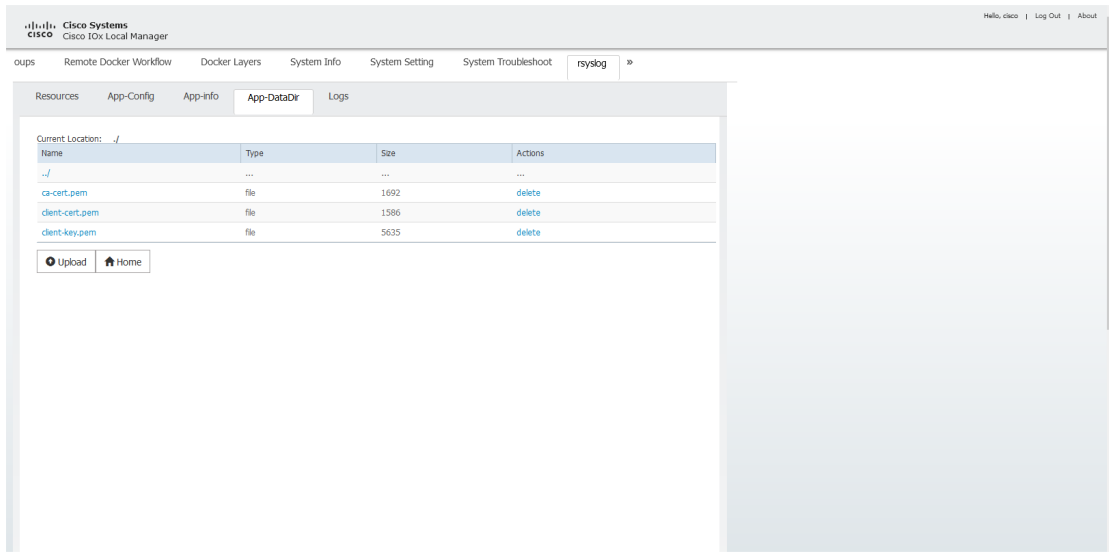


Figure 3.13: Data directory of the *syslog relay* application on Cisco IOx Local Manager, containing three *.pem* files

As outlined in the SECTION 3.1.1, the target router, IR1101, has minimalist (and shared) capabilities: 1255 CPU units, 862MB of DDR4 memory, and 701MB of flash storage memory. It is impossible at this stage to define a typical resource profile, mainly because the environment is not (yet) determined. Depending on the number of connected IOT devices, the message rate, size, frequency, but also the desired lifetime of the logs in case of internet outage, or even the speed of the forwarding process. Nonetheless, it is possible to define a series of standard *resource profiles* that would cover most of the needs in terms of functionality and performance. *The approach follows Cisco’s recommendation for resource allocation*[42].

Table 3.1: The 3 standard resource profiles

	Memory (MB)	CPU (units)
Small	64	200
Medium	128	400
Large	256	600

Depending on the profile, it will be possible to determine a set of parameters that best optimize the resources. Please note that the CPU units allocated to an application is the minimum guaranteed. However, memory is a hard limit and going beyond will result in a kill signal to the application[42].

Regarding the persistent storage, which is of no use for the application, it is set to its default value of 10 MB. As a note, Cisco advises limiting storage resources per application to about 20 MB[25] to minimize flash wear on the router.

3.2.2 Docker

Docker is an engine that automates application deployment into containers[43]. Its tooling and ecosystem can be used to develop applications for IOx, as seen in the previous section. Relying on such environment is key for reusability, portability, and scalability.

For ease of comprehension, the Docker image configuration is provided below:

```

❶ 1 FROM multiarch/qemu-user-static:aarch64 as qemu
❷ 2 FROM arm64v8/alpine:latest
3
4 ARG CONFIG_FILE
5 ARG EXTRA_CONFIG_FILE
6
7 MAINTAINER Egon Scheer <e.scheer@student.uliege.be>
8 LABEL Description="Reliable_syslog_relay_for_IOx"
9
10 COPY —from=qemu /usr/bin/qemu-aarch64-static /usr/bin
11
12 RUN echo "http://dl-cdn.alpinelinux.org/alpine/edge/main" \
13     >> /etc/apk/repositories
14
❸ 15 RUN apk —no-cache update \
16     && apk add —no-cache —purge —uU \
17     bash \
18     logrotate \
19     rsyslog \
20     rsyslog-tls \
21     rsyslog-mmnormalize \
22     rsyslog-elasticsearch \
23     tzdata \
24     && rm -rf /var/cache/apk/* /tmp/*
25
26 ENV TZ=UTC
27
28 COPY ${CONFIG_FILE} /etc/rsyslog.conf
29 COPY ${EXTRA_CONFIG_FILE} /etc/rsyslog.conf.d/
30
31 VOLUME [ "/var/lib/rsyslog" ]
32
33 COPY start.sh /
34
❹ 35 ENTRYPOINT [ "./start.sh" ]

```

Listing 3.1: Project's Dockerfile

One of the main constraints for running an application in IOx is that the architecture of the docker image must be identical to that of the router, in this case aarch64. Moreover, Docker only supports building for the host platform. To ease such limitation, the Dockerfile's base image ❶ is set to the emulator and virtualizer `qemu` which emulates the machine's processor and enables it to run a variety of guest operating systems.

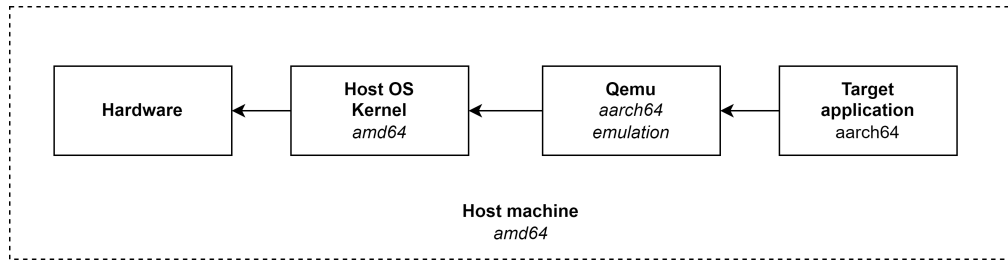


Figure 3.14: Qemu emulation workflow on an amd64 host machine.

To lessen the memory footprint of the Docker image, the minimal docker image based on Alpine Linux ❷ is set on top of qemu. This mechanism is called a multi-stage build and allows docker images to be layered on top of one another. The one below is referred to as the parent image and the bottom one is called the base image[43, p. 62].

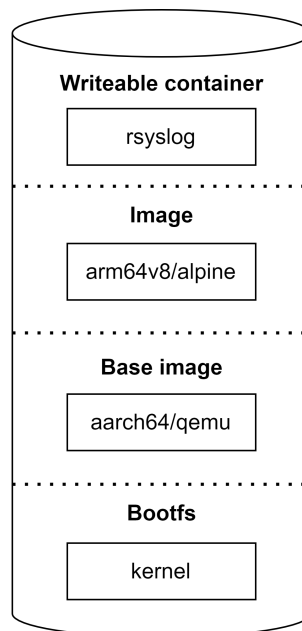


Figure 3.15: The project's Docker filesystem layers. Inspired from [43, p. 63].

To run rsyslog under the Alpine distribution for arm64, the daemon itself has to be compiled and packaged ❸ under this same architecture. Fortunately, the rsyslog engine and a large majority of its plugins are available, up to date, and installable

from the Alpine Linux package manager[26].

At last, the container entry point is bound to the wrapper shell ❹ presented in SECTION 3.2.2.

3.2.3 Rsyslog

Instead of recapping the implementation's in-depth explanation from SECTION 3.1.6, this section will focus on the configuration of rsyslog and the tuning of its parameters. The rsyslog daemon is configured through scripts called **RainerScript**[30, p. 31]. RainerScript is a programming language used to process network events and set event processors. For more information, the complete language manual is available here [30, p. 31].

Again, for ease of comprehension, rsyslog's main configuration is given below:

```

1  # INCLUDES
2  include (
3      file="/etc/rsyslog.conf.d/*.conf"
4      mode="optional"
5  )
6
7  # MODULES
8  module(
9      load="imudp"
❶ 10      threads='echo $UDP_THREADS'
❷ 11      timeRequery='echo $UDP_TIME_REQUERY'
❸ 12      batchSize='echo $UDP_BATCH_SIZE'
13 )
14
15 # LISTENER
16 input( type="imudp" port="514" ruleset="sendRemote" )
17
18 # DEFAULT RULE
19 global(
20     workDirectory="/var/lib/rsyslog"
21     DefaultNetstreamDriver="gtls"
22     DefaultNetstreamDriverCAFile='echo $CA_CERT_FILE'
23     DefaultNetstreamDriverCertFile='echo $CLIENT_CERT_FILE'
24     DefaultNetstreamDriverKeyFile='echo $CLIENT_KEY_FILE'
25 )
26
27 # TEMPLATE
28 template(

```

```

29     name="metadata_syslog "
30     type="string "
31     string="<%PRI%>1 [...] %timereported%] %msg%\n "
32 )
33
34 # MAIN MESSAGE QUEUE
35 main_queue(
4 36     queue.type='echo $QUEUE_TYPE'
5 37     queue.size='echo $QUEUE_SIZE'
6 38     queue.workerThreads='echo $QUEUE_WORKER_THREADS'
7 39     queue.dequeueBatchSize='echo $QUEUE_DEQ_BATCH_SIZE'
8 40     queue.discardMark='echo $QUEUE_DISCARD_MARK'
9 41     queue.discardSeverity='echo $QUEUE_DISCARD_SEVERITY'
42 )
43
44 # FORWARDING RULE
45 ruleset (name="sendRemote" parser=["rsyslog.rfc3164"]) {
46     action(
47         queue.type="direct "
48         name="send_remote "
49         type="omfwd "
50         template="metadata_syslog "
51         target='echo $REMOTE_HOST'
52         port='echo $REMOTE_PORT'
53         protocol="tcp "
54         action.resumeRetryCount="-1"
55         StreamDriver="gtls "
56         StreamDriverMode="1 "
57         StreamDriverAuthMode="x509/name "
58         StreamDriverPermittedPeers='echo $PERMITTED_PEERS'
59     )
60 }

```

Listing 3.2: Rsyslog main configuration file `rsyslog.conf`

The configuration is self-explanatory and does not require any particular comment. The remainder of this section will thus be dedicated to identifying, defining, discussing, and finally assigning a value or at least bounding the parameters. The parameters of interest are considered to be the set of variables that influence the performance or the policies (e.g. dropping strategy) on the syslog relay. The remaining parameters are for basic configuration and can be adjusted in the IOx local manager (see SECTION 3.2.1).

Firstly, the parameters related to the UDP input. It is possible to operate on

three parameters: the number of threads ❶, the rate at which the time is obtained ❷, and the size of a batch ❸.

1. The number of worker threads to process incoming messages might be needed for high throughput. UDP is connectionless and tend not to be as multithreaded as TCP servers[44, p. 423]. Because the OS' receive buffer is finite, threading can help improve performance and reduce message loss. It should be noted that when there are too many threads, performance might decrease.

The documentation advises[30, p. 156] to set the value up to the number of cores²² and has a hard upper limit at 32. We will thus consider **the interval** [1, 4].

2. Obtains the precise time only once every n-times. Because the kernel frequently returns the same time²³ twice when there has been no optimization[30, p. 167], **the value is thus set to 2**. The higher the value, the less accurate the timestamping becomes. *As a reminder, this parameter is intrinsically linked to the precision of the `timegenerated` field (see **SECTION 3.1.6**), and thus so is the order of the messages.*
3. The maximum number of UDP messages that can be obtained with a single OS call. A reasonably large batch size can lower system overhead and boost performance for systems with heavy UDP traffic. This option should not be used excessively, though. The manual advises against setting it higher than 128[30, p. 168]. The default value is 32. As a result, **the interval** [32; 128] **will be considered**.

Next is the primary message queue which accepts 6 parameters: the type of queue used ❹, the queue size ❺, the number of worker threads ❻, the size for batch dequeuing ❼, the discard threshold ❽, and the discard severity ❾.

4. The type of queue that will be used. It can either be a *fixed array* or a *linked list*. In the first data structure, pointers to queue entries are stored in a fixed, preallocated array. The pointer array is fairly compact on its own (the

²²The IR1101 datasheet does not provide the number of cores, but running the command "`show process CPU platform sorted`" on the router showcases 4.

²³Rainer Gerhards notes in [32, p. 6] that he frequently acquired the same timestamp for a relatively large group of messages: depending on message sizes, he may see this for several hundred messages.

majority of the space is taken up by the messages). A fixed array provides the highest run-time performance since it requires the fewest CPU cycles.

The second data structure is the polar opposite as every housekeeping structure is allocated dynamically. However, this guarantees that memory is only allocated when it is required, albeit adding some burden to runtime processing. Linked list are desirable for queues that face occasional message bursts.

To ensure the lowest possible memory footprint and in view of the relay's environment, the queue type is **set to linked list**.

5. The maximum number of messages in the queue. The value is expressed in terms of message and not memory size. The size of a message is determined mostly by its content and originator. Most messages should not take up more than 1KB of memory[30, p. 65]. Please keep in mind that each queue entry has an overhead of 8 bytes on 64bit systems. The documentation states that the queue size must be between 100 and 500,000[30, p. 65]. Considering a coefficient α of the RAM, dedicated to the good functioning of the rsyslog engine, p the resource profile used (p_{mem} in bytes), and 1KB per message, the queue can reasonably contain:

$$Q_{max}(\alpha, p) = \frac{(1-\alpha) \times p_{mem}}{1000} \text{ (in messages)}$$

Table 3.2: Maximum number of messages in queue with $\alpha = 0.25$.

	Queue size
Small	48,000
Medium	96,000
Large	192,000

The wrapper shell (see SECTION 3.2.2) ensures that the queue size can be allocated without exceeding the resources defined in the IOx application. For example, it is not possible to request a queue of 50,000 messages when there is only 32MB of allocated memory²⁴. This is necessary because the IOx mechanism (which is actually relayed to Docker using *cgroup*) deployed to control memory consumption is an *Out-Of-Memory killer*[45][42]: a last chance mechanism that is built into the Linux kernel in case of memory overflow and that kills the process at will. Following the example, if one considers a constant burst of more than 32MB, the engine will try to increase

²⁴Assuming that a message takes 1KB of memory, and not considering the cost of the engine for its own use, it would require at least 50MB of memory.

its queue but will be punished each time by the OOM (killed). Knowing that the engine startup can take several seconds²⁵, it is likely that it will loop on its startup without ever processing the slightest message. Unfortunately, rsyslog does not provide a feature to define a consumption limit. However, in the present situation, memory usage is mainly driven by the size of the queue. It is therefore essential to verify that it is within the margins of the available resources.

6. The maximum number of worker threads that can be run in parallel. Worker threads are started and stopped on an as-needed basis. Assuming that at time t there are k threads, a new $thread_{k+1}$ will spawn if the current queue size reaches $\frac{Q_{max}}{k}$ messages or more[30, p. 229]. *As a reminder, worker threads parse messages, build the output string, and forward them (please refer to SECTION 3.1.6 for more details).*

As the specification does not provide a bound for the maximum amount of workers, the parameter will be **set in the interval** $[1; ?]$.

7. The maximum batch size for dequeue operations. Larger batch sizes yield better results as it decreases locking calls. Depending on the current queue size, a batch might hold fewer messages (the parameter only defines the upper-bound). Using batching, the main memory might have more messages than the expected Q_{max} . Indeed, if the queue is full when a worker pulls a batch of messages (length B), the UDP input may completely refill the queue before the worker finishes its tasks, causing the RAM to contain $Q_{max} + B$ messages. In worst-case scenario, considering k workers and a maximum batch size B , the maximum queue size becomes:

$$Q_{max}(\alpha, k, B, p) = \frac{(1-\alpha) \times p_{mem}}{1000} - k \times B$$

Assuming that a coefficient γ of Q_{max} is allocated to batches, we can derive B_{max} :

$$B_{max}(\alpha, \gamma, k, p) = \frac{\gamma \times Q_{max}(\alpha, p)}{k} \text{ (in messages)}$$

8. The point at which rsyslog starts discarding *less significant* messages (defined by the parameter ⑨). Its value is expressed in number of messages and is bounded[30] by the interval $[0.8 \times Q_{max}; Q_{max}]$. This parameter does not influence performance, it simply allows messages to be prioritized in case of a full queue. This value is **set to the minimum**, $0.8 \times Q_{max}$.

²⁵Due to the structure allocation, connection with the central server, and more.

Table 3.3: Maximum batch size for dequeue with $\alpha = 0.25, \gamma = 0.05$.

	Queue size	Batch size
Small	45,600	$2400 \times k^{-1}$
Medium	91,200	$4800 \times k^{-1}$
Large	182,400	$9600 \times k^{-1}$

9. The Syslog severity level that defines the *less significant* messages. This parameter is left to the discretion of the user (editable in the IOx configuration). By default, **it is set to 4**.

Code	Severity
0	Emergency: system is unusable
1	Alert: action must be taken immediately
2	Critical: critical conditions
3	Error: error conditions
4	Warning: warning conditions
5	Notice: normal but significant condition
6	Informational: informational messages
7	Debug: debug-level messages

Figure 3.16: Syslog severity levels as defined by RFC 3164.

When the threshold of the parameter **8** is reached, all incoming and queued messages with a severity equal to or greater (numerically speaking) than indicated are discarded.

The two tables below summarize the possible values of the parameters (by profile, see TABLE 3.1), respectively those fixed and those dependent on other parameters and subject to an interval:

	UDP input	Main message queue		
	Time requery	Type	Discard mark	Discard sev.
All profiles	2	linkedlist	$0.8 Q_{max}$	4

Table 3.4: Values of the different fixed parameters.

	UDP input		Main message queue		
	Threads	Batch size	Queue size	Worker threads	Deq. batch size
Small	[1; 4]	[32; 128]	45600	$k \in [1; ?]$	$2400 k^{-1}$
Medium	[1; 4]	[32; 128]	91200	$k \in [1; ?]$	$4800 k^{-1}$
Large	[1; 4]	[32; 128]	182400	$k \in [1; ?]$	$9600 k^{-1}$

Table 3.5: Values of the different dependent parameters with $\alpha = 0.25, \gamma = 0.05$.

3.3 Log analysis

Following the analysis carried out by Rajiullah et al.[24] on the syslog messages of one of their servers in their university department, it seems relevant to transpose their approach to this situation. Unfortunately there is no public dataset at this stage that is specific to the syslog messages of one (or more) IOT devices²⁶. It is therefore difficult to derive their characteristics, especially as they are extremely sensitive to the environment. Indeed, let us take for example Loghub[47] which is a large collection of publicly available system log datasets. One can see on FIGURE 3.17 that, depending on the type of system, the quantity of messages for the same duration varies enormously. Moreover, even within the same system there can be a huge disparity.

System	Description	Time Span	#Messages	Data Size	Labeled
<u>Distributed systems</u>					
HDFS	Hadoop distributed file system log	38.7 hours	11,175,629	1.47GB	Yes
	Hadoop distributed file system log	N.A.	71,118,073	16.06GB	No
Hadoop	Hadoop mapreduce job log	N.A.	394,308	48.61MB	Yes
Spark	Spark job log	N.A.	33,236,604	2.75GB	No
Zookeeper	ZooKeeper service log	26.7 days	74,380	9.95MB	No
OpenStack	OpenStack infrastructure log	N.A.	207,820	58.61MB	Yes
<u>Supercomputers</u>					
BGL	Blue Gene/L supercomputer log	214.7 days	4,747,963	708.76MB	Yes
HPC	High performance cluster log	N.A.	433,489	32.00MB	No
Thunderbird	Thunderbird supercomputer log	244 days	211,212,192	29.60GB	Yes
<u>Operating systems</u>					
Windows	Windows event log	226.7 days	114,608,388	26.09GB	No
Linux	Linux system log	263.9 days	25,567	2.25MB	No
Mac	Mac OS log	7.0 days	117,283	16.09MB	No
<u>Mobile systems</u>					
Andriod	Andriod framework log	N.A.	1,555,005	183.37MB	No
HealthApp	Health app log	10.5 days	253,395	22.44MB	No
<u>Server applications</u>					
Apache	Apache web server error log	263.9 days	56,481	4.90MB	No
OpenSSH	OpenSSH server log	28.4 days	655,146	70.02MB	No
<u>Standalone software</u>					
Proxifier	Proxifier software log	N.A.	21,329	2.42MB	No

Figure 3.17: Loghub datasets details[47].

For example, looking at the Linux OS logs[47], which contain 264 days and more than 25,000 messages, we observe after parsing that the average message size is 92 bytes, its content is 70 bytes and the inter-arrival time is 1250 seconds (~ 21 min.).

²⁶Although there seems to be a huge interest[46] and popularization in IOT datasets, especially for risk and attack assessment, there are none that focus on the syslog messages they produce.

Unfortunately, there is no information on the priority of messages. While the first two data points are consistent with the Rajiullah et al. study, the inter-arrival time is much larger than its counterpart (which was $\sim 1.5s$). The reason for this is simple: the data retrieved by the study[24] was from a server and not a source. Indeed, the more senders there are, the shorter the interval between any two messages. To support this argument, let us analyse together the *Linux operating system* dataset (from Loghub’s collection) which in view of its metrics could be transposed to an IOT device behaviour. By plotting the number of messages per day, as depicted in FIGURE 3.18, it becomes clear that the messages from the Linux machine are actually generated in groups and not in a constant fashion.

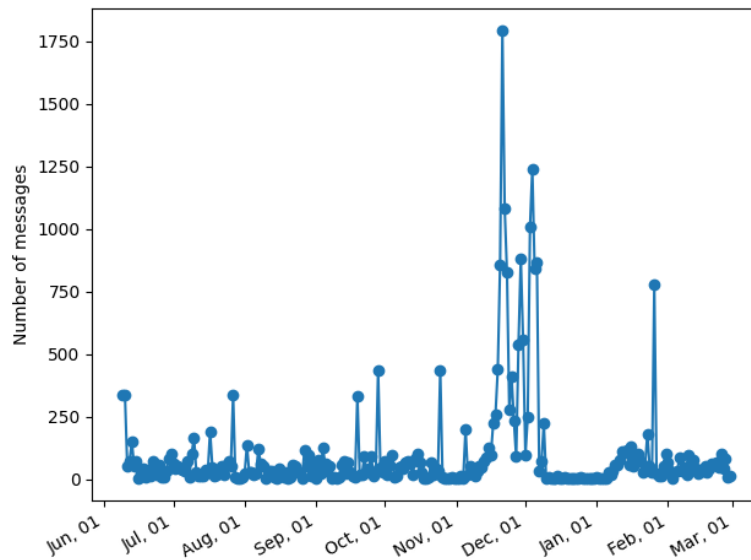


Figure 3.18: Number of messages per day based on the *Linux* dataset.

By fluctuating the minimum interval between two groups, which was previously arbitrarily set at one day, the ideal interval can be established: that is, the one that gathers the most messages per group, that separates the groups as good as possible in terms of time space, and that within its group has a low message inter-arrival. FIGURE 3.19 outlines these criteria.

The ideal grouping interval region seems to be between 15 minutes and two hours. Above these values the inter-arrival within the groups explodes and those below shred the grouping. As such, the one-hour interval seems to be a good compromise between size and sparseness. Indeed, the average number of logs in a group is 27, the average interval between two groups is 32,442 seconds (~ 9 hours), and the average inter-arrival between two messages is 57 seconds.

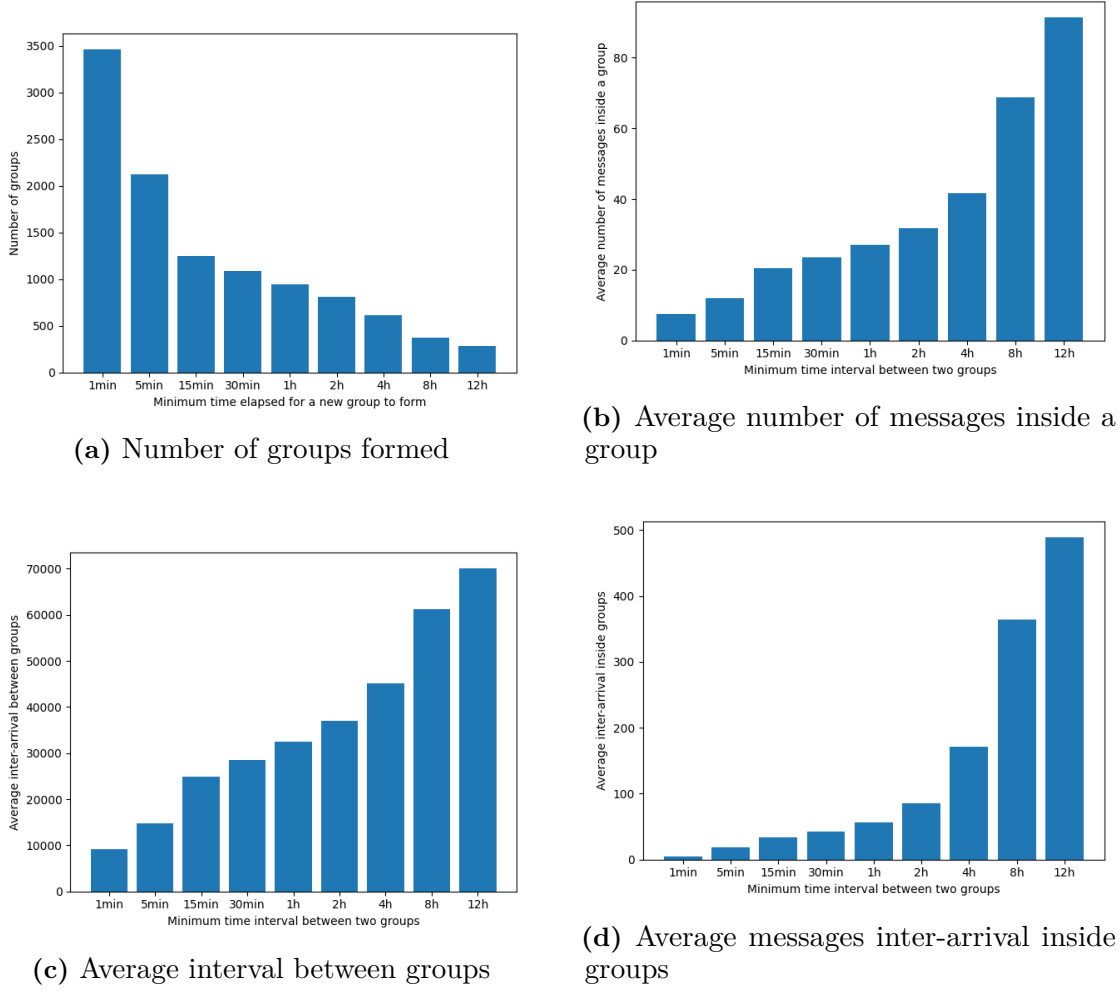


Figure 3.19: Metrics (y axis) derived from the different ways of bundling messages into groups according to the minimum elapsed time required between two consecutive groups (x axis).

This set of metrics is valuable because another way of looking at groups is to consider them as burst periods, which can be used to model a bursty traffic using Markov chain (to name one). Moreover, this analysis shows the importance of a dataset, understanding its origin, and especially its suitability in the context to which it will be applied.

In theory, theory and practice are the same. In practice, they are not.

— Benjamin Brewster [48, p. 202]

4

Evaluation

Contents

4.1	Purpose	51
4.2	Environment setup	52
4.2.1	Testbed specifications	55
4.2.2	Testbed parameters	57
4.3	Functional tests	58
4.3.1	Reliability	58
4.3.2	Store and forward	62
4.3.3	Chronological order	65
4.4	Performance	68
4.4.1	Cellular networks	69
4.4.2	High-speed low-latency network	71
4.4.3	Queue build-up prediction	76
4.4.4	Results	79

4.1 Purpose

This chapter is dedicated to defining and creating ways to ensure that the solution meets its design, behaves as intended, and how well it does so. The idea behind this evaluation is to answer two main questions:

1. Does the implementation follow the key objectives of the project?
2. How well does it perform?

The former question will be answered by a set of scenarios aimed at undermining the key objectives. The latter will be evaluated by operating on a set of variables—the network state, message behaviour, implementation configuration, and many others—for 3 standard profiles to determine their respective performance.

4.2 Environment setup

An environment that is both cohesive and, most importantly, practical is required to conduct these tests correctly. Something that is replicable for the readers. It should ideally be portable and independent of any operating system. The testbed is presented below.

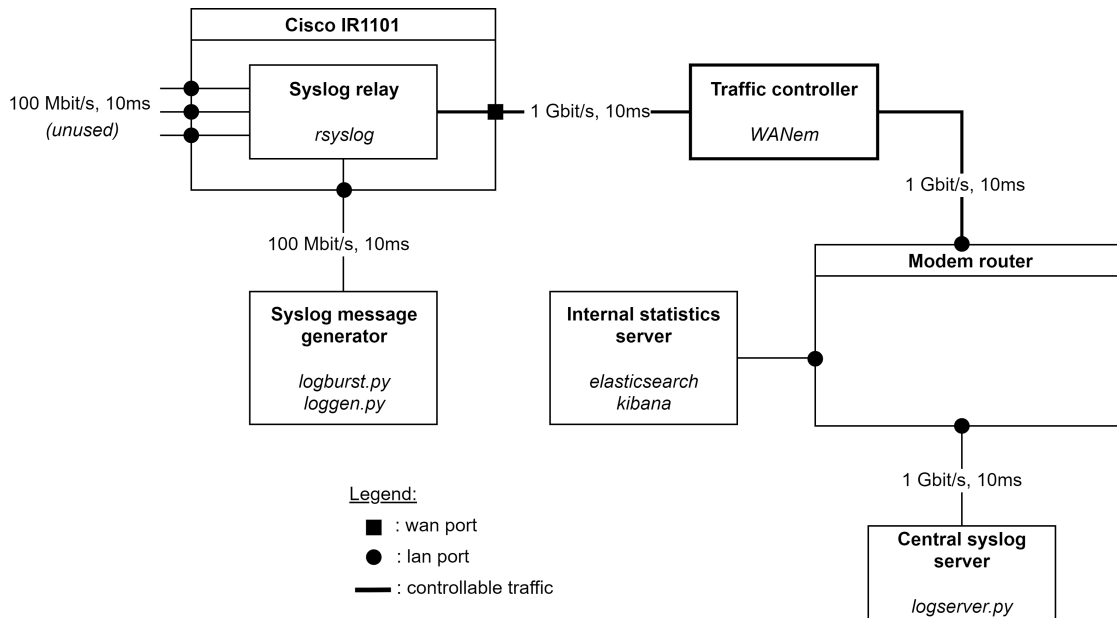


Figure 4.1: Testing environment topology.

This approach closely follows the architecture designed in SECTION 3.1.5, but with some slight additions and simplifications where possible. In terms of simplifications, the main ones are:

1. The central syslog server implementation (called **logserver**) must not be the bottleneck and therefore only does the bare minimum: no format checking and minimal parsing. Only statistics (internal counters) are returned periodically to be informed about the current state of the server. This shortcut offers a way to compete with enterprise configurations that can handle huge amounts of messages.

2. The procedure for sending logs from IOT devices is greatly simplified. Given that the messages arrive in UDP, and therefore no connection or response is required, and that no disparity is made on the basis of the emitter, it is reasonable to think that sending from a single source or from several does not change much¹ in this test environment. This solution makes it easy to increase the number of devices but also to decouple on the other LAN ports of the Cisco router if the 100Mbit/s limit is reached.

Two modes are available for message generation. The first (called **loggen**) follows the distributions and probabilities detailed in SECTION 3.3 to generate messages for n devices. Each device has its own clock (randomizable). A sequence number (atomic variable) is affixed to the messages to trace them and record the order. The seed for randomness is configurable. The generator can be set to bursty mode which simulates standby periods followed by bursts (as discussed in SECTION 3.3). It follows a two-state Markov model as shown on the figure below, with $p = \frac{57}{3244} \approx 0.00175$ and $q = \frac{1}{27} \approx 0.037$

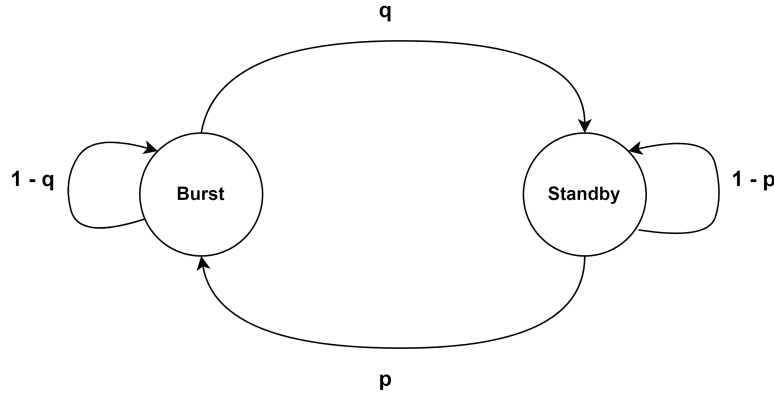


Figure 4.2: A 2-stage Markov chain that captures burst behaviour.

The second mode (called **logburst**), provides a volume of messages (custom size) per second for just-in-time testing. To ensure the highest throughput, the message is already pre-built and encoded, and does not vary for the whole burst duration (configurable). If necessary (has a rate loss), it is possible to insert the sequence number and/or generate important messages (with a given probability, but assignment is deterministic). The *token bucket* algorithm, visible below, is employed to ensure that message transmissions conform to defined limits on bandwidth and burstiness.

¹As long as the source is capable of providing the equivalent in terms of message rate for n devices.

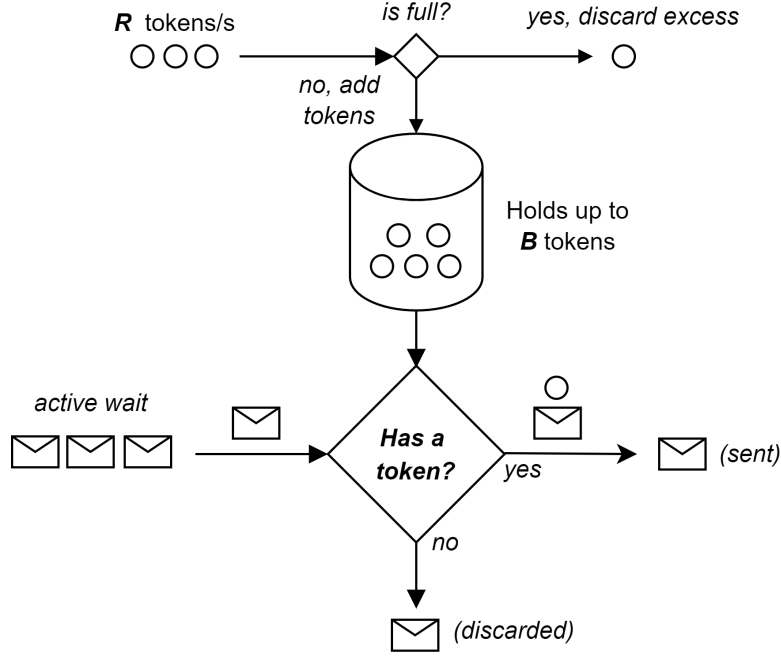


Figure 4.3: The token bucket algorithm which allows or denies messages depending on the levels of traffic required.

As for the additions made to the original architecture, mainly to improve the quality of the evaluation, the following can be noted:

1. While the two main actors of the evaluation—the message generator and the server— will provide valuable measurements, it is not possible at this stage to know the state of the relay which would be detrimental for the performance phase. Fortunately, `rsyslog` provides a module to periodically dump the internal counters. It is sufficient to provide an HTTP endpoint to feed it with the data. For simplicity, the server is powered by the *Elasticsearch*[49] engine, in combination with the *Kibana*[50] visual engine, in order to remotely manipulate² the metrics through a web interface. The service runs in a Docker container. The module can be activated via the configuration. By default, the interval between two reports is 10 seconds.
2. For these tests to be meaningful, it is necessary to have an emulation of the *Wide Area Network* (WAN) as close as possible to reality: network delay, packet loss, packet corruption, disconnections, packet re-ordering, jitter, etc. To this end, a controller is positioned between the Cisco router and the modem

²The tool will be used to reconstruct events, isolate tests and quickly iterate on the data to find interesting graphs. Once done, these are converted to `.csv` and processed by a script using the `matplotlib` library to produce quality renderings.

to simulate specific link qualities. It is powered by *WANem*[51] (Wan Area Network EMulation), a software running on *Knoppix*[52], a Debian-based Linux distribution. It can be executed from a bootable USB stick or inside a VM. After configuring the interfaces to be bridged and assigning an IP to the bridge, it is possible to interact with the software remotely from a web interface as illustrated on **FIGURE 4.4**. Note that rules can be restricted to a specific flow only, e.g. based on the source or destination IP address.

Interface: eth0		Packet Limit: 1000 (Default=1000)		Symmetrical Network: Yes	
Bandwidth	Choose BW	other		Other: Specify BW(Kbps) 0	
Delay		Loss		Duplication	
Delay time(ms)	0	Loss(%)	0	Duplication(%)	0
Jitter(ms)	0	Correlation(%)	0	Correlation(%)	0
Correlation(%)	0			Correlation(%)	0
Distribution	-N/A-			Gap(packets)	0
Idle timer Disconnect		Type	none	Idle Timer	
Random Disconnect		Type	none	MTTF Low	MTTF High
Random connection Disconnect		Type	none	MTTF Low	MTTF High
IP source address	any	IP source subnet		IP dest address	any
				IP dest subnet	
				Application port if any	any

☐ Display commands only, do not execute them

Figure 4.4: WANem software advanced configuration panel.

Behind the curtain, WANem uses the *tc*[53] utility to configure the kernel packet scheduler. It is important to have an intermediary who sits between the two networks so that it can shape traffic going to the relay as well as out to the central syslog server. Moreover, as stated in [54], it is easier to create traffic control rules for traffic flowing out of an interface because one can control when the system sends data, whereas controlling when one receives data requires the creation of a second intermediate queue to buffer incoming data (much slower).

4.2.1 Testbed specifications

To achieve a maximum transparency in this testbed, and thus minimize environment bias, the specifics of the software, hardware, and network used are listed in **TABLES 4.1** and **4.2**.

As far as the network is concerned, all the links in the testbed are Ethernet connections, either via an on-board port, USB 3.0 or a NIC card, all of which

Table 4.1: Software specifications.

	Specifications	Comments
Message generator and central syslog server	Python 3	Dep. in <code>requirements.txt</code> .
Cisco router	Cisco IOS XE 17.8.1	/
Rsyslog engine	v.8.2206.0-r0	Compiled under aarch64.
WANem	v.3.0 beta 2	From bootable usb.
Elasticsearch & Kibana	v.8.3.3	/

Table 4.2: Hardware specifications.

	Specifications
Cisco router	1 1Gbit WAN and 4 100Mbit LAN, spec. details here 3.1.1.
Modem router	Bbox 3v+ with a 400MHz Dual Core, 256MB of RAM, and has 1 1Gbit WAN and 4 1Gbit LAN (see [55]).
Message generator	Intel Core i5-7200U 2.5GHz, 8GB DDR4 2133MHz, and has 1 USB3 Gbit ethernet port. Linux distribution.
Traffic controller	Intel Core i7-2600 3.4GHz, 8GB DDR3 1333MHz, and has 2 1Gbit ethernet port. Linux distribution.
Central syslog server and statistics server	AMD Ryzen 5 3600 (6-Core) 3.59 GHz, 16GB DDR4 2400 MHz, and has 1 1Gbit Ethernet port. Linux distribution. <i>Due to a lack of available machines for this testbed, the internal statistics server runs inside a bridged VM with 6144 GB of RAM and 2 core allocated.</i>

guarantee a 1Gbit/s downstream rate. However, in order to certify the quality of the network and establish practical bandwidth limits, multiple measurements were taken using `iperf3`[56] and *My Traceroute*[57] (MTR) tools.

A first measurement was made between the message generator and the IOx application³ in UDP, via the "`iperf3 -u -c <IP> -b 200M`" command, and shows an average bitrate of 95.6 Mbit/s. The script (`logburst`) inside the message generator can produce an average of 310,000 messages/s (of 256 bytes each), which is equivalent to 793.6 Mbit/s and is well beyond the link limit. With the sequence number affixed, this drops to 270,000 messages/s or 691.2Mbit/s.

The second test was performed between the IOx application and the central syslog server, firstly to determine the link capacity (via `iperf3` in TCP), which provided an average bitrate of 240 Mbit/s. This is far from the theoretical bandwidth, probably

³Tests involving interacting with the IOx application shell are done in ssh and then via the "`app-hosting connect app <app-name> session`" command to connect to the container.

due to the traffic controller (and the modem router) posted between the two, but still more than sufficient for the testbed (cellular edge or 3G networks). Another measurement was conducted, again between the same machines, but this time to check that the controller is functional. We set the maximum bandwidth to 1Mbit/s, the loss rate to 1%, and a delay of 250ms. The iperf3 tool (in TCP mode) showed us that the average bitrate is 950 Kbit/s. Using MTR, which combines traceroute and ping⁴ tool, it indicated after a thousand ICMP messages a loss rate of 1.1% and an average delay of 250ms. This confirms the proper functioning of the traffic controller. *Note that this verification will be carried out before each test (functional or performance) to ensure the correctness of the environment.*

4.2.2 Testbed parameters

This section contains the set of parameters that will be varied throughout the evaluation. The engine variables, as discussed in SECTION 3.2.3, are a prime example. Below is a list of these *interaction levers*:

1. The quality of the network is one of the most influential parameters in the evaluation and should be as close to reality as possible. Given that in practice the router will rely on a cellular network to access the Internet, we will borrow the values from the WebRTC⁵ project's source code to simulate an EDGE and 3G network. *The numbers below were initially collected from Google data[58].*

Table 4.3: Four configurations containing the characteristics of a network connection.

	Rec. bandwidth	Send bandwidth	Delay	Packet loss
3G, Average Case	780Kbit/s	330Kbit/s	100ms	0%
Edge, Average Case	240Kbit/s	200Kbit/s	400ms	0%
3G, Lossy Network	780Kbit/s	330Kbit/s	100ms	1%
Edge, Lossy Network	240Kbit/s	200Kbit/s	400ms	1%

2. Three resource profiles have been defined in SECTION 3.2.1. They are parameters that determine the share of resources allocated to the IOx application (CPU, RAM, and storage). They will be used in the performance tests as a baseline to categorize three types of usage. After the evaluation, each profile will have an ideal set of parameters associated with it.

⁴MTR relies on *Internet Control Message Protocol* (ICMP) "Time Exceeded" and "Echo Reply" messages to compute network metrics.

⁵WebRTC (*Web Real-Time Communication*) is an open-source API used in various web browsers to enable real-time communication. Its usefulness in this context is that it offers different profiles to simulate a network (is featured in Google Chrome's developer tools, Firefox's inspector, etc.).

3. The message generator can also be seen as a parameter, mainly to alternate between the *realistic* (`loggen`) mode and the *burst* (`logburst`).
4. And last but not least, the parameters of the `rsyslog` engine, presented in details in SECTION 3.2.3.

4.3 Functional tests

This section is dedicated to the assessment of the key objectives defined in SECTION 3.1.3. Despite the fact that these criteria were used to guide the implementation decisions, it still is necessary to confirm that they are upheld in actual usage.

4.3.1 Reliability

As highlighted many times—see SECTION 2.1.3 2.1.3 3.1.3, the relay is not a hundred percent reliable because a variety of situations do not guarantee reliability for the messages: abrupt crash of the router, use of the UDP transport protocol by the senders, loss of connection with the relay, absence of acknowledgment on the application layer, hardware limitation of the router, and many others. In the present context, a few mechanisms exist to mitigate these issues: pipeline all queues to a permanent storage, dump messages to a *Network Attached Storage* (NAS) during downtime, use an application layer protocol such as RELP instead of TCP, or even additional central syslog server(s) for failover. The FIGURE 4.5 below highlights these areas of unreliability (in bold). *Note that the (abrupt) shutdown of the server is not depicted because otherwise the whole flow would simply be unreliable (except for the main queue that could be spared if piping through the disk).*

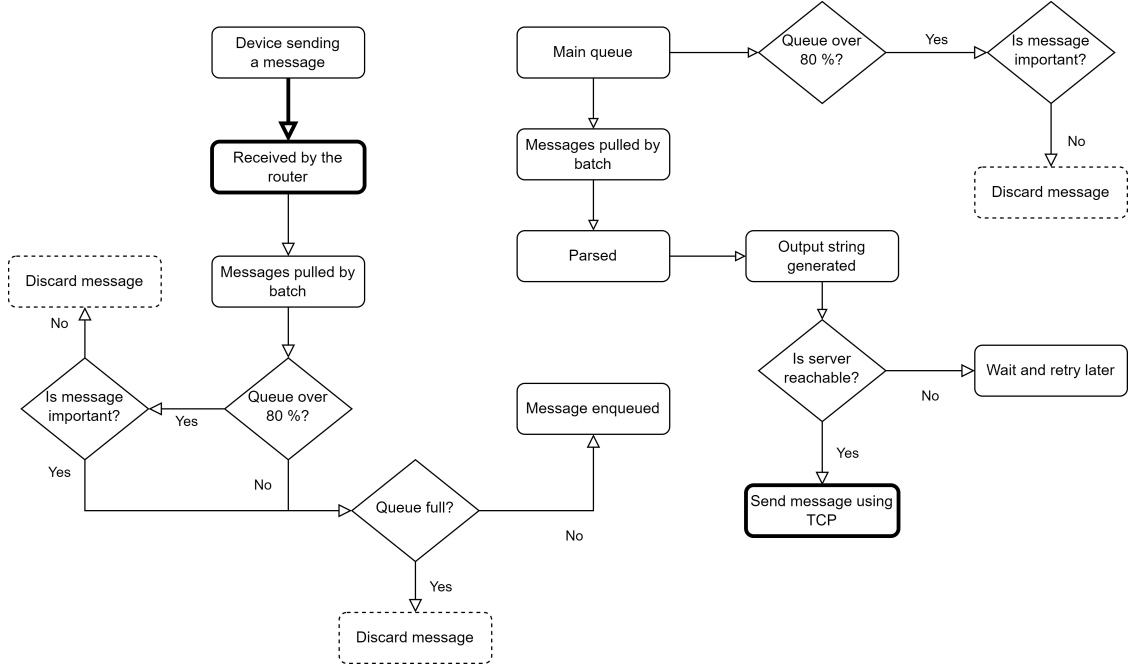


Figure 4.5: Relay’s messages flowchart, highlighting unreliable areas in *bold* and the dropping strategy in *dotted-line*.

Dotted-lines illustrates the dropping strategy in place to optimize (the limited) space in the event of a burst: the *important* messages are given precedence and others are discarded. This behaviour is recommended[5, p. 27] by the RFC 5424 standard and its effectiveness is discussed in the following section (see 4.3.2).

Although these areas of unreliability cannot be ignored, they represent a minimal disruption to the daily message traffic and transmission process. The example below demonstrates the proper functioning of the relay. Messages are generated using the `logburst` script with the *small* resource profile and the *lossy 3G* network configuration. At this point, it is impossible to determine the ideal parameters, thus we will arbitrary choose the minimum values of TABLE 3.5. The script is set to 100 messages/s (of size 256 bytes) and with sequence number affixed to messages. *I would like to remind readers that the aim here is not to highlight performance but simply to showcase the reliability of the system.*

In order to demonstrate reliability, the generator affixes sequence numbers to messages which enables the server to identify any loss. Running the test for 300 seconds showed no loss as visible on FIGURES 4.6 and 4.7.

```

Statistics: msg-count=17506, impt-msg-count=0, time=201 (sec), avg-rate=87.0945 msg/sec
Statistics: msg-count=18514, impt-msg-count=0, time=211 (sec), avg-rate=87.7441 msg/sec
Statistics: msg-count=19502, impt-msg-count=0, time=221 (sec), avg-rate=88.2443 msg/sec
Statistics: msg-count=20510, impt-msg-count=0, time=231 (sec), avg-rate=88.7879 msg/sec
Statistics: msg-count=21504, impt-msg-count=0, time=241 (sec), avg-rate=89.2282 msg/sec
Statistics: msg-count=22509, impt-msg-count=0, time=251 (sec), avg-rate=89.6773 msg/sec
Statistics: msg-count=23504, impt-msg-count=0, time=261 (sec), avg-rate=90.0536 msg/sec
Statistics: msg-count=24513, impt-msg-count=0, time=271 (sec), avg-rate=90.4539 msg/sec
Statistics: msg-count=25511, impt-msg-count=0, time=281 (sec), avg-rate=90.7865 msg/sec
Statistics: msg-count=26512, impt-msg-count=0, time=291 (sec), avg-rate=91.1065 msg/sec
Statistics: msg-count=27512, impt-msg-count=0, time=301 (sec), avg-rate=91.402 msg/sec
Statistics: msg-count=28513, impt-msg-count=0, time=311 (sec), avg-rate=91.6817 msg/sec
Statistics: msg-count=29520, impt-msg-count=0, time=321 (sec), avg-rate=91.9626 msg/sec
Statistics: msg-count=30099, impt-msg-count=0, time=331 (sec), avg-rate=90.9335 msg/sec
Statistics: msg-count=30099, impt-msg-count=0, time=341 (sec), avg-rate=88.2669 msg/sec
Statistics: msg-count=30099, impt-msg-count=0, time=351 (sec), avg-rate=85.7521 msg/sec
Statistics: msg-count=30099, impt-msg-count=0, time=361 (sec), avg-rate=83.3767 msg/sec
Statistics: msg-count=30099, impt-msg-count=0, time=371 (sec), avg-rate=81.1294 msg/sec
Statistics: msg-count=30099, impt-msg-count=0, time=381 (sec), avg-rate=79.0 msg/sec
Statistics: msg-count=30099, impt-msg-count=0, time=391 (sec), avg-rate=76.9795 msg/sec
Statistics: msg-count=30099, impt-msg-count=0, time=401 (sec), avg-rate=75.0599 msg/sec

```

Figure 4.6: Central syslog server's console after receiving 30,100 messages.

```

Statistics: count=28115, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28217, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28319, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28421, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28523, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28625, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28727, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28829, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28931, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29033, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29135, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29236, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29337, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29438, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29538, impt-counter=0, rate=100 msg/sec, throughput=0.0256 MB/sec
Statistics: count=29639, impt-counter=0, rate=100 msg/sec, throughput=0.0256 MB/sec
Statistics: count=29740, impt-counter=0, rate=100 msg/sec, throughput=0.0256 MB/sec
Statistics: count=29841, impt-counter=0, rate=100 msg/sec, throughput=0.0256 MB/sec
Statistics: count=29942, impt-counter=0, rate=100 msg/sec, throughput=0.0256 MB/sec
Statistics: count=30043, impt-counter=0, rate=100 msg/sec, throughput=0.0256 MB/sec
Cancelling the token...
Statistics: count=30099, impt-counter=0, rate=55 msg/sec, throughput=0.0141 MB/sec

```

Figure 4.7: Message generator's console after sending 30,100 messages (300s timeout).

The internals of the relay is outlined below. As one can see on FIGURE 4.8 no messages were lost nor dropped. As a reminder, the *forwarding action* (the solid line on the figure), explained in detail in SECTION 3.1.6, takes care of pulling the messages from the main queue, parsing them and transforming them into formatted strings, then sending them to the TCP layer. *For the curious readers, you can also see the effect of a single thread on the size of the main queue (one producer for one consumer), it almost never has time to fill. A queue size close to zero indicates that the flow is tense and fast (no busy waiting).*

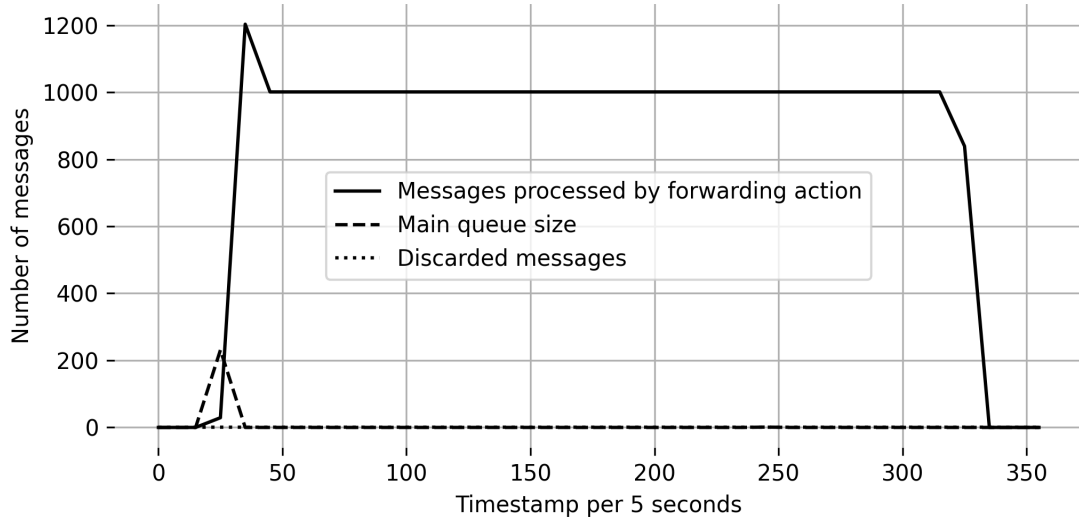


Figure 4.8: The relay's internal counters evolution over the 300s test and a rate of 100 messages/s.

The second graph (FIGURE 4.9) highlights the cumulative sum of the input⁶ and output⁷ counters which as expected add up to 300,100 messages.

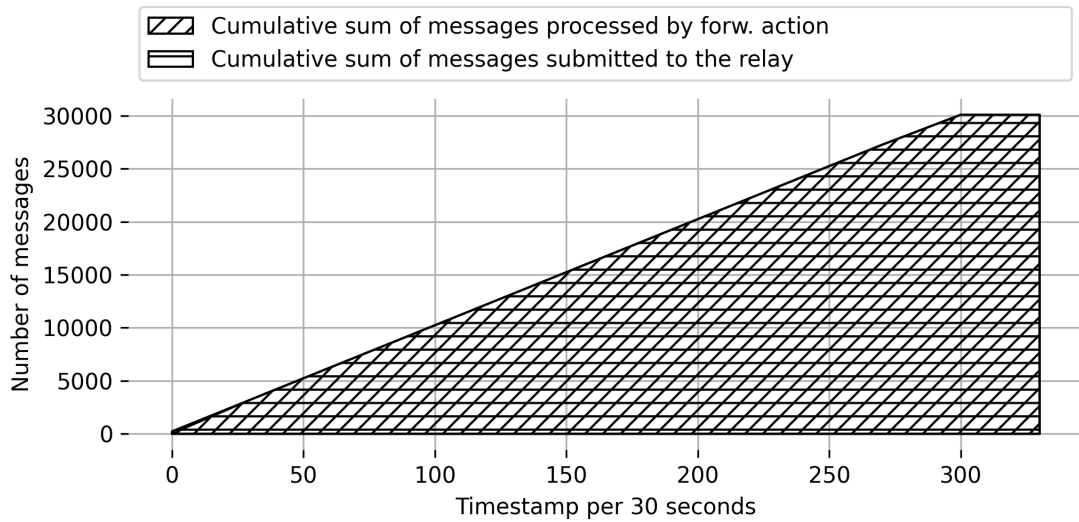


Figure 4.9: Cumulative sum of the relay's input and output counters. Both have a maximum value of 300,100 messages.

Although this experience is a tad meagre, the remaining set of tests will also serve as evidence of the correctness of the reliability process.

⁶The number of messages pulled from the UDP buffer by `recvmsg()`[59], which supports multiple datagrams per syscall, and submitted to the main queue.

⁷The number of messages pulled and processed by the forwarding action.

4.3.2 Store and forward

As mentioned above, guaranteeing reliability is a challenging task, but it is still possible to be competitive, within the limits of the available (hardware) resources, in transmitting messages to the destination. Without going into detail⁸, the idea behind store and forward is to use the available memory resources to store as many messages as possible when the server is unreachable. Coupled with the above is a dropping strategy to maximize the quality⁹ of the information stored. We will first test the dropping mechanism and then the management of the relay during a downtime. The configuration of the testbed is identical to that of SECTION 4.3.1.

If we define r as the number of messages sent per second, t as the time in seconds, d as the discard mark between 0 and 1, s as the maximum queue capacity, and finally p as the probability of having an important message, then the size of the queue at time t necessarily follows this equation:

$$Q(r, t, d, s, p) = \min\{rt, ds\} + \min\{\max\{rt - ds, 0\} \times p, (1 - d)s\} \quad (4.1)$$

Based on this, we can define the number of messages discarded at time t :

$$D(r, t, d, s, p) = \begin{cases} 0 & \text{if } Q(..) \leq ds, \\ rp & \text{if } ds < Q(..) < s, \\ r & \text{otherwise.} \end{cases} \quad (4.2)$$

From 4.1 and 4.2, a simulation of the dropping strategy is drawn:

⁸The options have been developed extensively in SECTION 3.1.3.

⁹Quality in the sense of message importance, defined by the priority of the message, or more precisely the severity it contains. The closer it gets to 0, the more its value as information increases.

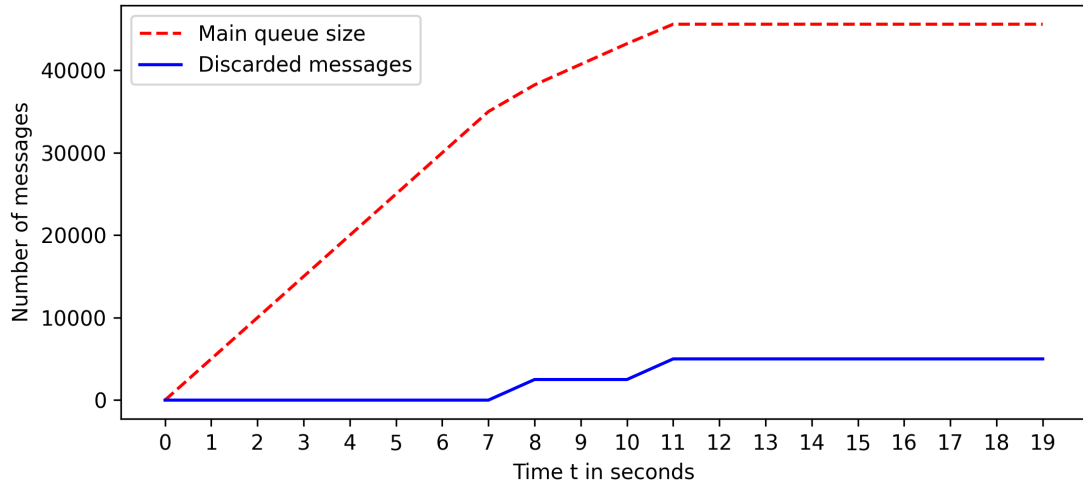


Figure 4.10: Simulation of the dropping strategy for 20 seconds with $r = 5000$, $d = 0.8$, $s = 45600$, $p = 0.5$. We can see very well the staircase effect induced by the dropping strategy (solid line).

With this in mind, a test with the same parameters as in FIGURE 4.10 and a probability of *important*¹⁰ a message of 0.5 was launched. To maximize the accuracy of the statistics counters, the interval between two reports has been set to 1 second.

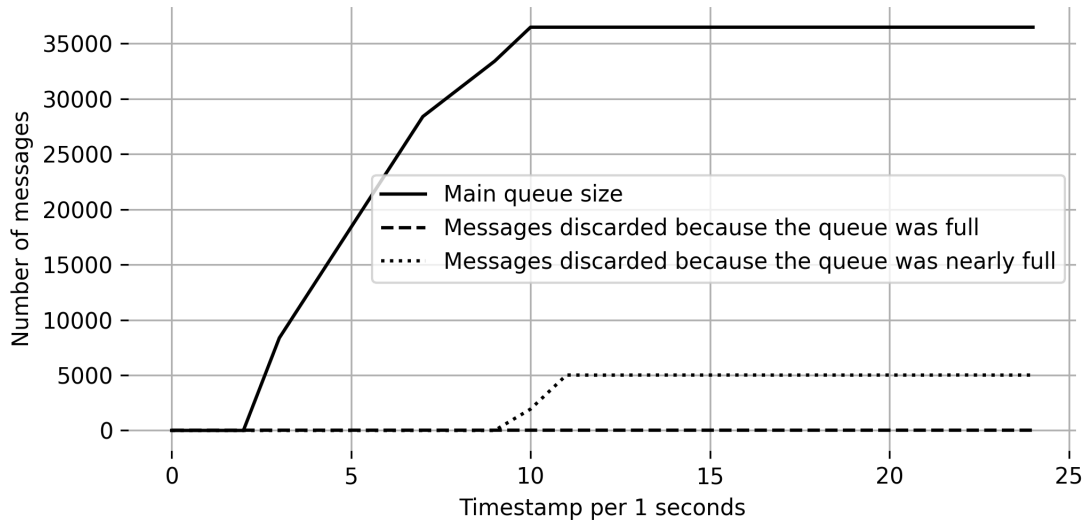


Figure 4.11: The relay's queue size evolution second by second, with the dropping strategy threshold set to 80% of the queue. Same parameters as FIGURE 4.11.

Unfortunately, as can be seen in FIGURE 4.11, the dropping strategy does not behave as it should. The first indication of a malfunction is the absence of the staircase

¹⁰The queue size will never exceed 80% if no important message arrives.

effect that should normally occur between seconds 7 and 11, but the primary evidence is that the queue never reaches 45,600 messages while important messages are continuously sent (and received). Everything suggests¹¹ that this problem is not related to the implementation but to rsyslog's engine itself. Several avenues such as changing the version of the engine, using a different message generator, analysing the messages at the router entrance and in the application, or downgrading to a minimalist configuration did not bear fruit. An issue has been opened on the rsyslog Github with the tracker ID #4960¹². The upside is that this feature is a strategy and only influences the "quality" of the messages stored in the queue. Indeed, this mechanism is independent of all other relay functionality and can be activated or deactivated at any time. *Please note that from now on and until the end of this paper, tests will be performed without the dropping strategy.*

On the other hand, the downtime is flawlessly¹³ managed as indicated by this graph showing the reconnection attempts to the central syslog server. Timestamps are based on the logs provided by the IOx application during a downtime of ~ 10 hours:

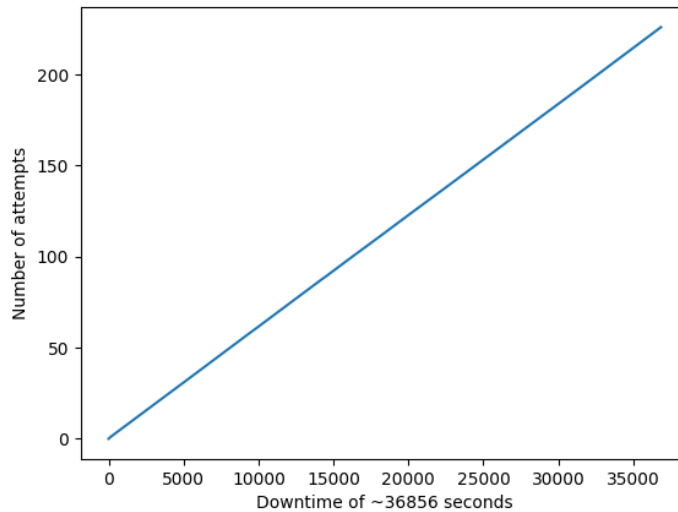


Figure 4.12: Number of reconnection attempts made during a downtime period of more than 10 hours. Resume interval of 30 seconds with a ceiling of 1800 seconds.

¹¹Although the rsyslog engine parses the messages perfectly, the dropping mechanism considers that each message is of severity 7 and can therefore be dismissed (whereas the discard severity in the configuration of the IOx application has been set at 4). In addition to not being able to differentiate messages, the direct consequence is that the queue never fills up completely.

¹²See <https://github.com/rsyslog/rsyslog/issues/4960> to find out more about the issue's current status.

¹³Follows the line shown in FIGURE 3.9 with $C = 1800$, $R = 30$.

Finally, a test is conducted to assess the message recovery after a period of downtime. Messages are generated using the `logburst` script with the *small* resource profile and the *lossy 3G* network configuration. The central syslog server is shut down, 100 messages are sent for 60 seconds, followed by 120 seconds of waiting before the server is turned back on. The result graph can be visualized in FIGURE 4.13.

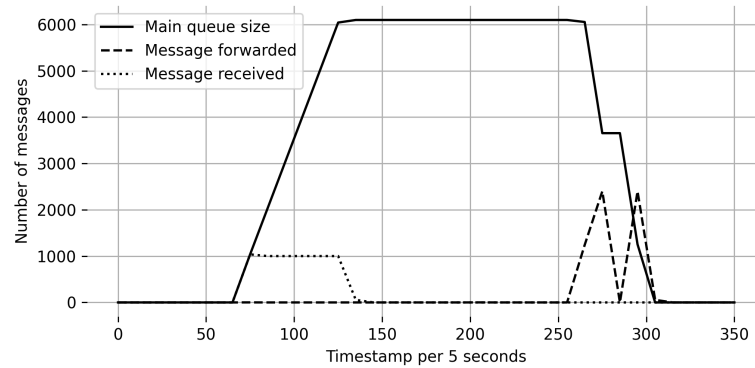


Figure 4.13: Behaviour of the relay during a catch-up test. 100 messages are sent every second for 60 seconds. After 120 seconds, the central server is switched on.

As expected, the relay behaved correctly and retained all 6,099¹⁴ messages sent during the outage, then successfully transmitted them upon reconnection to the central syslog server.

4.3.3 Chronological order

The problem of chronological order was studied in detail in SECTION 3.1.7 and highlighted the limitations of "orderability" arising from the poor accuracy of the timestamp contained in incoming messages. If it had been possible to obtain the time of creation of the messages to the thousandth of a second¹⁵, it would have been conceivable to estimate[60] the internal clock of the devices using the time of arrival in the relay and the message's delay. Various methods exist to estimate the one-way delay but they all require[36] a reply from the IOT device. The simplest[36] would be for example to send an ICMP message of type "Echo Reply"[61] message (but requires TCP/UDP port 7 open).

¹⁴Normally there should only be 6,000 messages but the message generator consumed an extra second before stopping.

¹⁵Without this piece of information, any attempt to "synchronize" with the IOT device's clock will have a margin of error of up to one second.

However, if we assume that the devices are close enough to the relay and on a simple local network with few connections—no router, middle-box, or switch in-between—then there is little to no risk of packet reordering. Consequently, even if the order is not guaranteed within the relay during message forwarding, they will have been timestamped on arrival in the UDP input and can be placed back into the timeline accordingly (on the central server). The timestamp of the original message is also retained to reconstruct the time-frame of a single device accurately. Both paradigms are demonstrated below.

The testbed configuration is identical to the previous ones apart from the message generator mode which changes. For this test we will use the `loggen` generator to simulate 10 independent IOT devices with each its own clock. For a minimum of throughput in a short period of time, we will use the distributions from SECTION 2.2.2. The generator assigns a unique (shared) sequence number to each message sent, allowing us to compare the true order with the one derived from our approach.

For readers who wish to recreate the sample, the seed used is 123456789. *Note that to avoid parallelism randomness, sub-seeds are derived automatically for each device.* We used the `--store` parameter of `logserver` to save received logs on a file and assess them a posteriori. The duration of the test is 300 seconds.

```
Seed used: 123456789
Configured device 0 with clock 2016-11-07T09:43:13
Configured device 1 with clock 1971-06-11T08:21:11
Configured device 2 with clock 2000-06-07T21:41:10
Configured device 3 with clock 2017-02-23T22:53:58
Configured device 4 with clock 1987-09-16T23:40:50
Configured device 5 with clock 2015-11-01T07:23:12
Configured device 6 with clock 1985-03-14T19:14:00
Configured device 7 with clock 2002-06-30T16:10:29
Configured device 8 with clock 1976-09-06T01:36:11
Configured device 9 with clock 2011-02-13T01:06:49
Starting 10 device(s)...
Starting time: 2022-08-14 23:07:36.298576
Done
Statistics are displayed every 5 second(s)...
Statistics: last-seq=39, last-impt-seq=2, count=40, time=5 (sec), avg-rate=8.0 msg/sec
Statistics: last-seq=83, last-impt-seq=6, count=84, time=10 (sec), avg-rate=8.4 msg/sec
Statistics: last-seq=111, last-impt-seq=7, count=112, time=15 (sec), avg-rate=7.4667 msg/sec
Statistics: last-seq=148, last-impt-seq=10, count=149, time=20 (sec), avg-rate=7.45 msg/sec
Statistics: last-seq=185, last-impt-seq=10, count=186, time=25 (sec), avg-rate=7.44 msg/sec
Statistics: last-seq=232, last-impt-seq=14, count=233, time=30 (sec), avg-rate=7.7667 msg/sec
Statistics: last-seq=263, last-impt-seq=21, count=264, time=35 (sec), avg-rate=7.5429 msg/sec
Statistics: last-seq=302, last-impt-seq=22, count=303, time=40 (sec), avg-rate=7.575 msg/sec
Statistics: last-seq=332, last-impt-seq=25, count=333, time=45 (sec), avg-rate=7.4 msg/sec
```

Figure 4.14: The `loggen` script instantiating 10 devices, each in a thread with its own (random) clock.

```

Statistics: last-seq=1454, last-impt-seq=127, count=1455, time=200 (sec), avg-rate=7.275 msg/sec
Statistics: last-seq=1483, last-impt-seq=129, count=1484, time=205 (sec), avg-rate=7.239 msg/sec
Statistics: last-seq=1518, last-impt-seq=134, count=1519, time=210 (sec), avg-rate=7.2333 msg/sec
Statistics: last-seq=1562, last-impt-seq=138, count=1563, time=215 (sec), avg-rate=7.2698 msg/sec
Statistics: last-seq=1596, last-impt-seq=138, count=1597, time=220 (sec), avg-rate=7.2591 msg/sec
Statistics: last-seq=1630, last-impt-seq=139, count=1631, time=225 (sec), avg-rate=7.2489 msg/sec
Statistics: last-seq=1663, last-impt-seq=141, count=1664, time=230 (sec), avg-rate=7.2348 msg/sec
Statistics: last-seq=1696, last-impt-seq=144, count=1697, time=235 (sec), avg-rate=7.2213 msg/sec
Statistics: last-seq=1731, last-impt-seq=146, count=1732, time=240 (sec), avg-rate=7.2167 msg/sec
Statistics: last-seq=1754, last-impt-seq=148, count=1755, time=245 (sec), avg-rate=7.1633 msg/sec
Statistics: last-seq=1788, last-impt-seq=150, count=1789, time=250 (sec), avg-rate=7.156 msg/sec
Statistics: last-seq=1819, last-impt-seq=152, count=1820, time=255 (sec), avg-rate=7.1373 msg/sec
Statistics: last-seq=1858, last-impt-seq=154, count=1859, time=260 (sec), avg-rate=7.15 msg/sec
Statistics: last-seq=1892, last-impt-seq=159, count=1893, time=265 (sec), avg-rate=7.1434 msg/sec
Statistics: last-seq=1919, last-impt-seq=160, count=1920, time=270 (sec), avg-rate=7.1111 msg/sec
Statistics: last-seq=1968, last-impt-seq=163, count=1969, time=275 (sec), avg-rate=7.16 msg/sec
Statistics: last-seq=2003, last-impt-seq=165, count=2004, time=280 (sec), avg-rate=7.1571 msg/sec
Statistics: last-seq=2042, last-impt-seq=168, count=2043, time=285 (sec), avg-rate=7.1684 msg/sec
Statistics: last-seq=2071, last-impt-seq=170, count=2072, time=290 (sec), avg-rate=7.1448 msg/sec
Statistics: last-seq=2102, last-impt-seq=175, count=2103, time=295 (sec), avg-rate=7.1288 msg/sec
Statistics: last-seq=2135, last-impt-seq=177, count=2136, time=300 (sec), avg-rate=7.12 msg/sec
End time: 2022-08-14 23:12:36.300801
Closing...

```

Figure 4.15: loggen script closing after sending 2,136 messages in 300 seconds.

FIGURES 4.14 and 4.15 showcase the 10 devices internal counters over time. Altogether, 2,136 messages were sent, which gives a little over 7 messages/s. The collected logs were analysed in a tiny script to ensure that the order given by the timestamp affixed to the relay (called `timegenerated`) is the true order. As expected from the ideal environment in which the messages were generated—a direct Ethernet connection to the router, the messages are correctly sorted based on the timestamp. However, 6 of them could not be classified because they were identical. Bearing in mind that the time requery¹⁶ of our configuration is 2, the experiment was repeated with the minimal value proposed by the engine: 1. The results are summarized on FIGURE 4.16.

```

$ python3 ./rfc5424_parser.py
File 14-08-2022_23-00-23.log contains 2136 messages
Summary: 0 missing message(s), 0 date(s) are misplaced, 5 d
ate(s) ex-aequo (cannot break a tie), 0 date(s) could be un
tied using device localtime

$ python3 ./rfc5424_parser.py
File 14-08-2022_23-22-23.log contains 2136 messages
Summary: 0 missing message(s), 0 date(s) are misplaced, 0 d
ate(s) ex-aequo (cannot break a tie), 0 date(s) could be un
tied using device localtime

```

Figure 4.16: Audit of the two log files populated during the test on the chronological order. Both have the same configuration except for the *time requery* which is 2 for the first file and 1 for the second.

¹⁶Time requery is one of rsyslog's parameters. A requery of n means that once every n messages a `time()` syscall is issued. Details can be found here in SECTION 3.1.6.

Using a time requery of 1 we managed to reduce collisions by 500%¹⁷. In some cases, the use of the local timestamp (called `timereported`) of the IOT device can be used as a confirmation or untying when two messages are very closely-related. Assuming a chain of messages following each other by a few thousandths of a second, a valid indicator of the quality of the order of this chain would be that all messages belonging to the same device follow the same order as given by the `timereported`. In the example above this situation never occurred (the collisions came from different devices).

Finally, although performance is not the subject of this section, the bottom graph provides some insights on the state of the relay for the two time requery values. Unfortunately, the few messages sent are not enough to highlight its computing footprint. In fact, as stated in the documentation[32, p. 6], it would take several hundred messages per second to see a significant difference.

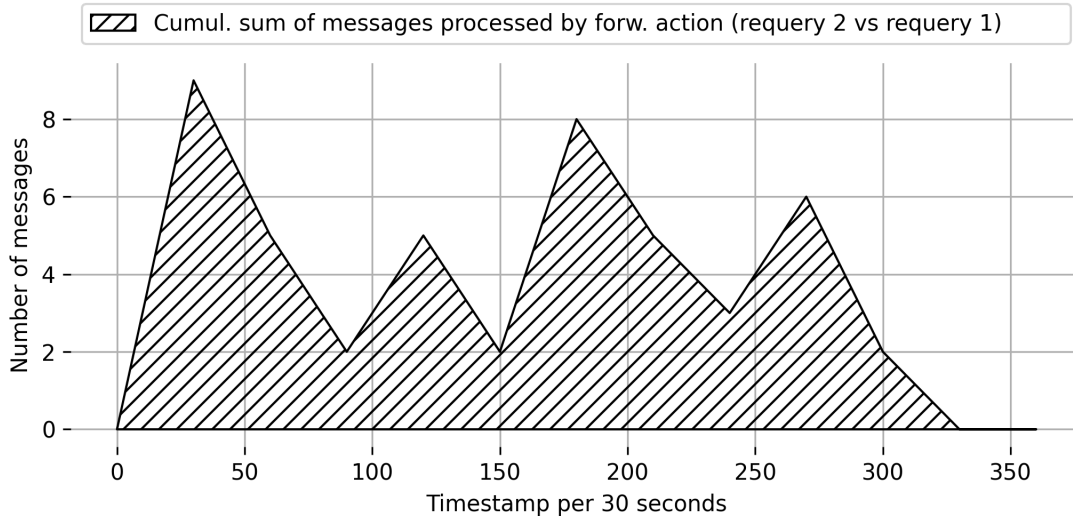


Figure 4.17: Two tests with identical configuration except for the *time requery* which is respectively 2 on the left and 1 on the right. Displays the number of forwarded messages (cumulative sum) that requery 2 has over requery 1 (at equal time). On average, requery 2 is ~ 4.6 messages ahead of requery 1 (a percentage lead of $\sim 0.2\%$).

4.4 Performance

In this section, we will evaluate the performance of the implementation. As presented in SECTION 3.2.1, the evaluation will be based on three resource profiles that define three different types of usage. This approach follows Cisco’s recommendations[42]

¹⁷Given the sample size, this should be treated with caution.

for creating IOx applications and allows a set of resources to be encapsulated under a unique name that is consistent across other IOx platforms (hence the term CPU unit). The assessment of the profiles is based on the following procedure: we will start with the most restrictive profile, assess it and identify its limits. If necessary, we will move on to the next profile to alleviate potential resource drain. The network emulation will be conducted by means of the two lossy configurations: EDGE and 3G cellular networks. All tests are performed in a flow-through fashion from the `logburst` script up to the router's LAN port hardware limit of 100Mbit/s ($\sim 40,000$ messages/s).

4.4.1 Cellular networks

The small profile is the first to be assessed. It sets the CPU resources to 200 units and the RAM to 64MB. We start with the minimum configuration, which is detailed on TABLE 4.4.

	UDP input		Main message queue		
	Threads	Batch size	Queue size	Worker threads	Deq. batch size
Small	1	32	45600	1	2400

Table 4.4: Starting parameters for the small profile.

EDGE cellular network

Let us arbitrarily choose the weakest network configuration, which is EDGE in lossy. This specifies a sending bandwidth of 200Kbit/s, 400ms delay, and 1% loss. From this limit, a first 300 seconds test is carried out with a message rate close to the bandwidth. Considering an average size of 256 bytes per messages, that is 100 messages/s. The results, visible on FIGURE 4.18, reveal that the configuration is more than adequate. Indeed, on (c) we can notice that in terms of efficiency two syscalls (left y-axis) are executed for only one message received (right y-axis), hence one can be convinced that at least one syscall out of two is empty. In reality there are many more "empty"—or nearly so—calls because in an ideal¹⁸ situation, given that `recvmsg()` [59] batches by means of 32 messages, there would be only one call for 32 messages. We can also observe on (d) that the input's worker thread, which pushes messages to the main queue (the dashed line), is perfectly synchronized (staggered) with the one that processes and sends the messages (the solid line). The queue size filled to a maximum of 6% of its capacity. Of the 30,100 messages sent, all were collected by the server. Its average receive rate was ~ 90 messages/s.

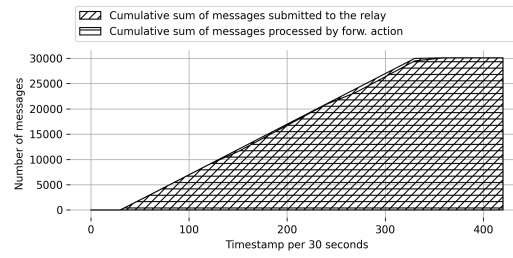
¹⁸This ideal scenario is also an "alarming" indicator that the OS buffer is under heavy stress (contains at least 32 messages on each call) and might very well be saturated (meaning messages are dropped).

```

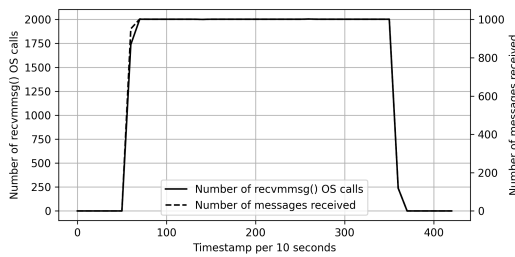
Statistics: count=28234, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28336, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28438, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28540, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28642, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28744, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28846, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=28948, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29049, impt-counter=0, rate=100 msg/sec, throughput=0.0256 MB/sec
Statistics: count=29150, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29251, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29352, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29454, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29555, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29656, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29757, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29858, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=29959, impt-counter=0, rate=101 msg/sec, throughput=0.0259 MB/sec
Statistics: count=30060, impt-counter=0, rate=100 msg/sec, throughput=0.0256 MB/sec
cancelling the token...
Statistics: count=30100, impt-counter=0, rate=39 msg/sec, throughput=0.01 MB/sec
End time: 2022-08-15 22:41:23.948831

```

(a) `logburst` script running for 300 seconds with a message rate of 100 messages/s. In total 30,100 messages were sent.



(b) The 30,100 messages were successfully received and processed without noticeable delay (the surfaces overlap perfectly).



(c) Number of `rcvmsg()` OS calls performed vs actual messages received, single-threaded). At least one out of every two calls was empty.



(d) Small profile internal counters with a minimalist configuration. The maximum size reached by the queue is 2,735 ($\sim 6\%$) and the average 757 ($\sim 1.5\%$).

Figure 4.18: Small profile performance test with a minimalist configuration on a lossy EDGE cellular network.

3G cellular network

Using the same configuration but this time with the 3G lossy network, which reflects a link capable of 330Kbit/s upstream, 100ms delay, and 1% loss, we encoded the `logburst` rate set to 175 messages/s (256 bytes per messages) and executed the test for another 300 seconds. The experience is equivalent (see **FIGURE 4.19**), if not better. Indeed, the 300ms reduction in delay has a direct impact on the queue (see dashed line on (d)) which is almost always empty. This effectively suggests that the sampling done every 10 seconds is not short enough to capture the producing consuming process (except for the very first time). This is a strong evidence of a steady flow. On the server side, an average rate of 160 messages/s is reached, very close to the theoretical limit.

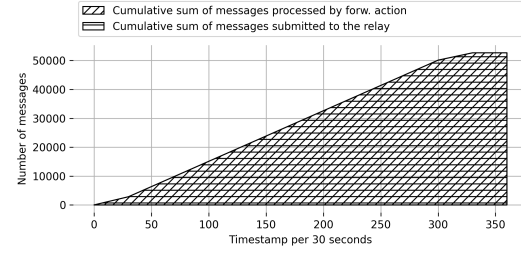
The surprising behaviour of this configuration is in fact totally sensible. One could indeed imagine that increasing the number of threads would only have a positive impact, and that having only one is a waste of the available cores. In this context, these are more likely to damage performance (on the ingress and egress). Indeed,

```

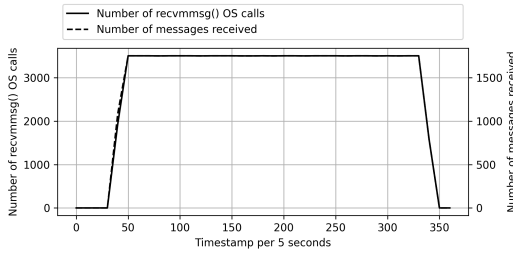
Statistics: count=49270, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=49446, impt-counter=0, rate=175 msg/sec, throughput=0.0448 MB/sec
Statistics: count=49623, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=49799, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=49975, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=50152, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=50328, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=50505, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=50681, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=50858, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=51034, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=51210, impt-counter=0, rate=175 msg/sec, throughput=0.0448 MB/sec
Statistics: count=51387, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=51563, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=51740, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=51916, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=52092, impt-counter=0, rate=175 msg/sec, throughput=0.0448 MB/sec
Statistics: count=52269, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=52446, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Statistics: count=52623, impt-counter=0, rate=176 msg/sec, throughput=0.0451 MB/sec
Cancelling the token...
Statistics: count=52674, impt-counter=0, rate=51 msg/sec, throughput=0.0131 MB/sec
End time: 2022-08-16 00:00:12.106227

```

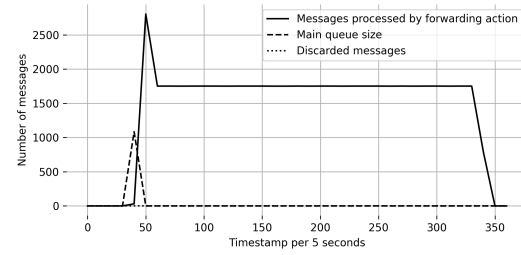
(a) logburst script running for 300 seconds with a message rate of 175 messages/s. In total 52,674 messages were sent.



(b) The 52,674 messages were successfully received and processed. No noticeable delays (the two areas perfectly overlaps).



(c) Number of `recvmsg()` OS calls performed vs actual messages received (one thread). At least one out of every two calls was empty.



(d) Small profile internal counters with a minimalist configuration. The maximum size reached by the queue is 1,084 ($\sim 2.5\%$) and the average 30 ($\sim 0.05\%$).

Figure 4.19: Small profile performance test with a minimalist configuration on a lossy 3G cellular network.

raising the number of threads can be detrimental because they are all fighting for the same resource: messages. The overhead of threads fighting for the lock on the queue slows down everyone and forces small batch sizes¹⁹. Deciding on the size of a batch is a question of balancing overhead versus throughput. In a way, they follow the same philosophy as the one applied to jumbo frames[62]. There are benefits from the upsizing but it is not linear: we are damping²⁰ the per-batch overhead across more messages.

4.4.2 High-speed low-latency network

Before losing the reader in further explanation, I propose to test this configuration foot to the floor, i.e. in continuous burst of 100Mbit/s, to review the maximum

¹⁹When a thread requests a batch, whether it is in the syscall `recvmsg()` scenario or the main queue, it will respond with the appropriate request or whatever it has on hand (meaning less).

²⁰Halved going from 1 to 2 as the overhead per message is 50%, even less going 2 to 3 as the overhead is 33%, 4 = 25%, 5 = 20%, 6 = 16%, etc.

capacity and performance of this configuration. After a small blank test, the interface seems to accept only 10MB/s, or more or less 39,000 messages/s (for a message size of 256 bytes). The restriction imposed by the traffic controller has of course been deactivated.

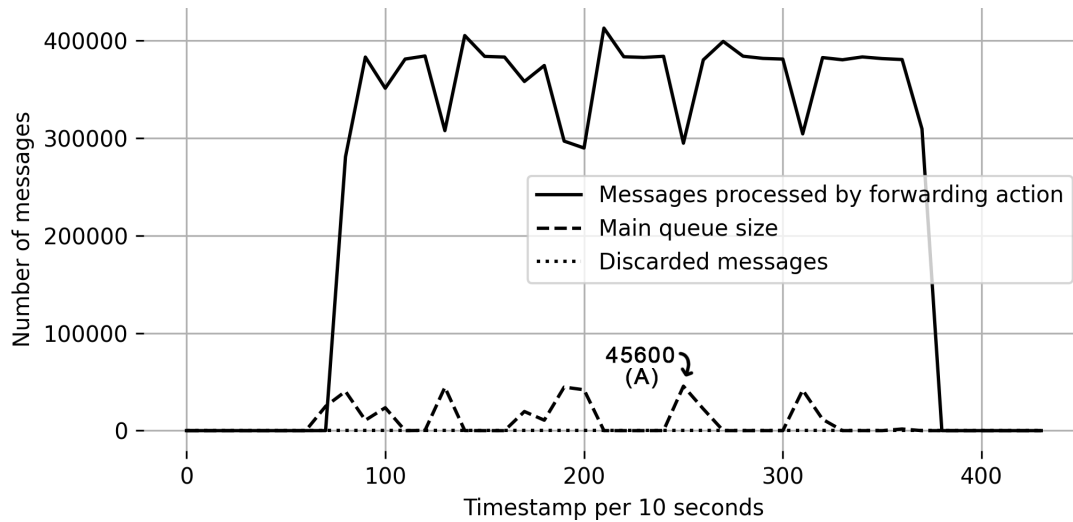


Figure 4.20: Small profile internal counters under a 10MB/s burst. On average, 247,422 messages are sent to the server every 10 seconds. No messages were dropped.

FIGURE 4.20 illustrates the state of the relay during this benchmark. The main queue size reached the maximum value of 45,600 once, visible on the figure as point (A), and contains on average 8,727 messages ($\sim 19\%$). Once again one can see the producing (dashed line) consuming (solid line) effect which happens *almost* always before the main queue is full and is indicative of a process that can cope with the burst. Although no messages were discarded (dotted line), it is possible that at that moment (A) when the queue was full, messages could not be retrieved by the input thread: the queue being full, the upstream process can do nothing but wait for space to be made (potentially leaving room for losses).

However, the following FIGURE 4.21 supports that calls to the RX buffer per batch return messages each time as the line of received messages (dotted line) encloses the line of OS calls (solid line). If the OS buffer was overloaded then for n `recvmsg()` calls there would be around $32 \times n$ messages received by the relay. In the present case there is never more than a factor of 2.

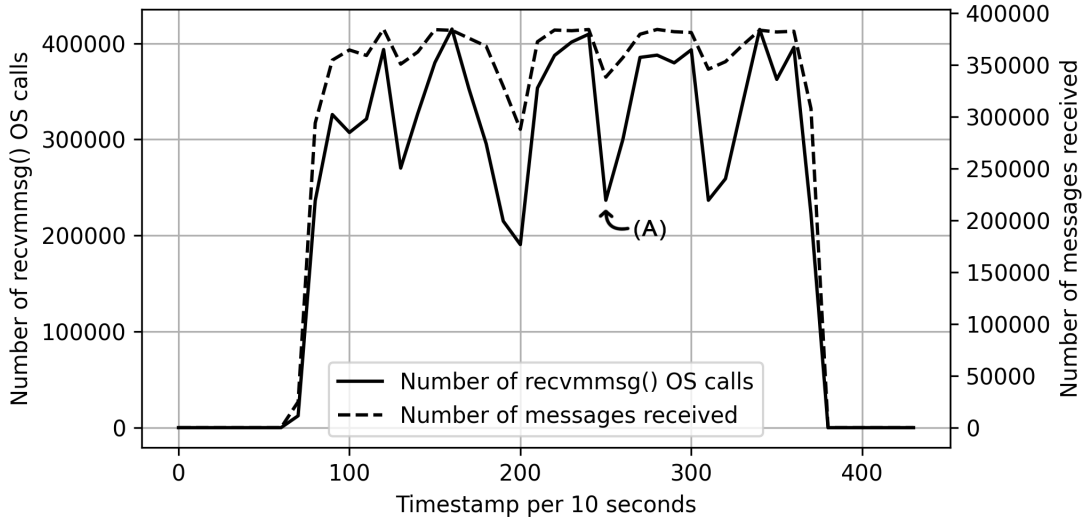


Figure 4.21: Number of `recvmmsg()` OS calls performed vs actual messages received (one thread). Each call returns at least one message. Point (A) denotes the event in which the queue size maximum capacity was reached. Every 10 seconds, an average of 247,421 messages are received and 225,101 calls are made.

Admittedly, the above-mentioned inference was correct. Indeed, despite the performance of this configuration continues to surprise, this time it was unable to retrieve all the messages sent. As visible on the FIGURE 4.22 and 4.23, the generator indicates that 11,570,815 messages were passed on, but the server reports only 10,886,549 messages.

```
Statistics: msg-count=8918820, impt-msg-count=0, time=340 (sec), avg-rate=26231.8235 msg/sec
Statistics: msg-count=9282993, impt-msg-count=0, time=350 (sec), avg-rate=26522.8371 msg/sec
Statistics: msg-count=9666002, impt-msg-count=0, time=360 (sec), avg-rate=26850.0056 msg/sec
Statistics: msg-count=10047458, impt-msg-count=0, time=370 (sec), avg-rate=27155.2919 msg/sec
Statistics: msg-count=10429660, impt-msg-count=0, time=380 (sec), avg-rate=27446.4737 msg/sec
Statistics: msg-count=10744245, impt-msg-count=0, time=390 (sec), avg-rate=27549.3462 msg/sec
Statistics: msg-count=10886549, impt-msg-count=0, time=400 (sec), avg-rate=27216.3725 msg/sec
Statistics: msg-count=10886549, impt-msg-count=0, time=410 (sec), avg-rate=26552.5585 msg/sec
Statistics: msg-count=10886549, impt-msg-count=0, time=420 (sec), avg-rate=25920.3548 msg/sec
```

Figure 4.22: logserver script receiving an average of 27,215 messages/s, for a total of 10,886,549 messages.

```
Statistics: count=11390869, impt-counter=0, rate=38500 msg/sec, throughput=9.856 MB/sec
Statistics: count=11429349, impt-counter=0, rate=38480 msg/sec, throughput=9.8509 MB/sec
Statistics: count=11467830, impt-counter=0, rate=38481 msg/sec, throughput=9.8511 MB/sec
Statistics: count=11506292, impt-counter=0, rate=38462 msg/sec, throughput=9.8463 MB/sec
Statistics: count=11544766, impt-counter=0, rate=38474 msg/sec, throughput=9.8493 MB/sec
Cancelling the token...
Statistics: count=11570815, impt-counter=0, rate=26049 msg/sec, throughput=6.6685 MB/sec
End time: 2022-08-16 01:16:00.825378
```

Figure 4.23: logburst script which sent 11,570,815 messages in 300 seconds.

That's 684,226 messages missing, i.e. a net loss of $\sim 6\%$. The internal counters of the rsyslog engine do not indicate any loss due to a full queue (see FIGURE 4.24). The router's *Fast-Ethernet* port also reports no loss (FIGURE 4.25). If it wasn't for the previous analysis done on the input (FIGURE 4.21), we could think that the ingress is having difficulties in absorbing the burst.

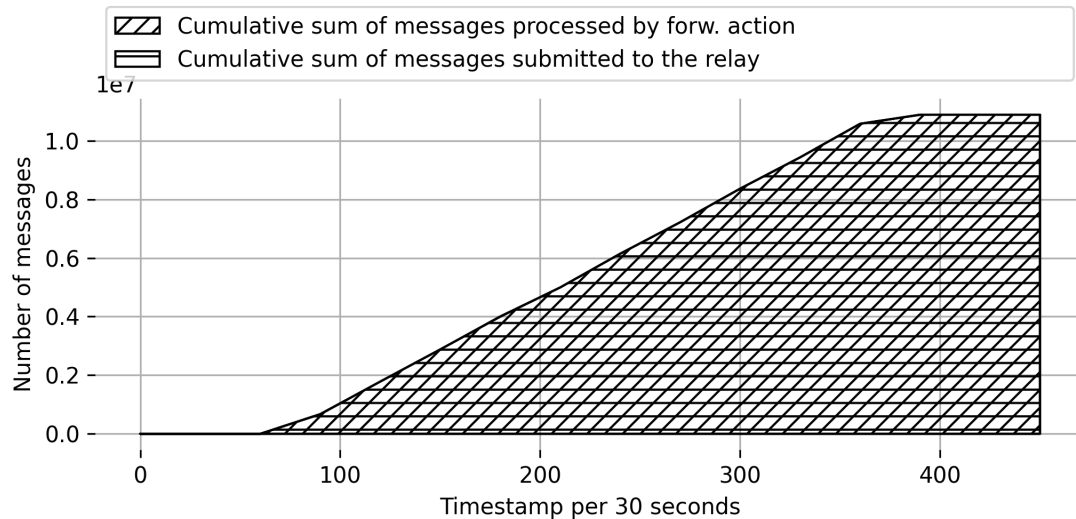


Figure 4.24: Every message received by the relay has been transmitted, totalling 10,886,549 messages. It can be observed that the forwarding process induces almost no delay (overlap).

Input Information	
Input Queue	0/375/0/0
CRC	0
Input Rate[Packets/sec]	35662
Last Input	6d16h
Output Information	
Output Queue	0/40
Protocol Drop	0
Output Rate[Packets/sec]	0
Last Output	00:00:24
General	
Interface Name	FastEthernet0/0/1
Mac Address	a03d.6e75.f501
Duplex	Full-duplex
Speed	100Mb/s
MTU	1500 bytes
Delay	100 usec
Bandwidth	100.00 Mbps
RxLoad	221/255
TxLoad	1/255
Reliability	255/255

Figure 4.25: General information about the `FastEthernet0/0/1` port where the message generator is connected. Image taken during the burst from the router's web interface. The RX buffer is under heavy load (rxload at 221/255 for an input rate at $\sim 35,000$ packets/s) but retains some space.

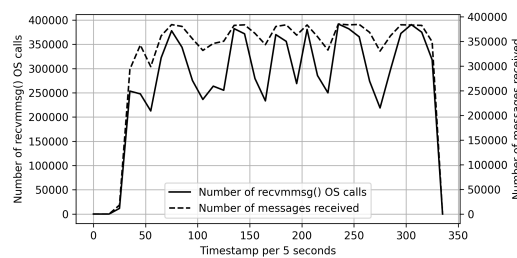
While it is tempting to increase the number of threads, after checking the load on the CPU (via the `top` command), we notice that only 11% is used for the input worker thread and 17% for the main queue's worker. It is therefore irrelevant to decouple their workload.

```
Mem: 3324868K used, 662696K free, 217732K shrd, 159528K buff, 935152K cached
CPU: 44% usr 22% sys 0% nic 15% idle 0% io 1% irq 16% sirq
Load average: 3.98 2.43 1.96 9/457 27
```

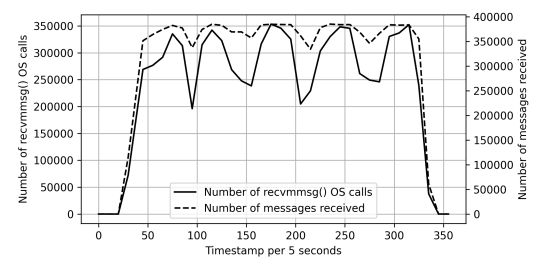
PID	PPID	USER	STAT	VSZ	%VSZ	CPU	%CPU	COMMAND
13	0	root	S	25692	1%	0	17%	{rs:main Q:Reg} /usr/sbin/rsyslogd -n
10	0	root	R	25692	1%	1	11%	{imudp(w0)} /usr/sbin/rsyslogd -n
12	0	root	S	25692	1%	0	0%	{rs:stats:Reg} /usr/sbin/rsyslogd -n
9	0	root	S	25692	1%	1	0%	{in:impstats} /usr/sbin/rsyslogd -n
1	0	root	S	25692	1%	0	0%	/usr/sbin/rsyslogd -n
19	14	root	S	2788	0%	0	0%	/bin/bash
14	0	root	S	1744	0%	1	0%	/bin/sh
27	19	root	R	1676	0%	1	0%	top

Figure 4.26: Rsyslog's threads under load (using 'H' inside `top` command).

Although the number of input messages collected appears to be higher, as one can observe from FIGURE 4.27, increasing the size of the input batches does not decrease the loss of messages (the server was able to collect 10,845,940 and 10,943,053 respectively). Compared to FIGURE 4.21, the average ratio of messages per call went from 1.01 to 1.16 for a batch of 64 messages and 1.27 for a batch of 128 messages. This empirical evidence confirms our intuition about the factor linking the number of messages received and the number of OS calls: if it is not at least close to 32, then increasing the batch size on the ingress will have no effect.



(a) Number of OS calls performed vs messages received (one thread). Batch size of 64. Every 10 seconds, an average of 318,998 messages are received and 275,352 OS calls are made.



(b) Number of OS calls performed vs messages received (one thread). Batch size of 128. Every 10 seconds, an average of 307,210 messages are received and 242,432 OS calls are made.

Figure 4.27: Rsyslog's input thread pulling messages from the RX buffer by batch of 64 and 128. Dotted lines represent messages received and solid lines the number of `recvmsg()` calls.

As a last resort, perhaps the batching on the main queue side is too large and it spends some of its time waiting for TCP to deliver its messages. By reducing its

size, we would maximize the parallel efficiency: while TCP is busy, the worker thread is too. After a test with a batch size of 1200, then another of 600, we obtained respectively 11,247,432 and 11,335,718 messages (only a 2% loss). This is 4% less compared to the previous configuration. We also notice that the main queue's thread reaches the 20% CPU usage (3% increase from previous setup), demonstrating once more that the threads were not at fault by accepting additional workloads.

At this point it seems relevant to me to upgrade to the *medium* profile, firstly because the current configuration seems to be showing its limits and increasing the queue size will inevitably relieve the data flow, but also because the small profile in the last tests used between 25 and 30 percent of the CPU, i.e. between 300 and 350 CPU units. This is only possible because Cisco IOx allows an application to consume more CPU resources than those reserved for its launch as long as no one else claims them. It is thus preferable to pre-allocate them to ensure consistent performance.

Unsurprisingly, with the basic configuration and a batching for the main queue of 1200, the medium profile enables the relay to receive and transmit all messages generated. This cap also announces the limit of the testbed, or at least my hardware limit. It is obvious that adding more message generators on the other LAN ports will require configuration changes. The general approach would be to vary the batches (input and output) and only consider adding additional threads if it is found that a thread is reaching (or approaching) 100% of the CPU.

4.4.3 Queue build-up prediction

As a final step in this profiling, we can try to predict the message throughput that we are able to support infinitely (sustain mode) without any queue build-up. For this purpose, the logburst script has been improved to allow the rate to grow over time (configurable). A rest period has also been introduced to ensure that the relay queue is empty before moving to the next higher rate. The test was performed on a 512MB configuration (i.e. a capacity of 384,000 messages) to avoid any flattening of a potential peak, for 3,600 seconds with a burst period of 60 seconds and a quiet period of 30s, and a maximum rate of 39,000 messages/s. There are thus 40 periods and a difference of 975 messages between each. FIGURE 4.28 highlights this test.

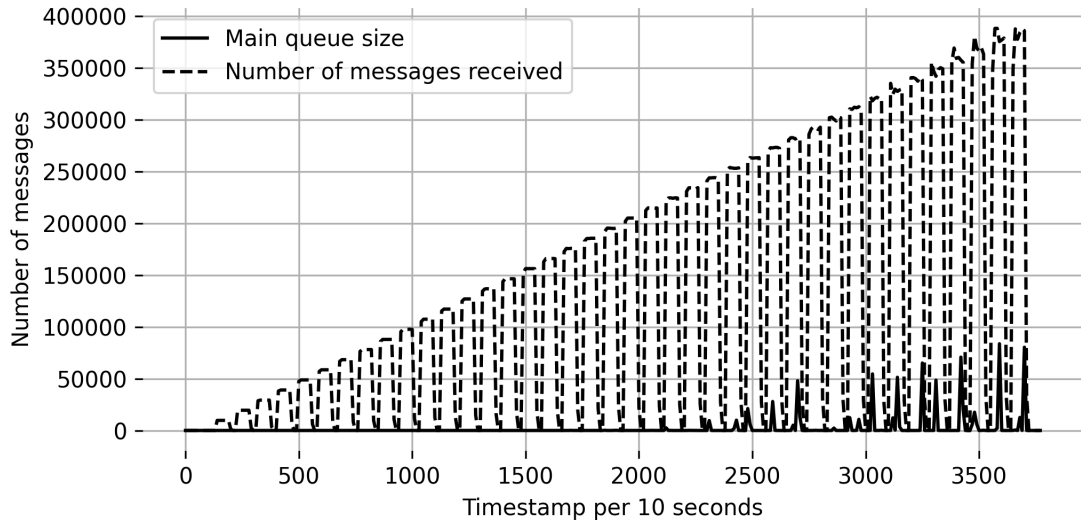


Figure 4.28: Test performed to gather data on the evolution of the size of the main queue as a function of the number of messages received. Note that the number of messages received (dashed line) is an average over 10 seconds.

The 40 periods are highlighted on this figure as summits of the dashed line and with the size of the queue increasing inside (solid line). The FIGURE 4.29 showcases a close-up view of a few periods.

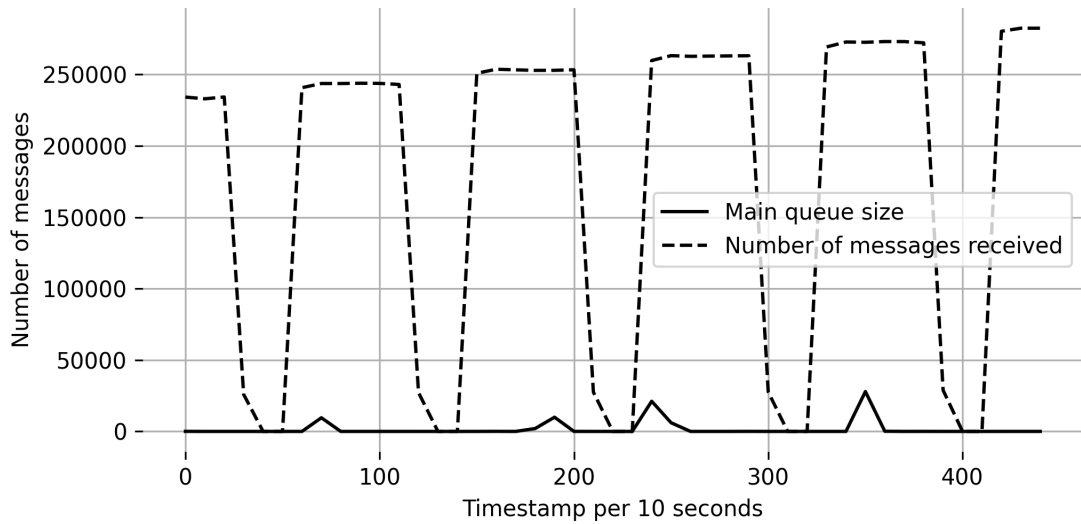


Figure 4.29: A close up view of FIGURE 4.29, one can clearly see the periods of burstiness followed by moments of stillness. The solid line represents the main queue size containing very indicative episodes of burst absorption with an escalation in size followed by a steep decline to zero.

A mathematical regression method is used to establish a relationship between the number of messages received and the queue size. The data retrieved above

are first parsed to isolate the dominant values, i.e. the maximum queue sizes encountered, then plotted for analysis. From the said graph, a trend close to a quadratic function can be identified. Using the `sklearn`[63] library available in Python, it is possible to create a polynomial regression model (of the second degree, using the `LinearRegression` and `PolynomialFeatures` class) and use it to predict values that we do not possess. The scatter plot and the regression curve can be viewed in FIGURE 4.29.

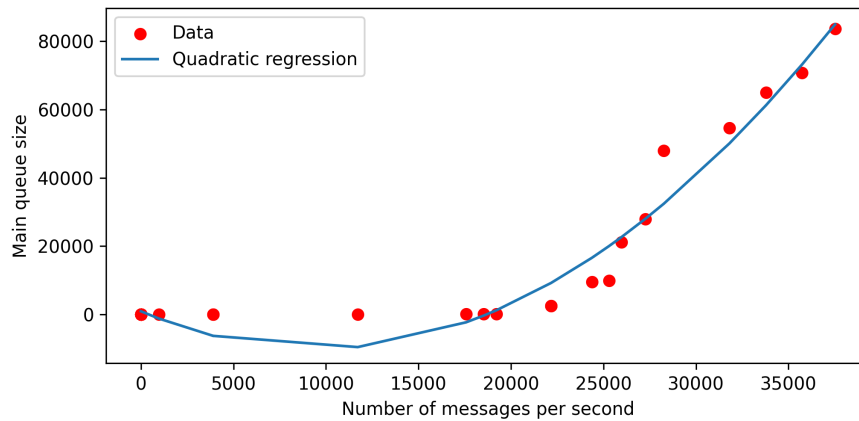


Figure 4.30: Quadratic regression curve based on the observed data (maximum values only) from FIGURE 4.29.

From this regression curve, it is possible to predict the behaviour of the queue for higher values. FIGURE 4.31 summarizes the possible values for a rate up to 100,000 messages/s.

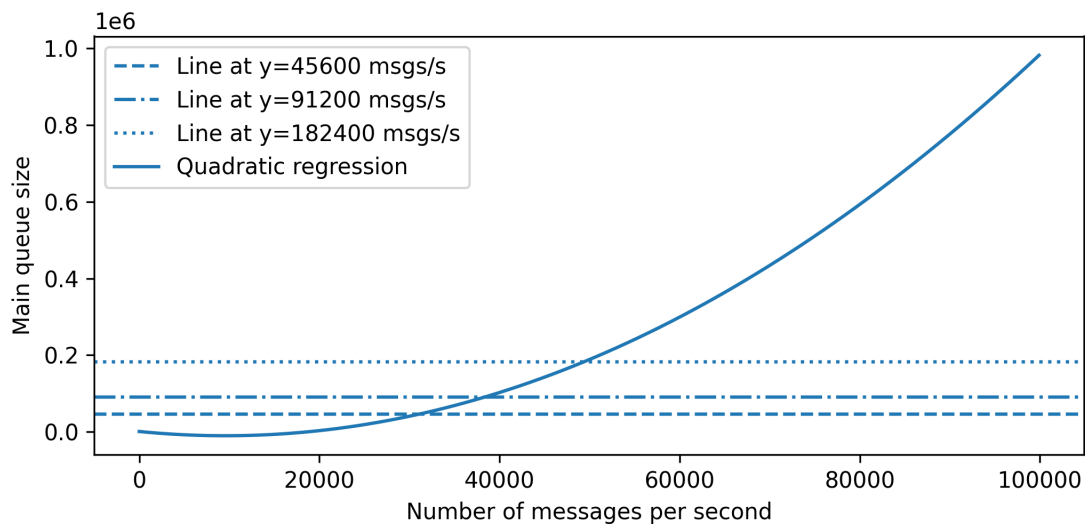


Figure 4.31: Quadratic regression curve predicting values for a rate up to 100,000 messages/s. The horizontal lines represent the three resource profiles.

Before using these predictions as a reliable source of information, I think it is important to point out a few factors that were overlooked during this test. Firstly, the use of maximum values is probably a bit extreme in the sense that a weaker queue could also absorb the start of a burst but more slowly. Secondly, this test reflects a particular configuration which is probably not adequate past certain limits. Conversely, one could also reasonably assume that the current settings are ideal for any implementation below or equal to a message rate of 10 MB/s (39,000 messages) as we have tested it. Therefore, any tendency put forward by this configuration would reflect an idyllic prediction that may not be achievable in practice (for example, because thread decoupling is not linear and generates losses).

4.4.4 Results

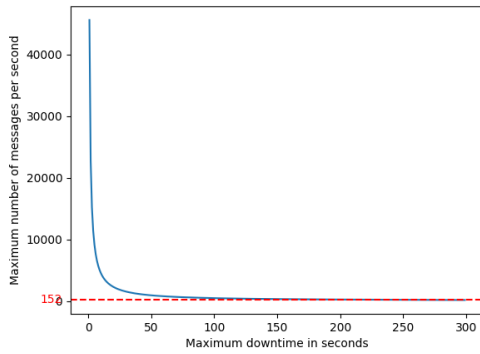
The results of this profiling phase highlight the performance achievable on the basis of a minimalist configuration. In the end, the medium and large profile were (almost) not necessary. However, performance has little or no influence in the current context, firstly because message flow is limited by the quality of the network used, and secondly because in the end if the server is not reachable it is the RAM that will determine the reliability of the configuration. Without access to a disk, the number of messages that the relay can ingest is known in advance. For the sake of argument, let us review how much downtime (depending on the message rate) each profile can absorb. The results are depicted on **FIGURE 4.32**. If, for example, one wishes to preserve messages for 5 minutes in the small profile, at most 152 messages can be absorbed per second.

Using this principle in reverse, it is also possible to measure the time it takes to completely discharge (recover) the queue when the connection is recovered (**TABLE 4.5**).

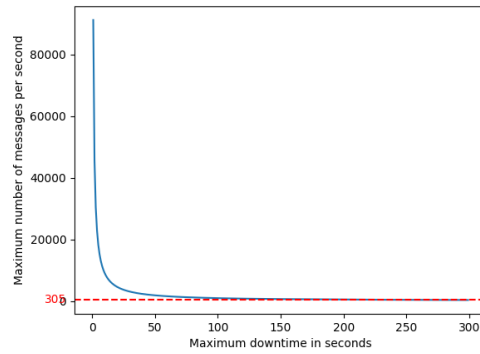
Table 4.5: Time in seconds needed to completely empty the queue per profile (message size of 256 bytes).

	Edge network	3G network
Small	~ 470s	~ 283s
Medium	~ 940s	~ 566s
Large	~ 1,880s	~ 1,133s

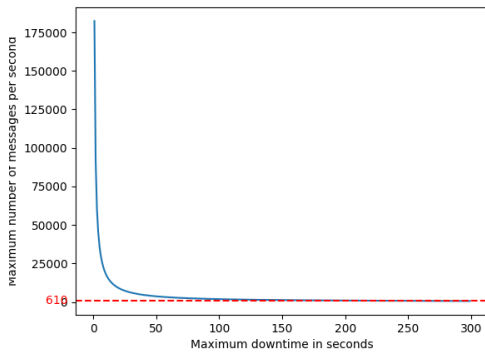
It should also be noted that the *time requery*, after evaluation, showed no weakness in terms of performance when switching from 2 to 1. The CPU usage did not exceed 20% and all messages sent were transmitted to the syslog server. *However*,



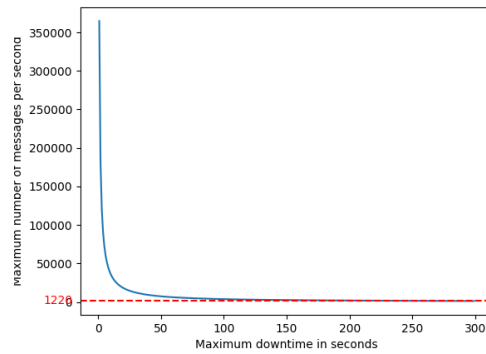
(a) 64MB configuration (45,600 message queue). For a downtime of 5 minutes the rate must not be greater than 152 messages/s.



(b) 128MB configuration (91,200 message queue). For a downtime of 5 minutes the rate must not be greater than 305 messages/s.



(c) 256MB configuration (182,400 message queue). For a downtime of 5 minutes the rate must not be greater than 610 messages/s.



(d) 512MB configuration (364,800 message queue). For a downtime of 5 minutes the rate must not be greater than 1220 messages/s.

Figure 4.32: Maximum manageable lossless downtime based on the number of messages received per second. The average message size used is 256 bytes. This calculation does not include worker thread(s) internal buffer.

be aware that the paper written by R. Geirhald, "Going up from 40K messages per second to 250K"[32], stresses the importance of this parameter which was one of the main causes of the slowdown of the rsyslog engine during the v4 improvements.

In the end, the basic configuration on the smallest profile passed the evaluation with flying colours. The parameters used seem to be the most appropriate for the present context and environment. The medium and large profile can therefore be modelled on the small one, with a correct adjustment²¹ of the queue size and

²¹I believe it is still crucial to keep 25% of the memory allocated by IOx as a safety margin for the rsyslog engine in order to prevent any OOM killer. See TABLE 3.2 for details.

possibly the batch size or the number of threads if performance requires it. *Please note that the configuration and parameters are not static and can be adjusted as desired²² from the IOx web interface.*

²²Depending on the usage, the maximum downtime, the recovery speed, the other applications present in the IOx environment, and an infinity of other reasons.

5

Conclusion

Contents

5.1	Challenges faced	83
5.2	Future works	84
5.3	Final words	85

5.1 Challenges faced

During the realization of this project, several challenges of different sizes have been overcome. The first challenge was to find scientific reading about syslog, the protocol and what is currently considered the state-of-the-art. It is hard not to link the lack of literature to the very state and evolution of syslog as a standard that has always been rooted in its foundations, inflexible. Similarly, while datasets about IOTs are getting more and more interest, when creating tools to generate relevant syslog messages I could only find few if any samples about device or application logs. Regarding the implementation itself, I'll mainly note the "nested layer" effect that Cisco IOx introduced: building an aarch64 application on which the rsyslog engine must be available, inserting it in a minimalist qemu emulatable container for compilation, packaging it for the IOx environment, and finally creating a "link" to drive the first layer (i.e. rsyslog) from the router's user interface. All these layers proved to be a difficulty during diagnostics, especially to identify the why and how (like the OOM killer to name one). Lastly, I would note the challenge of setting up a test environment that included many different actors, each influencing the network in some way, but all working towards the same goal.

5.2 Future works

In the current state of the solution there is substantial room for improvement. These are detailed below.

1. It seems interesting to me to improve the verbosity of the relay's health status. Currently it is possible to retrieve internal counters using Elasticsearch but this solution is not adapted to a realistic environment, it would be preferable to log the information in the data flow itself. Deleted messages and connectivity loss are prime examples of metrics to be reported to the central server. Along the same lines, it might be relevant to transmit the router's internal logs.
2. User (administration) experience was a valuable aspect of this project. The configuration from the local manager of the router is sufficiently verbose to provide debugging mechanisms, for example. As features are added, it may be worthwhile to customize them for the administrator and thus avoid having to repackaging the application at each change.
3. Several optimizations are possible concerning the relay's message flow. I am thinking in particular of the compression of messages which could be initiated when the server is not reachable, thus considerably increasing the capacity of the queue. The rsyslog engine offers native compression solutions. Similarly, it is also possible to configure additional central servers in the event that the first one is unreachable (failover). Finally, if many log messages are multi-line then the plain TCP syslog framing can be switched to a character other than the LF (`'\n'`) frame delimiter, thus avoiding unnecessary message fragmentation. For example, in RFC5425[18] the size of the message is included in a header that precedes each frame.
4. It is possible to configure queues such that they only dequeue (process) messages at specific times. When there is a constrained amount of bandwidth on the network path to the central server, for example, this is handy for transferring the majority of messages only during off-peak hours.
5. Regarding alternative protocols, it is worth noting the use of PR-SCTP to exploit the priority properties of syslog messages during loss recovery. The authors of [24] also report that PR-SCTP performs better than TCP in terms of average message transfer delay. The RELP protocol also offers some interesting functionalities, mainly the acknowledgement on the application layer through the use of a backchannel[30, p. 160].

6. Although the aspect of persistent storage has been discussed several times in this work, it has never been implemented. The rsyslog engine offers a variety of features related to data persistence, such as piping queues to a disk or the assisted mode to leverage the burst period occupying all the memory space. This *over*-reliability is discussed in detail in the paper written by R. Gerhards, "Rsyslog Design and Internals"[31].

5.3 Final words

In this paper, we have defined, designed, and implemented a reliable and efficient solution to overcome the shortcomings introduced by the use of the UDP transport protocol. In particular, we have enabled devices connected to the relay to transmit their messages in a reliable, partially ordered, and confidential manner, regardless of the state and quality of the network used. Beyond the key objectives, the solution offers more than reasonable performances for a low resource consumption. Its portability and configurability allows it to be easily ported to other incubators using the IOx environment.

Regarding the test environment, we were able to deploy the relay in real conditions and evaluate excellent behaviour in cellular networks but also high speed low latency networks. The whole ecosystem—the message generators, the central server, the statistics collector, the certificate generator, the application itself, and many other tools—is available for the reader to quickly iterate on a custom configuration. This foundation makes future research and improvement much more accessible.

Appendices



Project source code

Contents

A.1 Overview	89
A.2 Structure	89

A.1 Overview

In agreement with my industrial supervisor, Emmanuel Tychon, the whole project and the scripts used during the work are publicly available in the following Github repository: <https://github.com/e-scheer/syslog-relay>.

The IOx application, the message generators, the central server, the statistics collector, and many other files are accessible from this link and are organized in separate folders to facilitate your search. The source code, configurations, and scripts are commented in detail for the more curious reader.

A.2 Structure

The structure of the repository is divided into five parts. These are detailed below:

1. The generation of certificates using the `certtool` tool. Note that the CA is also generated and its certificate self-signed. Also, the configuration values such as the *common name* are used to authenticate the client and the server, thus be careful when setting them up.

2. All I/O scripts related, including the central syslog server and the message generators used for the evaluation of the implementation. Their functionalities are explained in detail in the directory (see `README.md`).
3. The IOx application itself, containing the Docker configuration files and the rsyslog engine. Its packaging is automated via the `build.sh` script.
4. The statistics collector as a service including two dockerized servers: Elasticsearch and Kibana.
5. All the remaining scripts such as those used for the analysis of the Linux dataset or to certify the order of the received messages. There are also command lines and notes related to the proper implementation of a testbed.

References

- [1] A. Chuvakin, K. Schmidt, and C. Phillips. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management*. Elsevier Science, 2012. URL: https://books.google.be/books?id=Rf8M%5C_X%5C_YTUoC.
- [2] Eric Allman. *INTERNET HALL of FAME INNOVATOR*. Online; accessed November 18, 2021. 2014. URL: <https://www.internethalloffame.org/inductees/eric-allman>.
- [3] Bryan Costales and Eric Allman. *Sendmail, 3rd Edition*. 3rd ed. O'Reilly Media, Inc, 2002.
- [4] C. Lonvick. *The BSD Syslog Protocol*. RFC 3164. RFC Editor, Aug. 2001.
- [5] R. Gerhards. *The Syslog Protocol*. RFC 5424. RFC Editor, Mar. 2009. URL: <http://www.rfc-editor.org/rfc/rfc5424.txt>.
- [6] A Min Tjoa et al. *Availability, Reliability and Security for Business, Enterprise and Health Information Systems : IFIP WG 8.4/8.9 International Cross Domain Conference and Workshop, ARES 2011, Vienna, Austria, August 22-26, 2011. Proceedings*. Jan. 2011.
- [7] *syslogd(8) Linux User's Manual*.
- [8] *logger(1) Linux User's Manual*.
- [9] *syslog(3) Linux User's Manual*.
- [10] *syslogd(8) Linux User's Manual*.
- [11] *klogd(8) Linux User's Manual*.
- [12] *syslog.conf(5) Linux User's Manual*.
- [13] Rainer Gerhards. *Why does the world need another syslogd?* Online; accessed November 30, 2021. 2007. URL: <https://rainer.gerhards.net/2007/08/why-doesworld-need-another-syslogd.html>.
- [14] Larene Le Gassick. *Analyze syslog messages with Seq*. Online; accessed December 02, 2021. 2020. URL: <https://blog.datalust.co/seq-input-syslog/#rfc3164>.
- [15] G. Klyne and C. Newman. *Date and Time on the Internet: Timestamps*. RFC 3339. RFC Editor, July 2002.
- [16] Anand Deveriya. *Network administrators survival guide*. 1st ed. Indianapolis: Cisco Press, 2006.
- [17] A. Okmianski. *Transmission of Syslog Messages over UDP*. RFC 5426. RFC Editor, Mar. 2009.

- [18] F. Miao, Y. Ma, and J. Salowey. *Transport Layer Security (TLS) Transport Mapping for Syslog*. RFC 5425. RFC Editor, Mar. 2009.
- [19] J. Salowey et al. *Datagram Transport Layer Security (DTLS) Transport Mapping for Syslog*. RFC 6012. RFC Editor, Oct. 2010.
- [20] D. New and M. Rose. *Reliable Delivery for syslog*. RFC 3195. RFC Editor, Nov. 2001.
- [21] *Embedded syslog manager configuration guide, cisco IOS release 15s - reliable delivery and filtering for syslog*. Online; accessed December 07, 2021. Sept. 2017. URL: <https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/esm/configuration/15-s/esm-15-s-book/reliable-del-filter.html>.
- [22] R. Gerhards and C. Lonvick. *Transmission of Syslog Messages over TCP*. RFC 6587. RFC Editor, Apr. 2012.
- [23] Hiroshi Tsunoda et al. “A Prioritized Retransmission Mechanism for Reliable and Efficient Delivery of Syslog Messages”. In: *Proceedings of the 2009 Seventh Annual Communication Networks and Services Research Conference*. CNSR '09. USA: IEEE Computer Society, 2009, pp. 158–165. URL: <https://doi.org/10.1109/CNSR.2009.33>.
- [24] Mohammad Rajiullah et al. “Syslog performance: Data modeling and transport”. In: *2011 Third International Workshop on Security and Communication Networks (IWSCN)*. 2011, pp. 31–37.
- [25] DevNet Cisco. *Platform Support Matrix - IOx - Document*. Online; accessed March 04, 2021. URL: <https://developer.cisco.com/docs/iox/#!/platform-support-matrix>.
- [26] Alpine Linux Development Team. *Alpine linux packages*. Online; accessed March 24, 2021. URL: https://pkgs.alpinelinux.org/packages?name=rsyslog*&branch=edge&repo=&arch=aarch64.
- [27] Rainer Gerhards. *Rsyslog will remain GPLv3 licensed*. Online; accessed March 06, 2022. Jan. 2012. URL: <https://rainer.gerhards.net/2012/01/rsyslog-will-remain-gplv3-licensed.html>.
- [28] *tcp(7) Linux User's Manual*. Mar. 2021.
- [29] G. Hohpe and B.A. WOOLF. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. The Addison-Wesley Signature Series. Prentice Hall, 2004. URL: <http://books.google.com.au/books?id=dH9zp14-1KYC>.
- [30] Rainer Gerhards. *Rsyslog Documentation*. Version 8.26.0. Online; accessed July 11, 2022. 2017. URL: <https://readthedocs.org/projects/rsyslog/downloads/pdf/stable/>.
- [31] Rainer Gerhards. *Rsyslog Design and Internals*. Tech. rep. Dec. 2009. URL: <https://download.rsyslog.com/design.pdf>.
- [32] Rainer Gerhards. “Rsyslog: going up from 40K messages per second to 250K”. In: (Sept. 2010).
- [33] Roberto Gomez, Jorge Herrerias, and Erika Mata. “Using Lamport’s Logical Clocks to Consolidate Log Files from Different Sources”. In: vol. 3908. Apr. 2006, pp. 126–133.

- [34] K. Dooley and I.J. Brown. *Cisco IOS Cookbook*. Cookbook Series. O'Reilly Media, Incorporated, 2007. URL: <https://books.google.be/books?id=w-1SAAAAAAAJ>.
- [35] A.S. Tanenbaum and M. van Steen. *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017. URL: <https://books.google.be/books?id=c77GAQAACAAJ>.
- [36] Luca De Vito, Sergio Rapuano, and Laura Tomaciello. “One-Way Delay Measurement: State of the Art”. In: *IEEE Transactions on Instrumentation and Measurement* 57.12 (2008), pp. 2742–2750.
- [37] Han van der Aa, Henrik Leopold, and Matthias Weidlich. “Partial Order Resolution of Event Logs for Process Conformance Checking”. In: *CoRR* abs/2007.02416 (2020). arXiv: 2007.02416. URL: <https://arxiv.org/abs/2007.02416>.
- [38] I. Grigorik. *High Performance Browser Networking: What Every Web Developer Should Know about Networking and Web Performance*. O'Reilly Media, 2013. URL: <https://books.google.be/books?id=KfW-AAAAQBAJ>.
- [39] Kenneth Nawyn. “A Security Analysis of System Event Logging with Syslog”. In: (Jan. 2003).
- [40] *Performance Guideline for syslog-ng Premium Edition 6 LTS*. One Identity Support. Online; accessed July 12, 2022. June 2019. URL: <https://support.oneidentity.com/fr-fr/technical-documents/syslog-ng-premium-edition/6.0.20/performance-guideline-for-syslog-ng-premium-edition-6-lts>.
- [41] Nasko Oskov. *TLS overhead*. Online; accessed July 5, 2022. Mar. 2010. URL: <http://netsekure.org/2010/03/tls-overhead/>.
- [42] DevNet Cisco. *Application Resource Profiles - IOx - Document*. Online; accessed July 20, 2022. URL: <https://developer.cisco.com/docs/iox/#!application-resource-profiles/resource-profiles>.
- [43] J. Turnbull. *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, 2014. URL: <https://books.google.be/books?id=4xQKBAAAQBAJ>.
- [44] E.R. Harold. *Java Network Programming: Developing Networked Applications*. O'Reilly Media, 2013. URL: <https://books.google.com/books?id=LXsgAQAQBAJ>.
- [45] *Runtime options with memory, cpus, and gpus*. Online; accessed August 4, 2022. Aug. 2022. URL: https://docs.docker.com/config/containers/resource_constraints/.
- [46] Online; accessed August 6, 2022. URL: <https://kwseow.github.io/#iot-datasets>.
- [47] Shilin He et al. *Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics*. 2020. URL: <https://arxiv.org/abs/2008.06448>.
- [48] Yale University. *The Yale Literary Magazine*. vol. 47. Herrick & Noyes, 1882. URL: <https://books.google.be/books?id=iJ9MAAAAMAAJ>.
- [49] *Elasticsearch: The Official Distributed Search & Analytics engine*. URL: <https://www.elastic.co/elasticsearch/>.

- [50] *Kibana: Explore, visualize, Discover Data*. URL: <https://www.elastic.co/kibana/>.
- [51] Hemanta Kumar Kalita and Manoj Nambiar. “Designing WANem : A Wide Area Network emulator tool”. In: Feb. 2011, pp. 1–4.
- [52] *Knoppix linux live CD*. URL: <http://www.knoppix.org/>.
- [53] *tc(8) Linux User’s Manual*.
- [54] *Traffic control*. Online; accessed August 6, 2022. Dec. 2014. URL: https://www.funtoo.org/Traffic_Control.
- [55] Astel SPRL Grégoire Bourguignon. *Belgacom dévoile la nouvelle B-box 3*. Mar. 2013. URL: https://www.astel.be/info/belgacom-devoile-la-nouvelle-b-box-3_4271.
- [56] Vivien GUEANT. *Iperf - The ultimate speed test tool for TCP, UDP and SCTP*. URL: <https://iperf.fr/>.
- [57] BitWizard. *My Traceroute (MTR)*. July 2020. URL: <https://www.bitwizard.nl/mtr/>.
- [58] *Script for constraining traffic on the local machine - external/webrtc - git at google*. Online; accessed August 10, 2022. URL: https://chromium.googlesource.com/external/webrtc/+/refs/heads/master/tools_webrtc/network_emulator/emulate.py.
- [59] *recvmmsg(2) Linux User’s Manual*.
- [60] Dongkeun Kim and Jai-Yong Lee. “One-way delay estimation without clock synchronization”. In: *Ieice Electronic Express* 4 (Dec. 2007), pp. 717–723.
- [61] J. Postel. *Internet Control Message Protocol*. STD 5. RFC Editor, Sept. 1981. URL: <http://www.rfc-editor.org/rfc/rfc792.txt>.
- [62] Shaneel Narayan and Raymond Lutui. “Network Performance Evaluation of Jumbo Frames on a Network”. In: Dec. 2013, pp. 69–72.
- [63] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.