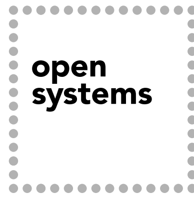

Messaging Challenges in a Globally Distributed Network

Raphael Thomas Seebacher

Master Thesis

MA-2013-02

Supervised by

Open Systems AG

David Schweikert
Stefan Lampart

ETH Zürich

David Gugelmann
Prof. Dr. Bernhard Plattner

September 13, 2013

Messaging Challenges in a Globally Distributed Network
by Raphael Thomas Seebacher (rse@open.ch)

Copyright © 2013 Open Systems AG, Switzerland.
All rights reserved.

MA-2013-02

This master thesis was under the patronage of

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ETH Zürich
Institute TIK
Communication Systems Group
ETZ Building
Gloriastrasse 35
8092 Zürich

and proudly conducted in the industry at

Open Systems AG
Räffelstrasse 29
CH-8045 Zürich
`https://www.open.ch`

Commit 4b51b5af4a9d4b3ca0922840521bf638d0ab7b69

Abstract

When operating more than 2600 hosts distributed in over 175 countries around the globe, as it is the case for Mission Control™ Security Services of Open Systems AG, efficient messaging is absolutely crucial in order to be able to monitor hosts, detect and react to incidents and, hence, to remain in control. With the current messaging architecture reaching its limits, this master thesis takes on investigating this architecture's properties to then design, prototype and evaluate a next generation architecture.

As messaging had evolved from simple remote procedure calls to object-oriented middleware, the current messaging architecture, named the Grand Unified Monitoring Architecture (GUMA), was built using the sophisticated Internet Communications Engine (ICE), a representative of the object-oriented middleware family. In our research, we found that the GUMA is trying to provide a solution for both monitoring, as well as messaging. Consequently, its architecture is rather complex and is currently growing out of manageability due to tight coupling, while lacking important features, such as bidirectional, reliable and persistent messaging and simple extensibility. Furthermore, ICE provides very sophisticated features that go far beyond the actual needs for Mission Control™ Security Services.

Based on the results found in our analysis of the current messaging architecture, we require a next generation architecture to be both modular and easily extensible, while still remaining simple and future-proof. Research on related work revealed that messaging solutions have further grown into message-oriented middleware. Therefore, we today have new building blocks readily available for designing and implementing this next generation architecture.

The newly designed architecture, as proposed in this thesis, consists of four key components providing a messaging layer completely transparent to use. At present, it is based on the Simple Text Oriented Messaging Protocol (STOMP) and independent instances of Apache Apollo message brokers, but designed such that these parts may be replaced in the future, without requiring changes outside the messaging layer. Contrary to the current architecture, applications may simply be built on top of this messaging architecture without requiring any changes being applied within the messaging layer itself; a consequence of proper separation of concerns and as loose coupling as possible. The new messaging architecture is furthermore designed to allow for arbitrary message payloads, giving the applications maximum flexibility. Additionally, support for remote procedure invocation is built into the messaging architecture, which facilitates, for example, host querying, distributed routing lookups and more. Finally, the new architecture is designed such that it also accounts for upcoming migration challenges: It may run in parallel to the current messaging architecture and allows for a highly granular host-by-host, process-by-process migration, having thus the least disruptive effect on Mission Control™ Security Services.

Acknowledgements

First of all, I would like to sincerely thank the management board of Open Systems AG, as well as Stefan Lampart, Senior Vice President Human Resources, for having given me the rare opportunity of writing my master thesis with this extraordinary company, whose spirit I was able to experience every single day I was in the office.

My special and most genuine thanks go to the distributed management team, namely its head, David Schweikert, as well as Konrad Bucheli, Ramon Schwammberger and Thomas Sturm for their tireless support regarding the many ideas I came up with, the insightful discussions on the problems I faced and all their feedback regarding my work. It was a real pleasure to work with that team indeed. In particular, the mentoring effort shown by David Schweikert is worth noting and I am grateful for having had him as a tutor, thereby having been able to profit from his profound knowledge and experience.

Even though this master thesis has been written at Open Systems AG, no less thanks are due to Prof. Dr. Bernhard Plattner, head of the Communication Systems Group at ETH Zürich, who rendered this thesis possible on the part of ETH Zürich and provided valuable inputs during the intermediate presentations. I am also deeply grateful for having had David Gugelmann as my supervisor on behalf of ETH Zürich. He offered many insights from a perspective outside of Open Systems AG, which proved to be of inestimable value and allowed for a more comprehensive treatment of my assignment.

Last, but most certainly not least, I would like to express my gratitude to all the employees of Open Systems AG for the interest they showed in my thesis, the advice they offered and the interesting discussions we had. Finally, my best thanks go to everyone I wasn't able to mention here explicitly, but who none the less contributed to the success of my master thesis.

So Long, and Thanks for All the Fish.[1]

[1]The rather humorous way the current messaging architecture bids farewell, when terminating its command line control interface. The phrase originates from the well-known book "The Hitchhiker's Guide to the Galaxy" written by Douglas Adams in 1979.

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 Open Systems AG 1
1.1.1 Mission Control™ Security Services 1
1.2 Assignment and Motivation 3
1.3 Contribution of this Thesis 4
1.4 Outline of this Thesis 4

2 Related Work on Middleware and Messaging **5**

2.1 Introducing Middleware 5
2.1.1 Classification of Middleware 6
2.1.1.1 Middleware Categories 6
2.1.1.2 Additional Classification Dimensions 7
2.2 Towards Message-Oriented Middleware 7
2.2.1 Remote Procedure Calls 8
2.2.2 Object-Oriented Middleware 9
2.3 Messaging and Message-Oriented Middleware 10
2.3.1 Message Queues 11
2.3.2 Messaging Models 11
2.3.2.1 Point-to-Point 12
2.3.2.2 Publish-Subscribe 12
2.3.3 Messaging Protocols 13
2.3.3.1 Java Message Service (JMS) 13
2.3.3.2 Advanced Message Queueing Protocol (AMQP) 13
2.3.3.3 Message Queueing Telemetry Transport (MQTT) 13
2.3.3.4 Simple Text Oriented Messaging Protocol (STOMP) 14
2.3.4 Message Broker 14
2.3.4.1 ZeroMQ (ØMQ) 15
2.3.4.2 Open Source Implementations 15
2.4 Benchmarking Messaging Middleware 16
2.5 Practical Applications of Message-Oriented Middleware 17
2.5.1 Messaging at CERN 17
2.5.2 Financial Applications 17
2.5.3 Wireless Sensor Networks 17
2.5.4 Further Applications 18

3 The Current Messaging Architecture Reviewed 19
3.1 Overview . . . 19
3.2 Components . . . 21
3.2.1 gumafilter . . . 21
3.2.2 gumaclient . . . 21
3.2.3 glacier2router . . . 23
3.2.4 gumaserver . . . 23
3.3 Evaluation . . . 23
3.3.1 Gathering Data . . . 23
3.3.2 Load-Balancing Evaluation . . . 24
3.3.3 Data and Message Rate Analysis . . . 26
3.3.4 Message Content Analysis . . . 28
3.3.5 Heartbeat-based Measurements . . . 30
3.4 Issues and Limitations . . . 31

4 A Proposal for a Next Generation Messaging Architecture 33
4.1 Requirements . . . 33
4.1.1 Transparency . . . 33
4.1.2 Bidirectional Communication . . . 33
4.1.2.1 Sending Messages from MSS Hosts . . . 33
4.1.2.2 Query MSS Hosts . . . 34
4.1.3 Reliable Messaging . . . 34
4.1.4 Scalability . . . 34
4.1.5 Further Non-Functional Requirements . . . 34
4.2 Designing the New Architecture . . . 35
4.2.1 Enabling Messaging from MSS Hosts . . . 35
4.2.2 Introducing the Messaging Gateway for MSS Hosts . . . 35
4.2.2.1 Persistent Message Queueing . . . 36
4.2.3 Allowing Message Consumers to Scale . . . 37
4.2.4 Integrating Host Querying into the Architecture . . . 38
4.2.5 Redundancy and Load Balancing . . . 39
4.2.6 Limitations of the Architecture . . . 40
4.3 Towards the Implementation of a Prototype . . . 42
4.3.1 Choosing a Message-Oriented Middleware . . . 42
4.3.1.1 The Messaging Protocol . . . 42
4.3.1.2 Selecting a Message Broker . . . 42
4.3.2 Interface Specification . . . 43
4.3.2.1 Passing Messages to the Messaging Gateway . . . 43
4.3.2.2 The Format of a Query . . . 45
4.3.2.3 Passing Information to Query Scripts on MSS Hosts . . . 46
4.3.3 A Perl-based Prototype . . . 46
4.3.3.1 Developing a STOMP Client . . . 47
4.3.3.2 Implementing the Messaging Gateway . . . 47
4.3.3.3 The Perl Module for Message Consumers . . . 50
4.3.3.4 The Host Querying Perl Module . . . 50
4.3.4 Availability, Reliability and Security Considerations . . . 51
4.3.5 Towards Deployment and Migration . . . 51

5 The Next Generation Messaging Architecture Prototype Evaluated 53
5.1 Setting up a Test Environment . . . 53
5.2 Evaluation of the Prototype . . . 53

5.2.1 Testing the Rate Limiter 54
5.2.2 Determining the Architecture's Delay 56
5.2.3 Evaluating Load Balancing 56
5.2.4 Failure of a Message Broker 60
5.2.5 Effect of a Network Outage 62
5.3 Evaluation Summary 64

6 Architecture Comparison, Future Work and Conclusion 65
6.1 Architecture Comparison 65
6.2 Opportunities for Future Work 66
6.2.1 Scientific Research 66
6.2.2 At Open Systems AG 66
6.3 Conclusion 67

List of Figures 71

List of Tables 73

List of Acronyms 75

References 77

1

Introduction

In the 1990s the Internet started to grow more and more popular among the public domain, as many companies slowly started making use of its advantages: Private and corporate websites commenced to emerge, emails were introduced and branch offices were connected to their headquarter over the Internet. The growth of the Internet itself and its popularity has been unbowed ever since. Unfortunately, but logically due to its growing popularity, the Internet also became of interest for malicious entities. While first occurrences were largely uncoordinated, isolated, based on simple attack vectors and against a large number of targets, we today face highly sophisticated, well-coordinated, well-funded and specific attacks against a well-defined target. In other words, the typical attackers are no longer curious individuals but rather companies, intelligence services and governments.

These days, we cannot possibly imagine life without the Internet. It has become a critical infrastructure, as daily operations largely depend on the availability of the Internet and the security of the appliances connected to it. Having efficient and effective security mechanisms available is, hence, absolutely crucial for business continuity. However, as a plethora of threats and different types of attackers exist nowadays and while the technology the Internet is based upon is growing in complexity, security is more and more difficult to achieve. Therefore, many companies have today outsourced their IT security infrastructure and trust managed security services offered by companies like Open Systems AG.

1.1 Open Systems AG

Open Systems AG is a company headquartered in Zürich, which has specialized in Internet security for more than 20 years. Apart from its managed security services named Mission Control™ Security Services, presented in the subsequent section, Open Systems AG runs large-scale international virtual private networks (VPNs) of large and midsize non-governmental organizations (NGOs) and companies from all industry sectors. In Gartner's MarketScope [6], an analysis of managed security service providers in Europe is presented. Therein, Open Systems AG received a positive rating and scored the highest proven customer satisfaction. At present, Open Systems AG operates over 2600 devices in more than 175 countries around the globe and monitors these hosts continuously, also relying on a security operation center (SOC) in Sydney, Australia.

1.1.1 Mission Control™ Security Services

Open Systems AG offers a variety of comprehensive and modular security services, known under the name of Mission Control™ Security Services. Below the security services are listed

as advertised on their website[1], in order to provide an impression of their scope of operation:

- **Mission Control™ Application Shield**
  Protect your web applications from internet attacks, and ensure secure and granular access, while reducing operational costs and maintaining auditability.
- **Mission Control™ Firewall**
  Separate multiple security zones, define authorized connections and transparently implement security policy with a centralized auditable relay station.
- **Mission Control™ Internet Proxy**
  Separate and organize web access according to a security policy, and protect users from direct attacks to their browsers.
- **Mission Control™ Security Gateway**
  Provide your branch offices with efficient site and communication protection to meet global objectives and local needs.
- **Mission Control™ Passport**
  Implement strong authentication and manage access authorization centrally. Reliably prevent misuse of passwords and eliminate the dangers of key logger phishing and specific attacks.
- **Mission Control™ Intrusion Detection (NIDS)**
  Scan the network continuously to detect and prevent intrusions and security breaches.
- **Mission Control™ Email Shield**
  Protect your email infrastructure effectively from overloading, spam, malware (viruses) and attacks from the internet.
- **Mission Control™ Client VPN**
  Work with the same comfort and security as in the office – at home or on the road.
- **Mission Control™ Load Balancer**
  The Mission Control™ Load Balancer allows the active use of redundant resources. Organizational resources such as application and proxy servers can be put into operation in parallel to increase the capacity and reliability of the services provided to internal or external clients. Typical usage scenarios include the load balancing of reverse proxy servers, application servers, and forward proxy server farms.
- **Mission Control™ BGP Router**
  Ensure stable and provider-independent operation of the Border Gateway Protocol (BGP) in your organization and reduce the operational overhead associated with changing an Internet Service Provider (ISP).
- **Mission Control™ Network Services**
  Use a single point of contact to manage and monitor all your ISPs centrally, and rightsize your global network to your business requirements. Benchmark your providers with the Mission Control™ lines in more than 800 cities in over 175 countries.

While the aforementioned services are selled to and therefore recognized by customers, it is important to note that a sophisticated and highly scalable configuration management and monitoring architecture lies at the core of Mission Control™ Security Services. Without such an architecture, operations at this large a scale would hardly be feasible, if at all.

[1] `https://www.open.ch/Content/1/1/Services.aspx`, last visited on 13.09.13

1.2 Assignment and Motivation

The monitoring architecture, named the Grand Unified Monitoring Architecture (GUMA), includes a messaging layer between more than 2600 remote devices and the central infrastructure of Open Systems AG. In this large, globally distributed system, messaging is both very important, but also rather challenging due to the scale and partially unreliable network connections. It is needless to say, that correct operation and availability of the GUMA is absolutely essential, as it is the eye of Mission Control™, the instrument for receiving messages from more than 2600 remote hosts, to then derive their statuses therefrom, which in turn enables detection of security incidents and allows for adequate reactions.

Consequently, Open Systems AG's security engineers are always improving both the GUMA, as well as Mission Control™ Security Services and introduce new features to the GUMA by challenging its architectural and technical limitations. Problems on remote hosts, which may for example cause that host to crash, result in messages, which are currently queued on that host, to be lost. This in turn complicates the identification of the problem's root cause, as potentially valuable information is no longer available. Furthermore, the load balancing mechanism is not optimal in that hosts are sticky, i.e. they do not periodically reconnect and, hence, prevent the load from being equally distributed amongst the servers that process the messages. Additionally, the current implementation was found to have drawbacks regarding maintainability, complexity, extensibility, as well as strong coupling between its various components, which more and more hinders further development.

This master thesis should therefore analyze the current messaging architecture and make proposals to improve its efficiency and reliability. To be more precise, we have been assigned the following four tasks, the importance of each weighted in terms of time to be allocated:

No.	Time	Description
1	10%	Study introductory material and related work regarding messaging, including the evolution of messaging protocols.
2	25%	Define an evaluation methodology to assess the performance, i.e. efficiency and reliability, of a messaging architecture and evaluate the messaging architecture currently operated by Open Systems AG.
3	50%	Design and implement a prototype for a next generation messaging architecture.
4	15%	Assess the performance of the next generation messaging architecture prototype and make a comparative analysis with the architecture currently in use.

Table 1.1: The assignment in terms of four tasks.

Apart from only analyzing and improving the current messaging architecture, the next generation architecture also has to account for bidirectional messaging, a new requirement. Having bidirectional messaging available allows for simple and fast querying of multiple hosts, e.g. regarding the status of their network interfaces. Additionally, persistent messaging has to be integrated into the new architecture as well, in order not to loose messages.

The huge number of hosts managed by Open Systems AG, as well as the worldwide distribution of their locations, provide a unique environment to put the prototype of the new messaging architecture to the test and potentially gain unique insights regarding messaging in general and the scalability of deployed products in particular.

1.3 Contribution of this Thesis

Based on the performance analysis of the current messaging architecture, as well as on reports by senior security engineers, we first inferred requirements for a new messaging architecture, then continued with its design, before finally evaluating suitable message-oriented middleware and implementing a prototype. The implemented prototype has the potential to, after further tuning for the production environment, replace Open Systems AG's entire messaging architecture. The newly designed architecture shows a considerable reduction in complexity and improvement in terms of maintainability. It is designed such that it provides a transparent messaging layer, and therewith allows for applications to be built on top of it without requiring knowledge of the messaging architecture's internals.

A message gateway on the remote hosts provides access to that messaging layer. It further provides persistent messaging by storing messages locally in an SQLite database until they are acknowledged, which in turn provides reliable messaging. To prevent the message queue in the message gateway from overflowing, for example during a network outage, messages may be given a queueing policy, such that only the newest message of a given type is retained in the queue, or that they are not queued at all. This reduction of messages, as well as a rate limiting algorithm in the message gateway prevent accidental flooding of the messaging architecture. Further, the message gateway has been equipped with an improved failover and load-balancing algorithm.

The new functionality for bidirectional messaging renders a whole new category of applications possible. So far hosts had to be queried based on shell scripting and `ssh`, a complicated and rather inefficient endeavour. The new query API now permits remote execution of predefined scripts on remote hosts. To allow for adequate querying of hosts, we have implemented support for setting a query scope. The same scope is already used internally for configuration management. It is thus, e.g., possible to query all Mission Control™ Email Shields of a given company.

We were even able to make a small contribution to the world of open source software with the development and public release of the perl module named `AnyEvent::STOMP::Client`. The asynchronously operating module `AnyEvent::STOMP::Client` provides an event-based client for the Simple Text Oriented Messaging Protocol (STOMP), currently our messaging protocol of choice for the new messaging architecture.

In a nutshell, we allow Open Systems AG's Mission Control™ engineers to work more efficient by providing a simple but powerful future-proof messaging architecture and enable new applications to be built without pain.

1.4 Outline of this Thesis

While this very chapter introduced and motivated the assignment, highlighted our contributions and provided the outline of our thesis, we delve into the world of middleware and messaging in Chapter 2 to introduce related work as well as the most important messaging concepts. In Chapter 3 we first present metrics to assess the performance of a messaging architecture and then evaluate the current messaging architecture of Open Systems AG, based on these metrics. The requirements for the next generation messaging architecture, the proposal of its design and the implementation of the prototype are illustrated in Chapter 4. An analysis of the performance of the new architecture, obviously based on the aforementioned metrics, is given in Chapter 5. Chapter 6 finally concludes this thesis, provides a comparison between the current and the new messaging architecture and shows opportunities for future work.

2

Related Work on Middleware and Messaging

A first analysis of our assignment, presented in the preceding chapter, particularly of our task to design and implement a prototype of a new messaging architecture, reveals the need for a piece of software that resides between custom high-layer applications and potentially multiple types of operating systems, network protocol stacks and hardware. This type of software is commonly known as middleware.

In this chapter we therefore first give a brief introduction of the middleware domain in Section 2.1, focusing on distributed applications. We show different types of middleware including their key concepts in Section 2.2, before delving into the world of message-oriented middleware in Section 2.3. In Section 2.4, we explore benchmarking possibilities for message-oriented middleware, and show practical applications of message-oriented middleware in Section 2.5. The purpose of this chapter is to give an overview of middleware and related work in order to identify and introduce the concepts relevant for our thesis.

2.1 Introducing Middleware

When researching the field of middleware, one notices rather quickly that quite a notable plethora of definitions does exist, with many only covering a very specific part of middleware. Consequently, a generally accepted definition of the term middleware is yet to be established. We therefore discuss selected definitions of middleware in the following[1], with the first definition originating from Gartner's IT-Glossary[2]:

Definition (Middleware). Middleware is the software glue that helps programs and databases (which may be on different computers) work together. Its most basic function is to enable communication between different pieces of software.

While this definition is rather brief, it points out the very essence of middleware glueing pieces of software together and allowing them to communicate. A more accurate definition of middleware is provided in the Free On-Line Dictionary Of Computing (FOLDOC)[3]:

Definition (Middleware). Software that mediates between an application program and a network. It manages the interaction between disparate applications across the heterogeneous computing platforms. The Object Request Broker (ORB), software that manages communication between objects, is an example of a middleware program.

[1] Note that this discussion is partly based on http://www.middleware.org/whatis.html (last visited on 13.09.13).
[2] `http://www.gartner.com/it-glossary/middleware/`, last visited on 13.09.13
[3] `http://foldoc.org/middleware`, last visited on 13.09.13

The FOLDOC points out nicely the middleware property of being a mediator and its ability to integrate heterogeneous platforms. The best definition of middleware, however, was found in [29]:

Definition (Middleware). Middleware can be thought of as software providing a set of enabling services that reside between applications and the underlying operating systems, network protocol stacks and hardware. Middleware allows multiple processes running on one or more hosts to interact transparently across a network and can also enable and simplify integration of heterogeneous software and hardware components.

From our point of view, the last-mentioned definition is the most profound, while still being reasonably brief. It not only points out essence of middleware of integrating heterogeneous systems, but also includes important integration properties like transparency and abstraction. As emerged from the above definitions of middleware, the range of potential applications of middleware is extremely broad. Typical use cases for middleware include, [23]:

- **Reusing legacy software**, where legacy applications are integrated into the enterprise-wide information system by connecting them to a standardized inter-application exchange bus. Due to the proprietary nature of their interfaces, a wrapper is used to translates the proprietary interface to a standardized interface compatible with the exchange bus.
- **Mediation systems** are typically referred to as systems responsible for various tasks like monitoring, logging and executing remote functions in a distributed environment with multiple devices interconnected by a network. The topology of that network is usually abstracted, resulting in the various devices being connected logically over a message bus, usually with asynchronous communication.

Furthermore, in [23], middleware is boiled down to the following four characteristics, consistent with the definitions given above and common to all middleware:

- **Hiding distribution**, i.e. the fact that an application is usually made up of many interconnected parts running in distributed locations.
- **Hiding** the **heterogeneity** of the various hardware components, operating systems and communication protocols that are used by the different parts of an application.
- **Providing standard and high-level interfaces** to application developers and integrators, so that applications can easily interoperate and be reused, ported, and composed.
- **Supplying** a set of **common services** to perform various general purpose functions, in order to avoid duplicating efforts and to facilitate collaboration between applications.

2.1.1 Classification of Middleware

Consistent to the lack of a common definition of middleware, a universal classification scheme for middleware is yet to be agreed upon. However, most classification schemes show a certain degree of overlap, e.g. the ones proposed in [14, 21, 29]. It is important to note that most of the following classification dimensions may not be orthogonal and overlap to a certain degree. There is, hence, some amount of arbitrariness in classifying middleware, depending on one's perspective.

2.1.1.1 Middleware Categories

A reasonably good way of classifying middleware is to categorize it by intended domain of application of the middleware. The following categories are common to both [14] and [29]:

- **Procedural Middleware** typically implements remote procedure calls, where a client invokes a procedure on a server and obtains the procedure's return value. The synchronized nature of communication in procedural middleware is generally considered a drawback. Section 2.2.1 provides additional information.
- **Object-oriented Middleware** takes on the concepts of object-oriented programming and transparently offers synchronous access to objects that may reside either locally or on remote hosts. Further insights are presented in Section 2.2.2.
- **Message-oriented Middleware** focuses on asynchronous messaging scenarios between a message producer and one or more message consumers. Typically message-oriented middleware requires for a centralized message broker, or message transfer agent, which may be considered a disadvantage. A profound discussion is given in Section 2.3.

Further middleware categories found in other works include, amongst others, transactional middleware, service-oriented middleware and event-based middleware. These categories may, however, be seen as a part of one of the above categories. For instance, current message-oriented middleware implementations typically have built-in support for transactions and may therefore also be counted among transactional middleware.

2.1.1.2 Additional Classification Dimensions

Even though the above classification of middleware into separate categories already gives a very good separation, one may further continue classification along the following dimensions, as illustrated in [14]:

- Proprietary systems with vendor lock-in versus open source and standardized systems,
- Language specific versus language agnostic architectures,
- Synchronous versus asynchronous communication between the components,
- The degree of coupling between the different components,
- Scalability of each component and the system as a whole [4].

The very nature of our assignment (cf. Chapter 1), i.e. the fact that we have to provide for a messaging architecture, allows us to already focus on message-oriented middleware, without the need of first analyzing requirements for a new architecture, as messages are, obviously, the core element of message-oriented middleware. For the remainder of this chapter we will therefore be focusing on message-oriented middleware, after having given more elaborate insights into remote procedure calls and object-oriented middleware. We do, however, encourage the keen reader to further explore the good introduction into the middleware domain found in [23].

2.2 Towards Message-Oriented Middleware

While message-oriented middleware may be considered the most recent type of middleware, other types do exists alongside it, two of which will be explained in the following, as previously mentioned. Again, one may generally state, that several features of these types of middleware are not completely orthogonal and have even been observed to be converging.

[4] One of the many challenges regarding scalability being the C10K problem, cf. `http://www.kegel.com/c10k.html` for a rather neat discussion of the problem (last visited on 13.09.13).

2.2.1 Remote Procedure Calls

Even though the term middleware is fairly new, the concept of remote procedure calls has been known for quite some time. First thoughts have been kept hold of in [45] back in 1976. As computers started being networked, communication with remote devices became possible and applications were able to interact over the network. This communication, however, was intransparent to the programmer and had to be explicitly built into the application, even though transparency is indispensable in order to allow distributed systems to scale.

The powerful and yet astonishingly simple concept of remote procedure calls is introduced in [5] by Birrell et al. and offers an elegant way to solve the above problem. It allows an application to call a procedure on a remote device and masks this procedure call as if being a local one. The calling application is suspended and the callee is computing the procedure's return value. This request and reply mechanism is transparent to the programmer, while under the hood a client-server model forms the basis for communication and marshalling of the transferred data, i.e. the procedure parameters and return values, takes place to account for potential differences, e.g. in endianness, between the two networked hosts. Figure 2.1 shows the concept of a simple remote procedure call. In spite of the simplicity of the RPC concept, not all problems can be masked. For example, the network may be congested, or either the client or server might fail to do its job and crash. To account for this, the programmer has to handle the following failure modes: In the "at most once" mode, the RPC system tries to call the remote procedure once and return an error to the programmer in the event of failure. It is then up to the programmer to deal with the error and potentially retry. The "exactly once" mode on the other hand tries to call the remote procedure a few times before returning with an error. It is important to note that "exactly once" cannot be guaranteed.

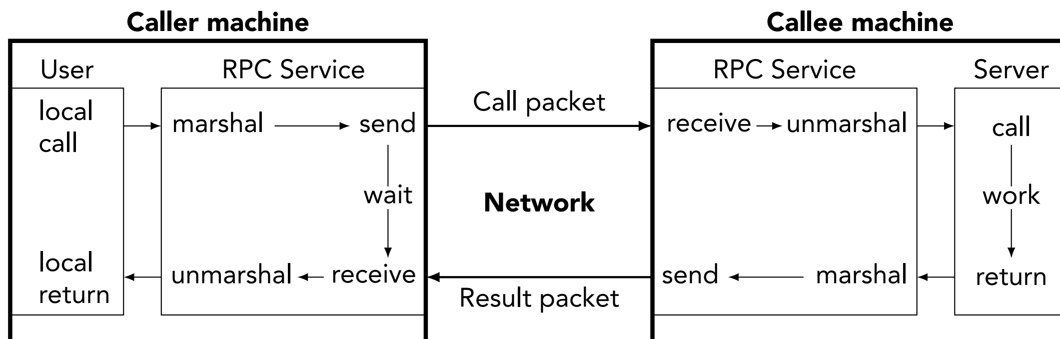

Figure 2.1: A simple remote procedure call, adapted from Fig. 1 in [5].

Remote procedure calls do, withal, suffer from a number of disadvantages. The most serious drawback of remote procedure calls is the synchronous nature of their request-reply communication, which results in a tight coupling between the client and the server. If, e.g., the server is suffering from a high load, a client calling a remote procedure may be forced to block for a long time. This in turn makes a multi-threaded architecture of the client necessary. It is also not possible to hide all problems that emerge due to the distributed nature of RPCs. In addition, implementations of the RPC mechanism are often highly vendor-specific and far from being standardized, resulting in incompatible implementations.

Much more detailed information regarding the implementation of remote procedure calls may be drawn from [5, 27, 40, 41]. As an example for remote procedure calls, one may, e.g., consider XML-RPC[5].

[5] http://en.wikipedia.org/wiki/XML-RPC, last visited on 13.09.13

2.2.2 Object-Oriented Middleware

While remote procedure calls represent the classic procedural programming style, object-oriented middleware introduced and promoted the possibility of using the more and more popular object-oriented programming model in order to build distributed applications. Nevertheless, it evolved fairly directly from remote procedure calls.

In object-oriented middleware, object can either be local or remote, as well as object references. Objects residing on remote machines are masked as if being local using a proxy object. The actual location of an object is, hence, transparent to the application. Behind the curtain the object request broker is responsible to map object references to object locations, i.e. to the remote hosts the object resides on and to pass information between the distributed applications. These general concepts are illustrated in Figure 2.2. Since distributed applications may be written in different programming languages, object-oriented middleware has to provide for some abstracted notation for objects, methods, parameters and inheritance. Therefore every object-oriented middleware consists of an interface definition language (IDL) providing this abstracted notation, that may then be compiled into the desired programming language, [28].

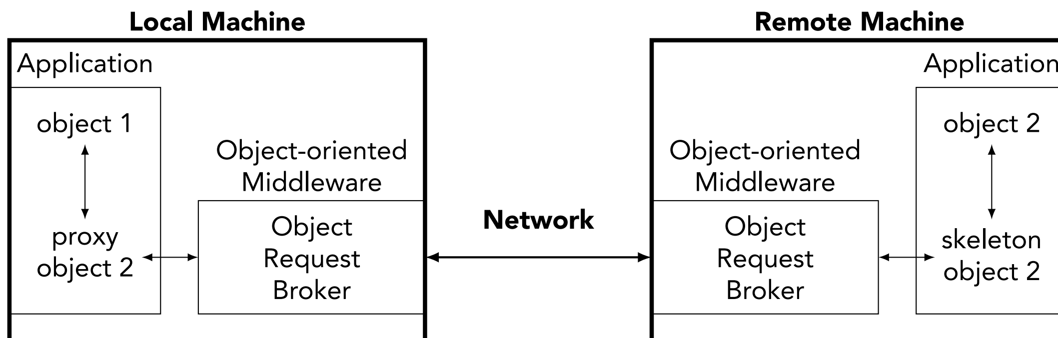

Figure 2.2: Generic object-oriented middleware architecture, adapted from [14].

Still, object-oriented middleware suffers from a series of deficiencies. As it is conceptually based on remote procedure calls, synchronous communication continues to be a problem, as well as rather tight coupling between the implementations. Modern implementations of object-oriented middleware, however, provide support for asynchronous communication. Furthermore a new challenge is introduced with the need for distributed garbage collection, since references to an object are typically also stored on remote machines. Finally, applications using object-oriented middleware tend to be rather static and heavy-weight, which one may consider a drawback for embedded devices and ubiquitous systems.

The Common Object Request Broker Architecture (CORBA)[6], specified by the Object Management Group (OMG), was the first representative of object-oriented middleware and remained the most important one for a considerable amount of time, [18, 43]. The CORBA offers the typical object-oriented middleware components as mentioned before, i.e. the CORBA IDL, an Object Request Broker (ORB), including an inter-Object Request Broker (ORB) protocol for communication between themselves. A prominent successor of CORBA is the Internet Communications Engine (ICE)[7], which lies at the core of the current messaging architecture of Open Systems AG, [17]. Another typical example of object-oriented middleware is the Remote Method Invocation provided by the Java programming language (Java RMI)[8].

[6] `http://www.omg.org/spec/CORBA/`, last visited on 13.09.13
[7] `http://www.zeroc.com/ice.html`, last visited on 13.09.13
[8] `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`, last visited on 13.09.13

2.3 Messaging and Message-Oriented Middleware

Just as remote procedure calls and object-oriented middleware, message-oriented middleware provides means for sharing data across processes within a distributed system. Contrarily, however, message-oriented middleware relies on asynchronous communication and therewith eliminates a major drawback of both remote procedure calls and object-oriented middleware, i.e. it allows for distributed applications to be more loosely coupled. On a logical level message-oriented middleware interconnects applications over a message bus or message channel and, hence, facilitates transport of information based on messages by abstracting the underlying physical network structure, which is shown in Figure 2.3.

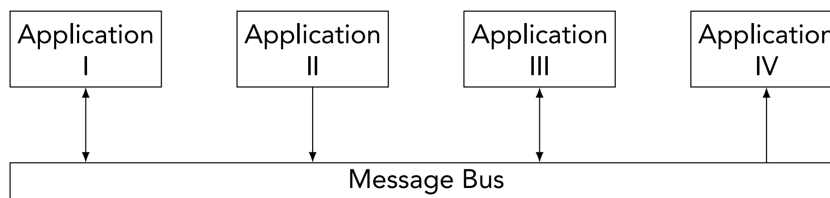

Figure 2.3: A logical message bus as provided by message-oriented middleware, based on `http://www.eaipatterns.com/Messaging.html`

Figure 2.4 illustrates the three distinct components message-oriented middleware consists of: the message producer, the message broker and the message consumer. Note that the message broker is sometimes also referred to as a message transfer agent. The message producer is responsible for generating a message and sending it to the broker. The broker, which may be considered a high-level message router, stores the message until a message consumer fetches the message for further processing. Consequently the producer and consumer are completely decoupled from each other, since they need not be available at the same time. Furthermore, it is important to note, that the broker may persistently store its message to, together with sending acknowledgements, provide for reliable messaging[9].

Figure 2.4: The three components typical message-oriented middleware consists of.

Message-oriented middleware offers key advantages in terms of coupling, reliability, scalability, availability. Due to the asynchronous nature of communication in message-oriented middleware, the interaction of different components, i.e. of message producers and consumers, is only loosely coupled, since they need not be online at the same time. This allows for faster response times, as applications do not have to block and wait for each other. Furthermore, decoupling reduces complexity of the applications and allows them to be isolated. The message broker, which acts as an intermediary between message producers and consumers, may further persistently store messages to prevent message loss in event of network or system failure. Besides the decoupling of their interaction, the decoupling of the performance characteristics of

[9]Note, however, that it cannot be guaranteed that a message is delivered exactly once, as nicely illustrated by the Two Generals Problem, e.g. in `http://en.wikipedia.org/wiki/Two_Generals'_Problem` (last visited on 13.09.13).

the different components also allows them to be scaled independently without affecting other parts of the distributed system. Finally, systems based on message-oriented middleware are also more robust against failures, as a failure in one component will not propagate through the entire system, but remain isolated.

A very good and more elaborate introduction into message-oriented middleware, including a comparison with remote procedure calls, is given in [9], as well as in [4]. Further information regarding enterprise application integration (EAI) and corresponding patterns may be found in [30] and in [19]. Note that EAI uses (message-oriented) middleware in order to glue different applications together and is thus not of particular interest for our thesis. It will therefore not be discussed any further.

2.3.1 Message Queues

Message queues are one of the most important concepts of message-oriented middleware. A message broker provides at least one message queue, but typically allows for the creation of virtually arbitrary many queues. When a sending a message to the broker, the producer chooses one queue as a destination for that message, given that multiple queues are available. Consumers, on the other hand, select a specific queue they want to work on, i.e. whose messages they want to retrieve and process. With multiple queue available, we have as many logical message channels between the producer and the consumer, despite the single physical connection between themselves and the message broker.

A message queue, as depicted in Figure 2.5, typically has multiple attributes. An essential attribute of a queue is its sorting algorithm, i.e. how messages are resorted within the queue upon the arrival of a new message. In practice first-in first-out queues are usually used. Other attributes include the name of the queue, which is used for its identification, a flag whether messages have to be buffered persistently, as well as the size of the queue itself. Access control may also be enforced for certain queues. Apart from normal message queues, several specialized queues are provided by message brokers. These may for example be temporary queues with a finite lifetime, or dead letter queues, which store messages that cannot be delivered or have expired.

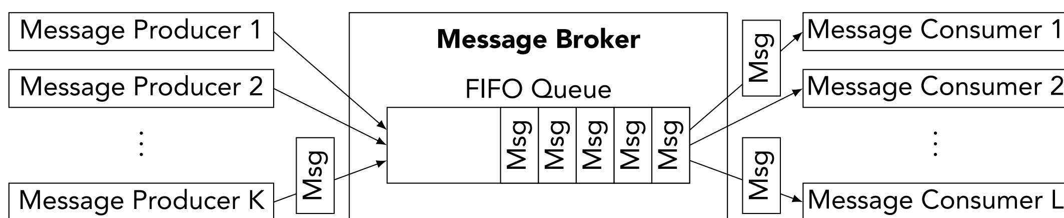

Figure 2.5: Message queue.

2.3.2 Messaging Models

There are two types of messaging models commonly available in the domain of message-oriented middleware. One model facilitates one-to-one communication channels, whereas the other provides for one-to-many communication channels. As each model has its advantages and drawbacks, a mixture of both models is typically being used in a practical setting. Note that it is a feature of the message broker, which messaging models it implements. While most brokers offer both models, there are some that only offer either of the messaging models.

2.3.2.1 Point-to-Point

In the so-called point-to-point model multiple producers and consumers attach to one or more queues in order to asynchronously exchange messages. The model guarantees that a given message is only consumed by a single consumer, even though multiple consumers may have attached themselves to the respective queue, as shown in Figure 2.6. Messages are, hence, exclusively consumed by any one of the attached consumers. By equally distributing messages between the consumers, the message broker introduces load balancing. Since consumers may acknowledge consumed messages, they gain the possibility to signal to the message broker when they are ready to consume the next message, which in turn allows the message broker to distributed messages between the consumers according to their capabilities.

In practice, one has to ensure that all consumers, which are attached to a queue, are of the same type, due to the exclusive consumption of messages and the unpredictability of the exact consumer that actually receives the message for processing. The typical application of the point-to-point model is the straightforward exchanging of messages between different parts in a system.

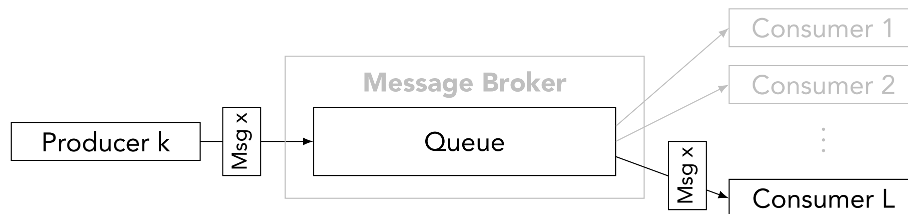

Figure 2.6: The point-to-point messaging model.

2.3.2.2 Publish-Subscribe

The publish-subscribe messaging model, depicted in Figure 2.7, provides for one-to-many, as well as many-to-many communication. Terminology is slightly different in this model. One speaks of a message publisher instead of a producer, a subscriber instead of a consumer and a topic instead of a queue. A publisher publishes to a topic it has information for, and subscribers subscribe to topics they are interested in. Upon the publishing of a message in a given topic, all of the subscribers will receive a copy of the message. A publish-subscribe message channel, hence, has broadcast semantics and is therefore typically used to disseminate information.

There are also some special features commonly available: Subscribers may additionally specify filters upon subscribing to a topic to further clarify what they are interested in. These filters may be content-based or publisher-based. Furthermore, the broker may retain the last message to provide new subscribers with the most recent message.

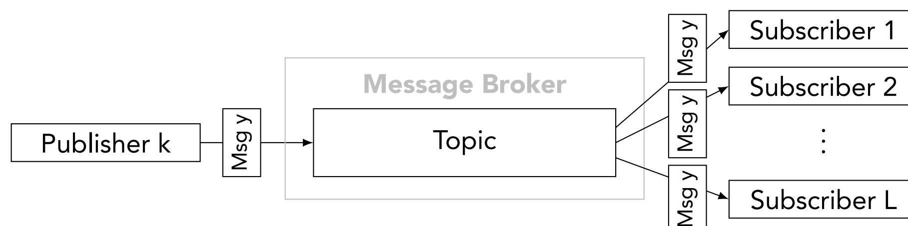

Figure 2.7: The publish/subscribe messaging model.

2.3.3 Messaging Protocols

While message-oriented middleware decouples producers from consumers, there is still a need for some standardization. The message broker and both the producer and consumer have to speak the same messaging protocol in order to be able to exchange messages. Even though, technically, different messaging protocols may be used between the producer and the message broker on the one hand, and between the message broker and the consumer on the other hand, one typically relies on the same messaging protocol for the sake of simplicity.

While early messaging protocols were proprietary and highly vendor-specific, we can today observe a trend towards more ubiquitous and standardized protocols. Despite that messaging protocols additionally differ in terms of verbosity on the wire and support for the aforementioned messaging models. Below, we present today's most important open messaging protocols, after having had a short glance at the Java Message Service (JMS). Note that there is a plethora of messaging protocols not mentioned here. Most of those are, however, very specific to either a certain programming language, or a certain vendor.

2.3.3.1 Java Message Service (JMS)

The Java Message Service (JMS) provides a standardized messaging application programming interface (API) to allow a client, written in Java, to interact with a message-oriented middleware, i.e. with a message broker. Even though the API supports both messaging models introduced above and has proven to be flexible and robust, the JMS is, however, limited in that it is specific to Java and has a certain degree of vendor lock-in. We consider this limitation to the Java programming language a drawback and will therefore not provide a more profound discussion of the JMS. Further information may be drawn from the official documentation[10].

2.3.3.2 Advanced Message Queueing Protocol (AMQP)

The Advanced Message Queueing Protocol (AMQP)[11] has its origins in the financial industry. Unlike JMS, which is a mere API, the AMQP specifies an efficient binary wire level protocol. This enables true interoperability across heterogeneous platforms and furthermore results in the protocol being programming language agnostic, as it is specified on the wire level without any coupling to any features of a specific programming language. Open standardization also ensured wide acceptance and results in AMQP being widely spread.

Most likely a consequence of many companies involved in the standardization process, the specification AMQP is complex and quite detailed. We consider this a disadvantage, as it renders implementations efforts more difficult and imposes a significant amount of complexity on the clients using the AMQP. Another imperfection lies in the evolution of the protocol, i.e. the lack of interoperability between the different protocol versions.

2.3.3.3 Message Queueing Telemetry Transport (MQTT)

Message Queueing Telemetry Transport (MQTT)[12] is a very simple messaging protocol designed for extreme robustness and efficiency. It is extremely lightweight in terms of bandwidth consumption, as protocol overhead is kept as low as possible. By requiring clients to specify their last will and a testament, MQTT is able to cope with high latency networks, frequent network outages, as well as unreliable networks. Furthermore, MQTT supports multiple QoS levels, including exactly-once message delivery based on a four step handshake. As MQTT assumes that clients are very limited in terms of processing power and memory, it imposes

[10] `http://docs.oracle.com/javaee/7/tutorial/doc/partmessaging.htm`, last visited on 13.09.13
[11] `http://www.amqp.org`, last visited on 13.09.13
[12] `http://mqtt.org`, last visited on 13.09.13

only very little requirements. A shortcoming of MQTT, however, is its lack of support of the point-to-point messaging model. This makes a potential use of the protocol for the purpose of this thesis improbable, as commonly both messaging models are used in practical applications. Typically, MQTT is used for internet of things applications, i.e. for constrained and embedded devices, which are typically deployed in large sensor and actuator networks.

2.3.3.4 Simple Text Oriented Messaging Protocol (STOMP)

Contrary to both AMQP and MQTT, the Simple Text Oriented Messaging Protocol (STOMP)[13] is not a binary, but rather a text-based messaging protocol, which results not only in increased verbosity on the wire, but also easier readability and therefore simpler debugging. Despite its verbosity, the STOMP is a lightweight and, most importantly, an extremely simple and easy to implement protocol, which results in wide interoperability, as well as a wide range of language bindings. Coming from the HTTP school of design, a typical STOMP frame is composed of a command on the first line, followed by colon separated key-value headers, a blank line and, optionally, a message body. Being so simple a protocol, a STOMP client may be implemented literally within hours.

A shortcoming of the STOMP was the decision to keep the protocol so simple at the expense of explicitness in specification. This resulted in different message broker implementations providing incompatible custom features, as these were deliberately not specified. It may therefore in some cases not be entirely straightforward to port code between different message broker implementations.

2.3.4 Message Broker

The by far most complex component within message-oriented middleware is the message broker, which acts as a mediator between message producers and message consumers. Not only has it to cope with a vast number of rapidly connecting and disconnecting clients, but also with a considerable amount of messages of various size, all with as little overhead and as much speed as possible. Implementing a message broker is thus a very challenging business and far beyond the scope of this thesis, as it requires profound knowledge of and experience in highly concurrent programming and various thread models in order to provide a stable and highly scalable message broker.

Typically a message broker offers multiple destinations, i.e. queues and topics, to which producers may send messages to and consumers may retrieve messages from. These destinations may either store the received messages transiently in memory, or persistently on a non-volatile drive. It obviously depends on the use case whether or not to store messages persistently, as it naturally comes with a performance penalty to store messages persistently. Common message broker implementations provide support for message transformation between different messaging protocols, thereby further improving interoperability. As a result of its ability to provide multiple destinations, a message broker is usually also referred to as a high-level message router.

One point, however, does require special attention when it comes to practical application of message brokers. As a mediator, the message broker naturally is the central entity between message producers and consumers and thereby becomes the single point of failure within the message-oriented middleware. In order to reduce this risk, one may employ a cluster of message brokers. Such a cluster causes, however, a considerable increase in complexity, since failover and recovery mechanisms between brokers are anything but simple and accidental message duplication has to be prevented.

[13] `http://stomp.github.io`, last visited on 13.09.13

Message Broker	Developer	Written in	License
Apache ActiveMQ	Apache Software Foundation	Java	Apache License 2.0
Apache Apollo	Apache Software Foundation	Scala	Apache License 2.0
HornetQ	Red Hat	Java	Apache License 2.0
Mosquitto	Roger Light	C	BSD License
RabbitMQ	VMware	Erlang	Mozilla Public License

Table 2.1: Characteristics of the chosen message brokers implementations.

2.3.4.1 ZeroMQ (ØMQ)

An exception in the world of broker-centric message-oriented middleware is ZeroMQ (ØMQ)[14], which offers a high performance, high throughput and broker-less messaging middleware. ØMQ essentially provides a transport agnostic socket library, i.e. with support for in-process, IPC, multicast and TCP communication, and offers more than thirty programming languages bindings. Despite its amazing performance[15], ØMQ is a rather low-level library and especially lacks support for message persistence, which would have to be implemented on top of ØMQ. Additionally a messaging protocol would have to be specifically designed, in order to build a fully featured ØMQ-based message-oriented middleware. The amount of work necessary to achieve this is far beyond the scope of this thesis and ØMQ is therefore not considered for further review. Nevertheless, we encourage the keen reader to have a glance at the extensive ØMQ guide[16], which offers interesting insights.

2.3.4.2 Open Source Implementations

As with messaging protocols, there is quite a plethora of message broker implementations available. Many of them are, however, proprietary, specific to a vendor and only implement a single, most likely also proprietary messaging protocol. Prominent representatives of such middleware includes IBM's MQSeries and Microsoft Message Queueing (MSMQ). In the following we will thus focus on message brokers that are open source and implement the messaging protocols discussed in the preceding subsection. Table 2.1 provides an overview of the chosen message brokers and their properties. Note that we focused on open-source message brokers that are still actively developed and maintained. A survey of the messaging protocols supported by the current version of the selected message brokers is given in Table 2.2.

Due to its lack of messaging protocol support, we exclude Mosquitto, as it only supports MQTT, which in turn only supports the publish-subscribe messaging model. HornetQ currently only implements support for STOMP, but has announced to support more messaging protocols in the foreseeable future. The Apache Apollo broker started out as a project to optimize Apache ActiveMQ's performance and has by now grown into a completely reimplemented message broker that shows much better performance than ActiveMQ. The chosen message brokers, except for Mosquitto, have been benchmarked using the STOMP messaging protocol[17] on different hardware. Based on that benchmark, as well as the messaging protocol support, we can draw the conclusion that the most promising candidates are both, Apache Apollo, as well as RabbitMQ.

[14]`http://zeromq.org`, last visited on 13.09.13
[15]`http://zeromq.org/area:results`, last visited on 13.09.13
[16]`http://zguide.zeromq.org`, last visited on 13.09.13
[17]`http://hiramchirino.com/stomp-benchmark/`, last visited on 13.09.13

Message Broker		AMQP	MQTT	STOMP
Apache ActiveMQ	5.8.0	✓	✓	✓
Apache Apollo	1.6	✓	✓	✓
HornetQ	2.3.0 Final	×	×	✓
Mosquitto	1.2	×	✓	×
RabbitMQ	3.1.5	✓	✓	✓

Table 2.2: Messaging protocol support of the chosen message brokers.

2.4 Benchmarking Messaging Middleware

With messaging middleware being closed-standard and proprietary, each vendor relied on its own benchmarking setup. Consequently, results emerging from these benchmarks were all but comparable and product comparison between different vendors was a daunting task. In order not to apply double standards when comparing messaging middleware, a generic benchmarks was required. It, nevertheless, has to be noted that while a generic benchmark is important for product comparison, it may not reflect the actual use case of the messaging middleware. In these cases a more specific benchmark would have to be designed.

First steps towards a standardized benchmark are presented in [42], where IBM's MQSeries is benchmarked and various performance metrics, as well as benchmarking test configurations are discussed. The notion of maximum sustainable throughput is introduced, with sustainable referring to a systems ability to maintain the current performance level for an unlimited amount of time. Efforts to provide an unbiased benchmark for evaluation of TIB/RV and SonicMQ is given in [24]. Metrics analyzed in the paper include message throughput and latency, the effect of the number of producers and consumers, and memory and CPU utilization during the benchmarking process. Still, this benchmark is not standardized, but tailored to that specific case, as the primary focus of the paper was the comparison of the two products.

With growing popularity of message-oriented middleware and with all major vendors adopting JMS, [35] and [34] proposed a generic benchmark named SPECjms, which provides a standardized means for evaluating message-oriented middleware via JMS. In contrast to proprietary benchmarks, SPECjms has a real-world scenario at its core and was designed to stress all critical services of a product, to generate reproducible results, to not have inherent scalability limitations and to not being optimized for a specific product. SPECjms is known to be the first industry-standard benchmark for message-oriented middleware, with major vendors having backed its development. A benchmark specifically for publish-subscribe-based message-oriented middleware is proposed in [33]. The benchmark named jms2009-PS is built on top of SPECjms. Amongst a discussion of the benchmark parameters, options for customization are presented, as well as a case study.

A set of benchmarks for financial applications based on AMQP, which originated in the financial services industry, are designed and evaluated in [39]. The benchmarks presented therein differ in the communication model, which is offered by AMQP, the number of producers and the number of consumers. As AMQP grew in popularity, the need for a more generic benchmark based on that protocol emerged. Based on an adapted version of the SPECjms and jms2009-PS benchmarks, [3] presents a method to evaluate AMQP-based middleware.

Based on the benchmarks presented above, a custom benchmark for specific use cases typically evaluates the latency to process a given batch of messages, the throughput in messages per second for a given load, the scalability and reliability of the system, its resilience against failures and outages, as well as the efficiency and overhead introduced by the middleware.

2.5 Practical Applications of Message-Oriented Middleware

Even though not always entirely evident, message-oriented middleware today powers many applications ranging from health care over supply chain management, wireless sensor network to financial applications. Unfortunately, little is documented about how message-oriented middleware is actually used, rather that it is used for a certain project. Furthermore, open source and open standard message-oriented middleware continues to emerge and yet remains to be widely adopted in the industry. Consequently, experience based on real-world large scale deployments is still to be gained, or, if already gained, corresponding research is to be made available to the public.

2.5.1 Messaging at CERN

A pleasant exception to the aforementioned lack of large scale deployments are the well-documented messaging efforts at CERN, with many documents publicly available. The CERN control system, as described in [12], is a three-tier architecture consisting of real-time processes reading data from sensors, data processing applications and graphical user interfaces. In order to allow for communication between the different components, the Controls MiddleWare (CMW) is used. The CMW itself consists of two products, i.e. a CORBA-based low level remote device access (RDA) application concentrating on low message latency and JMS-based message-oriented middleware with Apache ActiveMQ message brokers used for providing more high level services.

With a planned shutdown of CERN's Large Hadron Collider (LHC) for an entire year, an exclusive opportunity for reviewing and improving the CMW has arisen. Based on the analysis of middleware trends and market leaders given in [10], an improved RDA system is proposed in [11]. The new RDA system is based on ZeroMQ (ØMQ), which turned out to be the most suitable middleware for RDA according to the requirements specified in the paper. Furthermore, a new architecture for the distributed tracing facility, which is used to collect, analyse and correlate log events is proposed in [13]. The new facility is based on the STOMP, as well as ActiveMQ and has proven to be highly scalable in terms of performance.

With messaging growing more and more popular at CERN, [8] not only describes the current messaging service of the Worldwide LHC Computing Grid (WLCG) project, but also provides a prospect of how to use the messaging service in the future, including recommendations in terms of security, scalability, availability and reliability. Furthermore, valuable concepts regarding the design of messaging architectures and corresponding applications are discussed.

2.5.2 Financial Applications

The financial sector offers a prime area of application for message-oriented middleware, especially for financial market data delivery, apart from general EAI efforts. In [31] an information feed with financial market data is analyzed and an architecture for dissemination of that data is presented. AMQP remains, however, the most prominent example originating from the financial industry, as many large banks have been involved in its specification. Nevertheless, AMQP suffers from scalability issues. Approaches to overcome these limitations are presented in [26], where an architecture with broker federation is described to allow AMQP-based middleware to scale.

2.5.3 Wireless Sensor Networks

Another primary area of application for message-oriented middleware, which is increasingly researched, are wireless sensor networks (WSNs). With their network being significantly differ-

ent from traditional networks and their nodes significantly limited in available bandwidth and availability of power, WSNs pose additional challenges to middleware. [20] introduces and discusses the MQTT-S messaging protocol, which is an extension of the MQTT publish-subscribe protocol and accounts for the aforementioned restrictions in WSNs. The publish-subscribe-based middleware architecture Mires, which is specifically designed for WSNs, is introduced and analyzed in [36] and [37].

2.5.4 Further Applications

Message-oriented middleware is also increasingly used for web services. An example includes [25], where the design and implementation of WSMQ, a middleware designed to enhance the reliability of web services, is presented.

With increasing availability of mobile devices, their wireless communication capabilities facilitate flexible networks, which may be useful to provide for communication in case of emergencies. These networks do, however, pose new challenges for application design, as one has to account for delay tolerance. In [22] a delay-tolerant publish-subscribe message-oriented middleware is presented.

3

The Current Messaging Architecture Reviewed

This chapter is all about the Grand Unified Monitoring Architecture (GUMA), the current messaging and monitoring architecture of Open Systems AG. After having provided an overview of the current architecture in Section 3.1, we shed light on the components of the architecture in Section 3.2, evaluate the performance for several metrics in Section 3.3 to then pinpoint issues with and drawbacks of that architecture in Section 3.4. Even though this review of the current messaging architecture has not been the main focus of our thesis and has therefore been held reasonably brief, insights gained and issues discovered in this chapter build the foundation for the design of the new architecture, which is presented subsequently in Chapter 4.

3.1 Overview

Before exploring the peculiarities of the current architecture any further, it is important to gain an overview of the various components which are used for host monitoring and to differentiate the purpose of the current messaging architecture from that of the other services. As shown in Figure 3.1 with differently colored arrows, there are three basic means to communicate with Managed Security Service (MSS) hosts.

The secure shell (SSH) protocol is generally used in order to gain management access to MSS hosts from special SSH gateway hosts, while connections to the MSS hosts are monitored based on Internet Control Message Protocol (ICMP) Echo Requests and Replies and the `fping` utility. Contrary to the two aforementioned services, the Grand Unified Monitoring Architecture (GUMA) is the only channel that explicitly allows for communication back from the MSS hosts to the central infrastructure of Open Systems AG, as indicated with the yellow arrows in Figure 3.1. Whereas MSS hosts directly communicate with the central infrastructure, appliances rely on a MSS host as a relay in order to send information, as shown in Figure 3.1. The GUMA is, hence, the only possibility for MSS hosts to push information back to the central infrastructure and thereby becomes the eye of Mission Control™ Security Services, i.e. the instrument for allowing monitoring information to be transported to the central database. Note that all this communication may either be built on top of the plain internet protocol, using IPsec or even be running over an OpenVPN tunnel. All information gained from either connection monitoring or the GUMA is stored in `magmadb` database after having been processed appropriately.

When the GUMA has been developed in the late last decade, message-oriented middleware was not very mature yet, let alone standardized and open-source. Object-oriented middleware solutions, however, were quite mature at the time and consequently the GUMA was implemented based on the Internet Communications Engine (ICE) and the Qt framework[1].

[1] `http://qt.digia.com`, last visited on 13.09.13

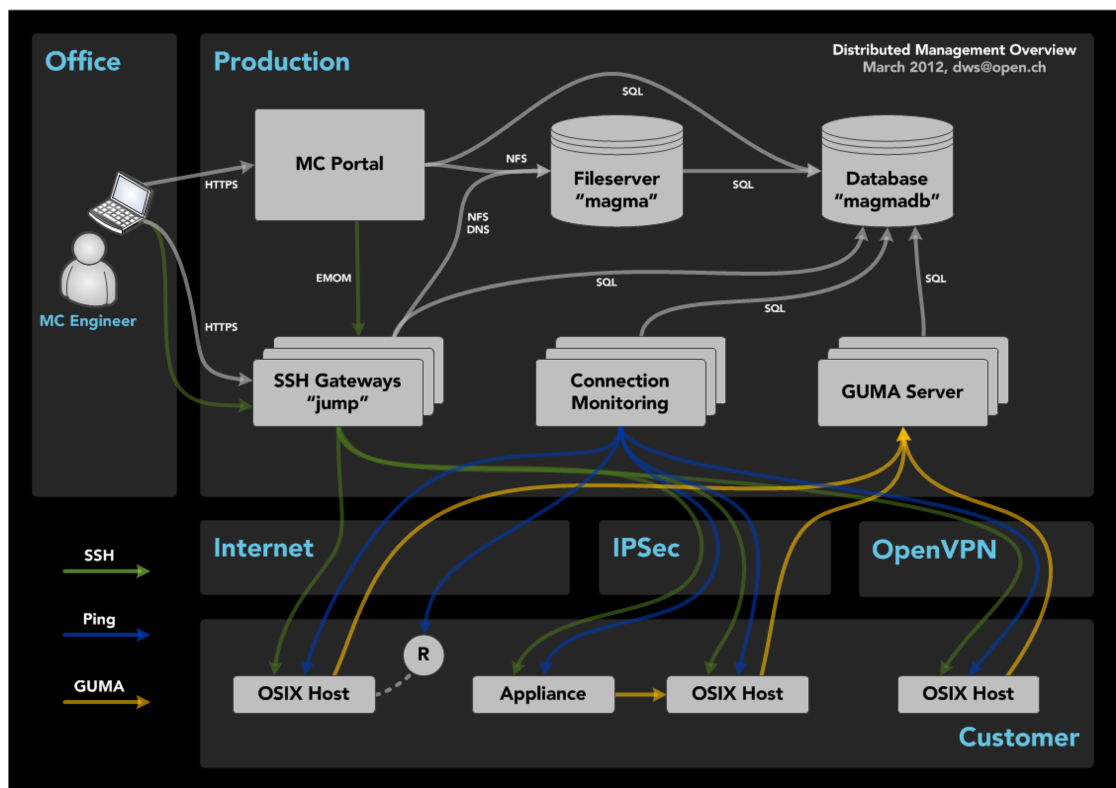

Figure 3.1: Distributed Management Overview, contributed by David Schweikert.

3.2 Components

Having shown an overview of the GUMA and especially its position within other systems in the foregoing section, this section provides a much more detailed view of the individual components of the GUMA. Figure 3.2 depicts these components and indicates their location, i.e. whether they reside on a MSS host at the customer or in the central management infrastructure of Open Systems AG, as well as it highlights the components implemented based on the ICE. We furthermore identify the three management hosts `chiba`, `dixie` and `magma`. The first two hosts, `chiba` and `dixie`, are characterized by their identical setup and are referred to as GUMAservers, whereas the latter host provides the `magmadb` database where the received information is stored, after it has been processed by either GUMAserver.

Despite `syslog-ng` being the primary source of information for the GUMA and the `magmadb` database providing the sink for information received and processed by the GUMA, we will not further discuss these components as a profound discussion would be out of scope of this thesis and since these components need absolutely not be included in a messaging architecture.

3.2.1 gumafilter

Compared to the traditional `syslog`, `syslog-ng` offers many more features. In particular it allows logs to not only be written to files, but also to pipes, i.e. to other processes. On all MSS hosts, `syslog-ng` has been configured with the `gumafilter` as a program log target, that is to forward logged events to the `gumafilter` for further processing.

The `gumafilter`, which is a script purely written in Perl, essentially tries to match received messages against a pattern library to then either drop the message, or to reformat it, add supplementary information and forward it to the `gumaclient`. The pattern library consists of a list of regular expressions with an associated signature, a rule whether or not to drop the message, as well as additional meta information such as identifiers of applications and hosts. If a message matches a given regular expression, the message is transformed into the Extensible Markup Language (XML) format and enriched with supplementary information retrieved from the corresponding entry in the pattern library. The assembled XML message is then passed to the `gumaclient`, given that the pattern library rule does not state to drop it. By default, messages that did not match a pattern are forwarded to the `gumaclient` in order to be able to learn about new messages and thereby to improve the pattern library. Clearly, the `gumafilter` is a component that belongs to the monitoring part of the GUMA and not on its messaging part. As we focus on messaging, the `gumafilter` will not be altered in the course of this thesis.

3.2.2 gumaclient

Written in C++ and based on the ICE framework, the `gumaclient` is responsible for the transport of messages from the MSS host to the central infrastructure. Internally, the `gumaclient` consists of a receiver thread, a non-persistent message queue, as well as a sender thread.

The receiver thread listens for TCP connections on port 7775 by default and accepts incoming messages in a simple line-oriented manner, i.e. it treats each received line as a single message. Without performing any syntax verification of the received message, it queues the messages for sending. Note that not only the `gumafilter` may send messages to the `gumaclient`, but also other processes. In practice, three additional processes send messages through the `gumaclient`: the network intrusion detection system `nids`, `keystats`, which collects various metrics to compute statistics, and the ISP link monitoring script `linkmon`. Messages sent by the `gumafilter`, however, form the majority, as shown in the following section, where message types are evaluated. The queues found in the `gumaclient` and the `gumaserver`

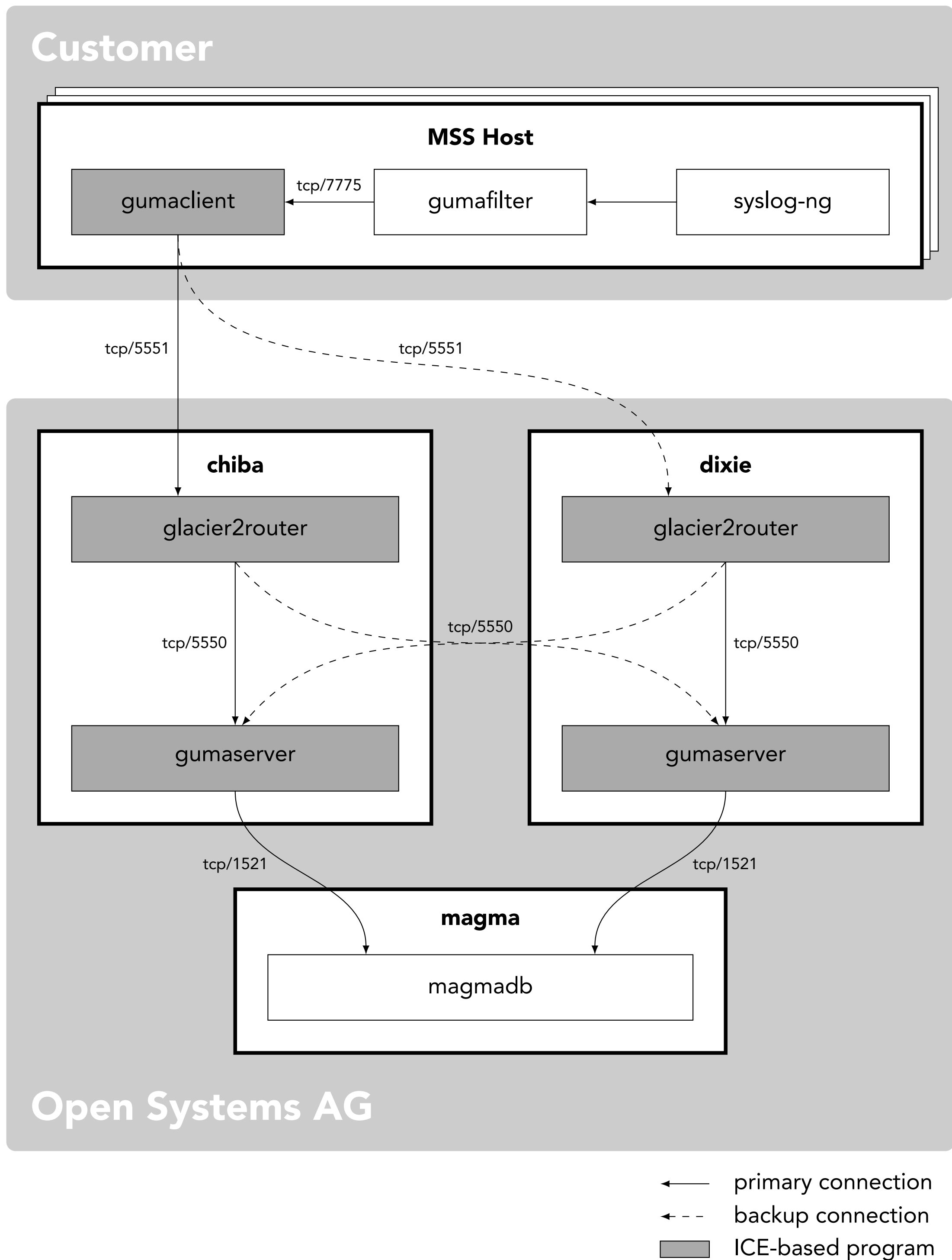

Figure 3.2: The Grand Unified Monitoring Architecture (GUMA) in detail.

are simple non-persistent FIFO queues of fixed size. If the queue is full and a new message arrives, the oldest message is discarded. The sender thread on the other hand is responsible for establishing a connection to either `gumaserver` instance, which are part of the central infrastructure, and to then send messages waiting in the queue of `gumaclient`. In order to send messages, the sender thread essentially invokes the `sendMessage` function on an object residing on the `gumaserver`. While, theoretically, the `gumaclient` might send its messages directly to the `gumaserver`, a `glacier2router` instance is joined up in circuit. Furthermore, a simple rate limiter ensures, that a specified message rate is not exceeded in the sender thread.

3.2.3 glacier2router

The `glacier2router` is a native component of the ICE, which acts as a connection concentrator. Instead of each `gumaclient` connecting directly to one of the `gumaserver` instances in order to send messages, the `glacier2router` accepts connections from multiple `gumaclients` and forwards incoming messages to a `gumaserver` instance over a single connection. Thereby, the `glacier2router` assumes the handling of the connection to each `gumaclient`, which may be of unstable nature. This removes a considerable burden off the gumaserver. Furthermore, the `glacier2router` implements failover in case of failure of a `gumaserver` instance.

3.2.4 gumaserver

Just like the `gumaclient`, also the `gumaserver` is not only based on the ICE and written in C++, but also consists of three components: a receiver thread, a non-persistent message queue and a database writer thread. While the receiver thread simply takes incoming XML messages and puts them in the message queue, the database writer thread is more complex. First the database writer thread checks the syntactical correctness of the message and discards the message in the event of failure. It then tries to determine the type of the message, extracts type-specific parameters from the message and invokes the type-specific stored procedure in the database with these parameters.

3.3 Evaluation

Evaluating the current messaging and monitoring architecture proved to be a difficult task, due to the fact that the GUMA is embedded in a production system. Being constrained in terms of methods of evaluation, access to the production system and with the evaluation not being the central task of our thesis, we naturally decided to frist collect and store messages that are transferred to the `gumaserver`, to then perform an offline analysis of these messages. In addition to this collection and analysis of messages, senior engineers have pointed out issues and drawbacks of the GUMA, which further helped in our evaluation. For the remainder of this section, we describe how we exactly gathered data, describe the dataset and illustrate the methods as well as the results of the evaluations.

3.3.1 Gathering Data

As mentioned above, we decided for an entirely passive evaluation with the least disturbance on the production system. We therefore dumped messages for a given amount of time to then proceed with an offline analysis of the captured data. We thus used `tcpdump` as shown below, in order to capture all data sent on port 5550 between each `glacier2router` instance and the corresponding `gumaserver` instance, i.e. on both `chiba` and `dixie`:

```
tcpdump -i lo -s 65535 -w guma-msg-[hostname].pcap port 5550
```

The collected dataset is roughly 1.5 GiB in size, which corresponds to a total of 7 113 479 packets. It has been collected during a typical working day, that is on the August 7, 2013. On chiba packets were captured from 06:02:42 UTC until 15:01:12 UTC, and on dixie from 06:02:43 UTC until 15:01:10 UTC. Subsequently to capturing raw packets, all sensitive information was removed from the dataset in order not to reveal any confidential information. Additionally, TCP packets without any payload, as well as packets with a payload other than a message transported by GUMA have been taken out of the dataset and messages have been added meta information. After these steps, the size of the dataset was reduced to 452.13 MiB, while still containing all 1 771 171 messages. Table 3.1 highlights some characteristics of the dataset collected for evaluation.

Note that this capturing is solely possible since the traffic between the glacier2router instance and the gumaserver instance is not encrypted, contrary to the traffic between remote MSS hosts and the glacier2router instances, where SSL/TLS and X.509 certificates are used in order to establish a secure channel.

3.3.2 Load-Balancing Evaluation

The goal of this evaluation is to determine how the load is shared between the gumaserver instances. We would expect it to be balanced equally amongst the instances, at least when compared not at a specific instant but over a larger interval of time.

Method

In order to obtain a measure of load balancing, we took the gathered data and split time into intervals of five minutes length. For each interval, we counted the number of messages per gumaserver instance, as well as the total number of messages. In addition to counting the number of messages processed per gumaserver instance, we additionally counted the size of all messages processed during a given interval for each gumaserver instance.

Results

The results are shown in Figure 3.3 for load balancing in terms of the number of processed messages, and in Figure 3.4 for load balancing in terms of the size of processed messages, respectively. In both figures, we have set H+0 to 07.08.2013 06:05:00 UTC and plotted each data point in the middle of the respective five minute interval. Furthermore, we decided only to show the first four hours of the measurements, since nothing of any significance changes for the remaining five hours of measurement, which are not shown in the figure.

Discussion

When looking at the results shown in Figure 3.3 and Figure 3.4, respectively, one immediately notices that the load is far from being evenly balanced between the two gumaserver instances. In fact, chiba has to consistently deal with approximately 25% more load compared to dixie over the entire period of data collection, i.e. for more than nine hours. This indicates that MSS hosts, once they have connected to either glacier2router instance, try to remain connected as long as possible. In other words, the MSS hosts are sticky and only reconnect to a glacier2router instance if absolutely necessary. According to security engineers at Open Systems AG, this may be due to chiba having a longer uptime compared to dixie.

We do furthermore note, that there is essentially no difference between the load balance shown in Figure 3.3 and Figure 3.4. That is, it does not matter whether we compute the load balance in terms of processed messages, or in terms of the size of the processed messages and further indicates that the majority of host show a similar communication pattern.

Characteristic	chiba	dixie	Total
Capture Time [s]	32 310	32 307	-
TCP Packets	4 302 930	2 810 549	7 113 479
Messages	1 110 756	660 415	1 771 171
Data Size [MiB]	968.740	591.235	1559.976

Table 3.1: Characteristics of the dataset collected for evaluation of the GUMA.

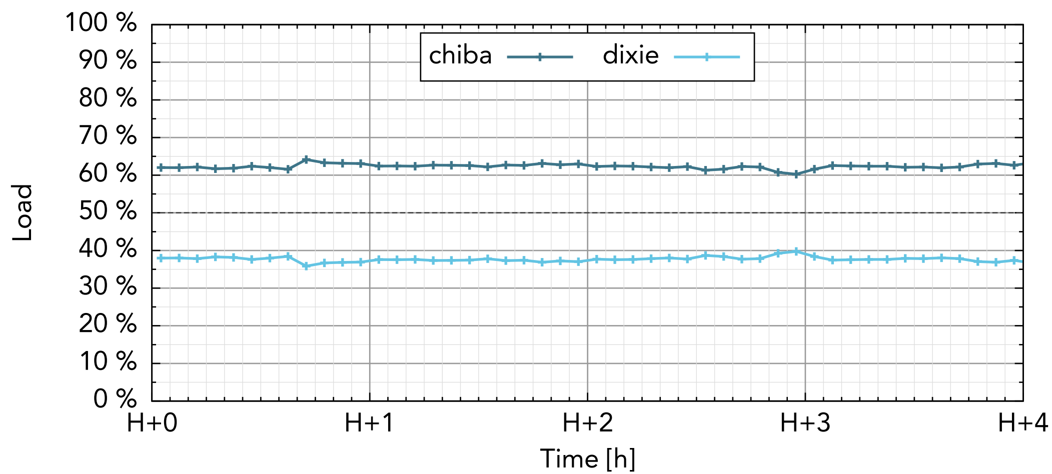

Figure 3.3: Load balancing measurements based on the number of processed messages.

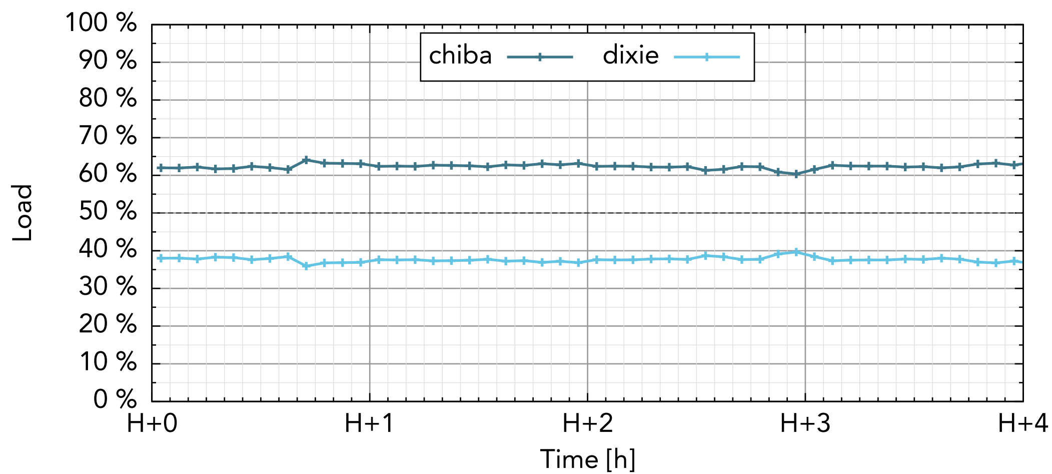

Figure 3.4: Load balancing measurements based on the size of the processed messages.

3.3.3 Data and Message Rate Analysis

As the load balancing evaluation compared the load of both `gumaserver` instances relative to each other, we are yet lacking an evaluation of a quantitative measure. A next evaluation is therefore dedicated to determine how much is actually sent by the MSS hosts, in terms of both, messages per second as well as bytes per second. Note that since we have already evaluated the load balancing between the `gumaserver` instances, we are only interested in analyzing the aforementioned rates for the whole architecture and not for `chiba` and `dixie` separately, nor for specific MSS hosts.

Method

To compute the message rate, as well as the data rate, we consider intervals of one minute in length. For each interval we count both the number of messages and the cumulative size of the messages and then divide both numbers by the length of the interval. Additionally, we computed the distribution of both rates.

Results

As an overview, the average of the message and the data rate, computed over the entire dataset, is given in Table 3.2. More details are revealed in Figure 3.5 and Figure 3.6. For both figures, we have set `H+0` to `07.08.2013 06:03:00 UTC` and plotted each data point in the middle of the respective interval. As the computed rates are of periodic nature, with a period of approximately 30 minutes, they are only plotted for two periods in both figures. Figure 3.7 and Figure 3.8 show the distribution of both rates for a bin width of half a second. It is important to note, that the distribution has been computed not only for the two periods shown in Figure 3.5 and Figure 3.6, but for the entire dataset.

Rate	**chiba**	**dixie**	**Total**
Average Message Rate [msg/s]	34.380	20.443	54.823
Average Data Rate [KiB/s]	15.638	9.279	24.917

Table 3.2: Average message and data rate.

Discussion

We can identify two different periods in both Figure 3.5 and Figure 3.6. One period is approximately of length 30 minutes, whereas the other is of length five minutes. Furthermore, we observe that many small messages are sent half-hourly by comparing both figures. We also note that the messages, which are sent every five minutes, are not that many but larger in terms of message size. The peaks, which occur half-hourly, originate from messages sent by `keystats`, an application, which sends statistical figures back to the central infrastructure. The other peaks, which show every five minutes are due to defect RRD files. Every five minutes an application wants to access these files and sends a corresponding error message to the `syslog`. Consequently, the log message is passed to the `gumafilter` instance, which then sends it via the `gumaclient` instance to the central infrastructure.[2] From the average message rate shown in Table 3.2, we can compute the average message rate per MSS hosts. Assuming 2500 MSS hosts, we get roughly 1.3 messages per minute per MSS host, as well as 10 bytes per second per MSS host. Note that each MSS hosts sends one heartbeat message per minute.

[2] Needless to say, this defect had been given high priority and has already been resolved.

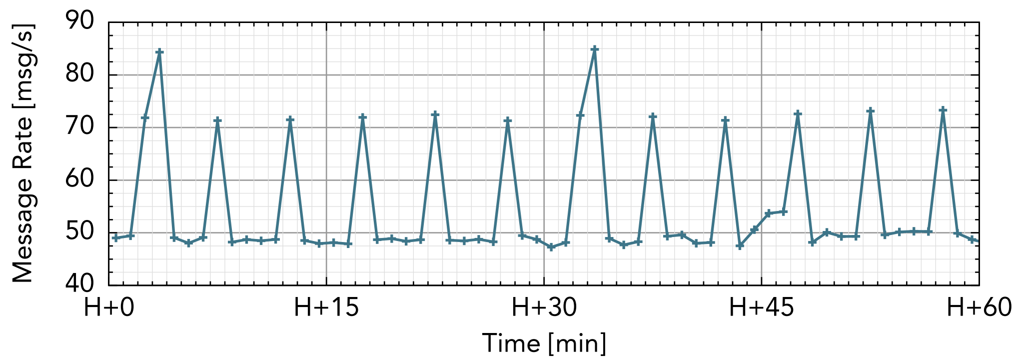

Figure 3.5: Message rate over time.

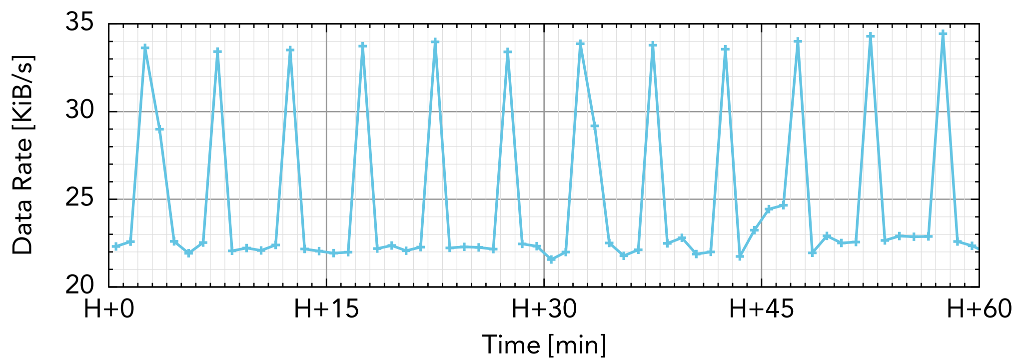

Figure 3.6: Data rate over time.

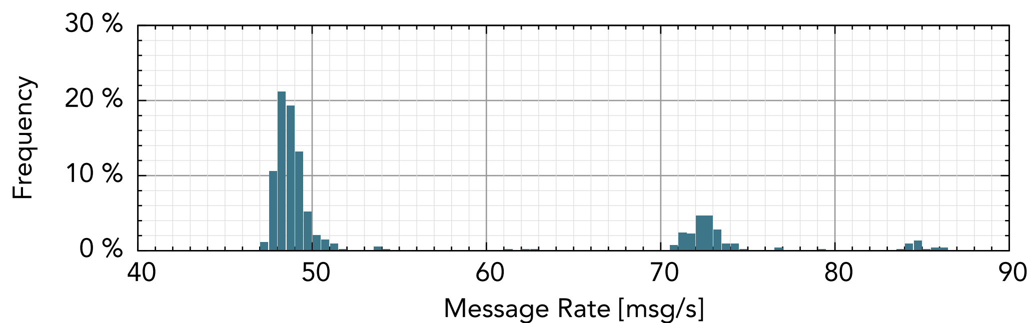

Figure 3.7: The distribution of the message rate measured at the central infrastructure.

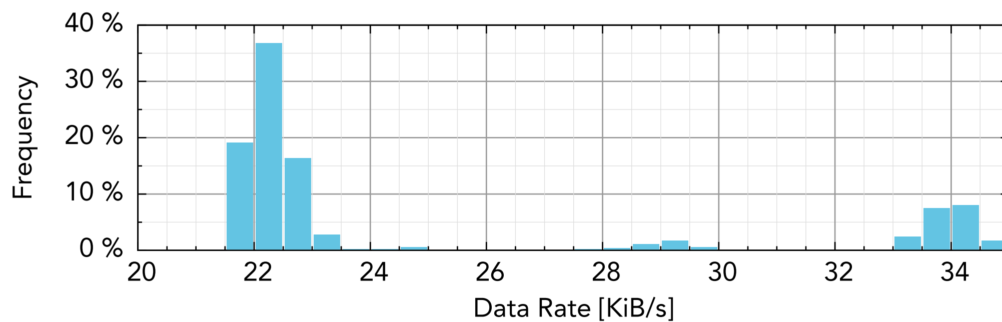

Figure 3.8: The distribution of the data rate measured at the central infrastructure.

3.3.4 Message Content Analysis

Up to now, we have analyzed how the load is balanced between both gumaserver instances and what data rates they have to deal with. In the following evaluation, we go farther and determine what the actual content of the captured messages is. That is, we assess what types of messages are transferred and to which extent.

Method

Each message transported via a gumaclient instance is formatted as an XML string, with the outmost tag being either <msg></msg> or <msg type="[type]"></msg>. The [type] is chosen by the application that sends the message to the gumaclient. If no type is given, then the message was sent by the gumafilter. As a first analysis, we extracted that type, or substituted it with gumafilter if it is not given, from each message and counted the occurrence of each type in the whole dataset.

As described in Section 3.2.1, the gumafilter assigns each message a signature, according to a predefined pattern library. Based on all messages sent from the gumafilter, we determined the occurrence of each GUMA signature.

Results

How many messages the different applications have sent to the gumafilter, i.e. the result of the first analysis regarding the messages types, is shown in Figure 3.9. Which GUMA signatures are sent most frequently is shown in Figure 3.10. Note that the latter figure only shows the four most frequent GUMA signatures, which do, however, make up approximately 99.6% of all messages sent through gumafilter.

Discussion

Figure 3.9 reveals that the gumafilter is by far the largest message source for the gumaclient. All other applications, that is keystats, linkcapacity and nids are only responsible for about 3% of the transported messages. Both keystats and linkcapacity send messages regularly from each MSS hosts, i.e. half-hourly for the former and every two hours for the latter. Messages originating form the network intrusion detection system (NIDS) are extremely rare and occur in an irregular manner. It is thus reasonable, to provide a more in-depth analysis of messages sent by the gumafilter.

The analysis of the occurrence of different GUMA signatures, depicted in Figure 3.10, shows basically two frequent signatures: NURSE:HEARTBEAT:OK and GUMA:UNKNOWN, which together account for almost 98% of all gumafilter messages. The message for the signature NURSE:HEARTBEAT:OK originates from a special script, which is running on every MSS host and designed to send a heartbeat message every minute to the gumafilter. Naturally, these heartbeat messages are amongst the most common ones. When the gumafilter is not able to match a message against any pattern in its pattern library, it forwards the message by default, as previously explained Section 3.2.1. Such messages are assigned the signature GUMA:UNKNOWN. The high amount of this signature in Figure 3.10 is also due to the broken RRD files mentioned in the foregoing section, since the corresponding error message is not registered and assigned a special signature in the pattern library of gumafilter. The third most frequent signature is TMON:UPDATE, which is assigned to messages that report the status of the VPN tunnels of the respective MSS host. AUTH:UPDATE2 finally is the fourth most common signature. It is attached to messages that report events originating from the RADIUS daemon. All remaining signatures have been merged into the other group. All these messages together occur with less than half a percent.

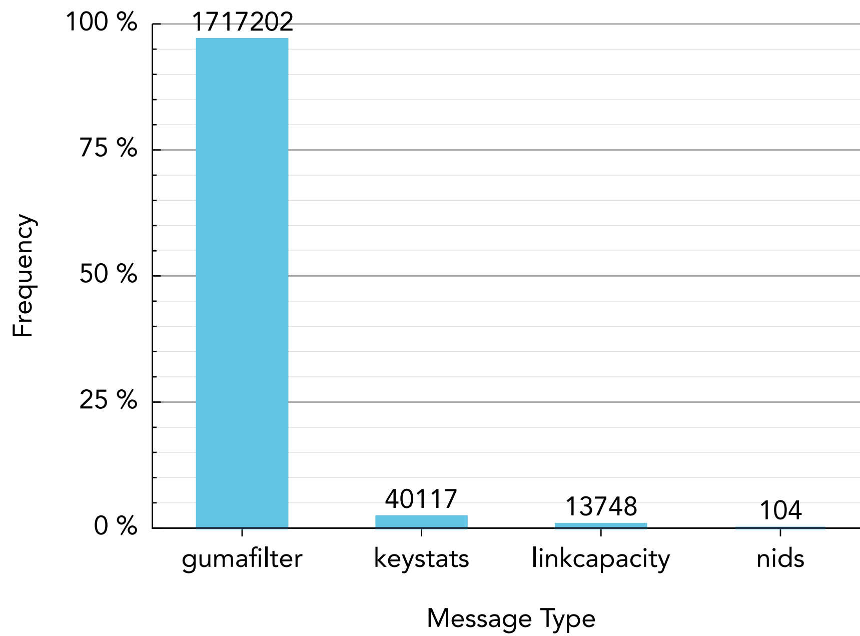

Figure 3.9: The frequency of the different message types as received by the GUMA servers. These types directly correspond to the message origins on the remote hosts.

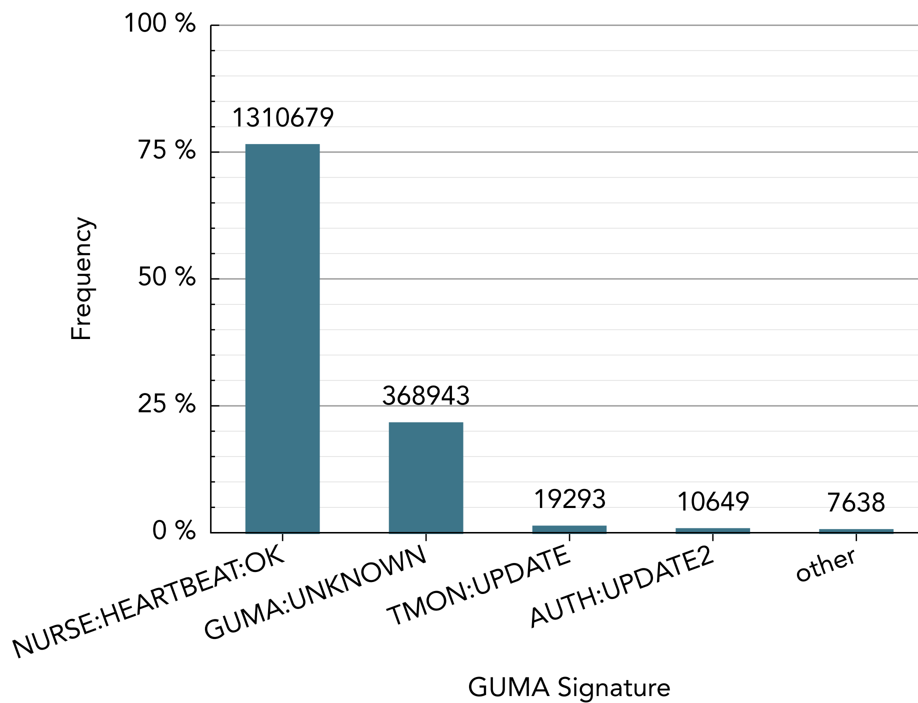

Figure 3.10: The frequency of the most common GUMA signatures.

3.3.5 Heartbeat-based Measurements

In the preceding section we have discovered that heartbeats, that is messages originating from the `gumafilter` with the signature `NURSE:HEARTBEAT:OK`, make up about 74% of all messages in the dataset collected for our evaluation. Since these heartbeats are additionally sent regularly every minute from every host, they are a prime target for evaluating the GUMA further. We may, hence, use that knowledge and analyze deviations from that heartbeat-specific pattern. Based on these heartbeats, we perform two reviews, one to determine the heartbeat delay and the other to estimate potential heartbeat loss.

Heartbeat Delay

In order to get a measure of the delay that the GUMA introduces, we compute the delay of each received heartbeat as follows:

$$\text{delay} = t_{\text{heartbeat capture}} - t_{\text{heartbeat generation}}$$

where $t_{\text{heartbeat capture}}$ is obtained using `tcpdump`'s `-tttt` flag and $t_{\text{heartbeat generation}}$ corresponds to the timestamp found in every heartbeat. This metric is, however, potentially not very meaningful, since the timestamps found in heartbeats unfortunately lack subsecond precision.

Plotted in Figure 3.11 is the distribution of the computed heartbeat delay from zero seconds until two seconds for a bin width of 50 milliseconds. This interval includes approximately 96.4% of all heartbeats that were recorded in the dataset. As suspected, the data shown in Figure 3.11 is rather inconclusive. Since the timestamp in the heartbeat lacks subsecond precision, we have an uncertainty in the measurement of one second. Consequently, all computed delays less or equal to second are useless. Additional uncertainty is introduced if the MSS hosts failed to synchronize their clock using the network time protocol (NTP). The sole conclusion we may draw from these delay measurements is that, generally, the delay is not more than a few seconds, which is considered sufficient for an architecture like GUMA.

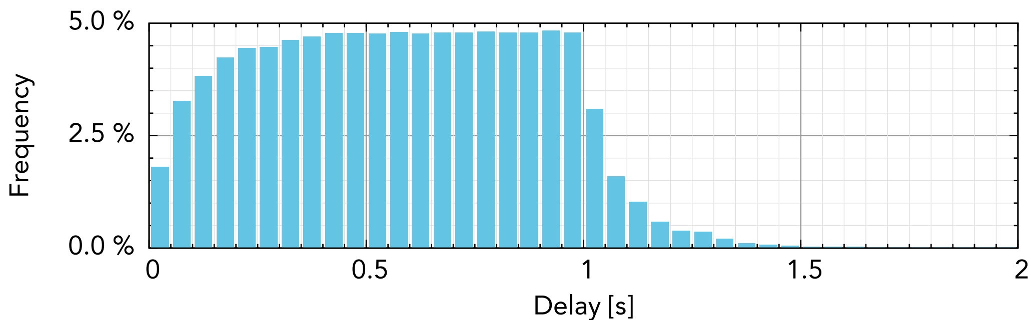

Figure 3.11: Distribution of the heartbeat delay.

Heartbeat Loss Estimation

To assess the reliability of the GUMA, we take advantage of the fact that every host sends a heartbeat every minute. We may thus compute the expected number of heartbeats for host i, using the formula

$$\text{expected heartbeats}_i = \frac{t_{i,\text{last heartbeat}} - t_{i,\text{first heartbeat}}}{\text{heartbeat interval}} + 1,$$

where $t_{i,\text{first heartbeat}}$ ($t_{i,\text{last heartbeat}}$) denotes the time when the first (last) captured heartbeat of host i was generated and the heartbeat interval equals sixty seconds. With that approach we are, obviously, not able to detect loss before (after) the seemingly first (last) heartbeat. Detecting this type of loss would require a much more complex loss estimation algorithm and we thus choose to accept that drawback of our approach. An estimation of the heartbeat loss for host i is then computed based on the subsequent formula, where captured heartbeats$_i$ denotes the amount of heartbeats of host i that have been captured for evaluation.

$$\text{estimated heartbeat loss}_i \quad = \quad \text{expected heartbeats}_i \; - \; \text{captured heartbeats}_i$$

Given the estimated heartbeat loss of every host, we assigned each host to a group as shown in Table 3.3. We notice that the majority of hosts, i.e. 2298 hosts, do not show any heartbeat loss at all. Up to 10 heartbeats were lost by 51 hosts. A loss of more than 10 heartbeats occurred on 10 hosts, most of which are located in Africa and typically suffer from unreliable network connections, e.g. via a satellite link. The host which lost 305 heartbeats was found to have considerable trouble with its internet service provider (ISP) and was thus unable to deliver all heartbeats.

Lost Heartbeats	Number of MSS Hosts
0	2298
1-10	51
11-100	6
101-124	3
305	1

Table 3.3: Estimated loss of heartbeats.

3.4 Issues and Limitations

We conclude this chapter on the Grand Unified Monitoring Architecture (GUMA), by summing up the foregoing evaluation and, in addition to that, by providing insights from senior security engineers to point out issues and limitations of the GUMA. When the GUMA was developed in late 2007, mature open-source message-oriented middleware was yet to emerge and therefore an architecture based on object-oriented middleware, i.e. the ICE, was created. Despite the powerfulness of the Internet Communications Engine (ICE), only a very limited subset of its capabilities are actually used in the GUMA. One may thus state that the underlying architecture, i.e. the ICE, is too sophisticated compared to how it is actually used within the GUMA.

In Section 3.3.2, we have seen that the GUMA lacks real load-balancing and that the MSS hosts are sticky. Furthermore there is no persistence of messages within the architecture, neither in the `gumaclient`, nor in the `gumaserver`. If either process is killed or the corresponding host is rebooted, then all messages are lost.

As has become apparent in the precedent sections, the GUMA does not only provide for a messaging architecture, but in fact also contains components responsible for monitoring. We consider this suboptimal. While on the MSS hosts we may identify the `gumafilter` as part of the monitoring architecture and the `gumaclient` as part of the messaging architecture, the `gumaserver` has to account for both, messaging and monitoring. This increases the complexity of the `gumaserver` considerably, as it is forced to be aware of message types, has to handle

each message and invoke the correct stored procedures on the database. With each additional message type, the development of the `gumaserver` becomes more involved and the probability of error rises. However, not only the development of the `gumaserver` poses challenges, but also its operation in the production environment. As it is so central to the entire architecture, a failure in one of the two `gumaserver` instances are highly critical and needs to be resolved fast, in order not to endanger monitoring services relying on the GUMA. If a failure occurs in practice, the respective `gumaserver` instance is just restarted, which causes all messages currently queued on that instance to be lost. Furthermore, with the utilization of stored procedures to store each message, high load is put on the database. This architecture based on the `gumaserver` and stored procedures in the database lacks proper separation of concern and has thus the fundamental drawback of not being able to individually scale according to the frequency of different message types. That is, the architecture cannot accommodate for the fact that heartbeat message are much more common than all other messages. Another drawback of the GUMA is the fact that it has been developed in C++, which is undesirable as today mostly Perl is used as a programming language, as well as the lack of sufficient possibilities for diagnosis and monitoring, especially of the `gumaserver`.

In addition to the drawbacks just discussed, new requirements have arisen. A new architecture must account for persistent messaging, on both the MSS host, as well as in the central infrastructure. The most important new requirement, however, is bidirectional communication. Today, the GUMA does only support one way communication, that is, from the MSS host back to the central infrastructure.

4

A Proposal for a Next Generation Messaging Architecture

The main contribution of this thesis, that is the design and implementation of new messaging architecture for Open Systems AG, is presented in this very chapter. We state the requirements for that architecture, admittedly in a quite abstract manner, in Section 4.1. Based on these requirements, as well as the issues of the GUMA that we discovered in the preceding chapter, we go on with the design of the new architecture in Section 4.2. In Section 4.3 we then provide insights into the implementation of a prototype of the new messaging architecture.

Before we continue with the analysis of requirements, it is important to highlight the difference between messaging and monitoring once again. While the GUMA accounted for both, monitoring and messaging, the new architecture is to focus purely on the latter. It is, hence, an enabling architecture for a monitoring system, that may be built on top of the messaging architecture, rather than being tightly coupled to the monitoring system. For the reasons indicated in Chapter 2, we rely on building blocks and concepts of message-oriented middleware in order to design the next generation messaging architecture.

4.1 Requirements

4.1.1 Transparency

Even though this requirement may sound entirely trivial, it is essential to ensure future proofness, as well as to maximize usefulness of the new architecture. With a transparent messaging architecture, we refer to the fact that applications are not required to have any knowledge of the messaging architecture's internals when using its capabilities. They do solely need to be aware of the interface to the messaging architecture. Additionally, the architecture has to be entirely agnostic regarding the messages it is asked to transport. Transparency further ensures future proofness, in that the internals of the messaging architecture may change without requiring any changes outside the messaging layer.

4.1.2 Bidirectional Communication

4.1.2.1 Sending Messages from MSS Hosts

As provided by the GUMA, the new messaging architecture also has to enable MSS hosts to send arbitrary messages to the central infrastructure of Open Systems AG. This use case can straightforwardly be implemented based on the point-to-point messaging model, which was introduced in Section 2.3.2.

4.1.2.2 Query MSS Hosts

Whereas the GUMA just had to fulfill the aforementioned use case, we require the new messaging architecture to additionally support a request-reply scenario. More precisely, we want the architecture to support querying of a subset of MSS hosts from the central infrastructure and getting the corresponding answer back from each reachable MSS host in the subset. Note that we do explicitly not require every single MSS host in the respective subset to answer the query, but rather those who are available at the time of querying. An example use case would be to query selected MSS hosts for the status of their VPN tunnels.

4.1.3 Reliable Messaging

We require the new architecture to be reliable. This means that in practice the utilized messaging protocol has to provide some mechanism for acknowledging messages, once they are successfully delivered. To further improve reliability of the overall architecture, we require different components of the new architecture to have built-in persistent message queues. While typical message broker implementations are already equipped with that functionality, it also needs to be implemented for the message queues on MSS hosts. Additionally, we want message queues on MSS hosts to provide for a queueing policy, where one may choose whether or not a message is to be stored persistently. If we take the regular heartbeat messages as an example and assume the connection between the MSS host and the central infrastructure to be unavailable, we want to be able to specify that the queue on the MSS host has to retain just the most recent heartbeat.

4.1.4 Scalability

Not only do we want the new architecture to scale well with the increasing number of MSS hosts, but we equally require each component to be able to scale separately. We thus demand a modular architecture with very well separated components. We want to account for the fact that some messages are more common than others, e.g. heartbeat compared to the messages of the NIDS, and provide appropriate resources for handling each message type.

4.1.5 Further Non-Functional Requirements

Maintainability Today, a majority of the applications developed by the Distributed Management Group of Open Systems AG are written in Perl. Naturally, we desire the new architecture to rely on that same programming language.

Vendor Independence We require the new architecture to be built based on open source software, as well open and standardized protocols.

Load Balancing Contrary to the GUMA, the new architecture has to balance the load of messages equally between both message brokers, as well as message consumers.

Safety In a production environment, we naturally want to have suitable failover mechanisms in place to be able to tolerate failure of any type of component of the messaging architecture.

Security Finally, we require our architecture to be secure, more precisely we especially require a confidential channel between all MSS hosts and the central infrastructure, as well as authentication of each component and integrity of messages that are transmitted, which is typically provided by transport layer protocols.

4.2 Designing the New Architecture

To design our new architecture, we straightforwardly take on the concepts message-oriented middleware that we introduced previously in Section 2.3. To allow the reader to comprehend how the new architecture was designed, we deliberately describe this process in a constructive manner, rather just presenting and discussing the finished architecture. Along the road to the final architecture, we highlight decisions we had to take including other design options we had.

In order to better illustrate the emerging architecture, we assume that, for the entire design process, there are two applications that need to send their messages to the central infrastructure. These are a heartbeat application, which sends a small status message every minute and the `syslogfilter` (formerly known as `gumafilter`), that forwards `syslog` entries given that they match a certain pattern. Nevertheless, the final architecture is to support arbitrary applications and is anything but restricted to these two applications.

4.2.1 Enabling Messaging from MSS Hosts

The two messaging models introduced in Section 2.3.2 allow us to fulfill the requirement for bidirectional communication quite easily. For now, we will first enable one way communication and integrate host querying into the architecture later.

To allow applications on the MSS host to send messages to the central infrastructure, we can use point-to-point messaging model as shown in Figure 4.1. Each application on the MSS host acts a message producer and sends its messages to the broker, which resides in the central infrastructure. Since message brokers typically support multiple message queues, it makes sense that each application uses its own message queue, even though thats not absolutely necessary. To process the messages that arrived on the message broker, we may either use one large message consumer, that handles all messages, or multiple application-specific message consumers. Note that the former case would roughly correspond to how the `gumaserver` works today. While this first draft of the new architecture as shown in Figure 4.1 would perfectly work in practice, it does not yet account for several requirements we stated above.

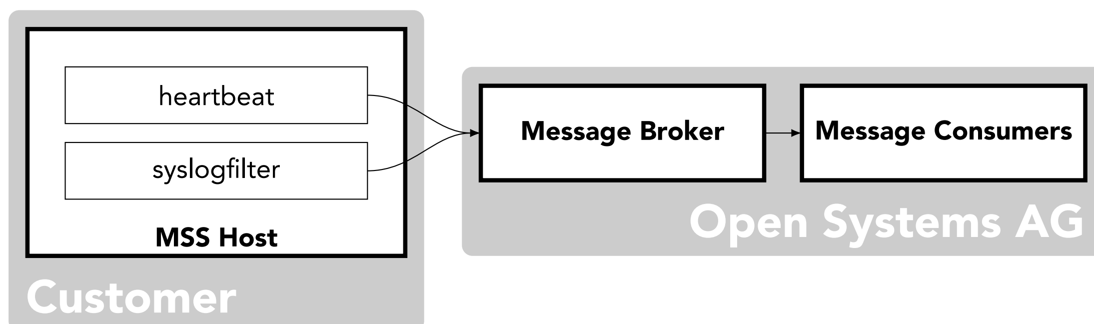

Figure 4.1: A first draft of the new messaging architecture.

4.2.2 Introducing the Messaging Gateway for MSS Hosts

On MSS hosts, we do lack transparency so far. In practice this means that every producer has to be aware of the message-oriented middleware, that is, to be precise, to implement the messaging protocol and have knowledge of the message broker's features. At the latest when replacing the message-oriented middleware with a successor, this lack of transparency would require changes in each application that relies on the messaging architecture. Furthermore

every message producer would currently establish its own connection to the message broker, which increases the number of incoming connections from MSS hosts tremendously and thus limits scalability of the entire architecture. It is evident that such an approach is suboptimal.

To overcome the latter issue and add transparency, we simply introduce an additional component to the architecture, that is similar to the `gumaclient` in the GUMA. This component, which we refer to as the messaging gateway, abstracts the message-oriented middleware and provides a simple interface for applications to send messages from the MSS host to the central infrastructure. With all messages now routed through this messaging gateway, we require the gateway's interface to provide for a subset of a messaging protocols features. In particular, the interface has to have some notion of a message's destination, such that the gateway may then route the message to the correct queue on the message broker. Additionally, the interface has to support reliable messaging. This boils down to the support of acknowledgements for messages, such that applications know, when the messaging gateway component was able to successfully handle the corresponding message. A further implication of the introduction of the messaging gateway is that the message broker now just sees one connection per MSS host, regardless of how many applications actually send their messages to the messaging gateway. Figure 4.2 shows the messaging architecture after having introduced the messaging gateway.

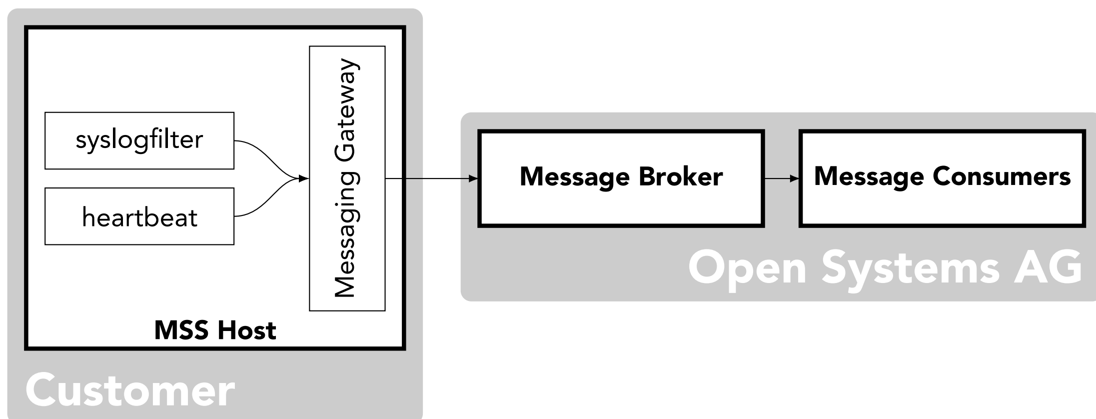

Figure 4.2: The messaging architecture with the newly added messaging gateway.

4.2.2.1 Persistent Message Queueing

With the messaging gateway available on every MSS host, the integration of a persistent message queue becomes a simple task, whereas without the gateway, each application would have had to implement its persistent message queue separately. Having a central message queue in the gateway does, however, require its interface to also support a notion of a queueing policy, in order to allow applications to specify how they want their messages to be queued, as well as a message lifetime, to limit the size of the message queue. Note that the queueing policy is applied per destination, as if each application had its own message queue.

The combination of the messaging gateway and an integrated persistent message queue has another key advantage. Since the interface provides for acknowledgements, it may now acknowledge messages to applications as soon as they are persistently queued, and not only once the message broker has sent its acknowledgement for the message. This decouples applications on the MSS host from the message-oriented middleware, that is, they may send their messages in an asynchronous manner, not having to care about how and when the messaging gateway actually interacts with the middleware.

4.2.3 Allowing Message Consumers to Scale

On the message consumer side of the new messaging architecture, we face similar challenges as on the producer side on the MSS hosts. That is, we have to account for reliability, transparency as well as scalability. In our evaluation of the GUMA, we have identified that the `gumaserver` does not completely fulfill these requirements. Most notably, it does not scale well in terms of message types, and puts a huge load on the database.

To now allow the message consumer side to scale appropriately, we choose to rely on multiple message-specific consumers, rather than on one large message consumer. Each message queue is to be worked off by a specific message consumer application. This decoupling allows message consumers to be horizontally scaled independent of each other, depending on their respective needs. Furthermore, the decoupling and specialization of message consumers not only results in smaller message consumer applications, which are thus much easier to develop, maintain and debug, but also allows for simpler extension of the messaging architecture. Note that decoupled message consumers need not necessarily run on the same host, which is thus increases flexibility of the new architecture further. Specialized message consumers finally make it feasible to move computation from stored procedures in the database to themselves and consequently free the database from unnecessary load.

Thus far, we have enabled message consumers to independently scale to their needs, but we have not yet considered transparency and reliability. To do so, we could basically choose an approach similar to the messaging gateway. Contrary to MSS hosts, however, we do not necessarily need asynchronous communication between the message consumers and the message broker. Actually, such an approach would tremendously increase complexity of the architecture, especially when it comes to reliable messaging. We therefore decided to make the architecture transparent by just providing a wrapper API that abstracts the messaging protocol and just provides the functions needed for consuming messages, as depicted in Figure 4.3. This decision implicates that every message consumer establishes its own connection to the message broker, which we accept, since the number of message consumer processes is well limited and they also reside in the central infrastructure. To finally account for reliable messaging, the functionality of the API needs to include the possibility to acknowledge messages upon successful consumption.

With the requirements now also applied to the message consumers, we have successfully designed an architecture that fulfills the use case of messaging from the MSS host to the central infrastructure together with most related non-functional requirements. In the subsequent section, we extend this architecture to enable bidirectional communication.

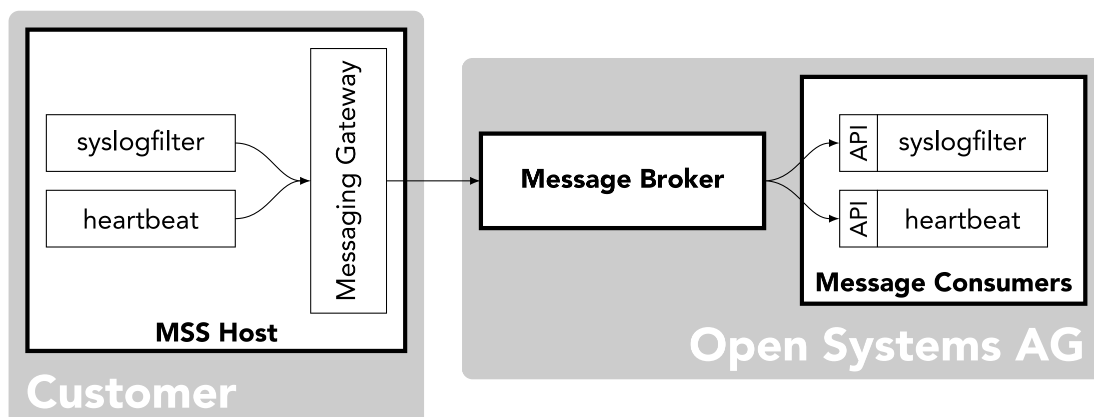

Figure 4.3: The messaging architecture with improved message consumers.

4.2.4 Integrating Host Querying into the Architecture

The second use case of host querying, presented in Section 4.1.2, is more involved compared to the first use case. The main challenge with host querying is to provide a means of sending the query to the respective MSS hosts. If querying had to be reliable, we would not get around having a dedicated query queue per MSS host. The querying application at the central infrastructure would have to include a logic to send the query to the appropriate queues. This approach, obviously, does not scale very well with the number of MSS hosts.

Fortunately, we are not required to reliably send queries to MSS host. As aforesaid, host querying is to be best effort, that is, queries are be sent to those MSS hosts, that are available at that very moment. We can therefore use the publish-subscribe model with a dedicated topic, where all MSS hosts subscribe upon connecting to the message broker and where we then can publish queries. In order to collect replies from each MSS host, we then again use the point-to-point model and may even rely on the already established messaging channel from MSS hosts back to the central infrastructure. With that concept of host querying, we need to integrate two components into our architecture, one component in the central infrastructure and the other one on MSS hosts.

At the central infrastructure, we need to provide an interface to the query architecture and still abstract the message-oriented middleware that lies at its core. For reasons similar to those for choosing an API for message consumers, we do also decide to provide an API for host querying. On the MSS host we need to be able to receive a query, compute the query's answer and finally send the answer back to the central infrastructure. Clearly, receiving the query and sending its answer are tasks for the messaging architecture, whereas the actual computation of a query's answer is not. We thus need to provide a component for receiving the query and an interface to the application that computes the query's answer. It makes sense to reuse the connection to the message broker that the messaging gateway has already established and, hence, to integrate the query component into the messaging gateway. In addition to sending messages to the broker, the messaging gateway will henceforth also subscribe to the aforementioned query topic in order to receive queries from the central infrastructure. Upon receiving a query, the messaging gateway will execute and monitor the application that computes the query's answer. Since a query requires a return queue to send its answer to and optionally also parameters, that the messaging gateway needs to pass these to the corresponding application. To not burden the messaging gateway further, the applications that computes a query's answer are to send the answer themselves, using the messaging

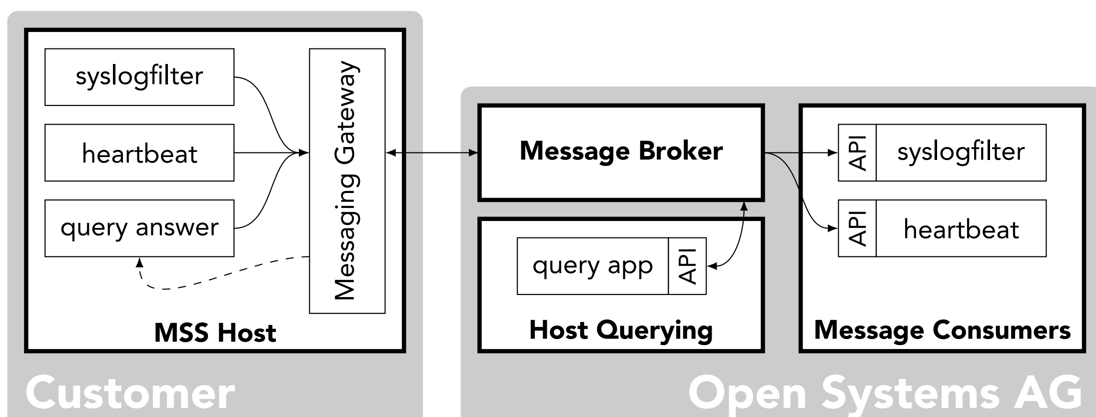

Figure 4.4: The messaging architecture with host querying integrated.

gateways interface. Note that, for reasons of security and reliability, the messaging gateway monitors the query applications that it invoked and terminates or even kills them if required. The new messaging architecture with querying built-in is shown in Figure 4.4.

In order to allow to query only a subset of MSS hosts, the API for host querying needs to support some notion of scope of an MSS host. Consequently, MSS hosts need to be aware of their respective scope and need to only answer those queries that have the same scope. A rather naïve approach to achieve this would be to have the messaging gateway doing the check whether the scope of a query matches that of the MSS host. Needless to say, such an approach would lead to a tremendous communication overhead, especially if only a single host is to be queried, since queries are always sent to all MSS hosts and then discarded by all but the specified host. A much better approach is to define a content-based filter upon subscribing to the aforementioned query topic. The message broker then only forwards a message to the messaging gateway, if the content-based filter matches the message. Obviously, this approach depends on the message broker providing for such a feature. Given that the message broker provides message filtering, which is the case for most message brokers, we need to specify how to represent a query's scope in the query itself, in order to be able to specify a corresponding filter when subscribing to the query topic on the message broker.

4.2.5 Redundancy and Load Balancing

The messaging architecture that emerged thus far already fulfills most of the requirements that we stated in the beginning of this chapter. One aspect, however, has not yet been given any thought yet. That is, we have not yet discussed how to protect the architecture from failures of its components, most notably failure of the message broker, followed by failures of message consumers or the host they are running on.

The typical approach to ensure the availability of the messaging architecture is to add redundancy to the system. We do basically have two options to do so. We can either provide a cluster of message brokers or make multiple independent message brokers available. However, we consider message broker clustering suboptimal for our messaging architecture, since on the one hand message broker implementations may not even support clustering and on the other hand clustering leads to a notable increase in the architecture's complexity. The other option to provide multiple independent message brokers is also suggested in [8] and offers a couple of key benefits. Despite being very simple, the architecture's availability is improved in that it works as long as one of the message brokers is available. Independent message brokers furthermore allow the architecture to linearly scale and additionally management of the architecture is simplified. An independent message broker may be added, stopped, upgraded or restarted at any time without having a disruptive effect on the messaging architecture as a whole. The downside of having independent message brokers is that each messaging gateway in the MSS host needs to be aware of every message broker. Consequently, if a new message broker is added at the central infrastructure, an updated configuration with the additional message broker included needs to be rolled out to the MSS hosts. The advantages of independent message brokers are, however, predominant.

With multiple message brokers available, the key challenge for both the messaging gateways, as well as message consumers is to decide to which message broker they should establish a connection to. As suggested in [8], message consumers use a consume-from-all pattern, that is, they connect to all message brokers and consume messages from the respective queue. There is thus no relation between a specific message broker instance and a given message consumer process. On MSS hosts the messaging gateways employ a produce-to-any pattern, i.e. they choose any of the available message brokers to connect to. By randomly choosing to which of the available message brokers the messaging gateway connects to, we implicitly

introduce load balancing. To further improve that load balancing mechanism, the messaging gateway can periodically re-establish the connection to any of the available message brokers chosen at random. Figure 4.5 gives an overview over the redundancy and load balancing concepts that we introduced in this section.

Despite the redundant and robust messaging architecture, several failure modes might occur, which we will briefly discuss in the following. Obviously, any component of the messaging architecture can fail. We can detect the failure of either an MSS host or just its messaging gateway by the absence of messages, especially of those that are to arrive regularly, that is, e.g., heartbeat messages. If just the messaging gateway fails, then applications on the MSS host can no longer send messages and have to decide themselves how to move on (e.g. just wait). Messages that have been queued within the messaging gateway are stored persistently and will survive the failure of the messaging gateway. In case of failure of the network connection to a MSS host, we have the messaging gateway on the MSS host persistently queueing messages until the network connection is back up again. Applications on the MSS host will not notice that the network connection failed.

More severe failures are those occurring in the central infrastructure. If a message broker fails, the messaging gateways on MSS hosts will immediately notice the broken connection and re-establish a connection to any of the remaining available message brokers. We can thus tolerate failure of up to $n-1$ message brokers, where n is the number of message brokers. Messages that were residing on the message broker at the time of failure are persistently stored, but unavailable as long as the message broker is not made available again. One has, however, to note that message brokers are not to be regarded as message stores, but rather as high-level message routers. Message consumer are thus to be scaled such that messages remain queued on the message broker as short as possible. Once the failed message broker is available again, message consumers will eventually reconnect to it and messaging gateways will also establish connections to that broker, due to the periodic connection re-establishment. Finally, if either the network between the message broker and message consumers, or the consumers themselves fail, we notice that the message queues on all message brokers are increasing in size, i.e. they are no longer being worked off. If, however, just a single message consumer fails, the respective queues will still be worked off as a consequence of the consume-from-all pattern introduced previously. We can thus also tolerate failure of all message consumers but one.

4.2.6 Limitations of the Architecture

Even though not that severe, the host querying architecture introduces a certain amount of communication overhead upon sending a query in some cases. If we consider Figure 4.5 and, for example, want to send a query to MSS hosts A, B and C, then the query API will send the query to each message broker. This can, however, be considered a minor design issue, since all traffic remains within the central infrastructure.

A much more serious limitation of the proposed messaging architecture is that reliable querying is quite hard to achieve, which is a direct consequence of choosing to deploy multiple independent message brokers instead of a message broker cluster. As mentioned above, reliable messaging would boil down to each MSS host having its own queue for queries. In our case with multiple independent message brokers, each MSS host would now have a query queue on each broker and consequently we would introduce the problem of having duplicates as queries are sent to all message brokers. A possible workaround would be to determine to which broker each MSS has connected to before sending the query. Since the messaging gateways periodically re-establish their connection to any of the message brokers, this workaround would lead to a race condition. Another workaround would be to require the messaging gateway to keep a state to be able to tell which queries have already been answered.

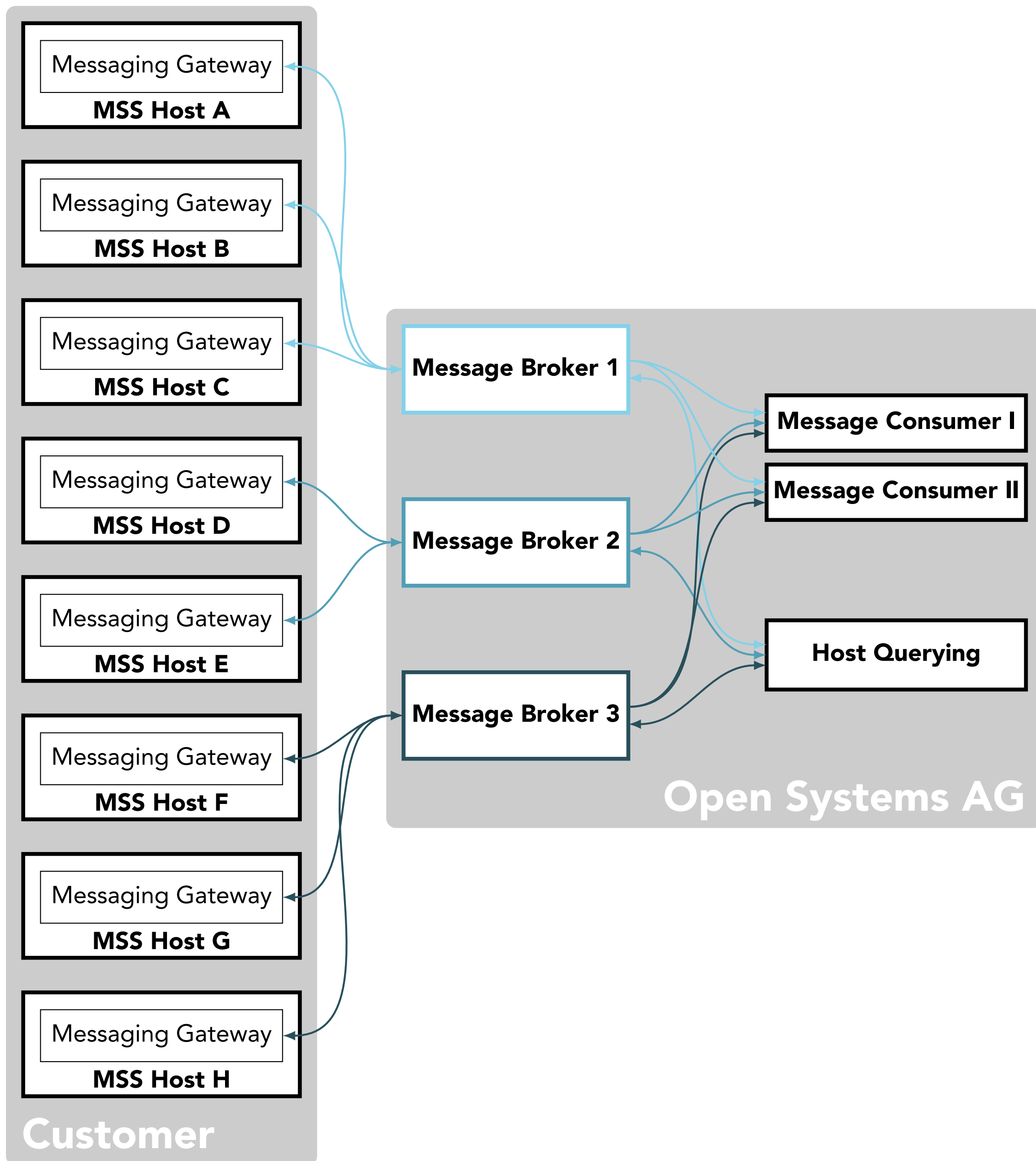

Figure 4.5: Redundancy and load balancing added to the new messaging architecture.

4.3 Towards the Implementation of a Prototype

Based on the foregoing, admittedly rather abstract design of the new messaging architecture, this section takes on the implementation of a prototype. We thus detail many things that we have just briefly explained above, most importantly the interfaces of the new architecture.

4.3.1 Choosing a Message-Oriented Middleware

Before going anywhere near implementing the new messaging architecture, we need to lay a proper foundation, that is, we have to select a suitable message-oriented middleware. In particular we have to choose a messaging protocol, followed by a message broker implementation that is compatible with the selected messaging protocol.

4.3.1.1 The Messaging Protocol

In Section 2.3.3 we have introduced the three messaging protocols AMQP, MQTT and STOMP. An overview of their characteristics is abstractly, but accurately depicted in Figure 4.6. Summing up the preceding section, we require a messaging protocol to especially support both messaging models and provide for acknowledgements. Therewith, MQTT already bows out of our selection of messaging protocols, since it only supports the publish-subscribe model, which leaves us with AMQP 1.0 and STOMP 1.2. Both messaging protocols fulfill all requirements we have, especially are there already implementations in the Perl programming language for both of them. We decided in favour of the STOMP, since it is much simpler than AMQP, which is also demonstrated in its specification that is about one-tenth in length compared to that of the AMQP. Furthermore, we are certain that the text-based STOMP is easier to debug in a production environment compared to the binary AMQP.

Figure 4.6: AMQP, MQTT and STOMP represented as moustaches, cf. [44].

4.3.1.2 Selecting a Message Broker

Having decided on using the STOMP, we now have to select a suitable message broker. In Section 2.3.4, where we introduced the concept of a message broker, we have already identified two potentially suitable message brokers, Apache Apollo and RabbitMQ. For the implementation of the prototype of the new messaging architecture, we decided in favour of Apache

Apollo for multiple reasons. First of all, the benchmark results given in [7] indicate superiority of Apache Apollo compared to RabbitMQ, as well as to the other message brokers introduced in Section 2.3.4. In addition to that, Apache Apollo was found to not necessarily transform messages internally, while RabbitMQ transforms messages to AMQP regardless of their original format, which may lead to more complexity. Furthermore, Apache Apollo has proven to be very simple to setup, whereas setting up RabbitMQ with STOMP was slightly more involved.

At this point, we advise the keen reader to delve into the specification of the STOMP version 1.2, as found in [38], the user manual of Apache Apollo 1.6 given in [2] and additionally also into the STOMP protocol manual of Apache Apollo as provided in [1]. Note that all three documents are very well limited in terms of their length and we thus consider that exploring them is a feasible undertaking.

4.3.2 Interface Specification

With the chosen message-oriented middleware at hand, we can move on towards the implementation of a prototype by specifying the new messaging architecture's interfaces to the outside. In particular, we have to provide for three different interfaces. Probably the most frequently used and thus most important interface is the one between applications on the MSS host and the messaging gateway. The other two interfaces are needed for host querying, in particular the interface for sending queries, i.e. how messages that contain a query are formatted, and last but not least the interface that defines how query parameters are passed to the application that computes a query's answer on the MSS host.

4.3.2.1 Passing Messages to the Messaging Gateway

In order to enable the exchange of messages between applications on MSS hosts and the messaging gateway, we need to specify three parts: First, by which means applications are to interact with the messaging gateway, second the format of messages, and third the format of acknowledgements. The keen reader will notice a certain degree of similarity between the following protocol and the STOMP protocol. We, obviously, have used STOMP as a source of inspiration for protocol design in order to also reduce the overhead of message transformation within the messaging gateway.

Interacting with the Messaging Gateway The concept of interprocess communication embraces several possibilities of making two processes on a given system exchange information. To enable applications to interact with the messaging gateway, we have chosen to rely on TCP sockets bound to a configurable port on the loopback interface and furthermore that applications are not authenticated by the messaging gateway. Applications may simply establish a TCP connection, send a message, receive the corresponding acknowledgement and disconnect from the messaging gateway. We chose to format the frames exchanged between the messaging gateway and applications in the style of the STOMP and consequently in that of the Hypertext Transfer Protocol. Below, we formally describe the format of frames using the augmented Backus-Naur Form (BNF) grammar, as it is also used in [15, Section 2.1].

```
LF             = <US-ASCII line feed>
CR             = <US-ASCII carriage return>
EOL            = [CR] LF
DIGIT          = <any US-ASCII digit "0".."9">
OCTET          = <any 8-bit sequence of data>
```

```
frame          = type "/" version EOL
                 *( header EOL )
                 EOL
                 *OCTET

type           = "OSAG-MSG" | "OSAG-ACK"
version        = DIGIT "." DIGIT

header         = header-name ":" header-value
header-name    = 1*<any OCTET except CR or LF or ":">
header-value   = *<any OCTET except CR or LF or ":">
```

Having formally defined the structure of frames, we further detail the structure by specifying mandatory and optional headers for both frame types. In the following we use inequality signs, i.e. < and >, to denote placeholders for actual values.

Message Format First, we detail the frame type for messages that applications may send to the messaging gateway, i.e. of OSAG-MSG. Essentially, we require such a frame to include some mandatory headers, while the frame body can be arbitrarily chosen by the application. Applications may further choose to add arbitrary many additional headers, as long as their names differ from the names of the mandatory and optional headers introduced below. A OSAG-MSG frame is of the following form, including all protocol-specific headers.

```
OSAG-MSG/<version>
destination:<destination>
content-type:<content-type>
content-length:<content-length>
expires:<expires>
queueing-policy:<queueing-policy>
queueing-ack:<queueing-ack>

<body>
```

The version currently has to equal 1.0. The header-names required in OSAG-MSG frames, including the corresponding header-values are described in Table 4.1. The <destination> describes the message queue, to which messages are sent to on the message broker. Applications can choose to set an arbitrary <body> of the message, but do need to specify the appropriate <content-type>, as well as the length of the <body> in bytes in the <content-length> header value. If a message is to expire, a timestamp in Unix Time can be set in <expires>. The message will, however, not expire unless this header is set. To set the policy for queueing messages, the queueing-policy header is to be used. Note that the queueing policy is applied per distinct destination. When an application wants to reliably send a message to the messaging gateway, the queueing-ack header can be used to instruct the messaging gateway to respond with an OSAG-ACK upon successful queueing of the respective message.

Acknowledgement Format Once the messaging gateway has successfully handled the message, i.e. once it has queued the message persistently in its message queue, an acknowledgement is sent back to the application that has sent the respective message. These acknowledgement messages are of the following form:

header-name	header-value(s)	Mandatory
destination	/queue/[A-z0-9_-]+	✓
content-type	MIME Type according to [16]	✓
content-length	Unsigned integer	✓
expires	Unix Time	×
queueing-policy	queue (default), retain-newest-<n>, where <n> is a positive integer, retain-newest, which is a shortcut for retain-newest-1, or no-queue.	×
queueing-ack	A nonce	×

Table 4.1: STOMP headers for messages sent to the messaging gateway.

```
OSAG-ACK/<version>
queueing-ack:<queueing-ack>
```

where the <version> corresponds to the version chosen by the application that has sent the message. The <queueing-ack> holds the queueing-ack value, which has been set in the message that is being acknowledged. As mentioned above, acknowledgements are only sent if the queueing-ack header has been set in the original message.

4.3.2.2 The Format of a Query

Queries that are sent from the central infrastructure to certain MSS hosts are essentially specially formatted STOMP messages. In practice, the headers given in Table 4.2 are to be included in a STOMP message for a query. The command header indicates, whether to start or stop the execution of the query application on the MSS host. Which application is actually executed on the MSS host in order to compute a query's answer is determined with the query header. For security reasons this header contains an alias for the application and not the path to the application itself, nor any executable code. The mapping between this alias and the actual executable application is given in the configuration file of the messaging gateway. In order to identify a specific query the query-id header is used and consequently it has to be allocated a nonce. To enable the invoked application to return the query's answer to the correct queue, the reply-to header has to be specified. The timeout parameter finally allows to restrict the execution time of the application invoked on the MSS host. The messaging gateway will thus terminate the application once timeout has been reached. The body of the query message can be chosen arbitrarily and will be passed to the application that is invoked by the messaging gateway on the MSS hosts, that receive the query.

header-name	header-value(s)
command	start or stop
query	query name
query-id	a nonce
reply-to	a valid destination
timeout	query timeout in seconds

Table 4.2: STOMP headers for query messages.

The Scope of a MSS Host To be able to send queries only to selected MSS hosts, we need a notion of scope of MSS hosts. Fortunately, the configuration management architecture used at Open Systems AG already provides different scopes for MSS hosts. Table 4.3 gives an overview over the available scopes. Each MSS host belongs to exactly one company, one VPN, one service type, one service and, obviously, one host. This association is registered in a database and may be retrieved in the form of `ids`. We can thus include the desired scope names together with the desired `id` as additional headers in a query message, in order to limit the scope of the query. If we do not want to limit the query with respect to one scope, we set the corresponding header value to `0`. Of course, different scopes may be used together to further refine the query. For example, one may send the query to all Email Shields of a specific company using both the `service_type` and `company` scope together. Note that all scopes have to match when multiple scopes are used.

Scope Name	Scope Description
`company`	All hosts of given company.
`vpn`	All hosts taking part in the same VPN.
`service_type`	All hosts of a specific type of service.
`service`	All hosts of a high-availability service.
`host`	One specific host.

Table 4.3: Scopes used for configuration management.

4.3.2.3 Passing Information to Query Scripts on MSS Hosts

The interface between the messaging gateway and the applications that compute the query's answer is rather simple. Once the messaging gateway receives a query, it extracts the name of the query given in the `query` header, determines the application associated with that name and invokes it. The messaging gateway then passes all necessary information to `STDIN` of the invoked application in the form

```
<reply-to>
<arguments>
```

where the first line consists of the parameter `<reply-to>`, which equals the destination specified in the respective query. All following lines, named `<arguments>` above, correspond to the body of the query message and may be used for whatever purpose required by the invoked application. Remember that the application that computes the query's result is to send the result via the messaging gateways interface, which has previously been specified.

4.3.3 A Perl-based Prototype

As we have already mentioned previously, the new architecture is ideally built using the Perl programming language, in order to simplify maintainability of the architecture. Apart from the message broker, which is an off-the-shelf component programmed in Scala, all remaining components can be implemented in Perl, not least due to the simplicity of the STOMP protocol.

A general challenge when developing such an architecture is concurrent programming. In order to avoid potential issues with thread-based programming in Perl, we decided on using the `AnyEvent` Perl module, which enables us to build event-based applications. The drawback

of using such an approach is, however, that all operations have to occur in a non-blocking manner. Fortunately, further modules that cover most use cases already exist.

4.3.3.1 Developing a STOMP Client

A STOMP client that both supports STOMP version 1.2 and that is compatible to the `AnyEvent` module was unfortunately not available. We thus developed that missing module ourselves, which resulted in the event-based STOMP version 1.2 client `AnyEvent::STOMP::Client`. Internally, the `AnyEvent::STOMP::Client` client is based on the Perl modules `AnyEvent` and `Object::Event`. With the release of that STOMP client on the Comprehensive Perl Archive Network (CPAN)[1] and on github.com[2], we contributed to the Open Source community.

We developed yet two additional Perl modules on top of our `AnyEvent::STOMP::Client` module, namely the `AnyEvent::STOMP::Client::All` module, which implements the consume-from-all pattern, and the `AnyEvent::STOMP::Client:Any` module, that provides the produce-to-any pattern[3]. Note that these modules include the implementation of the redundancy and load balancing concepts discussed in Section 4.2.5. With these three modules at hand, we can continue with the implementation of the remaining components of the messaging architecture's prototype.

4.3.3.2 Implementing the Messaging Gateway

Apart from the off-the-shelf Apache Apollo message broker, the messaging gateway is the most involved component of the entire new messaging architecture. Its prototype implementation consists of the application `msg-mss-gateway`, which basically is responsible for reading the configuration file and instantiating the messaging gateway with that configuration. Internally, the `msg-mss-gateway` relies on several Perl modules, which are all of non-blocking, event-based nature and offer object-oriented interfaces. A figure of the messaging gateway's internals is provided in Figure 4.7 and explained in the following.

The `OSAG::Messaging::MSS::Receiver` module establishes a TCP socket, listens for and handles incoming message frames. Upon receiving a valid `OSAG-MSG` frame, the module fires the `on_receive_message` event to advise the messaging gateway application of the message that has arrived. Furthermore a method for sending `OSAG-ACK` acknowledgement frames back to a connected application is provided. Additionally, the module fires events when applications connect to and disconnect from the established TCP socket.

With the `OSAG::Messaging::MSS::Queue` module, we implemented the persistent message queue for the messaging gateway. This module uses a SQLite database on the respective MSS host, in order to store received messages. Using a database with an SQL-based interface reduced the effort required to implement the queueing policy, which we specified afore. The module's core function are the `queue` method, obviously to store messages in the queue, and the event `on_message_queued`, which is fired once the corresponding message is persistently queued. Furthermore a method to clean up the message queue is provided.

Once a query is received from the message broker, it's the `OSAG::Messaging::MSS::Query` module's task to provide for query processing. The module relies on the `AnyEvent::Util` module to run the application that computes the query's result. Furthermore, it monitores the application and sends TERM or KILL signals, once the application's timeout is reached. Additionally, the module handles crashes of these applications and sends a corresponding error message to a predefined query error queue on the message broker. The module further

[1] `http://search.cpan.org/~raphi/AnyEvent-STOMP-Client`, last updated on 16.08.13
[2] `https://github.com/raphiniert/AnyEvent-STOMP-Client`, last updated on 23.08.13
[3] These two modules have not been released on CPAN, as they are yet to be finalized for public use.

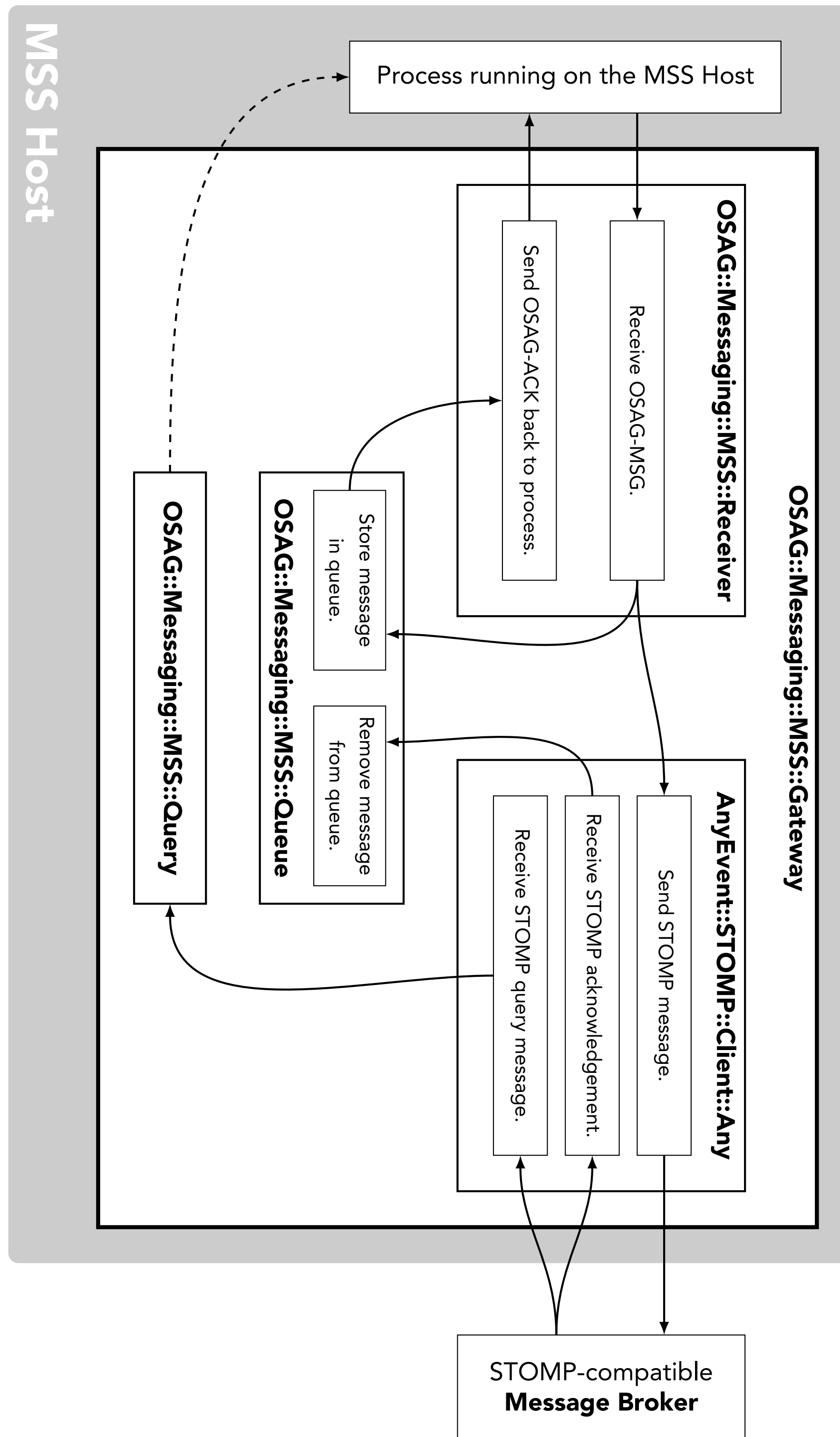

Figure 4.7: The messaging gateway's internals.

includes a method that provides a message selector, which is used when the messaging gateway subscribes to the query topic on the message broker and ensures that only queries with matching scope are consumed.

Finally, the `OSAG::Messaging::MSS::Gateway` module ties all the above modules together and additionally uses the `AnyEvent::STOMP::Client::Any` module for communication and connection handling with the message broker. The `OSAG::Messaging::MSS::Gateway` thus provides for the messaging gateway's functionality. In order to prevent accidental flooding of the message broker, the `OSAG::Messaging::MSS::Gateway` module implements rate limiting using the token bucket algorithm, as described in [46]. The token bucket algorithm essentially limits a rate to a predefined rate r in the long term and nonetheless allows for bursts of predefined size b. The rate limiting algorithm within the messaging gateway is implemented to limite the message rate and not the data rate. Figure 4.7 shows graphically how the aforementioned modules interact in the messaging gateway. Remember that all methods are non-blocking and thus do instantly return.

Sending a Message to the Messaging Gateway To ease the development of applications that use the messaging gateway to transmit their messages to the central infrastructure, we implemented the `OSAG::Messaging::MSS::Message` module, which provides an abstraction of the `OSAG-MSG` frame format that we specified in Section 4.3.2. Assuming that the messaging gateway listens for incoming frames on TCP port `12345`, an extremely simple, but intrinsically useless application could be implemented as follows:

A Simple Message Producer

```
1  #!/usr/bin/perl -wT
2
3  use IO::Socket::INET;
4  use OSAG::Messaging::MSS::Message;
5
6  my $socket = new IO::Socket::INET (
7    PeerHost => 'localhost',
8    PeerPort => '12345',
9    Proto => 'tcp',
10 );
11
12 my $msg = new OSAG::Messaging::MSS::Message(
13   '/queue/example',
14   'text/plain',
15   'queue',
16   {'foo' => 'bar'},
17   'lorem ipsum'
18 );
19
20 print $socket $msg;
21
22 $socket->close;
```

4.3.3.3 The Perl Module for Message Consumers

To facilitate the development of message consumers for a certain message queue, we implemented the Perl module `OSAG::Messaging::Consumer` based on the design proposed in Section 4.2.3. This module abstracts the messaging architecture, in particular it hides the details of the `AnyEvent::STOMP::Client::All` module, which is used behind the scenes to retrieve messages from all message brokers. An application may thus simply register a subroutine as a callback for the `on_message` event and then use the `ack` and `nack` methods to acknowledge a received message. The implementation of a rather basic, also intrinsically useless message consumer, which works off the message queue `/queue/example`, looks like this:

A Simple Message Consumer

```
1  #!/usr/bin/perl -wT
2
3  use OSAG::Messaging::Consumer;
4
5  my $consumer = new OSAG::Messaging::Consumer('/queue/example');
6
7  $consumer->on_message(
8    sub {
9      my ($header_hashref, $body, $ack) = @_;
10     $consumer->ack($ack);
11   }
12 );
13
14 $consumer->run;
```

4.3.3.4 The Host Querying Perl Module

We developed the `OSAG::Messaging::Query` module to simplify the implementation of host querying applications that are to be used in the central infrastructure. These applications may register callback methods for the `on_receive_query_result` and `on_receive_query_error` events that are fired by the `Query` module. To handle communicate with all configured message brokers, the module internally uses `AnyEvent::STOMP::Client::All`. For each new query, the module is designed to use a temporary queue as a reply-to destination for MSS hosts. A module for MSS hosts that aids in implementing applications, which compute a query's result, has not yet been developed. A sample application that invokes the query `example` on all available hosts of company `123` is shown below.

A Sample Host Query Script

```
1  #!/usr/bin/perl -wT
2
3  use OSAG::Messaging::Query;
4
5  my $q = new OSAG::Messaging::Query('example', {'company' => 123});
6
7  $q->on_receive_query_result(
8    sub {
9      my ($header, $body) = @_;
10     print "$header->{hostname} has replied.\n";
11   }
12 );
13
14 $q->start;
```

4.3.4 Availability, Reliability and Security Considerations

Many measures to provide availability, reliability and security have already been treated afore. This includes the discussion of failure modes in Section 4.2.5 as well as the implementation of a rate limiting algorithm in the messaging gateway to prevent flooding of the message broker. Furthermore the execution of query applications on MSS hosts is secured in that queries are terminated by the messaging gateway after a prespecified timeout and since all query applications, that is, to be specific, the mapping of the query's name to the actual query application, have to be preregistered on MSS hosts.

A requirement we have neglected so far concerns the security of the connection between MSS hosts and the central infrastructure, or, more precisely, between the messaging gateway and the message broker. Fulfilling this requirement turns out to be quite straightforward, by relying on SSL/TLS connections. Apache Apollo message brokers do, naturally, implement support for this type of connections and further provide support for certificate-based client authentication. All MSS hosts are already equipped with a private key and a corresponding certificate, that has been signed by the root certificate of Open Systems AG, and they additionally do also know the certificate of the authority that issues certificates for hosts of the central infrastructure of Open Systems AG. We thus just have to equip message brokers with a private key and a corresponding certificate that is signed by the aforementioned authority and configure all components to use SSL/TLS with the forecited certificates.

Another challenge is to monitor the message brokers, while not relying on a monitoring architecture, that in turn relies on the messaging architecture the message brokers are a component of. A potential approach could use the RESTful API of the Apache Apollo message brokers in order to centrally monitor their state completely independent of the messaging architecture they provide for, and thus independent of themselves.

4.3.5 Towards Deployment and Migration

Once the entire messaging architecture is fully developed, reviewed and tested by security engineers of Open Systems AG, it will be deployed in the production environment to eventually replace the GUMA. Especially the migration process of the monitoring part of the GUMA to the new messaging architecture poses additional challenges.

Deployment of the new messaging architecture requires that a special module for configuration building, i.e. a module that is able to generate a configuration file for every MSS host, is implemented and furthermore that hosts in the central infrastructure are setup to provide a platform for message brokers and message consumers. Furthermore rule changes on firewalls may be necessary to allow communication between the messaging gateways on MSS hosts and the message brokers. As soon as the messaging architecture is set up in central infrastructure, the messaging gateway may be rolled out to MSS hosts. This rollout may occur as granular as required, that is the messaging gateway may, for example, first be deployed just for a selected company. An advantage of the new architecture is that it is designed such that it may be operated alongside the existing GUMA without interfering with it.

To enable the migration of monitoring applications from the GUMA to the new messaging architecture, there is still some amount of development required, despite the modules we have provided. Once a given application is extended to support the new messaging architecture, it can just be rolled out as granular as desired, which is also a consequence of both architectures being able to run side by side. To first gain experience with the new messaging architecture in the productive environment and to also fine-tune its configuration, less critical applications, as for example `keystats`, are to be ported first to the new messaging architecture.

5

The Next Generation Messaging Architecture Prototype Evaluated

The prototype of the newly designed messaging architecture is put to the test in this chapter. For this purpose, we first introduce the setup of a test environment in Section 5.1 and then evaluate the new messaging architecture in different scenarios in Section 5.2. A brief summary of the evaluation is given in Section 5.3.

The evaluation is carried out in the Virtual Testing Environment (VTE) of Open Systems AG, a facility that is used to test the Open Systems Unix (OSIX) LX operating system, as well as various products that are developed before they make it into production. As the new messaging architecture is still a prototype that has not been reviewed and thoroughly tested, we chose to not yet deploy it in the production environment. Such an evaluation would also be very time-consuming and personnel-intensive as only certified security engineers are granted access to the production environment. Furthermore a key advantage of the new messaging architecture compared to the GUMA lies in its very design, which should have become apparent in the foregoing Chapters 3 and 4.

5.1 Setting up a Test Environment

In the VTE, we set up the two virtual machines `apollo-dev-1` and `apollo-dev-2`. On each of these machines, we installed and configured the message broker Apache Apollo version 1.6. To deploy the messaging gateway on a subset of the MSS hosts that are running in the VTE, we first extended the configuration management architecture to automatically generate and rollout a configuration file for the messaging gateway. We then created a package that includes the messaging gateway application itself and corresponding init scripts for starting and stopping the gateway. In addition to that, we also added a simple heartbeat application, which periodically sends messages through the messaging gateway. After having built the package, we rolled it out on 38 selected MSS hosts in the VTE, which make up, together with the two message brokers `apollo-dev-1` and `apollo-dev-2`, our environment for evaluation.

5.2 Evaluation of the Prototype

To test different components and mechanisms of the new architecture, we designed different scenarios. In particular, we evaluate how much delay is added by the entire architecture and test both the load balancer and rate limiter of the messaging gateway. Moreover, we investigate the behaviour of the architecture in different failure modes, i.e., in the event of message broker failure and during network outage between the message brokers and all MSS hosts.

5.2.1 Testing the Rate Limiter

Method

To test the rate limiter in the messaging gateway, we send messages to the gateway at predefined input rates and measure the output rate. To do so, we implemented an application, which sends messages to the messaging gateway with the rates and intervals given in Table 5.1. All messages are empty apart from a header, which includes the time the message was sent.

Interval [s]	Message Rate [msg/s]
$[0, 3]$	2
$[3, 5]$	10
$[5, 8]$	5
$[8, 15]$	2

Table 5.1: Message rates for testing the rate limiter.

The messaging gateway then forwards the messages to a dedicated queue on the message broker. A message consumer application concurrently retrieves messages from that queue and immediately sets a reception timestamp for each message. Based on these timestamps, we compute the input and output message rate using the reciprocal of the time difference between subsequent messages $i-1$ and i, that is,

$$\begin{aligned}\text{rate}_{\text{input}} &= \left(\text{timestamp}_{\text{send},i} - \text{timestamp}_{\text{send},i-1}\right)^{-1} \\ \text{rate}_{\text{output}} &= \left(\text{timestamp}_{\text{receive},i} - \text{timestamp}_{\text{receive},i-1}\right)^{-1}\end{aligned}$$

The messaging gateway's rate limiter is configured with rate $r = 5$ msg/s and burst size $b = 10$ msg. In order to visualize the effect of having no rate limiter, we disabled it and run the evaluation once again.

Results

The result is shown in Figure 5.1. The left column of the figure displays the results with the rate limiter being enabled, while the the right column those without rate limiter. For reasons of completeness, the figure includes the delay of messages over time, that is the difference between $\text{timestamp}_{\text{receive}}$ and $\text{timestamp}_{\text{send}}$.

Discussion

The results do exactly represent what we expected. Without the rate limiter enabled, the input and the output message rate are absolutely congruent and the message delay is essentially zero. With the rate limiter enabled, we can clearly observe how it allows for a burst and then limits the rate to the specified maximum, that is, 5 msg/s. After having let the burst of 10 messages through, we observe the rate limiter coming into action and consequently, the message delay starts to increase, as messages are held back. As soon as the input message rate drops to the specified maximum rate, the messaging gateway can keep up with sending messages again and thus the message delay becomes constant. We further note that the output message rate continues to be at the maximum rate, even after the input message rate has again decreased to a level well below the maximum rate. This effect shows the messaging gateway being finally able to work the remaining messages off its the message queue.

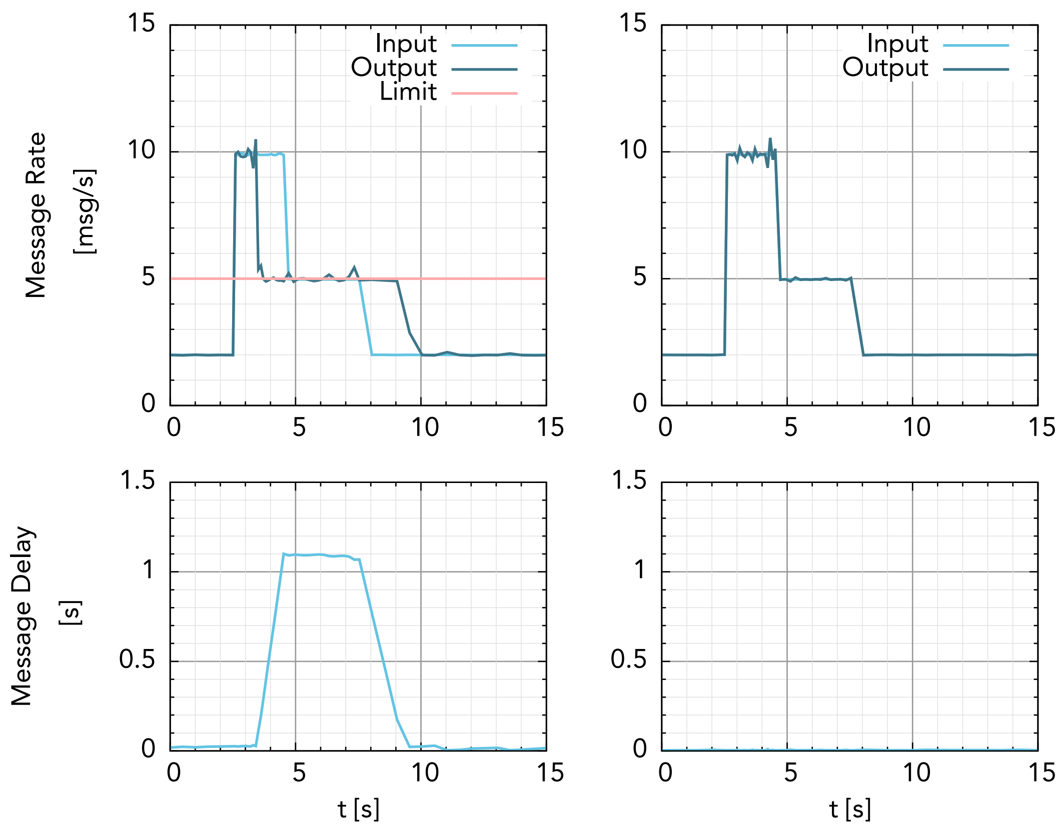

Figure 5.1: Measuring the input and output rate of the messaging gateway with the rate limiter enabled (left column) and disabled (right column).

5.2.2 Determining the Architecture's Delay

In the following scenario, we evaluate the delay that is introduced by the entire messaging architecture. That is, to be precise, the time it takes for a message from the application on the MSS host to the message consumer at the central infrastructure. In this scenario we assume the delay of the network to be virtually zero and design the setup accordingly. The resulting delay hence shows the actual overhead which is solely caused by the messaging architecture itself and not by the underlying network.

Method

A key challenge for delay measurements is time synchronization between different components, additionally to our assumption of a zero-delay network connection. We thus run all applications on a single host, i.e. on one of the two message brokers, to be able to use the same clock for time measurements and to only have to rely on the loopback network interface, which causes a delay of a few tens of microseconds.

To run the evaluation on `apollo-dev-1`, we first deployed the messaging gateway package on it and started the aforementioned heartbeat application, which sends a timestamped message to the messaging gateway every second. Additionally, we run a message consumer application on `apollo-dev-1`, which retrieves the messages sent by the heartbeat application from the respective queue on the message broker and timestamps the message again upon retrieval. The messaging delay of a specific message is then given by the difference of its two timestamps. We have run the measurements until we collected roughly 12 000 messages. Note that the message broker on `apollo-dev-1` has not been under any additional load while we conducted the evaluation.

Results

The outcome of this evaluation is depicted in Figure 5.2, which contains two plots. The top plot shows the distribution of the message delay for a bin width of ten microseconds. The bottom plot represents the corresponding cumulative distribution of the message delay. Furthermore Table 5.2 shows several percentiles, which have been rounded up to the next microsecond.

Discussion

The results shown in Figure 5.2 and Table 5.2 are quite impressive. The lowest measured message delay was as low as 1.809 milliseconds and the highest measurement was a delay of 112.970 milliseconds. The results show that three-quarter of all messages are received within 3.1 milliseconds and that 99 percent of the messages arrive no later than after 5.8 milliseconds, when assuming no delay is being introduced by the underlying network.

The delay introduced by the messaging architecture is thus at least two orders of magnitude smaller than the delay that is usually caused by the network, which typically is approximately only a few hundred milliseconds for wired links, but may be up to several seconds in the case of satellite links. We can, hence, conclude that the delay introduced by the new messaging architecture is largely insignificant to the overall message delay, given essentially no load on the message broker. How the message delay depends on the load of the message broker remains to be shown, preferably in a more realistic environment.

5.2.3 Evaluating Load Balancing

We continue the evaluation of the new messaging architecture with load balancing measurements. Recall that load balancing mechanism of the GUMA was found to be suboptimal.

Percentile	Message Delay [ms]
25	2.446
50	2.775
75	3.093
90	3.404
95	3.656
99	5.785

Table 5.2: Percentiles of the measured message delay.

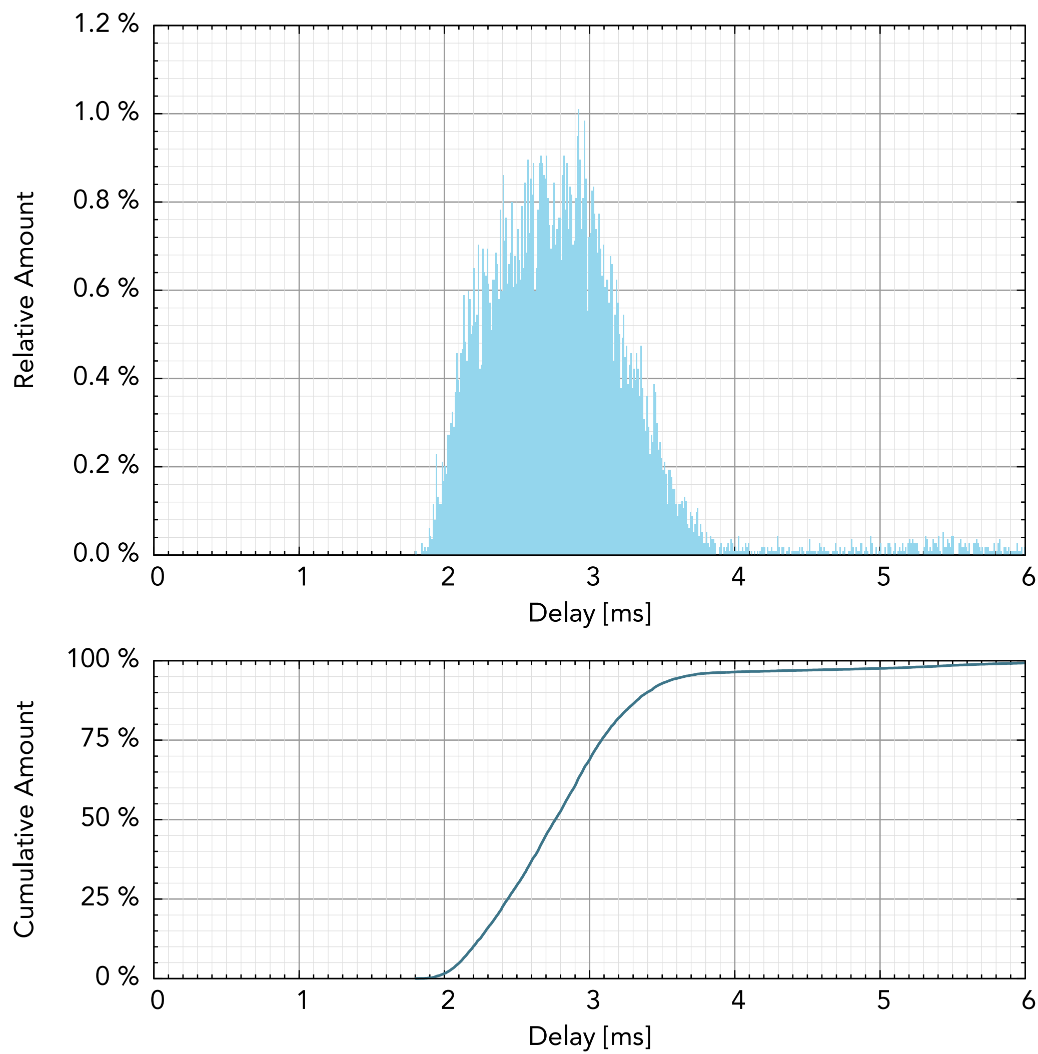

Figure 5.2: Message delay evaluation results.

Method

In order to determine how the load is balanced between the two message brokers `apollo-dev-1` and `apollo-dev-2`, we need to determine how many messaging gateways have established a connection to either broker. That is we compute the load based on the number of gateways that connected to a message broker and thus assume that all MSS hosts essentially show similar message sending patterns. To find out the number of messaging gateways that have connected to a message broker, we make use of the RESTful management API that Apache Apollo provides. We thus query both message brokers, i.e., `apollo-dev-1` and `apollo-dev-2`, every five seconds and extract the respective number of messaging gateways that have chosen to connect to that message broker. We have run these load balancing measurements only for 15 minutes, since the messaging gateways have been configured to reconnect periodically with a period randomly chosen between one and five minutes, which is rather frequent compared to production environments.

Results

Figure 5.3 shows the results of the load balancing measurements. The figure consists of three separate plots. The topmost plot depicts the load of the two message brokers relative to each other over time, that is, irrespective of the number of connected messaging gateways. The availability of each message broker over time is shown in the middle plot and will be of more importance in the two subsequent evaluations. The plot in the bottom shows the number of connected clients, i.e. the number connected messaging gateways, over time plotted for each message broker separately.

Discussion

Looking at the results shown in Figure 5.3, we notice that load balancing is clearly improved, compared to the results shown in Figures 3.3 and 3.4. We observe that the load balancing is no longer constant as it was the case in the GUMA, but that the balance changes randomly around the equilibrium of 50%, which is due to the randomized selection of any of the two message brokers, as implemented in `AnyEvent::STOMP::Client::Any`, and also due to the reconnect mechanism, which is integrated in the messaging gateway and causes it to periodically re-establish the connection to either message broker.

In the second half of the measurements, we notice that the difference in load is rather big. Even though extreme imbalance is unlikely, it is still possible. For example, it is theoretically possible that all messaging gateways choose to connect to the same message broker. The probability P that k out of n messaging gateways choose to connect to either message broker is the number of possibilities to choose k out of n messaging gateways times the probability that these k connect to either message broker times the probability that the remaining $n-k$ messaging gateways connect to the other message broker. This results in the formula:

$$P = \binom{n}{k} \cdot \left(\frac{1}{2}\right)^{k} \cdot \left(1 - \frac{1}{2}\right)^{n-k} = \binom{n}{k} \cdot 2^{-n}.$$

Due to the symmetry of the binomial coefficient, the above formula is also symmetric around the equilibrium, that is, around $\frac{n}{2}$. We also note that with an increasing amount of messaging gateways connecting to the message brokers, i.e. with increasing n, the probability of extreme imbalance is further reduced. Consequently the load will be more equally balanced in the production environment, where more than 2600 MSS hosts are operated, compared to the evaluation shown in Figure 5.3, where we just used 38 MSS hosts.

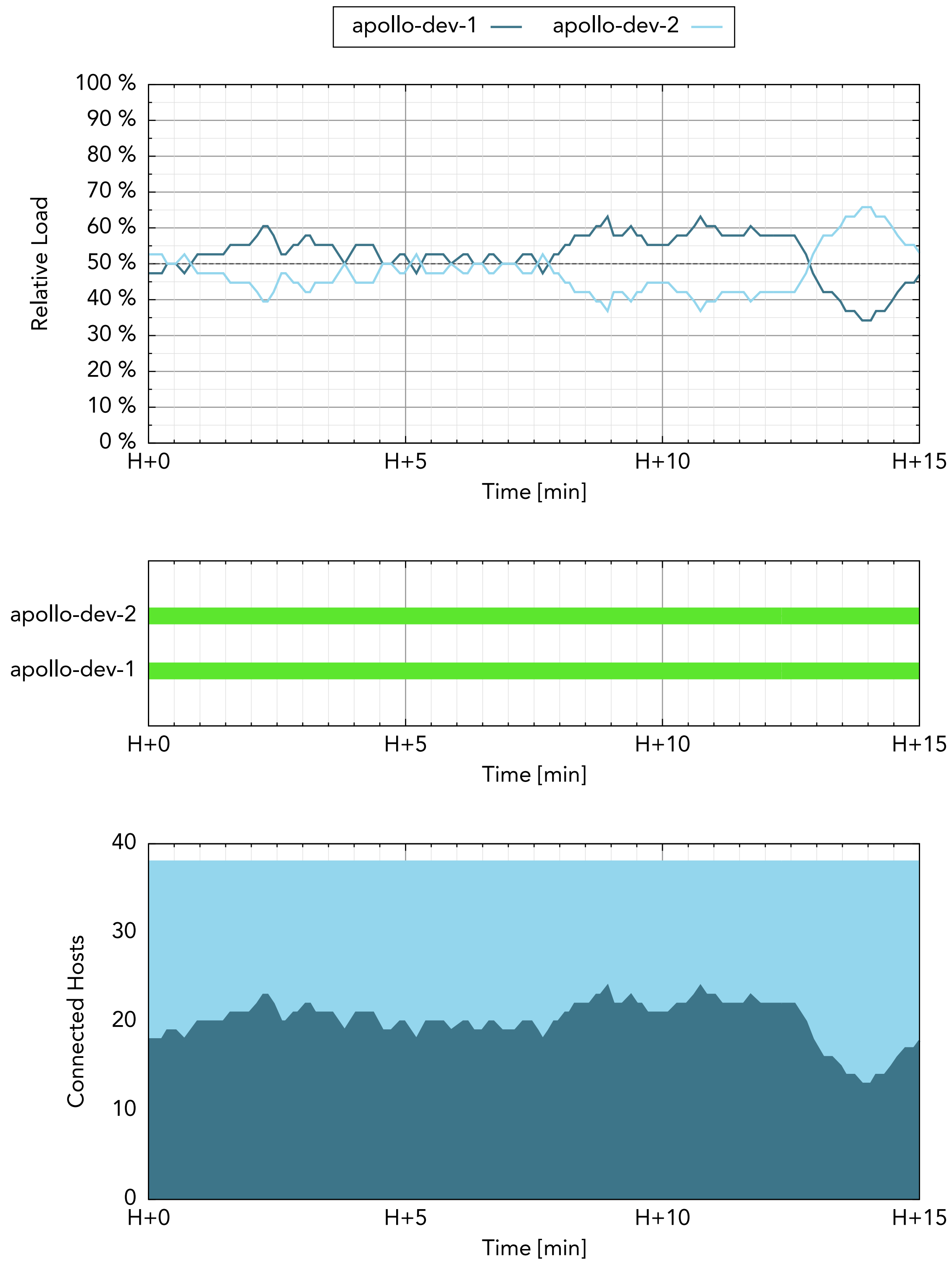

Figure 5.3: Results of the load balancing test.

5.2.4 Failure of a Message Broker

The messaging architecture's reaction to the event of temporary failure of a message broker is evaluated in the following scenario.

Method

We used the same setup as in the foregoing evaluation of the load balancing mechanism. That is, we again collected measurements from the REST API of the two message brokers. Approximately five minutes into the experiment, we simulated the failure of the message broker on `apollo-dev-1` by stopping the corresponding process. We restarted the message broker process after 40 seconds of simulated failure. After a quarter of an hour, we finally stopped collecting measurements, as a state, which is similar to the one before the message broker failure, has been reached again.

Results

The results are shown in Figure 5.4, where we used the exact same type of plots as described in the aforegoing evaluation of the load balancer. Note, however, that the plot in the middle now shows the simulated failure of `apollo-dev-1`. The messaging gateways are configured to re-establish their connection to the message broker periodically, choosing a period from an interval between one and five minutes at random.

Discussion

The bottommost plot in Figure 5.4 shows that the messaging architecture as a whole was always available for the messaging gateways. This can be deduced from the constant number of connected hosts. When the message broker on `apollo-dev-1` fails, we observe that all messaging gateways, which have lost their connection, immediately jump to the remaining message broker. It is to be determined whether this immediate switch to the remaining message broker may cause it overload and become unavailable as well. If that would be the case, we could add a randomized delay before messaging gateways re-establish their connection to the message broker.

Once the message broker on `apollo-dev-1` is available again, messaging gateways start to connect to it, which is due to the randomized connection re-establishing mechanism. Approximately five minutes after the failed message broker has become available again, we notice that the load is balanced evenly. We note that this delay equals the maximum of the connection re-establishment interval. Recall that the timer for connection re-establishment is reset every time a connection to a message broker has been established. Each messaging gateway thus re-establishes its connection at least once within that five minute interval and since both message brokers are available for the entire interval the load is being evenly balanced towards the end of the interval, no different from the load balancing situation in the precedent evaluation.

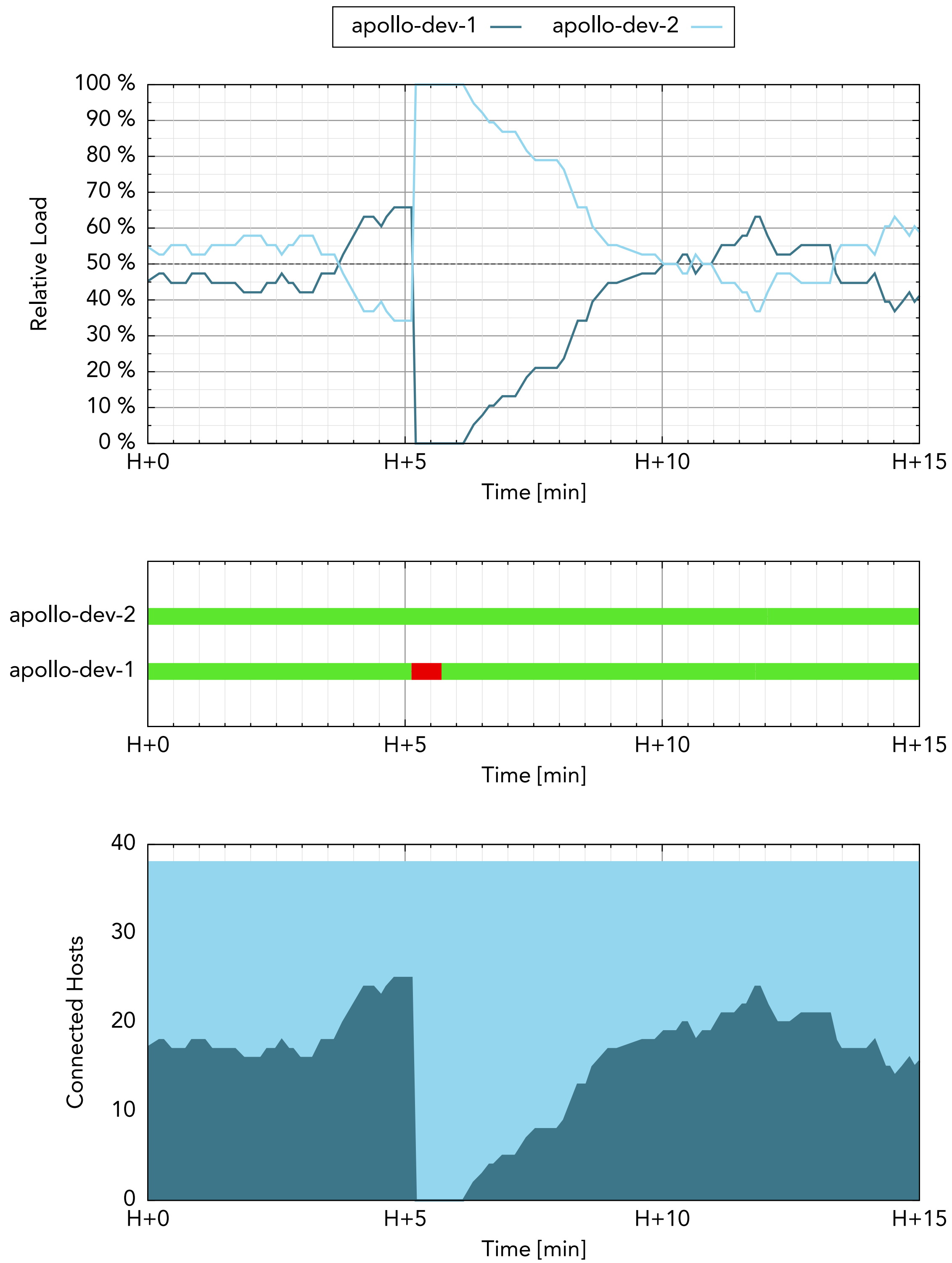

Figure 5.4: Testing the messaging architecture in the event of broker failure.

5.2.5 Effect of a Network Outage

The last scenario evaluates the new messaging architecture during a simulated outage of the network between all MSS hosts and the entire central infrastructure.

Method

Just as in the two foregoing evaluations, we use the REST API of the two Apache Apollo message brokers again in order to obtain measurements. To simulate such a network outage, we decided not to actually disable the corresponding network links, but to shut the message broker processes on both `apollo-dev-1` and `apollo-dev-2` down. This has the same effect on the messaging gateway, as well as our measurement application since both message brokers are just observed to not being reachable, irrespective of the actual cause. After collecting measurements for four minutes, we simulated a network outage of 135 seconds and then made the network available again, i.e., turned both message brokers back on. In total we performed measurements for a quarter of an hour.

Results

Figure 5.5 presents the results of this evaluation. The three plots shown in the figure are of the same types as those in the two preceding figures and will thus not be explained any further.

Discussion

Also in this scenario, we observe that the new messaging architecture shows a behaviour as we expected. Immediately upon the simulated failure of the network connections, we notice that no more clients are connected to the messaging infrastructure, which has become completely unavailable. When loosing its connection to the message broker, the messaging gateway tries to reconnect. If it, however, fails to re-establish the connection it performs a randomized exponential backoff and then tries to reconnect again.

As soon as the network connection is back up again, messaging gateways start reconnecting to the message brokers, which can be observed at 6 minutes and 30 seconds into the evaluation, as shown in Figure 5.5. Furthermore, we see that irrespective of how many messaging gateways are connected, the load is evenly balanced between the two message brokers for the entire duration of the evaluation. Roughly 150 seconds after the network outage, we find that all messaging gateways have re-established their connection to either message broker. This delay is approximately equal to the maximal backoff time of 128 seconds, which has been configured in the messaging gateway.

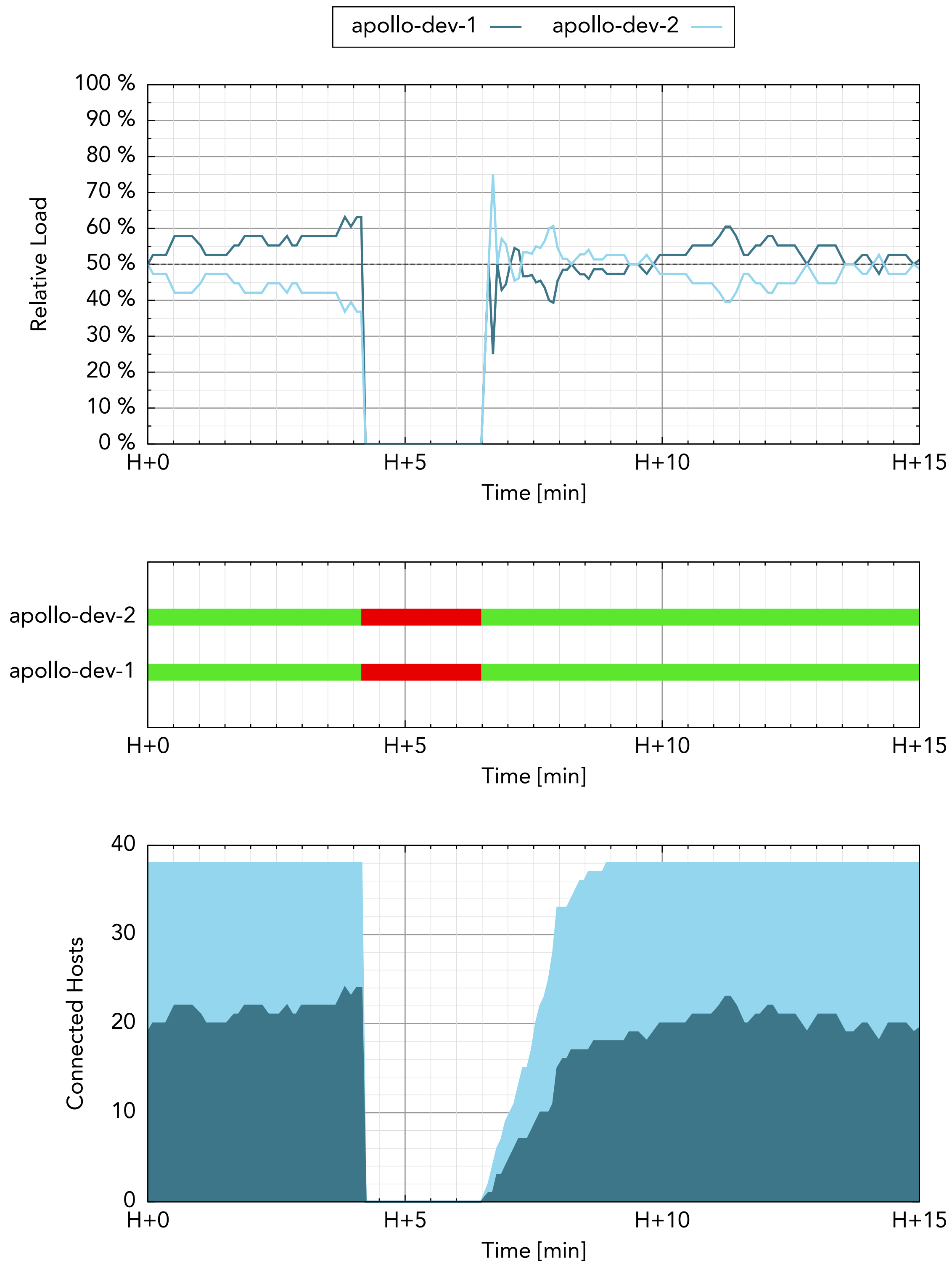

Figure 5.5: Testing the architecture during a network outage.

5.3 Evaluation Summary

In this chapter's evaluation we were able to demonstrate the solid performance of the prototype of the new messaging architecture within the Virtual Testing Environment (VTE) using two message brokers and 38 MSS hosts. Despite this good performance of the prototype, it is very important to note that the results of the evaluation need not necessarily hold in a production environment with more than 2600 MSS hosts.

Messaging gateways are effectively prevented from accidentally flooding the message brokers at the central infrastructure, by means of a rate limiter, which is based on the Token Bucket algorithm. The load balancing mechanism implemented in the messaging gateway is working, that is, it distributes the load across the available message brokers in a fair manner. Furthermore, we have shown that the delay introduced by the new messaging architecture is insignificant with respect to the delay introduced by underlying network connections.

In addition to that, we illustrated how the messaging architecture behaves in two failure modes. Even in the event of failure of up to $n-1$ of its n message brokers, the messaging architecture is able to continue providing for its messaging service. Once a failed message broker recovered, its load is continuously increased until the load is again balanced amongst the available message brokers. We have furthermore shown, that after a complete outage of the network connections between all message brokers at the central infrastructure and all MSS hosts, the messaging architecture is able to completely recover within a few minutes.

6

Architecture Comparison, Future Work and Conclusion

In this concluding chapter, we first provide in Section 6.1 a comparison between the Grand Unified Monitoring Architecture (GUMA) and the new messaging architecture, which we have developed and prototyped in the course of this thesis. Subsequently, Section 6.2 highlights opportunities for future work for both, scientific research, as well as ideas for further development of the new messaging architecture at Open Systems AG. Finally, we conclude this thesis in Section 6.3 by summarizing our most important contributions and providing a final outlook.

6.1 Architecture Comparison

The keen reader has already noticed, that the differences between Open Systems AG's current architecture, that is, the GUMA, and the new messaging architecture, which we designed and prototyped in this thesis, could hardly be bigger.

In the preceding evaluation in Chapter 5, we highlighted the solid performance of a simple load balancing algorithm that has been integrated in the messaging gateway. Contrary to that, the `gumaclients` were shown to be sticky, resulting in a continuous load imbalance for the `gumaserver` instances. We have further shown that the messaging delay introduced by the new messaging architecture lies below six milliseconds for 99% of all messages. Despite the evaluation of the message delay for the GUMA being inconclusive, the message delay caused by the new messaging architecture is at least as good that of the GUMA, if not better. Additionally, we illustrated that the new messaging architecture behaves predictably in different failure modes and is able to recover quickly from failures.

While the evaluation has revealed positive results for the new messaging architecture, the actual key advantage of the new messaging architecture over the GUMA lies in its improved design. Contrary to the GUMA, the new messaging architecture is designed to be highly modular and thus becomes very scalable, as its individual components may be scaled independently. The modularity of the architecture furthermore facilitates much better maintainability, since many small components are, obviously, easier to develop and test compared to one big application. In addition to that, the new messaging architecture is not only free of proprietary software, but also abstracts the messaging architecture towards applications, whereas the GUMA relies on the proprietary ICE and is tightly coupled with the applications that rely on it. MSS host querying finally introduced an entirely novel and powerful feature to the new messaging architecture. Arbitrary many MSS host can now be efficiently queried to obtain desired information. With the GUMA one would not have gotten around the overhead of establishing an SSH session with each MSS host. Overall, the new messaging architecture provides an ubiquitous messaging service for applications and allows for more sophisticated operation.

6.2 Opportunities for Future Work

There are two basic areas for performing future work. These are, in particular, rather theoretic contributions to scientific research and also more practical further developments of the messaging architecture at Open Systems AG.

6.2.1 Scientific Research

We essentially see three opportunities for future scientific research. Based on the prototype that has been implemented with the Apache Apollo message broker and the STOMP messaging protocol, the scalability of this prototype in a globally distributed environment, as provided with the MSS hosts of Open Systems AG, could be analyzed and eventually also be contrasted to the scalability of similar architectures. Additionally, one could furthermore compare the performance of message brokers other than Apache Apollo within the prototype of the messaging architecture, which has been implemented in this thesis.

In the future, we expect more messaging protocols and message broker implementations to appear, since message-oriented middleware will grow more and more popular. There will thus be an increased demand in instruments for comparing messaging protocols and message brokers. Consequently, future research may include the design and development of messaging protocol-specific benchmarks, similar to SPECjms provided in [35], in order to allow for comparison of different message broker implementations. In particular, a benchmark based on the STOMP could be designed.

A third opportunity for future work includes research on possible concepts of a reliable, scalable and, most importantly, delay tolerant request-reply architecture, which is based on message-oriented middleware. In the case of Open Systems AG, such an architecture would facilitate reliable host querying, which is a functionality that the architecture developed in this thesis cannot provide for. Such a functionality could be used, for example, to reliably push updated configuration files to MSS hosts, even if they are not permanently connected to the message brokers at the central infrastructure.

6.2.2 At Open Systems AG

Due to development of the new messaging architecture and its prototype, there are manifold opportunities for future work at Open Systems AG. First and foremost, however, the prototype needs to be thoroughly reviewed and tested, as well as extended in order to support different management functionalities, which have not yet been implemented. Finally, the Apache Apollo message brokers, which reside at the central infrastructure, need to be configured and set up. Not until then, the messaging gateways may also be deployed onto MSS hosts in the production network.

As soon as the messaging architecture is stable enough and deployed on a subset of the MSS hosts, at least in the testing environment, existing applications, which thus far relied on the GUMA, may be ported to the new messaging architecture. That is, each application has not only to be converted to act as a message producer, which sends its messages to the messaging gateway on MSS hosts, but applications are additionally required to provide for a corresponding message consumer component. Once first applications have been ported to the new messaging architecture, the performance of the entire messaging architecture and especially of the Apache Apollo message brokers can be analyzed.

Future work may also include the extension of the messaging architecture to provide another type of querying, that is, MSS hosts querying the central infrastructure. These types of queries would enable MSS hosts to, for example, pull certain configuration files from the cen-

tral infrastructure, contrary to today's approach, where the entire configuration is pushed from the central infrastructure to the MSS hosts. The new messaging architecture generally provides a ubiquitous messaging service, which not only enables, but also simplifies the development of a multitude of applications that we actually have not yet thought of.

6.3 Conclusion

In this master thesis on Messaging Challenges in a Globally Distributed Network, we first evaluated suitable middleware, in particular message-oriented middleware, and analyzed the current messaging architecture named the Grand Unified Monitoring Architecture (GUMA) at Open Systems AG. With the concepts of message-oriented middleware, as well as the results of the analysis of the GUMA, we stated requirements and then designed an entirely new messaging architecture, which stands out due to its high modularity, scalability, ease of maintainability and future-proofness. Moreover, we implemented a prototype of the new messaging architecture based on the Simple Text Oriented Messaging Protocol (STOMP) and Apache Apollo message brokers, and evaluated its performance. With the design of the new messaging architecture and the implementation of its prototype, we have been enabled to have considerable impact, since the new messaging architecture will, once reviewed and thoroughly tested, actually be deployed in the production environment of Open Systems AG and replace the current architecture, the GUMA.

With the new messaging architecture, we have laid the cornerstone for a new era of messaging at Open Systems AG and thus provide the security engineers of Open Systems AG with a future-proof messaging service, based upon which they may easily build their monitoring applications to further improve Mission Control™ Security Services[1].

[1] A catchy name for the new messaging architecture, however, remains to be found.

Appendix

List of Figures

2.1 A simple remote procedure call. 8
2.2 Generic object-oriented middleware architecture. 9
2.3 A logical message bus as provided by message-oriented middleware. 10
2.4 The three components typical message-oriented middleware consists of. 10
2.5 Message queue. 11
2.6 The point-to-point messaging model. 12
2.7 The publish/subscribe messaging model. 12

3.1 Distributed Management Overview 20
3.2 The Grand Unified Monitoring Architecture (GUMA) in detail. 22
3.3 Load balancing measurements based on the number of processed messages. . 25
3.4 Load balancing measurements based on the size of the processed messages. . 25
3.5 Message rate over time. 27
3.6 Data rate over time. 27
3.7 Message Rate Distribution 27
3.8 Data Rate Distribution 27
3.9 Frequency of Message Types 29
3.10 GUMA signature frequency 29
3.11 Distribution of the heartbeat delay. 30

4.1 A first draft of the new messaging architecture. 35
4.2 The messaging architecture with the newly added messaging gateway. 36
4.3 The messaging architecture with improved message consumers. 37
4.4 The messaging architecture with host querying integrated. 38
4.5 Redundancy and load balancing added to the new messaging architecture. . . . 41
4.6 AMQP, MQTT and STOMP 42
4.7 The messaging gateway's internals. 48

5.1 Measuring the input and output rate of the messaging gateway with and without rate limiter. 55
5.2 Message delay evaluation results. 57
5.3 Results of the load balancing test. 59
5.4 Testing the messaging architecture in the event of broker failure. 61
5.5 Testing the architecture during a network outage. 63

List of Tables

1.1 The assignment in terms of four tasks. 3

2.1 Characteristics of the chosen message brokers implementations. 15
2.2 Messaging protocol support of the chosen message brokers. 16

3.1 Characteristics of the dataset collected for evaluation of the GUMA. 25
3.2 Average message and data rate. 26
3.3 Estimated loss of heartbeats. 31

4.1 STOMP headers for messages sent to the messaging gateway. 45
4.2 STOMP headers for query messages. 45
4.3 Scopes used for configuration management. 46

5.1 Message rates for testing the rate limiter. 54
5.2 Percentiles of the measured message delay. 57

List of Acronyms

ØMQ ZeroMQ.
AMQP Advanced Message Queueing Protocol.
API Application Programming Interface.
BNF Backus-Naur Form.
CORBA Common Object Request Broker Architecture.
CPAN Comprehensive Perl Archive Network.
EAI Enterprise Application Integration.
FOLDOC Free On-Line Dictionary Of Computing.
GUMA Grand Unified Monitoring Architecture.
ICE Internet Communications Engine.
ICMP Internet Control Message Protocol.
IDL Interface Definition Language.
ISP Internet Service Provider.
JMS Java Message Service.
LHC Large Hadron Collider.
MQTT Message Queueing Telemetry Transport.
MSS Managed Security Service.
NGO Non-governmental Organization.
NIDS Network Intrusion Detection System.
NTP Network Time Protocol.
OMG Object Management Group.
ORB Object Request Broker.
OSIX Open Systems Unix.
SOC Security Operation Center.
SSH Secure Shell.
STOMP Simple Text Oriented Messaging Protocol.
VPN Virtual Private Network.
VTE Virtual Testing Environment.
WLCG Worldwide LHC Computing Grid.
WSN Wireless Sensor Networks.
XML Extensible Markup Language.

References

[1] Apache Apollo. Apollo 1.6 STOMP Protocol Manual. URL http://activemq.apache.org/apollo/documentation/stomp-manual.html.

[2] Apache Apollo. Apollo 1.6 User Manual. URL http://activemq.apache.org/apollo/documentation/user-manual.html.

[3] Appel, Stefan; Sachs, Kai and Buchmann, Alejandro. Towards benchmarking of AMQP. In Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, pages 99--100. ACM, 2010. URL http://www.dvs.tu-darmstadt.de/publications/pdf/AMQPDemo.pdf.

[4] Banavar, Guruduth; Chandra, Tushar; Strom, Robert and Sturman, Daniel. A case for message oriented middleware. In Distributed Computing, pages 1--17. Springer, 1999. URL http://cin.ufpe.br/~redis/intranet/bibliography/middleware/banavar-mom99.pdf.

[5] Birrell, Andrew D. and Nelson, Bruce Jay. Implementing remote procedure calls. ACM Transactions on Computer Systems (TOCS), volume 2(1), 1984. URL http://www.cs.princeton.edu/courses/archive/fall03/cs518/papers/rpc.pdf.

[6] Casper, Carsten. MarketScope for Managed Security Services in Europe, October 2012. URL http://www.gartner.com/technology/reprints.do?id=1-1CMHKJH&ct=121026.

[7] Chirino, Hiram. STOMP Benchmark Results. URL http://hiramchirino.com/stomp-benchmark.

[8] Cons, Lionel and Paladin, Massimo. The WLCG Messaging Service and its Future. In Journal of Physics: Conference Series, volume 396. 2012. URL http://mig.web.cern.ch/mig/pub/CHEP%202012%20-%20The%20WLCG%20Messaging%20Service%20and%20its%20Future%20(paper).pdf.

[9] Curry, Edward. Message-oriented middleware. Middleware for Communications, pages 1--28, 2004.

[10] Dworak, Andrzej; Charrue, Pierre; Ehm, Felix; Sliwinski, Wojciech and Sobczak, Maciej. Middleware trends and market leaders 2011. In Robichon [32], pages 1334--1337. URL http://accelconf.web.cern.ch/accelconf/icalepcs2011/papers/frbhmult05.pdf.

[11] Dworak, Andrzej; Ehm, Felix; Charrue, Pierre and Sliwinski, Wojciech. The new CERN controls middleware. Journal of Physics: Conference Series, volume 396:12--17, 2012.

[12] Ehm, Felix. Running a reliable messaging infrastructure for CERN's control system. In Robichon [32], pages 724--727. URL http://accelconf.web.cern.ch/accelconf/icalepcs2011/papers/wepkn006.pdf.

[13] Ehm, Felix and Dworak, Andrzej. A remote tracing facility for distributed systems. In Robichon [32], pages 650--653. URL http://accelconf.web.cern.ch/Accelconf/icalepcs2011/papers/wemau001.pdf.

[14] Evans, Dr. David. Middleware. http://www.cl.cam.ac.uk/teaching/1011/CDSysII/12-middleware.pdf, 2010.

[15] Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P. and Berners-Lee, T. Hypertext Transfer Protocol -- HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL http://tools.ietf.org/html/rfc2616. Updated by RFCs 2817, 5785, 6266, 6585.

[16] Freed, N. and Borenstein, N. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. URL http://tools.ietf.org/html/rfc2045. Updated by RFCs 2184, 2231, 5335, 6532.

[17] Henning, Michi. A New Approach to Object-Oriented Middleware, 2004. URL http://di.ufpe.br/~redis/intranet/bibliography/middleware/henning-methodology04.pdf.

[18] Henning, Michi. The rise and fall of CORBA. Queue, volume 4(5):28--34, 2006. URL http://cacm.acm.org/magazines/2008/8/5336-the-rise-and-fall-of-corba/pdf.

[19] Hohpe, Gregor and Woolf, Bobby. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman, Amsterdam, 1 Edition, October 2003. ISBN 978-0-321-20068-6.

[20] Hunkeler, Urs; Truong, Hong Linh and Stanford-Clark, Andy. MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks. In Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on, pages 791--798. IEEE, 2008. URL http://www.cin.ufpe.br/~redis/intranet/bibliography/middleware/hunkeler-publish08.pdf.

[21] Inspirel. Types of middleware, August 2010. URL http://www.inspirel.com/articles/Types_Of_Middleware.html.

[22] Jiang, Peng; Bigham, John; Bodanese, Eliane and Claudel, Emmanuel. Publish/subscribe delay-tolerant message-oriented middleware for resilient communication. Communications Magazine, IEEE, volume 49(9):124--130, 2011. URL http://www.cin.ufpe.br/~redis/intranet/bibliography/middleware/jiang-publish-2011.pdf.

[23] Krakowiak, Sacha. Middleware architecture with patterns and frameworks, February 2009. URL http://proton.inrialpes.fr/~krakowia/MW-Book/main.pdf.

[24] Maheshwari, Piyush and Pang, Michael. Benchmarking message-oriented middleware: TIB/RV versus SonicMQ. Concurrency and Computation: Practice and Experience, volume 17(12):1507--1526, 2005. URL http://byildiz.etu.edu.tr/bil519/resources/papers/mom_tibco_sonic.pdf.

[25] Maheshwari, Piyush; Tang, Hua and Liang, Roger. Enhancing web services with message-oriented middleware. In Web Services, 2004. Proceedings. IEEE International Conference on, pages 524--531. IEEE, 2004. URL http://cin.ufpe.br/~redis/intranet/bibliography/middleware/maheshwari-webservice04.pdf.

[26] Marsh, Gregory; Sampat, Ajay P.; Potluri, Sreeram and Panda, Dhabaleswar K. Scaling Advanced Message Queuing Protocol (AMQP) Architecture with Broker Federation and InfiniBand. Technical report, 2010. URL ftp://www.cse.ohio-state.edu/pub/tech-report/2009/TR17.pdf.

[27] Microsystems, Sun. RPC: Remote Procedure Call Protocol specification: Version 2. RFC 1057 (Informational), June 1988. URL http://tools.ietf.org/html/rfc1057.

[28] Nahrwar, Mort. Principles of Object-Oriented Middleware, 2005. URL https://www.os3.nl/_media/2005-2006/dia-files/part7.pdf.

[29] NATO Science and Technology Organization. Information management over disadvantaged grids, December 2007. URL http://www.cso.nato.int/pubs/rdp.asp?RDP=RTO-TR-IST-030.

[30] Oberste-Berghaus, Benjamin. Message Oriented Middleware: Technische Aspekte der Umsetzung von EAI, 2005. URL http://is.uni-muenster.de/pi/lehre/ss05/seminarEAM/MessageOrientedMiddleware.pdf.

[31] Oliveira, José Pedro and Pereira, José. Experience with a middleware infrastructure for service oriented financial applications. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, pages 479--484. ACM, 2013. URL http://gsd.di.uminho.pt/members/jop/pdfs/OP13.pdf.

[32] Robichon, Marie, editor. Proceedings of ICALEPCS2011. October 2011.

[33] Sachs, Kai; Appel, Stefan; Kounev, Samuel and Buchmann, Alejandro. Benchmarking publish/subscribe-based messaging systems. In Database Systems for Advanced Applications, pages 203--214. Springer, 2010. URL http://www.dvs.tu-darmstadt.de/publications/pdf/jms2009PS.pdf.

[34] Sachs, Kai; Kounev, Samuel; Bacon, Jean and Buchmann, Alejandro. Performance evaluation of message-oriented middleware using the specjms2007 benchmark. Performance Evaluation, volume 66(8):410--434, 2009. URL http://descartes.ipd.uka.de/fileadmin/user_upload/sdq/people/samuel-kounev/docs/08-PerfEval-SPECjms2007.pdf.

[35] Sachs, Kai; Kounev, Samuel; Carter, Marc and Buchmann, Alejandro. Designing a workload scenario for benchmarking message-oriented middleware. In SPEC Benchmark Workshop. 2007. URL http://www.dvs.tu-darmstadt.de/publications/pdf/Designing_workload_Scenario_MOM.pdf.

[36] Souto, Eduardo; Guimarães, Germano; Vasconcelos, Glauco; Vieira, Mardoqueu; Rosa, Nelson and Ferraz, Carlos. A message-oriented middleware for sensor networks. In Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, pages 127--134. ACM, 2004. URL http://www.cin.ufpe.br/~redis/intranet/bibliography/middleware/souto-sensor04.pdf.

[37] Souto, Eduardo; Guimarães, Germano; Vasconcelos, Glauco; Vieira, Mardoqueu; Rosa, Nelson; Ferraz, Carlos and Kelner, Judith. Mires: a publish/subscribe middleware for sensor networks. Personal and Ubiquitous Computing, volume 10(1):37--44, 2006. URL http://cin.ufpe.br/~redis/intranet/bibliography/middleware/souto-mires06.pdf.

[38] STOMP Specification. STOMP Protocol Specification, Version 1.2. URL http://stomp.github.io/stomp-specification-1.2.html.

[39] Subramoni, Hari; Marsh, Gregory; Narravula, Sundeep; Lai, Ping and Panda, Dhabaleswar K. Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand. In High Performance Computational Finance, 2008. WHPCF 2008. Workshop on, pages 1--8. IEEE, 2008. URL ftp://cse.osu.edu/pub/tech-report/2008/TR51.pdf.

[40] Tanenbaum, Andrew S. and van Steen, Maarten. Distributed Systems. Principles and Paradigms. Prentice Hall International, 1st International Edition, March 2003. ISBN 978-0-131-21786-7.

[41] Thurlow, R. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 5531 (Draft Standard), May 2009. URL http://tools.ietf.org/html/rfc5531.

[42] Tran, Phong; Greenfield, Paul and Gorton, Ian. Behavior and performance of message-oriented middleware systems. In Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on, pages 645--650. IEEE, 2002. URL http://www.cin.ufpe.br/~redis/intranet/bibliography/middleware/tran-mom02.pdf.

[43] Vinoski, Steve. CORBA: Integrating diverse applications within distributed heterogeneous environments. Communications Magazine, IEEE, volume 35(2):46--55, 1997.

[44] VMware vFabric Team. Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP, February 2013. URL http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html.

[45] White, J.E. High-level framework for network-based resource sharing. RFC 707, December 1975. URL http://tools.ietf.org/html/rfc707.

[46] Wikipedia. Token bucket -- Wikipedia, The Free Encyclopedia, 2013. URL http://en.wikipedia.org/w/index.php?title=Token_bucket&oldid=572345848.