Masaryk University
Faculty of Informatics

New Generation Logging Mechanism

Master's Diploma Thesis

Andrea Vašeková

Brno, 2013

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: doc. RNDr. Tomáš Pitner, Ph.D.

Acknowledgement

I would like to thank my consultant, Daniel Tovarňák, for his time, patience and constant feedback throughout the work on my thesis; my advisor, Tomáš Pitner, for his valuable comments and constructive, yet always kind criticism; and my family, friends and everyone else who has inspired and supported me during my studies.

Abstract

The aim of this thesis is to design and implement a logging mechanism that refrains from using natural language as the main medium for carrying logging information. Existing approaches to logging are discussed and set in the context of monitoring systems, in particular cloud monitoring. A structured logging mechanism is then presented, such that the logging data it produces is suitable for automated processing. Its features and implementation in Java are described in detail, together with instructions on how to use it for application logging. Finally, the results of performance analysis are reviewed. From the experiments it is concluded that the production of logs using the proposed logging mechanism introduces minimum to no overhead compared to unstructured logging, and the processing of structured logs in given format is very efficient as opposed to using custom regular expressions.

Keywords

structured logging, log processing, cloud monitoring, Ngmon

Contents

Chapter 1

Introduction

The gradual shift towards cloud computing experienced in the last years causes developers and researchers alike to question the routine. Many existing tools and techniques can be well adapted to cloud, yet the environment is special enough to benefit more from solutions tailored to its needs.

Monitoring poses a particularly intricate problem due to the very nature of cloud itself. Using virtualization in such a large scale as cloud does, and providing the consumers of cloud services with only a limited insight into the resources are probably the main reasons cloud monitoring has to be treated differently, since common approaches to distributed systems monitoring may not be sufficient. The Ngmon Project introduces a prototype implementation of a monitoring daemon for distributed systems, cloud especially, based on the requirements identified by Tovarňák and Pitner [40] as crucial for a new generation of monitoring information producers. This thesis was developed as a part of design and implementation works contributing to Ngmon.

In general, one of the most natural ways of gaining visibility into the state of monitored resources is by observing the logs. Logs produced by applications or systems represent a valuable source of monitoring information, therefore logging plays an important role in the whole monitoring process. However, log formats vary greatly among the developers. The majority of logs is produced in the form of natural language strings with no predefined structure, which makes them very difficult to process, let alone extract useful information from. This problem could be solved by bringing order into the way logs are produced and promoting structured logging over the traditional approaches.

This thesis elaborates on the current situation in the sphere of application logging and offers a possible solution to the problem of structured logs production. A logging component ready to be integrated in the Ngmon monitoring daemon is presented, such that it is capable of producing logs in a unified and extensible format and thus facilitates efficient subsequent processing of such logs.

The thesis is structured as follows: Chapter 2 discusses fundamental prerequisites for understanding the significance of our work, depicting the current state of cloud monitoring, and existing tools in this field. In Chapter 3, Ngmon, a prototype of an event-based monitoring daemon is described. Chapter 4 moves on from monitoring in general to the domain of log management as a more specific way of collecting and handling monitoring information. Particularly, it deals with the production, analysis and processing of logs. The last part of this chapter emphasizes the need for structured logging to fully exploit the potential of automated processing of logs. Chapter 5 then details the structured logging mechanism that was designed and implemented as the main outcome of this thesis, followed by instructions and recommendations related to its usage and customization (Chapter 6). Chapter 7 is dedicated to performance experiments and their results, comparing the proposed logging mechanism with unstructured logging; and finally, Chapter 8 summarizes the work.

Chapter 2

Monitoring the Cloud

2.1 Cloud Computing

Cloud computing has experienced a major breakthrough in the last decade. Although the main concepts date back as far as 1990s, only recently has it started to receive special attention from both experts and media.

According to the National Institute of Standards and Technology (NIST), "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [27]. Five characteristics are identified as essential for this model, namely: *on-demand self-service*, meaning the service provider does not need to be contacted in person for the consumer to gain access to the computing resources; *broad network access*, i.e. no special requirements are imposed on clients or network; *resource pooling* to support multi-tenant access; *rapid elasticity*, in the sense that the environment accommodates to the current demand and scales rapidly; and finally, cloud systems provide a *measured service* by monitoring and optimizing the resource usage. Naturally, other relevant characterizations exist; see for example the work of Gong et al. [18].

Four deployment models are recognized in cloud computing systems. In *public cloud*, the access to the cloud infrastructure is provided for general public. *Private cloud* is intended to be used by a single organization, while in a so-called *community cloud*, multiple related organizations share the infrastructure. It is also possible to arbitrarily combine all of the above; this is referred to as a *hybrid cloud*.

Clouds offer various types of services, most of which fall into one of the three major service models: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)*, and *Software as a Service (SaaS)*. IaaS provides virtual storage, virtual machines, networks and other hardware resources for the consumer to deploy arbitrary software to. PaaS gives consumers opportunities to create applications using tools or libraries supplied by the provider, but the service provider has control over the underlying infrastructure. In SaaS service model, the consumer is provisioned with applications to use; in all other aspects, the provider is in charge.

In addition to IaaS, PaaS and SaaS (sometimes referred to as the SPI model), other service models have been distinguished, consistently adopting the naming form of "X as a Service". These include for example Storage as a Service, Security as a Service, Compliance as a Service, Identity as a Service, Monitoring as a Service, and very interesting in the context of this thesis, Logging as a Service.

While cloud computing undoubtedly has many advantages, such as pay-for-use billing strategies or load balancing and backup features, it introduces various problems as well. Fox et al. [16] state the following obstacles to and challenges for the growth of cloud computing: availability of service, data lock-in, data confidentiality and auditability, data transfer bottlenecks, performance unpredictability, scalable storage, quick scaling, reputation fate sharing, software licensing. Barrie Sosinsky [36] adds some more, among them the problem of customization (cloud versions of applications tend to have fewer features), the latency due to sometimes non-trivial data transfers, questionable privacy and security, or regulatory compliance issues, especially when collaborating across multiple countries. Last but not least, cloud monitoring represents a significant challenge, and with respect to the topic of this thesis will be given further attention in the next sections.

2.2 Cloud Monitoring

Monitoring, as defined by Mansouri-Samani and Sloman [24], is the dynamic collection, interpretation and presentation of information about objects or software processes. Common uses for monitoring include debugging, testing, accounting, performance evaluation, resource utilisation analysis, security, and fault detection [20].

In cloud, monitoring is essential for providers as well as for consumers. It gives the producer a necessary insight into the state of the managed resources, thus facilitating the improvement of services and problems diagnosis. It also equips the consumer with means to inspect the performance or workload imposed on the utilized resources, and enables them to compare various cloud providers if they provide unified objective metrics. Examples of areas that benefit greatly from monitoring include capacity and resource management (resource planning, performance and data centre management, etc.), where the unpredictable and highly variable real state of the resources can only be obtained by monitoring; establishing a provider's billing criteria based on an observable metric, troubleshooting, or proving compliance to security regulations [2]. Foster et al. [15] claim that monitoring of cloud is more complicated than that of grid environments due to their different trust models and different limits on the level of user's access to the resources. On the other hand, they predict that cloud monitoring will become a less important problem with the evolution of the cloud and its progress towards self-maintenance, although it will certainly represent a significant challenge in the near future.

Providing reliable and fine-grained monitoring for cloud requires non-trivial effort. Due to the cloud making extensive use of virtualization, it is difficult to have full control over the monitoring of resources. Moreover, the consumer's access to monitoring information is usually very limited and may not be sufficient for adequate estimates of resource status. Hasselmeyer and d'Heureuse [19] identify several requirements for cloud monitoring systems:

- *Multi-tenancy*. It is important for the monitoring infrastructure to be able to deliver the same data to multiple consumers simultaneously, as well as ensure isolation in the sense that no consumer has access to information not addressed to them.

- *Scalability*. Large numbers of monitoring nodes, tenants, event notifications or types of monitoring information should not represent a significant problem for the monitoring system.
- *Dynamism*. Dynamic modifications in the environment, such as addition and removal of tenants or the information they are interested in, should be supported.
- *Simplicity*. The monitoring system should have an easy-to-use interface, and be well-maintainable.
- *Comprehensiveness*. The same monitoring infrastructure should be suited for multiple data types, notification sources and tenants regardless of their specific properties.

Monitoring systems can be attributed many other desirable properties, including: *timeliness*, the prompt detection of and reaction to the events; *autonomicity*, or self-management to a certain extent; *extensibility*; low *intrusiveness* as well as low impact on the performance of the monitored resource; *resilience*, the ability to sustain dynamic changes or component failures without affecting the operation; *reliability* and *availability*; and providing *accurate measures*, i.e. reflecting the reality as closely as possible [2].

The monitoring process can be divided into four main phases [23]: *generation* of reports on events detected in the system; *processing* of the monitoring information, e.g. conversion to a specific format, validation, or filtering; *dissemination* to the intended destination; and *presentation* and adaptation to a form interpretable by the consumer. Hoffner [20] adds two more: *collation*, combining messages from different parts of the system; and *logging* the monitoring data to enable later processing.

The next section deals with existing solutions to the problem of cloud monitoring as described above. In addition to dedicated cloud monitoring products, recent works begin to show interest in the *Monitoring as a Service (MaaS)* paradigm [3][28]; therefore both types of tools are mentioned.

2.3 Existing Solutions and Tools for Cloud Monitoring

Since the cloud computing paradigm essentially evolved from grid computing and distributed systems as such, monitoring systems designed for these can generally be applied to cloud monitoring as well. There has been a substantial amount of work related to grid monitoring, including for instance Ganglia, a scalable distributed monitoring system primarily intended for clusters, grids, and high-performance computing systems in general [26]; and the MonALISA (Monitoring Agents in A Large Integrated Services Architecture) system, which is an ensemble of autonomous agent-based subsystems cooperating to perform a wide range of monitoring tasks in large scale distributed applications [31]. Due to this thesis being focused on cloud monitoring, monitoring systems originally developed for grids will not be given any more attention at this point; for further information refer e.g. to the work of Zanikolas and Sakellariou [43].

As stated by Foster et al. [15], cloud has several specific characteristics that may require a slightly different approach to monitoring than a general solution for distributed systems can offer. The rest of this section therefore aims to provide a brief overview of several platforms and tools tailored to the particular needs of cloud monitoring. For a comprehensive survey of such tools see the work of Aceto et al. [2]; a less extensive listing can also be found in [33] or [35].

Amazon CloudWatch

Amazon CloudWatch[1] is a MaaS tool built on top of AWS (Amazon Web Services[2]) to monitor AWS cloud resources, such as Amazon EC2 and Amazon RDS DB, and customer-specific applications. It measures resource utilization and application performance, and provides means to visualize the data collected in the process as well as to define custom metrics. It is also capable of sending notifications or raising an alarm when a special event occurs.

Nagios

An open source monitoring platform for cloud infrastructures, Nagios[3] offers comprehensive monitoring of applications, services, operating systems, and networks. Multiple APIs are available to facilitate the development of extensions, and hundreds of them already exist. Nagios also supports multi-tenant access to monitoring data.

QoS-MonAAS

A MaaS facility designed for Quality of Service (QoS) monitoring on top of a generic cloud platform, QoS-MONaaS (QoS MONitoring as a Service [3]) allows to describe the performance aspects of interest (so-called Key Performance Indicators) in a Service-Level Agreement. When a discrepancy is detected, an appropriate notification is triggered. The authors claim that the importance of QoS monitoring will only rise, due to the consumers' constant desire to evaluate the actual state of the resources they are paying for.

Lattice

Cloud monitoring tools that were primarily developed for monitoring distributed systems usually count on a relatively slowly changing state of the underlying infrastructure. Contrary to these approaches, Lattice[4] addresses the problem of elasticity introduced by cloud monitoring and provides reliable monitoring services even in a dynamically changing environment [11].

[1] http://aws.amazon.com/cloudwatch/
[2] http://aws.amazon.com/
[3] http://www.nagios.org/
[4] http://clayfour.ee.ucl.ac.uk/lattice/

PCMONS

An extensible modular framework PCMONS[5] is specifically designed for monitoring private clouds. One of the objectives of PCMONS was to ensure its seamless integration into the organizations' existing management infrastructure. The first release (2011) is compatible with Eucalyptus[6] at the infrastructure layer and Nagios at the view layer, and it also allows extensions that support other cloud solutions [12].

[5] http://code.google.com/p/pcmons/

[6] http://www.eucalyptus.com/

Chapter 3

Ngmon

Tovarňák and Pitner [40] discuss current approaches to cloud monitoring along with the problems it faces, and introduce their own solution that focuses on the producers of monitoring information instead of observing the monitoring architecture as a whole. Stemming from the requirements for cloud monitoring systems identified by Hasselmeyer and d'Heureuse [19], but extending them, they propose a set of general criteria for the producers of monitoring information, namely:

- *Multi-tenancy*. To provide multi-tenant services, several requirements need to be met: *concurrency*, i.e. the use of the same resources by multiple consumers at the same time; *isolation* in the sense that none of the consumers can access monitoring information not destined for them; *integrity*, meaning the monitoring data cannot be modified once it is generated; and *proof of origin* of the monitoring information to ensure non-repudiation.
- *Unified representation of the monitoring information*. In order to facilitate distribution and collection, and even more significantly, processing and consumption of the monitoring information, the introduction of a standard, universally recognized representation could bring substantial benefits. Not only does the monitoring data exist in various forms, the systems also tend to deliver and store it differently based on the nature of the information that is carried. For example, logs are mostly stored as natural language entries in text files or databases, notifications and alerts are usually pushed directly to the consumer to draw their attention, measurements can be requested on an individual basis, etc. Therefore unifying the format of monitoring information among the producers helps reduce the complexity of subsequent processing and correlation of data from multiple sources.
- *Extensible data format*. Related to the problem of uniform representation mentioned above, the extensibility of the monitoring data format cannot be accomplished in a satisfactory manner as long as the information has no specific form. In such case to adequately process the data and obtain the information of interest means to employ some kind of a pattern-mining technique or even review the data manually. Utilizing data formats that are standardized, self-describing, based on an extensible schema, structured, and compact, renders the extensibility perfectly feasible and facilitates efficient processing.
- *Delivery channel*. The channel for the delivery of monitoring information should sup-

port all types of interaction identified in [39], i.e. publish-subscribe, query-response, and notification. It should also be capable of both synchronous and asynchronous data transfers.

Tovarňák and Pitner [40] also claim that there is a need to completely redesign monitoring, as opposed to creating yet another complex monitoring tool. To prove satisfiability of the abovementioned requirements, *Ngmon*[1] (New Generation MONitoring), a prototype of an event-based monitoring daemon, was designed and implemented.

The work of Spring [37][38] defines, based on the guidance issued by the Cloud Security Alliance [6], a layered cloud model, where the nature of each layer dictates a slightly different approach to monitoring. The model consists of seven layers: facility (i.e. the physical location of the hardware resources), network, hardware, operating system (OS), middleware, application, and user. Ngmon focuses on monitoring virtual machines spanning the top five layers, i.e. hardware, OS, middleware, application, and user. In the following, we describe the main characteristics of the monitoring daemon.

This thesis aims to contribute to the implementation of Ngmon, and although the solution detailed in subsequent chapters can be used autonomously, its primary objective was to be incorporated as a component in the daemon.

3.1 Unified Representation of Monitoring Information

To ensure a unified format as per the requirements set above, all kinds of monitoring information are produced in the form of *events* (event objects). Etzion and Niblett [13] define an event as "an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain". They note that the term is also used to refer to a programming entity that represents such occurrence.

Each event is represented as a JSON object, which has a uniquely determined type. An *event type* is "a specification for a set of event objects that have the same semantic intent and same structure" [13]. In other words, all events having the same type, i.e. belonging to the same class, are characterized by the same set of attributes and carry the same meaning. The structure of an event is properly defined by a JSON Schema. Such format is easily extensible with a custom JSON Schema, and it will be elaborated on later in this thesis.

3.2 Storage and Delivery Channel

Having the monitoring data in a unified format, the storage no longer depends on where the information originates; so all events can be collected in a single point (a UNIX domain socket) and processed the same way. The sensors publishing events into the UNIX domain socket are further authenticated in order to provide a proof of origin. The event objects are kept in a so-called event store, which is an encrypted, lightweight key-value database based

[1] http://github.com/ngmon

on Berkeley DB[2]. They are encoded in a JSON-compatible binary format SMILE [14] and cannot be modified once they enter the event store, thus ensuring integrity.

To comply with the requirements set for the delivery channel, Ngmon contains both a query evaluator component and a publish-subscribe component. The query processor accepts a simple query language enabling batch requests for information from the event store. In case of periodically querying the event store, it can also be regarded as an implementation of the notification paradigm. The publish-subscribe component is used to distribute the events of interest, specified by consumers in the form of subscriptions, as soon as they are issued by the producers. For the communication between producers and consumers, a custom lightweight frame-based protocol over TCP layer is used.

3.3 Access Control

Not all consumers are authorized to view everything, therefore a specific mechanism was implemented to control access to monitoring information. To be more precise, it is possible to define rules that restrict subscriptions at the time of their creation, and a new subscription will not be added if it is not in accordance with all constraints. Prior to working on this thesis, we focused on the implementation of the access control component, therefore this section tries to provide a gist of our results in this field.

The rules (also called *constraints*) are specified on a per-consumer basis and control access at the level of event attributes, for example "it is forbidden for user A to access events with type equal to B and values of attribute C greater than D". The access control component follows a liberal strategy, i.e. everything is allowed unless explicitly forbidden; therefore there are deny rules only.

A simple declarative language was designed to enable the formulation of these constraints. Given the user the particular constraint belongs to, and the event attribute it restricts, the requirements on the forbidden value are expressed using the language in the form of `#<operator> <arguments>`. Currently, seven operators are recognized: less than (`#lt`), less than or equal to (`#le`), greater than (`#gt`), greater than or equal to (`#ge`), equal to (`#eq`), prefix (`#pref`), and range operator (`#rng`). All operators require numeric values only, except for prefix and equals, which accept arbitrary string literals as well. Space is regarded as a delimiter, so in case it appears in an input, the whole string needs to be enclosed in apostrophes. Consider the following as examples of valid expressions:

```
#lt 0
#pref string
#eq 'quoted string'
#rng -10 10
```

When checking a subscription against all deny rules, the intervals denoted by the particular operators must be handled properly. For example, in case a constraint forbidding

[2] http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html

`A > 10` exists, a subscription for `A > 0` will be rejected due to its non-empty overlap with the mentioned constraint. Allowing to add the subscription and subsequently filtering just values greater than 10 could lead to confusion, since the consumer may be under the impression that no other values ever occur in the system, unaware of the fact that there may be a constraint causing the component to leave out some of them.

A custom parser for this language was implemented as well, for we believed that it would be more efficient than using regular expressions for parsing. The performance tests we conducted indeed showed that our parser processed the input 5 to 50 times as fast as the equivalent regular expression, depending on the length and type of the constraint.

Chapter 4

Logging and Log Management

Logs, often collected for system monitoring, debugging, and fault diagnosis purposes, provide a valuable source of monitoring information [30]. Reflecting important events (that happen in or have influence over the system) in log records helps maintain constant situational and informational awareness of the state of the system [7].

NIST defines log as "a record of the events occurring within an organization's systems and networks. Logs are composed of log entries; each entry contains information related to a specific event that has occurred within a system or network." [21]. Chuvakin [10] states that a log is what a device or a piece of software generates in response to a certain stimulus; while the stimuli vary depending on the producer of the log message.

Log entries typically consist of a timestamp, indicating the date and time of the occurrence of the event or its detection; the source of the event, such as an application that produced the event; and the actual message carrying detailed information. Some producers specify the importance (sometimes referred to as severity, priority, rank, level, category, etc.) of the event as well, together with other fields the consumer might find relevant.

Common uses for logging include resource management (hardware and software tuning and diagnosis), debugging and troubleshooting, intrusion and attack detection, forensics, and audit. In some cases, there are even special laws and regulations that enforce keeping logs for security reasons.

There are several challenges when it comes to logging. First, and probably the most significant, is that to date no standard representation for log messages has been established [10][7], and almost every producer of logging information uses a slightly different custom format. This results in problems with interoperability and correlation of logs from multiple sources.

Moreover, as the amount of logging data increases, an efficient storage and processing comes into question. In addition, cloud-based logging, which has been gaining attention over the past few years, has its own set of issues that are yet to be satisfactorily solved. These are related, among other things, to the decentralization and volatility of logs, their acquisition over multiple layers, retention policies, availability and accessibility, an appropriate level of details [25], and also to privacy protection and the overall security.

This chapter is dedicated to the concepts concerning log analysis and processing in general, as well as with respect to the particular nature of cloud computing and cloud monitoring. It also aims to provide a summary of existing approaches and comment on known issues, leading to the presentation of our own solution.

4.1 Log Production and Acquisition

According to Chuvakin [10], the ideal logs should contain at least information about what happened, when it happened, where it happened, who was involved, and where they came from. In addition, it would be useful to know where more information can be obtained, how certain it is that the event is reported correctly, and what else is affected. In a perfect scenario, advice on what should be done about the situation, what other events occurred that might be of relevance to this one, and an estimate of what will happen next would be very welcome, although it is not usually possible to include it.

The worst mistakes [8] found in logs can be summarized as follows: not logging what is essential; lacking context and important details; format unsuitable for humans or machines; inconsistent syntax even among the messages in the same application; and logging pieces of source code or confidential data. On the other hand, besides containing the necessary information, a useful log should be suitable for manual, semi-automated and automated analysis, regardless of whether the application that produced it is available for inspection; it should not represent a major performance obstacle; and it should be reliable in case it needs to be used as an evidence.

As it was implied above, currently there is no widely used standard defining the exact structure of log records. It is a common practice among developers to output the logs in the form of natural language constructs with variable parts corresponding to the information being recorded, such as "`User <userId> logged on`". Not only do these differ from system to system, from developer to developer; in case the logs are intended for machine processing this also requires additional (and often expensive) analysis of the records in order to extract the interesting piece of information. Normalization (i.e. conversion to a uniform, desired format, usually to facilitate processing) and correlation of events from different systems is very expensive if not impossible, and the processing often requires non-trivial data mining techniques.

For these reasons, there has been a growing ambition to standardize the form of log entries. MITRE's Common Event Expression (CEE) [7], for instance, aims to become a standard for unifying the categorization, terminologies and representation formats of the events being logged. The CEE architecture consists of CEE Profiles, defining the requirements for the structure of CEE Events; CEE Log Syntax (CLS), which dictates how CEE Events are represented and encoded; and CEE Log Transport, listing both mandatory and preferred characteristics of a log transport protocol for sharing events. The CLS intends to define how to encode a CEE Event into an event record, given a corresponding CEE Profile, to provide maximum interoperability with existing standards [29]. Moreover, it defines various kinds of event and event field encodings, emphasizing a straightforward translation between them due to the fact that all of them represent the same event structure. CEE also offers recommendations about the events and fields to be logged under various circumstances.

The Common Event Format (CEF) [5] is an older HP ArcSight[1] standard that again pro-

[1] http://www8.hp.com/us/en/software-solutions/software.html?compURI=1314386

motes interoperability between different event- or log-generating devices from various vendors. It defines a different syntax for log records, using the Syslog format as a basis and constraining the actual message field. There are several mandatory fields, delimited by a vertical bar; the last of them allows custom extensions. Other standardization efforts include OpenXDAS [2] or IBM's Common Base Event [32].

There are no definitive rules as to what should be logged, since it depends greatly on the circumstances and the nature of the application or system. However, some recommendations can be given. As per Chuvakin and Peterson [9], the following events should be logged: authentication and authorization, both successful and failed; changes in data and its attributes, especially privilege elevation; availability issues, startups and shutdowns, faults and errors, connectivity problems; resource issues, such as resource exhaustion, exceeded capacities, reached limits; invalid inputs and known potential security threats. Another piece of advice by Chuvakin [8] states that one should: log everything, retain the majority of the records, analyse enough of them, summarize and report on a subset, monitor some of the logs, and act on just a few.

In any case, logging systems produce considerable amounts of monitoring data, and the quantity will only grow in the future. Moreover, most of the logs produced need to be available for a certain period of time, based on the particular policies and regulations. This introduces challenges related to the storage and subsequent processing of such volumes of data.

Log records are commonly stored in various formats, ranging from simple text files, through binary and compressed files, to databases. To choose the most suitable option, many factors must be considered for the particular scenario, such as whether the particular format is both machine- and human-readable, whether it compresses well (mostly for archiving purposes), the availability of tools for viewing and processing its contents, storage capacities at the disposal and their price, writing and reading speed, etc.

One of the possible ways to cope with the large quantities of data is to use cloud storage or even cloud Logging as a Service (LaaS) tools. The choice again depends on the needs of the particular logging system, since the providers have various log acquisition methods, deployment models, log retention policies, searching and reviewing capabilities, custom alert services, and last but not least, billing options.

4.2 Log Processing

In order to better understand the monitoring information collected in the form of log records, various analysis tools and techniques are employed. Due to the great variability in log formats, there is no general way to analyse and process logs; the approach depends on the log syntax and form. Another complication is caused by the subjective nature of log messages; e.g. when filtering logs by severity, there is no guarantee as to the real importance of a message the producer decided to tag with a certain severity type. Also, there might be

[2] http://openxdas.sourceforge.net/

log records that are meaningless by themselves and only make sense if set in a particular context, or processed as a part of an event stream.

Overall, log analysis is not an easy task, but it needs to be performed for various reasons, for example for maintaining situational awareness of the managed resources, ensuring security and detecting attacks, diagnosing faults, or to comply with audit regulations about log reviews. Manual log analysis is usually not feasible, not only considering the quantity of log records to process, but also due to it being unlikely to provide a bigger picture or to correlate diverse logs from multiple sources. Manual analysis is not recommended unless a quick look at the logs is all that is necessary in a particular situation and the analyst is familiar with the format of the logs [10].

As the logs tend to be produced in the form of natural language strings, one of the most used techniques for manual processing of log files is to parse them using regular expressions. These can either be created manually in advance, based on the known form of the log record, or they can be built using various log abstraction methods. Log abstraction is a technique designed to isolate the static part from the dynamically changing parameter fields contained in a log message [30], which results in a set of regular expressions to be used for subsequent pattern matching. Some of log abstraction methods extract the regular expressions from the logging statements found when examining the source code [42], others use data clustering techniques [41][22][30] or other approaches. Not only do the results of these techniques rely on several assumptions that may or may not hold in particular cases (e.g. that the source code is available for inspection); they also involve the time-consuming process of parsing the logs by regular expressions.

The abovementioned approaches, however, are only suitable for cases when one knows what to look for in the log records. Unless there is a certain prior knowledge of the form or content of the logs, they are not applicable; in such cases, data mining comes into play. Peng et al. [34], for instance, present a log mining approach that takes advantage of the intrinsic characteristics of log messages, such as the fact that they are relatively short and they typically contain a timestamp, which assists in correlation of different messages. With data mining methods, it is possible to extract patterns from the collected data and get a bigger picture of what is happening in the system. For example, discovering several login failures followed by a successful login may imply a brute-force attack.

Before any kind of log analysis or processing is applied, the security and privacy issues must be considered. Log records often contain personal or confidential data, so measures should be taken to prevent compromise. Accorsi [1] distinguishes two threat models for privacy: outer and inner privacy, aiming at the latter, which he defines as an attacker's attempt to access private log data instead of collecting data about an individual manually. He then presents a tamper-evident secure remote logging approach for ensuring inner privacy.

Furthermore, cloud logging poses special problems and threats stemming from its nature. The collection of logs itself is not always possible, and the consumer's level of control over logs also depends on the service model. Zawoad et al. [44] introduce SecLaaS (Secure-Logging-as-a-Service) as a concept of providing access to logs while preserving the confidentiality of users and integrity of logs. They also emphasize that it is necessary to prevent tampering with logs, both on the side of cloud service provider and cloud consumers.

There is a wide range of tools specially designed to facilitate working with logs, and more of them still appear. The production of logs does not represent a particular problem, since every programming language has its logging frameworks (Log4j[3], Logback[4], or Java Logging API[5] can be mentioned as examples of possible alternatives for Java, which this thesis primarily focuses on), and it is not uncommon for a developer team to come up with its own logging solution. In addition to application logging only, system-level solutions such as Syslog or Windows Event Log provide a means for a centralized log collection.

Regarding log processing and analysis, however, the situation becomes more complicated. There is certainly not a lack of tools providing this kind of services, but typically they serve for filtering and searching free-form log records with natural language messages and thus the outcome of the analysis depends on the analyst's ability to pose the right questions. To give an example, Splunk[6], together with its cloud-service based version called Splunk Storm[7], is a log collection and search engine for unstructured, text-based logs. It makes use of its own custom-designed query language SPL (Search Processing Language), specially targeted at manipulation with large volumes of data. Splunk at first gathers the data, possibly from multiple locations, to a central repository and indexes it to improve the processing of subsequent queries. It can then perform a powerful analysis of the collected machine data and generate comprehensive reports, graphs and charts to visualize the results. Competitors and alternatives to Splunk Storm in the field of centralized cloud-based logging include e.g. Loggly[8], Logentries[9], Papertrail[10] or the open-source Logstash[11].

4.3 Structured Logging

Logs in a free-form natural language are the easiest for the developers to produce, but they represent a serious problem once they need to be processed. The most commonly used approach, i.e. parsing the logs by regular expressions, introduces a significant overhead; in addition, the messages subject to processing by regular expressions are very sensitive to changes, since even a small difference may result in the message not being matched.

The most interesting piece of information is usually the variable part of the message, as opposed to the static text accompanying it. In order to extract it, one basically needs to crawl through the unstructured text to find something the developer already had at their disposal in the first place. Therefore, instead of burying the essential data in human-readable padding, a different approach may be just what is needed to move towards a better logging.

[3] http://logging.apache.org/log4j/2.x/
[4] http://logback.qos.ch/
[5] http://docs.oracle.com/javase/7/docs/technotes/guides/logging/overview.html
[6] http://www.splunk.com/
[7] https://www.splunkstorm.com/
[8] http://www.loggly.com/
[9] https://logentries.com/
[10] https://papertrailapp.com/
[11] http://logstash.net/

Parallel to the log standardization efforts, although not so evident, there has been a discussion about structured logging. However, most of the recommendations revolve around preparing and logging the structured output (be it JSON, XML, or other formats) manually without any major changes to the existing logging tools. In other words, instead of representing the log message as a natural language string, the developer constructs the message in the desired form and just outputs a differently formatted log record. While it would certainly make a difference when processing such logs, the additional work it requires on the developer's part is probably a major obstacle to the widespread adoption of this approach. The rest of this section gives examples.

Founded in the beginning of 2012, the main objective of Project Lumberjack[12] is to enhance traditional Syslog logging with CEE syntax. JSON data is sent in a traditional Syslog message, preceded by a known prefix `@cee:` to indicate the message is CEE-encoded. When processing the logs, the message is either treated as an ordinary natural language string, or in case the CEE format is detected, the JSON part is parsed accordingly.

As a result of another initiative, Fluentd[13], a log collecting daemon for semi-structured JSON logs was created. It is designed for receiving, buffering and forwarding data to another destination. Fluentd log entries consist of three parts: a UNIX timestamp, a tag used for routing the message in the forwarding process, and the actual message in the form of JSON object. An advantage of Fluentd over other logging solutions using JSON is that there is no need for the developer to construct the JSON object; however, a map containing key-value pairs for all attributes to be logged must be created and passed to the logger. In our opinion, the obligatory construction of the map represents a significant inconvenience to the developer since typically, several lines of code are necessary to accomplish that, and in case of larger number of logs the code as a whole may appear polluted and disorganized.

Several other, mostly experimental, tools for structured or semi-structured logging are already available, such as Bunyan[14], a simple JSON logging library for Node.js; nevertheless, to the best of our knowledge, all of them require a non-trivial participation of the developer in order to obtain logs in JSON. Since we realize that a solution unpleasant to use will not be generally adopted, this is exactly the issue our logging mechanism wants to address, as it will be described in the next chapter.

[12] https://fedorahosted.org/lumberjack/

[13] http://fluentd.org/

[14] https://github.com/trentm/node-bunyan

Chapter 5

Structured Logging Mechanism Proposal

The reasoning behind the need for a completely new approach to logging was already stated in previous sections. Particularly, it should be noted that the manipulation of logs in natural language, as it mostly is the case even in modern systems, is a complicated, inefficient and difficult-to-maintain solution. A new approach refraining from the use of natural language as the main medium for carrying logging information would be more than welcome. However, for an innovative approach to be accepted by the developer community, it needs to be adequately straightforward to use compared to the currently existing ones, and it should be evident that it facilitates rather than complicates the usual process; otherwise, there is no reason to change. The aim of this thesis is to elaborate on the difficulties one must face when designing such approach and suggest a logging mechanism that attempts to solve the aforementioned problems.

From this chapter on, we use the term *logging* to denote *application logging* as opposed to system or security logging. In application logging, the primary source of information is the application code, and it is usually the developer who is in charge of deciding which events to log and in which form. This often leads to inconsistency among the logging statements due to multiple developers using different message formats and severity levels to represent the same event. Even strict logging policies cannot guarantee that the format of the logs will be uniform throughout the application. Our approach strives to address exactly these issues; focused on application logging and implemented in Java, it is intended to serve as an adapter for structured logging from Java applications.

As far as we know, currently developed tools and libraries do not provide structured logging in the sense that we understand it. We would like to achieve a structured output with minimal additional effort on the programmer's part; preferably, with as much of the configuration and the logging process automated as possible. We do not want to force the developers to construct the structured logs themselves, but instead provide the logger with the raw (i.e. not bundled in natural language string) data and let it take care of the adequate structure. This way it is also possible to change the structured representation in the background with no effect whatsoever on the logging statements already present in the code.

The logging mechanism presented in this chapter was designed and implemented primarily in the context of Ngmon, but the concepts are universal. As Chapter chapter 3 says, every type of monitoring information in Ngmon, logs included, is represented as an event; therefore we use the terms *event*, *event object*, *log* and *log record* interchangeably. The general form of an Ngmon event is shown in Figure 5.1. Note that it differs slightly from the event presented in [40]; it is because since then we have identified that the structure needed minor modifications in order to enable automated processing.

```
{ "Event":{
    "id":16051986,
    "occurrenceTime":"2012-04-11T08:25:13.129Z",
    "hostname":"lykomedes.fi.muni.cz",
    "type":"org.apache.httpd.request.GET",
    "application":"Apache Server",
    "process":"httpd",
    "processId":"4219",
    "severity":5,
    "priority":4,
    "payload":{
      "schema":"http://httpd.apache.org/v2.4/events.json#/definitions/GET",
      "schemaVersion":"2.4",
      "properties":{
        "resource":"/apache_pb.gif",
        "protocol":"HTTP/1.0",
        "response":200
      }
    }
} }
```

Figure 5.1: Sample JSON event object

Every JSON event object adheres to a general JSON Schema. Moreover, the object representing the `payload` part of an event object is described by another, specific JSON Schema, determined by the particular event type. Since the names and types of attributes differ among the event types, a description such as the JSON Schema is the only way to ensure unambiguous interpretation of the payload. Event types and their properties are not given nor restricted in any way; they can be tailored to the purposes of the particular application and defined by the developer. The logging mechanism we introduce in this thesis is responsible for producing the `payload` part of an event object, as it will be detailed later in this chapter.

Among the main design goals of our logging mechanism is a simple API and a solid performance. To facilitate its adoption, it is convenient for the API to resemble that of common logging frameworks. Since the logging module is intended for Java, we based the API on the well-known Log4j. As to the performance issues, not only should this approach lead to efficient processing of the logged information; it is also desirable for it to introduce minimum to no performance overhead in the actual logging process and so to provide performance comparable to that of unstructured loggers.

Since the logging component is intended to be no less developer-friendly than other logging frameworks in spite of its task to output the logs in JSON, it is desirable to achieve a high level of automation in creating the necessary constructs for this kind of logging. Therefore the one-to-one correspondence between logging methods and JSON Schemas describing them, implying the possibility to generate one if the other is given, is very welcome.

For the generation of resources, i.e. source files and JSON Schemas, a custom annotation processor is used. Annotation processors are basically Java classes that implement the `javax.annotation.processing.Processor` interface and as such can act as compiler plugins. They were introduced in Java 6 as a part of the Pluggable Annotation Processing API (JSR 269), replacing a standalone, non-standardized command line utility called `apt` (Annotation Processing Tool[1]) needed for processing annotations and executing annotation processors in Java 5. The API is called pluggable because the annotation processors can be dynamically attached to the compiler to perform a source code analysis, and subsequently carry out tasks such as code validation or generation of new resources. JSR 269 itself consists of two parts: an API for handling annotation processors (`javax.annotation.processing`) and an API providing a compile-time view of the sources (`javax.lang.model`) that plays a similar role as Java Reflection API does during runtime.

Our annotation processor is responsible for creating JSON Schemas based on the content of logging methods and vice versa. JSON Schema [17] is a specification defining the format of JSON data, analogous to what XML Schema is for XML documents. It is intended for description, validation and documentation of JSON data. The specification is still a work in progress; `draft04` was assumed as default for this thesis, being the latest available at the time, and all JSON Schemas used for our purposes must comply with this version of the specification.

The structure of the logging component, and as a result, its API, has undergone many changes. Finally, three variants were settled upon, each with a slightly different perspective on the relationship between entities and event types[2].

5.1 Event Types Tightly Coupled With Entities

At first, we approached the problem entirely from the point of view of monitored entities. A *monitored entity* (henceforth referred to as *entity* for short) is an object in the monitored domain disposing of certain qualities that render it interesting for monitoring; essentially any resource that can be considered useful, unique, and having a considerable lifetime and general use [43]. Zanikolas and Sakellariou [43] give processors, memories, storage mediums, network links, applications, and processes as typical examples of entities.

Assuming not all actions are applicable to every entity, each entity is unified with a separate logger, which defines the set of event types the entity supports. Hence when logging an event concerning a particular entity, only a subset of all event types in the system can be used.

5.1.1 Entity Classes and JSON Schemas

Each entity is represented by a class consisting exclusively of logging methods. By convention, the name of the class is the same as the name of the entity, and similarly, names of

[1] http://docs.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html

[2] The full source code of all three variants is available from https://github.com/ngmon/ngmon-json-logger

logging methods and their parameters correspond to the names of supported event types and event type attributes, respectively. Method overloading is forbidden because it does not completely agree with the concept of categorizing events into disjoint types. If two methods with the same name differ only in the list of their parameters, they are very likely to represent the same event type, just in different levels of detail.

Two different types of JSON structures are generated based on the information obtained from the entity class: JSON objects describing entities and JSON Schemas for event types. JSON objects describing entities conform to the schema in Figure 5.2, i.e. they contain an array of objects named `eventTypes`, where all logging methods from the entity class are listed.

```
{ "$schema":"http://json-schema.org/schema#",
  "title":"Entity",
  "type":"object",
  "properties":{
    "name":{
      "type":"string"
    },
    "eventTypes":{
      "type":"array",
      "items":{
          "type":"object",
          "properties":{
            "eventType":{
              "type":"string"
            },
            "schema":{
              "type":"string"
            }
          },
          "required":["eventType", "schema"],
          "additionalProperties":false
      }
    }
  },
  "required":["name", "eventTypes"],
  "additionalProperties":false
}
```

Figure 5.2: JSON Schema for entity objects

The schema requires each entity object to have only two attributes: `name`, which is the name or a certain identifier of the entity, not necessarily equal to the name of the entity class; and `eventTypes`, an array of objects specifying the path to schema for each particular event type (i.e. logging method).

All entity classes are located in package `logger` and its subpackages, while the generated JSON entity objects and JSON Schemas for logging methods can be found in subpackages of `entities` and `schemas`, respectively. The structure of subpackages in `entities` and `schemas` copies that of `logger`, i.e. if there is an entity class `EntityX` containing a method named `eventA` in package `logger.a.b.c`, the JSON object associated with this class will be placed to `entities.a.b.c` under the name of `EntityX`, and there will also be a JSON Schema named `eventA` in `schemas.a.b.c`. For the sake of simplicity, all three base packages lie at the root of their respective directories in line with the standard Maven[3] project structure; i.e. package `logger` directly in `src/main/java`; and packages `entities` and `schemas` in `src/main/resources`.

As Figure 5.3 shows, entity objects are essentially collections of references to event type schemas that are needed for matching an existing log record against its schema. The items of array `eventTypes` are characterized by two properties: `eventType`, the name of the method as declared in the entity class, and `schema`, the location and name of its schema.

```
{ "name":"EntityX",
  "eventTypes":[
    { "eventType":"eventA",
      "schema":"schemas/a/b/c/eventA" },
    { "eventType":"eventB",
      "schema":"schemas/a/b/c/eventB_2" },
    { "eventType":"eventC",
      "schema":"schemas/x/y/eventC" }
  ]
}
```

Figure 5.3: Sample JSON object describing `EntityX`

At first glance, it might not be obvious why these objects are needed at all - exploiting the convention over configuration paradigm and considering the absence of possibility to overload methods, the schema for each log record should be uniquely identified given the entity (or, to be more precise, the package it is defined in) and the name of the event type. Even more simply, we might have only used JSON Schemas for entities, incorporating subschemas for their supported event types such that a logged event for a particular entity would match exactly one of the subschemas. The primary reason the event type schemas are in separate files and not directly included in the entity schema is reusability. Multiple entities can, and often do, share the same event type, so linking all of them to the same event type schema ensures consistency and does not require changes in schemas in multiple locations. Resulting from this, several entities in the same package can declare the same logging method (i.e. having the same name, and the same parameter names and types) and only one physical file containing the underlying JSON Schema for it will be generated. If, however, another method with the same name but different parameters is created in the same package, the name of the method no longer serves as a unique identifier for the schema. In this case a new JSON Schema is created, its name composed of the event type name and an identification number appended to it with an underscore. This is where the references come in handy, since they provide an unambiguous mapping from method names to the equivalent

[3] http://maven.apache.org/

event type schemas.

It was very convenient for the idea of reusing event type schemas from the same package to be further generalized to any package. The conventions about package correspondence between entity classes and JSON structures generated from them were not to be bent or broken, however; therefore we introduced a custom annotation `@Namespace` that enabled specifying the target package for a particular event type schema. Its function can be illustrated using an example: Considering an entity class `EntityX` in package `logger.a.b.c` and its method `eventC` annotated with `@Namespace("x.y")`, the target package for the event type schema describing `eventC` will be `schemas.x.y`, as shown in Figure 5.3.

Finally it remains to define the structure of the event type schemas. These are very simple due to the fact that any additional constraints imposed on the event type attributes in the schema, such as the range of values or a string pattern, would be very difficult to reflect in the logging methods in a way that would enable for unambiguous translation back to the schema in case of changes. Figure 5.4 presents a schema corresponding to a method named `eventA` with two parameters called `param1` and `param2` of type `String` and `int`, respectively.

```
{ "$schema":"http://json-schema.org/schema#",
  "title":"eventA",
  "type":["object"],
  "properties":{
    "param1":{ "type":"string"  },
    "param2":{ "type":"integer" }
  },
  "required":["param1", "param2"],
  "additionalProperties":false
}
```

Figure 5.4: Sample JSON Schema for method `eventA`

All the aforementioned resources are generated by a Java annotation processor. The processor assumes the following:

- Entity classes (and these only) are located in package `logger` and its subpackages. Classes from other packages will not be taken into account.
- JSON files are located in accordance with the standard Maven project structure in `src/main/resources`; particularly: JSON objects corresponding to entity classes in package `entities` and its subpackages, and JSON Schemas corresponding to logging methods in package `schemas` and its subpackages.
- The structure of subpackages is synchronized across the three base packages the way it was already stated above.
- Only classes changed since the last build are processed.
- In case a JSON object does not exist for a particular entity class, it is generated; and vice versa.

- If both resources already exist, entity classes are granted higher priority so only the changes from entity classes will be reflected in the corresponding JSON files. If an entity class exists, it is guaranteed that no change in JSON resources will ever cause an update in it. This implementation was chosen because it was expected for the classes to be more conveniently changed than JSON structures.

Before the generation of files itself, the processor checks if all conditions imposed on the sources are met, namely it forbids method overloading and it only accepts JSON in the specified format. If a file does not satisfy these criteria, the execution of annotation processor terminates with a compilation error.

For JSON manipulation, Jackson JSON Processor[4], a high-performance JSON parser and generator, is used. The actual handling of JSON is quite straightforward; nevertheless, one thing should be noted. By design, JSON object is a set of key-value pairs that does not define any ordering on its attributes - therefore, for example, the order of parameters in a logging method does not necessarily match the one in its JSON Schema. In our case, however, it represented no particular problem. The only time it might matter is when using logging methods that were generated from JSON Schema, and even then if the order of parameters in a method is undesirable, it is sufficient to rearrange them in the signature of the method. Once the method exists, it will not be affected by its schema again.

5.1.2 API and Logging Process

Since there is no need to reinvent the wheel, the logging component has been designed from the beginning to serve as an adapter for other logging tools, not to become a standalone tool itself. In the version being discussed now, Log4j 2 is chosen as the underlying logging framework, its integration hardcoded in the component. We chose Log4j 2 as a de facto standard for Java logging. Although the original Log4j is no longer under development and the framework has been in fact replaced by Logback[5], Log4j 2 aims to revive its former glory by providing functionality comparable to, or in some aspects even surpassing, Logback [4].

The developers programming against our API do not need to know that Log4j 2 is in the background, nor do they have to configure it manually, for everything is preset. The implicit behaviour for the component is to log to a file; if however, logging to a file is not suitable in a certain context, the default settings can be overridden by creating a custom Log4j 2 configuration file `log4j2.xml` in the implementing project. For particulars on configuration options see the User's Guide [4], Chapter 6 - Configuration.

Each user-defined entity class as outlined above is a subclass of the abstract class `Logger` where methods that directly interact with Log4j are situated. By default, they produce log records in the form of `[date] [time] [severity] - [message]`, where message is the JSON object encapsulating given information about the entity and the particular event type. Figure 5.5 gives an example of a log message for an instance of event type `eventA`, logged from

[4] http://wiki.fasterxml.com/JacksonHome

[5] http://logback.qos.ch/

within entity `EntityX`. (Note that it is formatted for readability; in the log records, the JSON object is always output on a single line.)

```
{ "entity":"EntityX",
  "eventType":"eventA",
  "schema":"schemas/a/b/c/eventA.json#",
  "properties":{
    "param1":"abc",
    "param2":42
  }
}
```

Figure 5.5: Sample log message

As demonstrated above, the log record consists of the entity that a particular event is related to, its type, path to the JSON Schema that describes the event type, and finally an object composed of values for all attributes of the event type. It is important to include the path to the schema, so that the structure of the particular log record is determined instantly and the payload can be efficiently processed. Considering the already mentioned naming conventions for the location of schemas and their names, it is possible to parse the name of the event type out of the path to its schema file. Despite the fact that this renders stating the event type in the log record redundant, the event type is included to facilitate processing; but it may be subject to removal in later versions.

The actual logging statement to produce the message in Figure 5.5 is then constructed as follows:

```
LOG.eventA("abc", 42).error();
```

`LOG` is an instance of `EntityX`; method `error` represents one of the six severity indicators imitating the logging levels used by Log4j (`FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`). All logging methods in entity classes pass the values of their parameters to a generic logging method in the superclass; the severity methods at the end of the statement then confirm the operation and call the corresponding method of the underlying Log4j logger to perform the actual logging.

It must be admitted that the production of the JSON representation of the logged event was implemented in a relatively inefficient way due to the problem of correspondence between logging methods and their schemas. For the needs of future processing, it was necessary for the JSON log record to contain a reference to the schema describing it. However, since the name of the file containing the matching schema is not uniquely determined by the name of the logging method, for each log request the entity JSON object had to be parsed and searched in order to find the path to the event type schema in question.

The extreme overhead thus caused has been significantly reduced with the introduction of caching. For each entity, a collection of mappings from method names to paths to their schemas is now kept in memory, and so the entity JSON objects are only parsed the first time an entity is used for logging. It is also possible to control the loading of selected sets of these mappings manually by calling the static `initLoggers` method of the `Logger` class, giving it the fully qualified names of entity classes as parameters; or even using `initAll` to cache the method-schema pairs for all entities in the project.

In order to prevent mistakes and make the creation of logging methods more developer-friendly, the name of the method (i.e. the event type) and names of its parameters, both of which have to be recorded in the log message, are not sent to the superclass directly as parameters. It would force the developer to include multiple String constants in the method call (in addition to the actual values of the particular event type attributes) and strongly violate the DRY[6] principle, since the Strings would only represent what is already stated in the method declaration. Moreover, any typing error would cause discrepancies between the schema (generated from the method declaration) and log records (created using the values passed to the logging method), resulting in errors when processing the logs. To avoid such problems, the logging process takes advantage of Aspect-Oriented Programming (AOP). Using an AOP tool for Java, AspectJ[7], we implemented a custom aspect that is attached to logging method calls and extracts the necessary information from them. Each time a logging method from an entity class is called, the aspect intercepts its execution, retrieves the name of the entity class, its package, and the name of the method plus the names of method parameters, sends them to the `Logger`, and then returns control to the intercepted method.

The greatest downside of this model is the direct connection between entities and event types. Despite the reusability of schemas, tight coupling in this case clearly leads to flexibility and maintainability issues. For example, if a developer wants to define a new type of event applicable to all (or nearly all) entities, there are too many steps to perform: besides defining the event type by its JSON Schema, they also have to create a reference to this event type from each entity so that the entity could use it for logging; not to mention declaring the method in each and every corresponding class. The more repetitions, the more error-prone the process becomes.

5.2 Recommendation-Based Binding of Event Types to Entities

Mainly to eliminate the tedious task of updating a potentially great number of entities in order to add a new event type, we proposed an alternative approach. Entities are no longer equivalent to loggers; there is one central logger combining information about the entity and a particular event type into one record. Event types are stored independent of entities, logically aggregated into groups, and each group has its underlying JSON Schema. There are no schemas for entities; it is, however, possible to enumerate the event types supported by an entity. The logger then checks if the event type being logged is allowed to be in association with a certain entity. Contrary to the first approach, this one provides no guarantee for an entity to be used together with suitable event types only.

[6] Don't Repeat Yourself, as established by Andrew Hunt and David Thomas in *The Pragmatic Programmer: From Journeyman to Master.* (Addison-Wesley, 1999)

[7] http://www.eclipse.org/aspectj/

Note: when describing approaches that followed the entity-centred one found in Section 5.1, only the changes in design or implementation with regard to the first variant are mentioned. Concepts that are not explicitly stated are assumed not to differ from the first variant.

5.2.1 User-Declared Classes and JSON Schemas

Unlike the previous version, this one aggregates event types into so-called *namespaces* rather than binding them to a particular entity. A namespace is declared as a class annotated with `@Namespace` that contains logging methods (each with a unique name; method overloading is still prohibited) corresponding to the event types that belong to the namespace.

The `@Namespace` annotation is not to be confused with an annotation of the same name used in the previous version. Here, it is a class-level annotation that serves as an indicator for the annotation processor, meaning this class needs to be generated a schema for. This way the classes are no longer required to be located in a specific package; the annotation is sufficient for the processor to be able to recognize them. For each namespace, the annotation processor generates a single JSON Schema containing all its event types, which an event conforms to if and only if it matches exactly one of the event type subschemas defined in the schema. If a namespace does not exist for a certain schema, it is generated as well.

```
{
  "$schema":"http://json-schema.org/schema#",
  "title":"NamespaceXY",
  "type":"object",
  "oneOf":[
    { "$ref":"#/definitions/eventA" },
    { "$ref":"#/definitions/eventB" }
  ],
  "definitions":{
    "eventA":{
      "properties":{
        "param1":{ "type":"string"  },
        "param2":{ "type":"integer" }
      },
      "required":["param1", "param2"],
      "additionalProperties":false
    },
    "eventB":{
      "properties":{
        "a":{ "type":"boolean" }
      },
      "required":["a"],
      "additionalProperties":false
    }
  }
}
```

Figure 5.6: Sample JSON Schema for `NamespaceXY` containing two methods

The rules for overwriting generated classes or schemas stay the same as before, i.e. the classes are privileged and are never influenced by the schemas once they exist. The location of schemas is not changed either - all of them can be found in package `entities` and its subpackages, mirroring the location of namespaces they correspond to.

Another type of user-declared classes in this model are entity classes. An entity class plays the role of a filter, restricting the usage of certain logging methods in combination with the entity it represents. If not stated otherwise, all methods can be used. To override the default behaviour, the entity class has to contain an `enum` for each namespace it wants to limit, listing the names of all allowed logging methods from this namespace. The `enum` must be annotated with `@SourceNamespace`, giving the path to the namespace it refers to as a parameter of the annotation. In case the annotation is not present, or its value does not denote any existing namespace, the `enum` is ignored. No JSON files are necessary to support entity classes; the checks for compatibility between entities and event logging methods are carried out at compile time by the annotation processor based on the information available from the classes.

5.2.2 API and Logging Process

As a base for explanation on how the logging request is formed, consider the following logging statement:

```
Logger.debug(EntityX.class, NamespaceXY.eventA("abc", 42));
```

It has already been briefly mentioned that in this version, there is a single `Logger` providing the necessary interaction with Log4j to ensure the data is logged. It contains just static methods for all severity levels distinguished by Log4j (in the example, `debug`). A combination of an entity and an event type is passed on to the method, supplying all data for the subsequent log record. In case the entity is not specified (i.e. the first parameter is `null`), a warning is issued by the processor stating that no log will be produced. The code compiles successfully nevertheless, and it is up to the developer to decide whether to fix the statement that does not output anything.

The structure of logging statements and the format of entity classes enable for detection of illegal entity-method pairs at compile time. The annotation processor itself is not capable of performing such code inspection, because it only has access to method signatures and not their bodies. However, with the assistance of the Compiler Tree API[8], it is possible to analyse code the way it is needed for our purposes. The Compiler Tree API provides access to the abstract syntax tree (AST) of the source code, constructed by the compiler, where all Java constructs such as classes, method declarations, method invocations, variable assignments etc. are represented as a hierarchy of subtrees. The AST can be scanned and examined using the visitor design pattern, particularly by extending `TreeScanner`, `TreePathScanner`, or

[8] http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/

SimpleTreeVisitor, or directly implementing the com.sun.source.tree.TreeVisitor interface.

We implemented a custom MethodInvocationScanner that inspected method invocations in the same form as our logging statements and gathered information about them for the annotation processor to use in subsequent checks. To be more precise, every method invocation with exactly two parameters, the second of them being a method invocation itself, is broken down to components (the names of the objects, methods, parameters, etc.) and sent to the annotation processor. The processor's task is then to determine whether it deals with a logging statement at all, and if it does, compare the logging method from the second parameter to those enumerated in the entity given as the first parameter. The statement is rejected if and only if the entity contains an enum annotated with @SourceNamespace of the namespace in question, such that it does not list the logging method as one of its values. In such case the processor reports an error specifying the combination in conflict and the compilation fails.

Unfortunately, this technique is not completely reliable mostly due to the difficulties in mining the AST. The annotation processor and AST scanners can not cover all possible situations, and there is no way to strictly enforce the logging statements to be in the abovementioned form. Consider a log request with the same meaning as before, but created this way:

```
String eventJson = NamespaceXY.eventA("abc", 42);
Logger.debug(EntityX.class, eventJson);
```

MethodInvocationScanner will simply ignore the statement, because its second parameter is not a method invocation. Even if the implementation of the scanner was changed to allow this, it would require extensive and very costly manual processing of the AST to logically connect the two lines and detect the logging method to look for in the entity class. Furthermore, this is not the only means of bypassing the compatibility checks. Therefore the restrictions placed on logging methods defined in entity classes act merely as a recommendation and cannot be absolutely relied upon to be enforced by the processor. It must be observed, however, that despite the known flaws this approach has, it is still applicable in common scenarios if it can count on the structure of the logging statements.

The log records look almost exactly the same as Figure 5.5 presented in the previous version, the only difference being the path to the JSON Schema. Since the schema represents a whole namespace, i.e. it consists of subschemas for the individual logging methods, the path to the file is not sufficient for identification. Therefore navigation inside the schema is required, resulting in a/b/c/NamespaceXY.json#/definitions/eventA for the log record produced by the log statement used above. Analogously to the previous version, an aspect is responsible for sending method and parameter names to the Logger so only the values of the parameters seem to be needed.

As the above implies, an advantage of this variant over the previous one lies in more flexible manipulation with event types. On the other hand, since they are not strictly bound to entities, virtually any combination of the two is possible, regardless of whether it is reasonable or not. The design of the API also makes the logging statements rather complicated.

5.3 Event Types Arbitrarily Combined With Entities

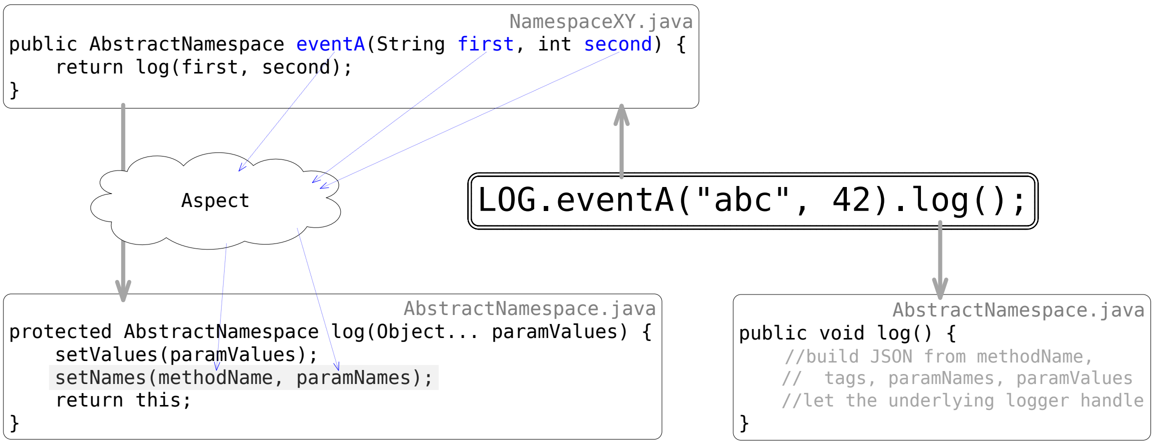
Trying to refine the specification of the logging component beyond the second variant, we identified that entities were not the ones being crucial in the logging process; it was primarily the event type that mattered. The logging module has been redesigned accordingly, focusing on the event type and treating entities just as an additional information.

Similarly to the second version, logging methods corresponding to event types are organized into namespaces, each of the namespaces backed up by a JSON Schema. The main difference between the two approaches lies in the logging process. Instead of a centralized logger, the namespaces themselves now act as loggers for a specific subset of event types.

Considering the fact that for each event type its context is determined by the namespace it belongs to, especially with conveniently designed namespaces it is often the case that no additional information is absolutely essential apart from the values of the event type attributes. Nevertheless, each event type can be assigned an arbitrary number of tags providing extra information, such as the entities it is related to. The question of compatibility between event types and entities has been dropped entirely in belief that developers, for their own sake, will not use unreasonable combinations; and in case an event type is absolutely forbidden to be used in association with a certain entity, the default behaviour of the logging methods can be extended with the necessary checks.

Abstracting from entities, this approach places namespaces at the core of the logging process. Defined in the same way as in the previous variant, namespaces are `@Namespace`-annotated classes that group logging methods together according to developer-defined principles. It is highly recommended that the namespaces extend class `AbstractNamespace` to ensure maximum reliability when applying components for automated processing, such as the pointcut of the aspect for extracting names of parameters or an annotation processor. Class `AbstractNamespace` encompasses data necessary for correct execution of the logging process, among the most important a reference to the logger implementation specific for the particular instance. A JSON Schema is maintained for each `@Namespace`-annotated class, exactly in the form shown in Figure 5.6. All conventions and requirements for the correspondence between JSON Schemas and namespaces stated in Section 5.2 hold for this version as well.

It should be observed that the processor detects classes to generate JSON Schemas from based on the `@Namespace` annotation rather than just working with all classes that extend `AbstractNamespace`. This is because semantically, the two are not completely interchangeable and can even be used separately to accomplish slightly different goals. For example, in case a particular subclass of `AbstractNamespace` is not annotated, it can exploit all advantages of structured logging, but no schema is generated for it. It might be useful in case a more sophisticated custom schema is already present and it is not desirable to have it overwritten by the processor; or if the class in question is a base namespace class containing



```
public AbstractNamespace eventA(String first, int second) {
    return log(first, second);
}
```

```
LOG.eventA("abc", 42).log();
```

```
protected AbstractNamespace log(Object... paramValues) {
    setValues(paramValues);
    setNames(methodName, paramNames);
    return this;
}
```

```
public void log() {
    //build JSON from methodName,
    //  tags, paramNames, paramValues
    //let the underlying logger handle
}
```

Figure 5.7: An illustration of the aspect's role in the logging process

common features designed to be further extended by namespaces. On the other hand, if the annotation alone is specified, it means the class will be generated an equivalent schema for, but for arbitrary reasons the functionality provided by `AbstractNamespace` is not needed.

In this variant, Log4j is no longer used as the default and only possible background logger. In projects using our logging component, at least one logging implementation (implementing method `log` in interface `Logger`) must be defined and set up in order to carry out the actual logging. It gives the developers adequate freedom to choose the logging technique or the storage appropriate for their logs (even different settings for various logging statements), and decouples the logging module from particular logging solutions and frameworks. On the other hand, moving the responsibility of establishing a logger to the using project we also lose the ability to explicitly define severity levels, since they differ among various logging frameworks. However, this is of no major concern to us since the severity methods were just an auxiliary and temporary solution. In our event-based representation of logs, each event type is implicitly assigned a severity based on its real-world meaning, e.g. event type `outOfMemory` is naturally more important than an informational message such as `userLoggedOn`.

An example usage:

```
NamespaceXY LOG = LoggerFactory.getLogger(NamespaceXY.class, new Log4jLogger());
LOG.eventA("abc", 42).tag("EntityX").tag("EntityZ").log();
```

`Log4jLogger` used in the example is a custom implementation of the `Logger` interface that delegates requests to Log4j. It can be seen that, in order to log, both the physical logger and a set of methods enforcing the structure of events (assembled in a namespace) must be present. It is further possible to add any number of tags (or none at all) to the logging statement, specifying e.g. entities concerned by the event. This way if an event applies to multiple entities, just one log record is created listing all of them, and no redundant copies are necessary. To indicate that the number of tags is definitive, method `log` has to be called last. Only this method actually initiates the logging; without it, the data is collected, but never output to the specified destination.

In case developers do not find this straightforward, minor modifications in the API can lead to logging statements where the confirmation method at the end is omitted for simplicity. By moving the tags to the front it is ensured that the output is always produced, as the logging logic can be incorporated directly in the logging methods:

```
LOG.tag("EntityX").tag("EntityZ").eventA("abc", 42);
```

Both approaches bring small benefits. In the first one, it is clarity; the logging method clearly stands out being the first one in the statement, so when reading multiple log lines with variable number of tags, the names of the logging methods are always aligned. In the second one, however, it suffices to call methods that carry logging data and no additional method calls need to be included, thus eliminating the possibility of misformulation of the logging statement.

Log records produced in any of the two ways contain fields specified by the settings of the particular underlying logger, and a message that looks analogous to:

```
{ "tags":["EntityX","EntityZ"],
  "schema":"a/b/c/NamespaceXY.json#/definitions/eventA",
  "properties":{
    "param1":"abc",
    "param2":42
  }
}
```

Figure 5.8: Sample log message

The last proposed approach provides the greatest flexibility, while maintaining the necessary requirements to perform structured logging. It is currently the preferred version. We believe it serves our purposes with no obvious flaws, solving all of the problems indicated before.

Chapter 6

Using the Proposed Structured Logging Mechanism

The main objective of this chapter is to provide instructions and guidelines for using the logging mechanism presented above. It focuses on the last mentioned alternative, which is believed to be readily usable with no significant disadvantages. To be able to log in a structured way using this mechanism, several requirements must be met; together with the conventions and assumptions, these are detailed in the following sections.

A fully configured sample project using the structured logging component can be found in the Ngmon Github repository[1].

6.1 Namespace Classes

As stated above, namespace classes play an indispensable role in the logging process. A namespace class is defined by the using project and contains a set of related logging methods. There are no restrictions as to what exactly the relationship between them means; that is left to the developer's judgement. It is recommended for a namespace class to both extend `AbstractNamespace`, which is a base namespace implementation provided by the logging component, and be annotated with `@Namespace` to indicate that a JSON Schema corresponding to the class should be generated.

The logging methods contained in a namespace class follow a general pattern that looks as follows:

```
public AbstractNamespace eventA(String param1, int param2) {
    return log(param1, param2);
}
```

Figure 6.1: Sample logging method

First, it should be noted that it is mandatory for the logging methods to return an object of type `AbstractNamespace` so that the logging statement using the particular method could be finished successfully. The API is designed in a way that requires the logging method to be called first, and then allows chaining multiple `tag` methods (thus adding an arbitrary number of tags to the log record), finishing the statement with a `log` method indicating that the log message is complete and no more tags will be added. Both these methods are declared in class `AbstractNamespace`. Since at some point, method `log` (also from

[1] https://github.com/ngmon/ngmon-json-logger

AbstractNamespace) is necessary to be called and passed the values of the attributes to be logged, the easiest way to accomplish both is to directly return the return value of log. Although the body of the logging method can be implemented differently, the default form generated from a JSON Schema in case the class does not exist is the one shown in Figure 6.1.

By convention, the name of the logging method is incorporated into the resulting JSON Schema for the enclosing namespace class. The method name is regarded as the name of the event type, therefore a reasonably descriptive one should be chosen.

There are two ways to obtain the namespace class containing the logging methods: it can either be directly implemented, or a JSON Schema with the corresponding name can be included in the project to generate the namespace class from; the choice depends solely on the developer's preferences. However, as mentioned in Chapter 5, a class will only be generated from a schema if it does not exist; no further changes to the schema are reflected in the associated class after its creation. Also, in case the JSON Schemas are created first and the namespace classes are left for the annotation processor to generate, it must be taken into consideration that the annotation processor is essentially a compiler plugin and works with a given set of source files; so unless there is at least one existing Java class in the project to be compiled, the annotation processor will not be launched and the namespace classes will not be generated.

6.2 JSON Schemas

Regarding the correspondence between the namespace class and its JSON Schema, it should be remarked that due to differences in the expression mechanism there are only a few items that can be unambiguously translated from one to the other, including the name of the event type (i.e., the name of the logging method) and the names and types of its attributes. These, and these only, are translated into the logging methods, regardless of the additional characteristics that may be included in the schema.

Moreover, as soon as the namespace class containing the logging methods exists, the schema has no longer any effect on it and is itself overwritten in case of changes in the class. Therefore, if it is desirable for additional constraints contained in the schema to be preserved, the only workaround involves removing the @Namespace annotation from the namespace class, and thus preventing the annotation processor from recognizing the class and overwriting the contents of the schema based on it. However, this may lead to inconsistencies and is not recommended unless it is certain that the events produced by the logging methods will conform to the customized schema. So far, JSON Schemas consisting just of event types and the names and types of their attributes have been identified as sufficient for our purposes; in case a need for other characteristics arises, a means of translating them to the corresponding logging method will have to be devised.

When generating schemas, the annotation processor adheres to the draft04 of JSON Schema [17], and all schemas created manually should also be compatible with this version of the specification. Due to their simple contents, there is little evidence that the generated schemas encompass new features introduced in draft04; nevertheless, they contain the newly supported keyword oneOf, so their draft04-compliance should be taken into account in case of validating against them.

All schemas are expected to be located in package events and its subpackages in the standard Maven resources directory (src/main/resources). As it was already stated, the package hierarchy and the exact location of a schema within the events package depends on the location of its corresponding class. Since no specific file extension for a JSON Schema file has been established, it is assumed for the files to use the .json extension.

6.3 Logging

In order to perform the actual logging, an instance of a structured logger must be requested from the LoggerFactory. Since the namespace classes serve directly as an alternative to loggers, a sample instantiation may look as follows:

```
NamespaceXY LOG = LoggerFactory.getLogger(NamespaceXY.class, new Log4jLogger());
```

The LoggerFactory must be provided with both the namespace, representing the structured part of the logging process, and a logger implementation (in the example above it is the Log4jLogger) that takes care of outputting the log records to a destined storage.

Although in the early versions of our logging component Log4j was used as a fixed background logger (see Section 5.1 and Section 5.2), the last proposed variant described in Section 5.3 only serves as an adapter for producing structured logging data and leaves the recording of the logs itself to a custom-defined logger.

There are no restrictions concerning the logger; it is possible to use an arbitrary logging framework. The custom logger (Log4jLogger) is only responsible for delegating the structured output to the underlying logging framework (i.e. Log4j in this case). It needs to implement interface Logger, which entails overriding method log having one String parameter so as to be able to receive the structured logged event from the namespace logger. After performing the necessary steps to obtain the JSON representation of a log record, the structured logger (an instance of NamespaceXY in the example above) calls log on the particular instance of custom logger associated with it, expecting that it in turn takes care of forwarding the log record to the underlying logging framework.

6.4 Build Process

A standard Maven project structure is assumed; to be more precise, all sources are expected to be located under src/main/java, and only resources collected in src/main/resources will be taken into consideration. A dependency on the particular version of the structured logging component should be declared in pom.xml.

To ensure that the annotation processor will put the generated namespace classes in the correct directory, a <generatedSourcesDirectory> of the maven-compiler-plugin (or an alternative, such as specifying the corresponding option in an IDE) must be set to `src/main/java`. This is because by default, annotation processors only put generated sources to the project output directory and so they are not readily accessible from the code. No output directory for generated JSON Schemas needs to be set, since in this case the process of determining the output location is controlled by the annotation processor.

It must also be made sure that the aspect taking care of retrieving additional parameters of the logging record is properly weaved into the code. Since the implementation of the aspect is located outside the using project, when building it, it should be explicitly stated that an external library of aspects is to be used. This can be achieved for example using the `aspectj-maven-plugin` and including the structured logging component in <`aspectLibraries`> in addition to declaring a dependency on it. For the aspect to be able to find the name of the logging method and the names of its parameters (see Section 5.1.2 for details), the compiler must be invoked with the option for generating debugging information enabled.

Chapter 7

Performance Evaluation

In this chapter, we evaluate the performance of our logging component based on the tests that were run, and compare it with the performance of traditional approaches to logging. The objective of the evaluation is twofold: to prove that the proposed logging mechanism does not impose any significant overhead in terms of performance during the production of structured logs, and to demonstrate the benefits it brings when it comes to processing the logs. To carry out the actual measurements, Google's micro-benchmarking framework called Caliper[1] was used, as it seemed well-suited for our purposes and very straightforward to use. The tests were run on a PC with Intel Core i5-2140M @ 2.30 GHz dual-core processor and 4 GB RAM, running the 64-bit version of Windows 7 and Java SDK 1.7.0.

The hypotheses we approached the measurements with were formulated as follows: (1) the cost necessary for constructing the JSON representation of an event does not have a considerable effect on the time it takes to execute a single logging statement; (2) contrary to parsing the logs in natural language using regular expressions, processing events in the form of JSON objects does not depend on the variability in event types present in the set of logs; so for a sufficient number of different event types, the former is much more costly.

7.1 Logging Performance

First, we wanted to measure the time it took to log a single structured event. Each of the three variants mentioned in Chapter 5 was tested, as well as an ordinary Log4j logger that served as a baseline for the comparison. In each case we had the logger output semantically the same structure, only represented in the means characteristic for the particular variant (see Figure 7.1). An underlying Log4j logger was used to perform the actual logging; and for testing purposes, it was configured to output the logs to a text file.

To be more specific, all loggers were set to log a JSON event similar to the one in Figure 5.5, i.e. having two parameters and being related to one entity; in case of a pure Log4j logging without any intervention from our component, the resulting JSON event was simply included as a string in the place of a log message. Since the cost of the logging statement might depend on the number of tags present in the third variant, two alternatives were measured, namely containing one tag and ten tags. Ten tags were estimated as a reasonable maximum one would use for a single statement.

[1] http://code.google.com/p/caliper/

```
(1) Logger LOG = LogManager.getLogger(this);
    LOG.debug({"tags":["EntityX"],
               "schema":"a/b/c/NamespaceXY.json#/definitions/eventA",
               "properties":{
                 "param1":"abc",
                 "param2":42
              }});
(2) EntityX entityX = new EntityX();
    entityX.eventA("abc", 42).error();
(3) Logger.error(EntityX.class, NamespaceXY.eventA("abc", 42));
(4) NamespaceXY LOG = LoggerFactory.getLogger(NamespaceXY.class, new Log4jLogger());
    LOG.eventA("abc", 42).tag("EntityX").error();
```

Figure 7.1: Testing log statements

Caliper itself takes care of an adequate warming up before the tests to stabilize the influence of Java garbage collector and Just-In-Time compiler. It also runs each test several times, deciding on the exact number of repetitions online, and only outputs the average over the particular number of repetitions. These average run times were further averaged over 10 invocations of Caliper for each of the variants to provide more reliable results. Table 7.1 shows the running times of the measurements aiming to support the first hypothesis. It lists the shortest and the longest time out of the 10 invocations of each benchmark, as well as the average over these 10 runs.

Five different scenarios for producing the logs were observed, namely: (1) executing a common Log4j logging statement; (2) using an entity class as the logger, as explained in Section 5.1; (3) specifying an entity together with the logged event in accordance with the variant described in Section 5.2; (4) logging by means of a so-called namespace logger (see Section 5.3) and tagging the logged event with the entity in question; and (5) the same, but tagging the logged event with ten entities. Figure 7.1 shows the pieces of code corresponding to scenarios (1) to (4); (5) is omitted for brevity, as it can be easily obtained from (4) adding the appropriate number of tags. The columns in Table 7.1 are also numbered according to this list.

Scenario	**1**	**2**	**3**	**4**	**5**
Best time	3.87	6.40	10.10	5.780	7.21
Worst time	5.32	11.10	11.10	6.200	11.20
Average time	4.26	7.88	10.72	6.057	7.79

Table 7.1: Log production time [μs]

As it can be seen in Table 7.1, unstructured logging using Log4j was the fastest of the scenarios measured. However, the difference between scenario (4), i.e. the last and most relevant variant proposed in this thesis, and logging using Log4j only amounts to around 2 μs per logging statement; so it can be stated that the logging performance overhead of this variant is reasonably small and the overall performance of the application will not suffer unless the number of logs produced per time unit is very large. Nevertheless, even then the inconvenience caused by a slightly more time-consuming production of structured logs is compensated by the advantages of processing and querying them.

Table 7.1 further shows that scenario (3) was the worst performing one, with times necessary to log a single statement slightly more than twice as long as Log4j. It also proves that the number of tags a logging statement is labelled with (see scenarios (4) and (5) for one and ten tags, respectively) indeed has an effect on the performance, although even as many as ten tags do not pose a major obstacle.

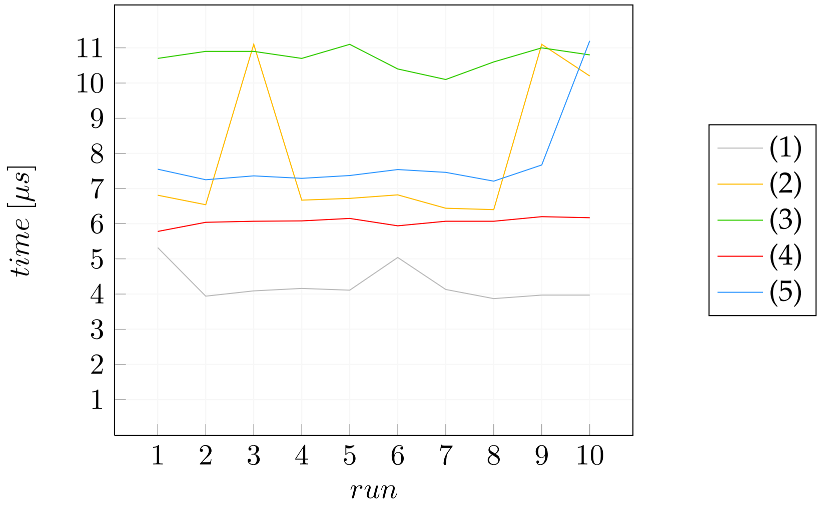


Figure 7.2: Logging performance

7.2 Processing Performance

In the second experiment we observed how the variability present in logs influenced the time necessary to process them. Two approaches were compared: parsing natural language log messages using regular expressions, and querying structured log messages.

The experiment was set up as follows: two sets of testing log records were generated; one of them contained logs in natural language with each log record similar to Figure 7.3, while the other imitated the output of a structured logger carrying the same meaning, such as the one shown in Figure 7.4.

```
2013-04-01 10:00:00,000 ERROR cz.muni.fi.sampleproject.Log4jLogger - Unable to connect
to host HostX1 as user UserX1. 1 attempt(s) left#1
```

Figure 7.3: Natural language log record

Both log record patterns described a simple event with three attributes (i.e. three variable parts contained within the fixed part of the natural language log message). To mimic a real logging output, one of the most common layouts for a log record was used, listing the date and time of the event occurrence, its severity level, and the name of the class that produced the event in addition to the log message itself.

```
2013-04-01 10:00:00,000 ERROR cz.muni.fi.sampleproject.Log4jLogger -
{ "pathToSchema":"a/b/c/NamespaceXY.json#/definitions/UNABLE_TO_CONNECT_HOST_AS_USER_1",
  "tags":["org.apache.hadoop"],
  "properties": {
    "host":"HostX1",
    "user":"UserX1",
    "attemptsLeft":1
} }
```

Figure 7.4: Structured log record

In order to simulate diverse event types, each log message was made unique by appending a number to its end (this is highlighted in blue in Figure 7.3 and Figure 7.4). In other words, we wanted to observe the worst-case scenario where no two log records belonged to the same event type. For each event type, i.e. for each natural language log message in our case, a regular expression was generated such that it only matched log records of that particular type. Figure 7.5 lists the regular expression that was generated to correspond to the natural language log record stated above (see Figure 7.3).

```
^.*? - Unable to connect to host (.*?) as user (.*?)\. ([0-9]++) attempt\(s\) left#1$
```

Figure 7.5: Regular expression matching the log in Figure 7.3

The three testing sets of mutually distinct natural language logs, regular expressions corresponding to them, and structured logs, were used in subsequent benchmarks. Two testing scenarios were designed:

1. Processing structured log messages in the form of JSON objects. For testing purposes, the value of attribute `attemptsLeft` was extracted, thus demonstrating the possibility to process structured logs in the proposed format efficiently. Each log message was passed to Jackson's[2] `ObjectMapper`, which returned a Java object representation of the JSON structure that could be simply queried using a `get` method.
2. Parsing natural language log messages using regular expressions. Since each log message in the set of natural language logs was unique by design, finding the one regular expression that matched it required a search through all available regular expressions (in the worst case). Processing all the logs in a particular set therefore had a time complexity of $O(n^2)$, where n was the number of regular expressions. When the applicable regular expression was identified, it was used to parse the desired value, which was again the number of attempts left, out of the log message.

[2] Jackson JSON Processor, http://wiki.fasterxml.com/JacksonHome

Since the time necessary to process the natural language logs was expected to depend on their number, several sets of testing logs were generated, comprising of 100 to 1500 unique log records. Table 7.2 indicates the processing times for selected sets of logs. Again, Caliper was run 10 times and the table only shows the average of the 10 measured values (the values are in milliseconds).

Number of event types	**100**	**200**	**300**	**500**	**1000**	**1500**
JSON event object	0.353	0.658	0.959	1.596	3.216	4.580
Natural language log	34.90	138.00	305.00	836.20	3271.00	7499.00

Table 7.2: Log processing time [ms]

It can be observed that the time required for processing the whole set of log messages using regular expressions grows rapidly with the number of different event types it contains. For more than 500 of them, the processing time already gets to seconds. The growth rate can be seen more clearly in Figure 7.6.

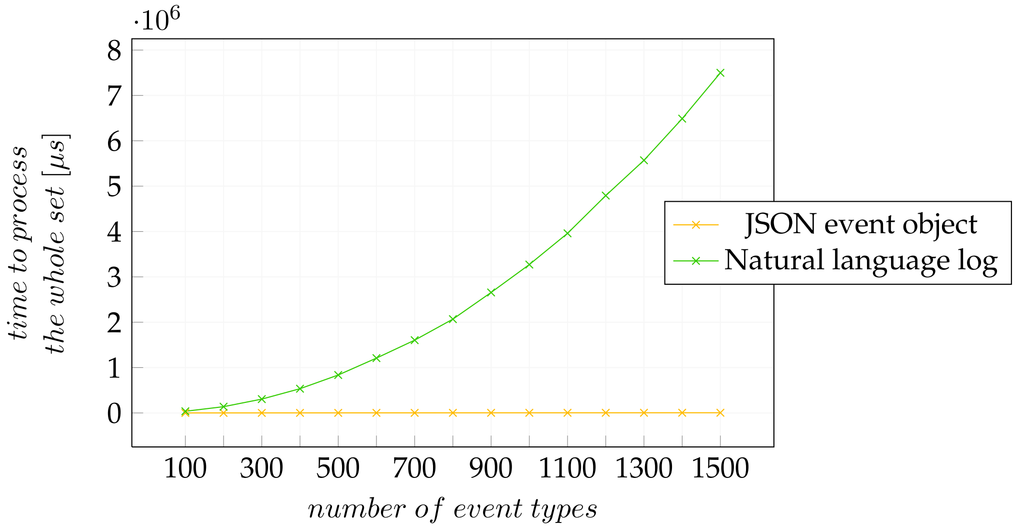


Figure 7.6: Log processing performance

The measurements confirm that with the increasing number of different log records, the processing time grows linearly for our approach, while the scenario utilizing regular expressions exhibits a quadratic growth. This was an expected behaviour. Due to the fact that the structured logs are processed uniformly regardless of their type, the time to process one of them can be considered constant. On the other hand, to be able to parse a natural language log using a regular expression, the latter has to be found first; and the time necessary to retrieve the right regular expression depends on how many of them are there to be searched. Figure 7.7 shows average times for processing a single log record in both scenarios.

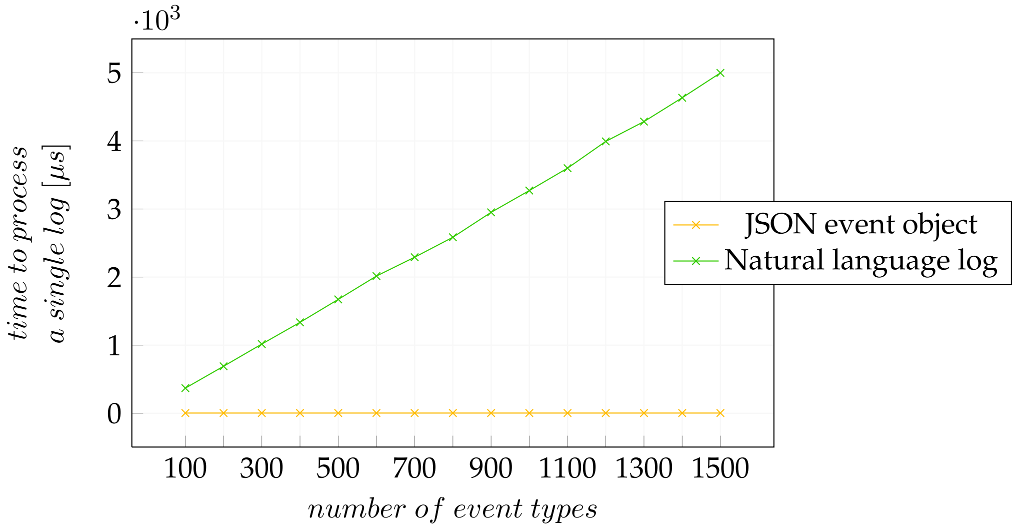


Figure 7.7: Log processing performance

7.3 Summary

From the two experiments described above, it stems that although producing structured logs using the proposed logging mechanism introduces a minor overhead, the disadvantage is outweighed by the advantages it brings when processing the logs. To search and parse the logs using regular expressions is expensive in terms of time, and in addition, it requires for the right regular expressions to be collected first for each type of log to be processed. Contrary to this, handling structured logs in the form of JSON event objects does not depend on the variability in the set of logs, for all of them can be queried in a uniform way independently of their event type.

Chapter 8

Conclusion

The main objective of this thesis was to devise a novel approach to logging, which would avoid using a natural language representation of the logged event. The necessity to provide such mechanism was motivated by the challenges logging (and monitoring in general) faces, particularly the need for a unified representation of monitoring information in order to allow an efficient automated processing, correlation and querying.

A logging component facilitating structured logging from Java applications was designed and implemented. It produces the logs in the form of JSON objects that can be efficiently parsed to obtain the information of interest. Each event type the application produces is unambiguously described by a JSON Schema, and every log record the logging component outputs conforms to exactly one such schema, uniquely identified in the log message itself. Thus the logs in JSON satisfy all the requirements for structured monitoring information; not only are they straightforward to produce, they are also easy to process.

Since the beginning of the work on this thesis, the logging component has evolved through two different stages to the current form, which represents the third, and in our opinion the best variant. Each of the two former variants introduced valuable ideas and a slightly different point of view at the logged events, but also had indisputable downsides; it was the intention to eliminate the flaws that led to constant improvement of the component, resulting in the final version. All three variants were described in detail in previous chapters so that they could be adequately compared and the background for certain design decisions argued more clearly.

It was also important to carry out measurements so as to prove that the introduced logging mechanism did not have unacceptable requirements in terms of performance. The experiments were targeted at observing the production of the logs, and also at the process of retrieving information from them. All three variants were considered, as well as an approach using a standard Log4j logger to log a manually constructed JSON event. The results showed that although the Log4j logger was the fastest of the four (which was only natural because there was no need to perform the operations necessary for constructing the JSON output), the last of our three versions got reasonably close in performance. Therefore, a conclusion was drawn that our logging component only had a negligible impact on the overall logging performance. Moreover, another experiment showed that when querying the logs, the structured variant was an outright winner. Contrary to parsing a set of free-form logs using regular expressions, which depends on the length of log records and the number of different types of them, obtaining the desired piece of information from our JSON logs required a simple method call, independent of the set of logs to process or the variability present therein.

Overall, it can be concluded that the thesis has achieved its primary goal. It proposed a logging component capable of producing structured logs represented by JSON objects, each of them described by an automatically generated and updated JSON Schema. The performance of logging in such way is comparable to that of standard logging frameworks, and as for processing, our approach aims for handling the logs in a way that outperforms parsing logs using manually created regular expressions significantly.

Bibliography

[1] ACCORSI, R. On the Relationship of Privacy and Secure Remote Logging in Dynamic Systems. In *Security and privacy in dynamic environments*. Springer, 2006, pp. 329–339.

[2] ACETO, G., BOTTA, A., DE DONATO, W., AND PESCAPÈ, A. Cloud Monitoring: a Survey. *Computer Networks* (2013).

[3] ADINOLFI, O., CRISTALDI, R., COPPOLINO, L., AND ROMANO, L. QoS-MONaaS: A Portable Architecture for QoS Monitoring in the Cloud. In *Signal Image Technology and Internet Based Systems (SITIS), 2012 Eighth International Conference on* (2012), IEEE, pp. 527–532.

[4] THE APACHE SOFTWARE FOUNDATION. *Apache Log4j 2 v. 2.0-beta4: User's Guide*, 2013-01-28. Retrieved March 3, 2013 from http://logging.apache.org/log4j/2.x/log4j-users-guide.pdf.

[5] ARCSIGHT, I. Common Event Format, Revision 15, 2009-07-17. Retrieved March 5, 2013 from http://mita-tac.wikispaces.com/file/view/CEF+White+Paper+071709.pdf.

[6] BRUNETTE, G., MOGULL, R., ET AL. Security Guidance for Critical Areas of Focus in Cloud Computing v3.0. *Cloud Security Alliance* (2009), pp. 1–76.

[7] THE CEE EDITORIAL BOARD. *Common Event Expression: Architecture Overview, Version 0.5*, May 2010. Retrieved March 5, 2013 from http://cee.mitre.org/docs/CEE_Architecture_Overview-v0.5.pdf.

[8] CHUVAKIN, A. Application Logging "Worst Practices", 2008-10-09. Retrieved March 5, 2013 from http://www.slideshare.net/anton_chuvakin/application-logging-good-bad-ugly-beautiful-presentation.

[9] CHUVAKIN, A., AND PETERSON, G. How to Do Application Logging Right. *Security & Privacy, IEEE 8*, 4 (2010), pp. 82–85.

[10] CHUVAKIN, A., SCHMIDT, K., AND PHILLIPS, C. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management.* Syngress, 2012.

[11] CLAYMAN, S., GALIS, A., AND MAMATAS, L. Monitoring Virtual Networks with Lattice. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP* (2010), IEEE, pp. 239–246.

[12] DE CHAVES, S. A., URIARTE, R. B., AND WESTPHALL, C. B. Toward an Architecture for Monitoring Private Clouds. *Communications Magazine, IEEE 49*, 12 (2011), pp. 130–137.

[13] ETZION, O., AND NIBLETT, P. *Event Processing in Action*. Manning Publications Co., 2010.

[14] FASTERXML, LLC. *Efficient JSON-compatible binary format: "Smile"*, 2013-05-12. Retrieved May 15, 2013 from http://wiki.fasterxml.com/SmileFormatSpec.

[15] FOSTER, I., ZHAO, Y., RAICU, I., AND LU, S. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE'08* (2008), IEEE, pp. 1–10.

[16] FOX, A., GRIFFITH, R., JOSEPH, A., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., AND STOICA, I. Above the Clouds: A Berkeley View of Cloud Computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28* (2009).

[17] GALIEGUE, F., AND ZYP, K. JSON Schema: core definitions and terminology, 2013-01-31. Retrieved May 01, 2013 from http://tools.ietf.org/html/draft-zyp-json-schema-04.

[18] GONG, C., LIU, J., ZHANG, Q., CHEN, H., AND GONG, Z. The Characteristics of Cloud Computing. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on* (2010), IEEE, pp. 275–279.

[19] HASSELMEYER, P., AND D'HEUREUSE, N. Towards Holistic Multi-Tenant Monitoring for Virtual Data Centers. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP* (2010), IEEE, pp. 350–356.

[20] HOFFNER, Y. *Monitoring in Distributed Systems*. Citeseer, 1993.

[21] KENT, K., AND SOUPPAYA, M. Guide to Computer Security Log Management. *NIST special publication 800–92* (September 2006).

[22] MAKANJU, A. A., ZINCIR-HEYWOOD, A. N., AND MILIOS, E. E. Clustering Event Logs Using Iterative Partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (2009), ACM, pp. 1255–1264.

[23] MANSOURI-SAMANI, M. *Monitoring of Distributed Systems*. PhD thesis, University of London, December 1995.

[24] MANSOURI-SAMANI, M., AND SLOMAN, M. Monitoring Distributed Systems. *Network, IEEE 7*, 6 (1993), pp. 20–30.

[25] MARTY, R. Cloud Application Logging for Forensics. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (2011), ACM, pp. 178–184.

[26] MASSIE, M. L., CHUN, B. N., AND CULLER, D. E. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing 30*, 7 (2004), pp. 817–840.

[27] MELL, P., AND GRANCE, T. The NIST Definition of Cloud Computing. *NIST special publication* (2011).

[28] MENG, S., AND LIU, L. Enhanced Monitoring-as-a-Service for Effective Cloud Management. *IEEE Transactions on Computers* (2012).

[29] THE MITRE CORPORATION. *CEE Log Syntax (CLS) Specification 1.0-beta1*, last updated 2012-08-09. Retrieved March 5, 2013 from http://cee.mitre.org/language/1.0-beta1/cls.html.

[30] NAGAPPAN, M., AND VOUK, M. A. Abstracting Log Lines to Log Event Types for Mining Software System Logs. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on* (2010), IEEE, pp. 114–117.

[31] NEWMAN, H. B., LEGRAND, I. C., GALVEZ, P., VOICU, R., AND CIRSTOIU, C. MonALISA: A Distributed Monitoring Service Architecture. *arXiv preprint cs/0306096* (2003).

[32] OGLE, D., KREGER, H., SALAHSHOUR, A., CORNPROPST, J., LABADIE, E., CHESSELL, M., HORN, B., GERKEN, J., SCHOECH, J., AND WAMBOLDT, M. Canonical Situation Data Format: The Common Base Event V1.0.1, 2003-11-04. Retrieved March 5, 2013 from http://www.eclipse.org/tptp/platform/documents/resources/cbe101spec/CommonBaseEvent_SituationData_V1.0.1.pdf.

[33] OLIVEROS, E., CUCINOTTA, T., PHILLIPS, S. C., YANG, X., MIDDLETON, S., AND VOITH, T. Monitoring and Metering in the Cloud. *Achieving Real-Time in Distributed Computing: From Grids to Clouds* (2011).

[34] PENG, W., LI, T., AND MA, S. Mining Logs Files for Data-Driven System Management. *ACM SIGKDD Explorations Newsletter 7*, 1 (2005), pp. 44–51.

[35] SMIT, M., SIMMONS, B., AND LITOIU, M. Distributed, Application-level Monitoring for Heterogeneous Clouds using Stream Processing. *Future Generation Computer Systems* (2013).

[36] SOSINSKY, B. *Cloud Computing Bible*, vol. 762. Wiley, 2010.

[37] SPRING, J. Monitoring Cloud Computing by Layer, Part 1. *Security & Privacy, IEEE 9*, 2 (2011), pp. 66–68.

[38] SPRING, J. Monitoring Cloud Computing by Layer, Part 2. *Security & Privacy, IEEE 9*, 3 (2011), pp. 52–55.

[39] TIERNEY, B., AYDT, R., GUNTER, D., SMITH, W., SWANY, M., TAYLOR, V., AND WOLSKI, R. A Grid Monitoring Architecture, 2002.

[40] TOVARŇÁK, D., AND PITNER, T. Towards Multi-Tenant and Interoperable Monitoring of Virtual Machines in Cloud. In *Proceedings of 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2012), pp. 436–442.

[41] VAARANDI, R. A Data Clustering Algorithm for Mining Patterns From Event Logs. In *IP Operations and Management, 2003.(IPOM 2003). 3rd IEEE Workshop on* (2003), IEEE, pp. 119–126.

[42] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 117–132.

[43] ZANIKOLAS, S., AND SAKELLARIOU, R. A taxonomy of grid monitoring systems. *Future Generation Computer Systems 21*, 1 (2005), pp. 163–188.

[44] ZAWOAD, S., DUTTA, A. K., AND HASAN, R. SecLaaS: Secure Logging-as-a-Service for Cloud Forensics. *arXiv preprint arXiv:1302.6267* (2013).