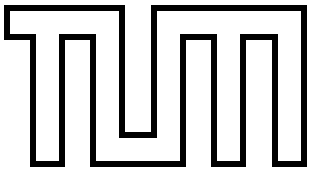


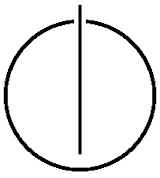
FAKULTÄT FÜR INFORMATIK

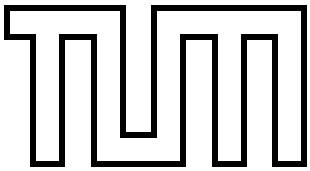
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Application Performance Monitoring of a scalable distributed Java web-application in a cloud infrastructure

Michael Rose





FAKULTÄT FÜR INFORMATIK

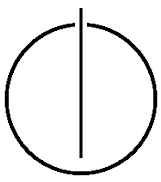
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Application Performance Monitoring of a scalable distributed Java web-application in a cloud infrastructure

Application Performance Monitoring einer skalierbaren verteilten Java Web-Anwendung in einer Cloud-Infrastruktur

Author:	Michael Rose
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Alexander Schneider, Dr. Thomas Büchner
Date:	August 15, 2013



I assure the single handed composition of this bachelor's thesis only supported by declared resources.

München, den Michael Rose

Abstract

Application Performance Monitoring (APM) is an essential aspect of software engineering and development. Inevitable for finding performance bottlenecks in an application or monitoring its current state while it is running in production, performance monitoring is often underestimated. However, monitoring CPU and memory consumption, file system and network traffic is not always sufficient. Application-level data is needed to be able to detect problems and solve them fast.

In this thesis we investigate how *APM* can be used for and integrated in an existing Java web-application, namely *Tricia* by *infoAsset*. The main focus hereby lies on covering the specific requirements posed by the application being scalable and deployed in a cloud infrastructure. We derive a variety of different but essential metrics for *Tricia* which can mostly be transferred to general Java web-applications. Furthermore, the required means of evaluating the gathered information are proposed as well as how they can be applied. We then take a look at already existing solutions for employing *APM*, including the examination of existing standards. Before advancing to the practical part, we conduct an interview with an employee of a large company who is experienced in *APM*. Afterwards, a complete *APM* solution is developed for *Tricia*. Key design and architectural decisions are presented in addition to important implementation details. Furthermore, we evaluate the solution in cooperation with *infoAsset* in the context of a real problem.

Contents

Outline of the Thesis

Part I: Background and Analysis

CHAPTER 1: INTRODUCTION

This chapter is an overview of the theoretical background. It defines the term *monitoring* in general and contains a description of *Tricia* which is used in the practical part of this thesis. Additionally, *scalability* and *distributed software* are explained.

CHAPTER 2: PROBLEM STATEMENT

We take a look at the given problem statement to see what we are trying to solve. An overview of the questions posed by the topic of this thesis is presented. Afterwards, the context in which the problem will be solved is outlined. Finally, the involved stakeholders are stated in combination with their respective motivation.

CHAPTER 3: ANALYZING THE PROBLEM

First, the key metrics to be observed by *Application Performance Monitoring* are presented in this chapter. Second, we describe the means of evaluation used to help draw conclusions from the gathered data. The end of this chapter sums up the results and contains a proposal on how the discovered means can be used together.

CHAPTER 4: RELEVANT TOOLS AND STANDARDS

Before advancing to the practical part of this thesis it is necessary to examine existing solutions. The key question to be answered is what advantages and drawbacks they have. This is required to later decide which of them can be reused when integrating *APM* in *Tricia*. The same procedure is performed for standards.

CHAPTER 5: INTERVIEW

The last chapter in the theoretical part is an interview conducted with an employee of a large company. He is experienced in *APM* with deployed applications. From his experience we derive some important aspects used in the practical part.

Part II: Employing APM in Tricia

CHAPTER 6: DESIGN DECISIONS

The first part of this chapter consolidates the current situation in *Tricia*. Based on this and the prior analysis, the key design decisions are made. This also includes selecting the frameworks and standard to use.

CHAPTER 7: ARCHITECTURE

After all the decisions are made we construct the overall architecture of the solution to be developed. This includes a separate monitoring tool as well as the adaptation of *Tricia*.

Part I.

Background and Analysis

1. Introduction

At the beginning of the thesis, important terms concerning the domain of *Application Performance Monitoring* are explained to provide the reader with information needed to fully understand the remaining parts. At first, the area of software monitoring is presented in general, before examining to the concrete meaning of *Application Performance Monitoring*. Afterwards, the context of the thesis as stated in the title — a scalable Java web-application in a cloud infrastructure — is dissected and detailed.

1.1. Monitoring

At first it is necessary to explain what monitoring means. One possible definition for *monitoring* is as follows:

> *"Monitoring is the process of maintaining surveillance over the existence and magnitude of state change and data flow in a system."* [3]

It is therefore a continuous task of checking specific properties of a system. Another important aspect is that monitoring is done in multiple ways [17]. It is always used in a non-intrusive manner, i.e. without affecting the monitored system. Monitoring is usually employed proactively — before any problems are reported. On the other hand there is the reactive situation — if a customer reports a problem and only subsequently monitoring is started. Furthermore, monitoring is primarily employed in production environments [17].

1.1.1. Types of Monitoring

Concerning software and the machines running it, monitoring can be done at a variety of levels. These levels are covered by distinct types of monitoring: [32]

- Availability Monitoring
- System Monitoring
- Performance Monitoring
- Server Monitoring
- Application / Transaction Monitoring
- End-User Monitoring

Performing *Availability Monitoring* means to ensure that systems, components, or single applications are running *at all*. It is the most basic form of monitoring and primarily used to ensure a working infrastructure. An example would be to check regularly if a server on the network can be reached. *System monitoring* refers to the lowest level dealing with hardware. Here, the availability of components and system resources is overseen and the amount of resources used is measured as well. The goal is to get an insight in the current state and behavior of the underlying machine. Some relevant metrics concerning system monitoring are *CPU load*, *network bandwidth* or *memory consumption*. *Performance Monitoring* is employed to observe the performance of a system. The relevant information in this case is how long specific tasks in the software take to be completed. The gathered measurements are often compared to predefined limits which have to be reached in order to fulfill certain requirements. An example of this might be the response time of a web-application, which the customer requires to be below $500ms$. *Server Monitoring* in turn handles the surveillance of an application server like *Tomcat*. These usually provide internal metrics themselves via *JMX* for example. The information collected in this case is relevant to the operator of the application server. *Application* or *Transaction monitoring* is done for one specific application. It is used to track a single request (*transaction*) while it is being processed. This way, very application-dependent metrics are gathered to identify problems in the software. The last one is *End-User monitoring*. Its purpose is to assure that all functionality is available to the user appropriately. In most cases this is handled by pre-recorded scripts simulating user interaction. One example for that type of monitoring is also tracking the overall page load time over the internet.

1.1.2. Application Performance Monitoring

Knowing what monitoring in general means, we now have to explain the essence of *Application Performance Monitoring* (*APM*). As its name implies it is a combination of *performance* and *application* monitoring. The purpose of *APM* is to monitor an application's performance employing application specific metrics [40]. This means that the application itself is adjusted to provide values suitable for expressing its state. Leveraging these values, it is then possible to see how an application performs or detect problems. If problems are discovered, *APM* then helps to identify them and finally solve the issues.

1.1.3. Distinction from Profiling

Monitoring has to be differentiated from *profiling* [40, 17]. Profiling describes the same process as monitoring, i.e. observing a system's state and data flow, but done in an intrusive way [17]. Therefore, the responsiveness of an application might decline or response times can increase drastically. Another difference is the environment profiling is used in. Typically, profiling is not employed in a production but a development environment [40, 17]. It is an action taken in order to check and optimize an application's behavior. Very often it is the next step after monitoring has revealed a problem [17]. The purpose of profiling in that situation is to find the root cause of the problem.

1.2. Tricia

The practical work of this thesis — employing *Application Performance Monitoring* in an existing Java web-application — will be done with *Tricia*, developed by *infoAsset AG*. We therefore give a description of what it is used for and then present an overview of how it can be deployed in different environments in combination with relevant architectural aspects. These are especially important for determining the scope of the solution to be developed in this paper as well as are reasons for oncoming decisions.

1.2.1. Overview of Tricia

Tricia is a web-based solution for collaborative project and information management as well as team work, suitable for companies from small to large sizes. It serves as a single point for storing various kinds of data, e.g. text content, contacts or even files, and gives the using team the opportunity to easily structure the information. *Tricia* also offers a powerful search to find what is needed fast and without much effort. This way the team's overall knowledge is made available to single members quickly in order to benefit from each others experience.

Furthermore, *Tricia* is adaptable to different scenarios like *Customer Relationship Management (CRM)* or *Enterprise Architecture Management (EAM)* [33]. It is, however, not limited to those concrete use cases. Due to its underlying *hybrid wiki* structure, customers can modify the information model and behavior to be compatible with their own information and organization structure or processes.

While it is developed by *infoAsset AG* since 2008, *Tricia*'s core functionality is based on a lot of research and evaluation in cooperation with TU München. Additionally, the experience gained from former customer projects with *Tricia*'s predecessor *infoAsset Broker* was an important groundwork to elaborate on.

1.2.2. Modes of Deployment

To suit the customer's needs, *Tricia*'s current architecture allows it to be deployed in different environments. One scenario is the deployment on a single server. This situation is represented in Figure 1.1.

For information storage, *Tricia* relies on a database server which is running on the same or potentially another machine. Apart from the relational database containing every information entered, there is an *Elasticsearch* search engine running. *Elasticsearch* is a very flexible open source search and analytics engine also used to provide the powerful search capabilities of *Tricia*. The key feature of this engine is its scalability — it can be configured to run in a *cluster*, i.e. the search index can be distributed across different machines. In the standalone scenario on a single server, however, *Tricia* does not use this setup but starts up its own internal *Elasticsearch* node without cluster configuration. Users can then use their browsers to connect directly to a running *Tricia*.

The other main type of deployment is inside a cloud environment. One practical example is the *Business Marketplace* of *Deutsche Telekom AG (DTAG)*[1]. Figure 1.2 shows how such a setup typically looks like. The main aspect is that there is not only one but many

[1] http://businessmarketplace.de

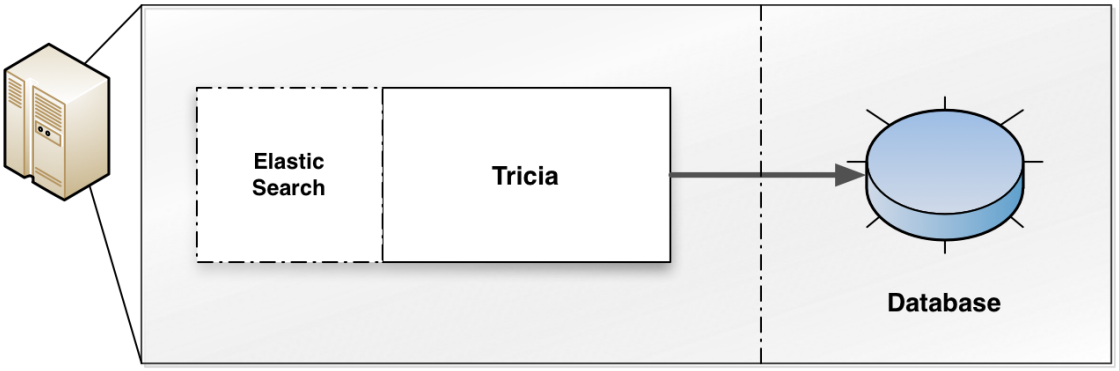


Figure 1.1.: Tricia Single Server Setup

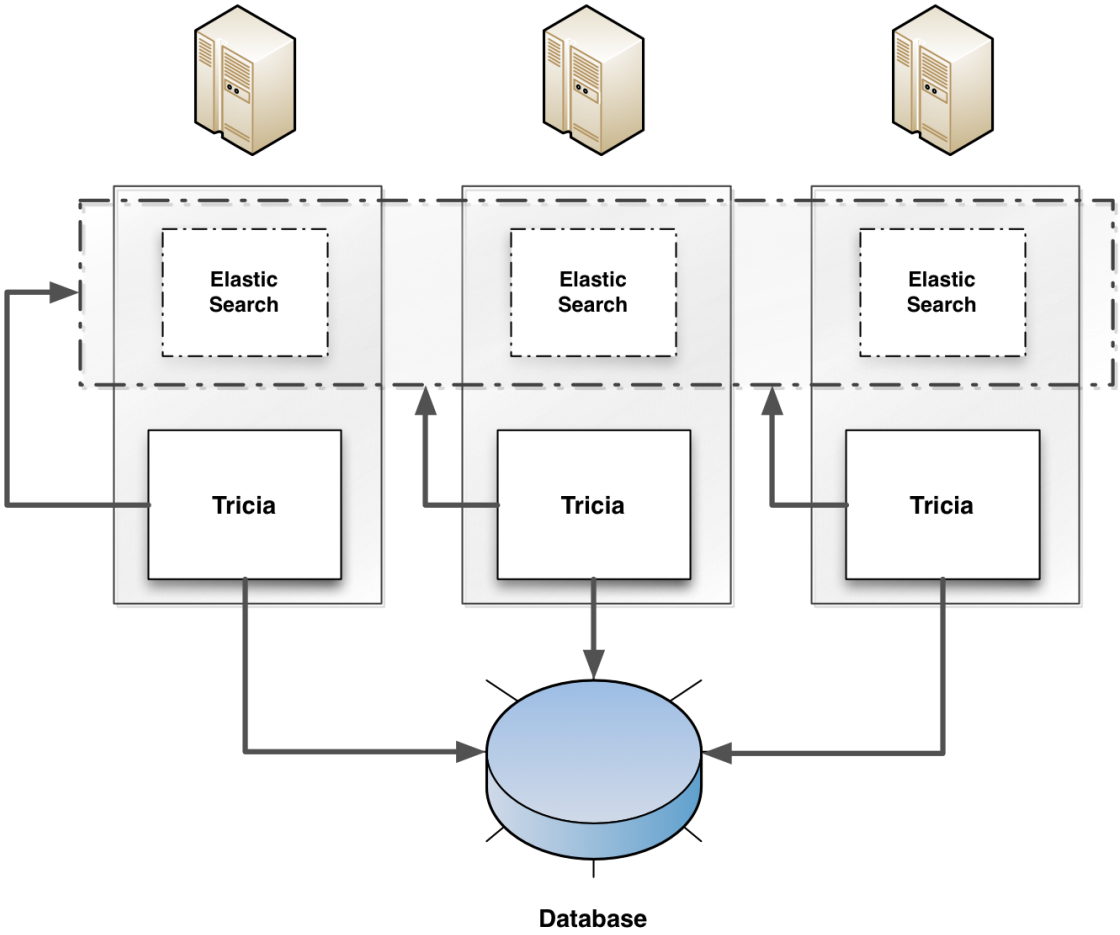


Figure 1.2.: Tricia Cloud Environment Setup

servers running, each with its own *Tricia* instance. As in the single server setup, a database for information storage is required. In this situation, there is a common one shared by all

Tricia instances. This way, every running *Tricia* can process any incoming request because the data is located at one point. Furthermore, *Elasticsearch* is also involved but now in its cluster configuration. To reduce the amount of servers needed, everyone of them runs both a *Tricia* and an *Elasticsearch* node. As mentioned before, the cluster itself takes care of distributing the search index among all nodes. Usually users do not connect to a single *Tricia* instance in a cloud environment directly but over a proxy (not shown in Figure 1.2). That is the same way *DTAG* also implemented it for their marketplace — users connect to a *load proxy* which tries to distribute all requests evenly among the different nodes in order to keep the average load as low as possible. When the proxy determined the right node, the *Tricia* instance running on the elected server gets the request.

1.3. Scalability and Distributed Software

This behavior leads to two important and required factors: *scalability* and *distributed software*. *Scalability* is a crucial aspect of handling an ever increasing demand for resources [35]. Basically, there are two different ways of scaling — *horizontally (scale up)* or *vertically (scale out)*. *Scaling up* means to make a system increasingly powerful. An example would be that a server has 4GB of RAM and a dual-core processor. In order to scale *vertically*, one replaces the server's hardware with, say, 16GB of RAM and a quad-core CPU. You therefore add more and more resources to a single node. On the contrary, *scaling out* requires to add more nodes, i.e. more servers. Instead of having one single server handle all the requests, one or two are added to the environment to take part of the load. Following the path of *horizontal scaling* ultimately leads you to the creation of a supercomputer-like setup.

In the context of *Tricia* and its deployment in the *Business Marketplace*, *horizontal scaling* is applied. Using this method also poses a very important requirement: every node has to be able to handle any request. As presented in Section 1.2.2, this is reflected in two key architectural aspects. First, there is one database shared by all nodes. Therefore the same information is available to every instance. Second, we have seen that search capabilities are provided by *Elasticsearch*. This framework was especially chosen for its cluster setup and thus its ability to scale. As most database systems are per se designed to be scalable, *Tricia* is consequently scalable as a whole system, too.

We are talking about three main parts, together forming the system *Tricia* — the database server, an *Elasticsearch* cluster and *Tricia* as application. We have also seen that these parts can all run on different machines at the same time. This exactly resembles the second factor mentioned at the beginning of this section: *distributed software* [8]. *Distributed software* describes that a software system is composed of components which can be located on different machines across a network. The components then interact with each other to altogether provide the system's abilities. Three key aspects of a distributed system can be stated as follows [8].

- *Concurrency of components* — When one component is busy doing a specific task, it does not automatically block others. Two different components can be active at the same time processing independent requests.
- *No global clock* — There is no explicit synchronization required for all components to

be working together.

- *Independent failure* — This should be a design goal when creating *distributed* software. The failure of one component of the system may not lead to the failure of other components. Practically, this is hardly feasible, especially in the context of a web-application with a database as information storage.

2. Problem Statement

This chapter presents the problem which is examined in the thesis. First, the motivation for doing *APM* is given. Following up, we describe the problem and put it in the context of *Tricia*. Afterwards, the involved stakeholders including their corresponding motivation are outlined.

2.1. Motivation

First, we want to give a short motivation why *APM* is necessary to do at all. *Tricia* is developed by *infoAsset*, where the following situation occurred. At that time, no monitoring was employed in *Tricia* in any way.

One day an employee of *infoAsset* discovered that two installations for customers were reacting very slowly. Consequently, the administrator of the instances' hoster was asked for help. Unfortunately, he could not identify any problems. The load of all involved systems seemed normal and their infrastructure did not show any issues. His only suggestion was to restart both instances.

The development team was left no other option than to do exactly that. They had no data on *Tricia's* behavior concerning these two installations. Fortunately, the problems were solved by a restart this time. However, it is unknown if a profound error in the application was the cause for the experienced slowdowns.

This is exactly the case where *APM* is needed. Without detailed information on the application's state and behavior at the time a problem occurs, it is almost impossible to find out the underlying cause. In order to be able to handle such issues, *APM* is an indispensable measure.

2.2. Problem Overview

The key question is how performance problems in an application can be solved by the help of *Application Performance Monitoring*. The origin and motivation is always the same: suddenly a problem arises without any reason. It is now vital to find the cause of the issue and solve it with a minimum amount of effort. The whole problem leads to four basic questions:

- *Are there any performance problems?*
- *Was it just a singular event?*
- *When do the problems occur?*
- *What is the cause of the issue?*

The first step is always to know *if* there are any problems at all. Before investing resources in polishing an applications performance, it is necessary to determine the need for such action. A very important aspect when talking about optimizing performance is *when* to do it. Donald Knuth once said: "*premature optimization is the root of all evil*" [3]. *Premature optimization* means to invest a lot of time to make code highly performant without knowing if this is required. Therefore, *APM* has the goal to give hints of when taking optimization action is really needed, i.e. when profound problems occur.

Secondly, as soon as a problem occurs, it has to be investigated if it was just a singular event. In case of the issue happening only once in a million executions, there is very little chance a profound problem in the application exists. However, even these events must be tracked. On the other hand, if an issue keeps arising every second time for example, a deeper investigation is definitely required.

When a problem has then been detected and verified to be pressing, the next step is to find out *when* it comes into play. Sometimes issues are related to a certain time of the day or specific to a single operation. Since this is one important indicator to restrict the problem's cause, *APM* should provide corresponding information, too.

The last question is of course the one asking the root cause. The cause has to be found out in order to properly solve the issue. Without knowing what was responsible for the problem, a developer cannot reliably find a solution. To help in resolving problems, appropriate forms of data and evaluation methods have to be determined.

These four questions are thoroughly analyzed. For every single one of them corresponding means of analysis are developed. They help answering those questions. Furthermore, it is required to analyze, how *Application Performance Monitoring* can then be successfully integrated in an already existing application.

Keeping in mind all requirements derived from the analysis of the above questions, an adequate solution has to be developed. There are already existing ones on the market but it is left to determine what can be reused. The main aspect is whether it is as easy as taking an off-the-shelf product or if a custom implementation is necessary.

2.3. Problem Context

The whole problem is tackled in the context of *Tricia*. *Tricia* and its basic architecture were already presented in Section 1.2. It is important to remember this context during the thesis. At all times, we refer back to aspects characteristic to *Tricia* when required.

As *Tricia* is a web-application, there is one key performance indicator: *response time*. This essentially defines the application's usability. The frustration of the user when working with the application can be related to the performance as outlined in [36]. This is depicted in Figure 2.1.

The vertical dashed line specifies the *normal* response time, i.e. what a user has to expect. In general this should not frustrate the user in any way at all. The first horizontal red line (**1**) is the *inefficiency* axes. When response time exceeds this threshold, the user is less able to focus on his work. Line **2** is deemed the *furstration* line. As soon as this limit is reached, a user actively realizes that response is slow. Forcing him to wait probably makes him think about other things, potentially even using a competitor. The last one (**3**) is the final line of *failure*. Response times greater than this either lead to a direct error in the browser

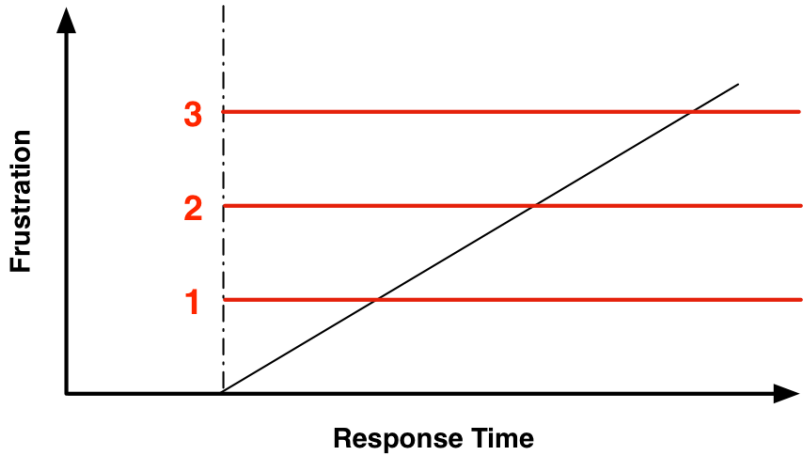


Figure 2.1.: Response time and user frustration (after [36])

because the request took to long, or the user himself refreshes the page. In the worst case a customer closes the browser window and is tired of waiting any more.

Knowledge of this behavior is therefore vital for *Tricia*. *Tricia* is designed and supposed to help customers get their work done faster. If the application itself prevents this by having slow performance, customers will be driven away. Ensuring high performance is thus also represents business value.

2.4. Involved Stakeholders

Before being able to properly analyze the given problem, we also have to identify the involved stakeholders. The analysis as well as the development of an adequate solution always needs to consider their interests. Two of them are identified quickly: the customer and the application's developer.

The customer has no interest in *APM* as of a data-centric viewpoint. He does not want to get any data of the application, e.g. response times or the number of requests. Talking about performance, he is just interested in having a *fast* and *usable* application. This also includes high availability, i.e. the application should be up and running at any time.

Furthermore, the developer is naturally interested in *APM*. His concerns are first to detect if there are any problems, and if so, he wants to be able to solve them. The developer therefore requires as much data as he can get to help him with analyzing issues. He is also responsible for ensuring the uptime and usability the customer demands. If problems occur it is the developers task to correct them as fast as possible. That is why the analysis and the developed solution are focused on the developer as a stakeholder of *APM*.

Since *Tricia* is deployed in cloud infrastructures, too, there is another stakeholder involved. A cloud infrastructure is hardly maintained by the developing company of an application but by a dedicated cloud provider. The provider gives a company access to all required resources and ensures appropriate maintenance and management of the infrastructure. Especially to determine the right amount of resources an application needs

and provide support for their customers, the cloud operators require information on the application's performance. The performance data serves as an indicator for them whether further actions need to be taken.

In the context of cloud environments, an immediate requirement is revealed. In all but every case a provider already has a monitoring system in place. This is used to ensure the availability of the infrastructure, i.e. that servers are running or the network is reachable. It is therefore important that the solution to be developed for *Tricia* can be plugged in into this system to provide the required data. Thus, a common standard has to be employed when implementing *APM* later on.

3. Analyzing the Problem

After the problem statement has been introduced, it is time to analyze it. First, we examine what metrics have to be collected in order to be able to extract valuable information. Afterwards, the four questions formulated in Section 2.2 are examined. For every one of them it is determined how *APM* can be used to answer them. We also work out the means of evaluation required to extract the necessary information from gathered performance data.

3.1. Key Metrics

As stated in Section 1.1, monitoring means to continuously observe a system. Since not everything can be observed as a complete thing, distinct *metrics* have to be determined. We therefore want to compile a set of key metrics best suitable for representing an application's current performance. Because this process cannot be generalized totally, we restrict ourselves to the context of *Tricia*. The findings presented in this chapter were backed by an operator of the *DTAG* cloud environment as well as Richard Weinberger, administrator of *Tricia* hosting at *SigmaStar*.

3.1.1. Requests

First of all, there are three different metrics directly related to the processing of requests: *response time*, *handler duration*, and the *requests rate*. Each of them is explained below and it is made clear why they are identified as key metrics.

In Section 2.3 we already elaborated on the importance of the response time of a web-application. It is *the* key metric to determine its performance perceived by the user. Of course, when measured with the users browser, side effects are also involved. These include the available network bandwidth and latency, for example. However, most of the side effects are beyond the developer's sphere of influence. Improving the application's performance would not help a user with a 56k modem too much.

Therefore the response time measured here is from the entry of a request in the application until the the response has been returned by the server. The response time includes the three steps parsing, processing and the generation of output. This is illustrated in Figure 3.1. The included steps have a gray background. Only processes which are directly affected by the application are measured.

In order to be able to differentiate better, one more metric is the *handler duration*. It is ancillary to the response time. In essence it is equal to the response time minus the two steps parsing and output generation as shown in Figure 3.1. With its help the execution time of the core business logic is tracked. It is necessary to employ two metrics in this case. If only the response time were measured, one could not detect if a problem occurred in the

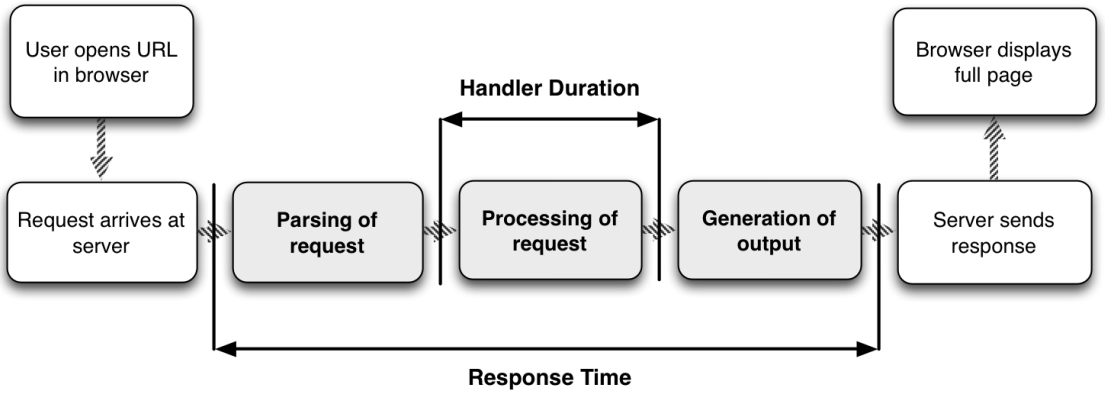


Figure 3.1.: Steps included in response time

business logic or the surrounding work. Having two metrics, one for the inner processing and one for the complete tasks, all steps can be assessed appropriately.

Moreover, the *request rate* must be measured. It is the rate of incoming requests, i.e. requests arriving *at the application*. This value is of special importance in the context of a cloud environment. Users do not directly access one of the cloud's servers. Their requests are sent to a *load proxy* responsible for delegating them in such a manner that all servers have the same load (see Section 1.2.2). As the result of a configuration problem, no requests might be delivered to the application itself, for example. This is then expressed in a request rate equal to 0. Consequently, continuously observing the request rate is highly important.

3.1.2. Database Operations

Just as a lot of web-applications make use of a database, *Tricia* also employs one for data storage. In cloud environments, one single database is shared across all instances. Since the majority of the data is located there, the database's availability is vital. Therefore, its performance has to be monitored closely. This is done by using two different metrics, one for the execution time and an additional rate.

The execution time includes all steps from creating a connection, if necessary, until the result is returned. Since the overall time to complete a database operation should be as low as possible, there is no use of tracking only the execution of a single command. Especially creating a connection can be a very time consuming operation. Also taking into account a possibly high network latency to the database server, it becomes clear that the overall execution time has to be measured. In *Tricia*, a lot of database operations may be triggered for a single request where even small delays sum up to a noticeable slowdown.

Furthermore, the rate of database requests has to be tracked. This is easily explained by the following example. Assume one database operation is executed every hour taking 1ms. Obviously, the average execution time is then equal to 1ms — absolutely no indicator of a problem. However, *Tricia* has a variety of jobs running in the background at fixed intervals. These usually execute a significant amount of database operations. That is why monitoring the execution time is not sufficient — a rate of 1 request/hour is definitely an alarming signal in the context of *Tricia*. It may indicate a severe problem concerning its

background jobs, for example.

3.1.3. Elasticsearch Operations

Apart from the database, *Elasticsearch* is another indispensable component. As stated in Section 1.2, it is used to provide the powerful search capabilities of *Tricia*. Even more so, a lot of functionality not related to a user searching for specific things is provided by *Elasticsearch*. As a result, the failure of *Elasticsearch* would imply the failure of *Tricia*. It is thus essential to monitor its availability, too.

In the same manner as with the database metrics, *Elasticsearch* should also be monitored by two distinct ones. The first one is to track the execution time of operations. Secondly, the rate of processed *Elasticsearch* requests is to be monitored. The reasoning is exactly the same as with the database operations in Section 3.1.2.

3.1.4. System Metrics

In addition to the application-related metrics mentioned above, system-related metrics also have to be observed. This is due to the fact that not all experienced performance problems are related to the application itself but are caused by resource shortages. Two main aspects have to be considered: *CPU* and *memory*.

The CPU is responsible for processing the tasks in the application. However, the application is in all but every case the only thing running on the machine. This is especially relevant in hosting environments where the application can be deployed alongside others. At some times, very computation demanding operations are executed outside the developer's influence. An example might be the generation of a server backup. This requires a lot of CPU time consequently lacking the application. As a result, requests may not be processed as fast as before and response time increases. What then shows as a performance problem to the end-user is not caused by a problem in the application itself. Therefore two CPU relevant metrics have to be observed: *system load* and *process load*. The former describes the overall load of the machine, i.e. how much work the CPU has to handle considering all running processes. The latter is restricted to the application's process, i.e. how much of the CPU time the application itself demands. The situation with creation of a backup would most likely result in a high system but steady process load.

A next very critical resource is memory. Almost always applications are hosted in virtual servers with a limited amount of memory. As soon as the memory is full, data will be *swapped*, i.e. stored on the hard drive. In the *DTAG* cloud for example, *swap* is disabled. Therefore when the memory is full and an application tries to allocate more, it will fail. That is why maintaining surveillance over the amount of *RAM* is required. Even in cases where *swapping* is enabled, the application experiences a drastic slowdown since hard drive reads/writers are extremely more time consuming than a corresponding memory operation.

3.1.5. Collection in Code

The collection in code is done by using *monitors*, as we call them. A *monitor* is used to measure a metric inside the application. To measure execution times for example, it is

wrapped around the monitored code. This can either be done manually by starting and stopping the measurement process or automatically. Some frameworks provide the ability to use Java annotations for tracking execution times [40]. There are also solutions which observe every method call themselves [27]. In essence, when speaking of a monitor, we refer to the measurement of a metric in code.

3.1.6. Summary

In this section various key metrics were identified. It was emphasized why each of them is required in order to draw conclusions on the application's state and performance. Nevertheless, the application itself has to contain more metrics. The developer is in need of very detailed performance data as formulated in Section 2.4. Therefore, every operation in code which may cause a problem has to be monitored for performance. As this is very application specific data, we do not give those metrics directly but every developer must determine them himself. These are often very low-level metrics embedded deep in the software. This section only identified key metrics which can also be transferred to other web-applications which refer to a higher level.

3.2. Means of Evaluation

Solely gathering data is not enough, it has to be used and evaluated. Therefore, the different means employed in order to benefit from the collected information must be worked out. The ultimate goal is to answer the four questions posed in Section 2.2 and repeated below:

- *Are there any performance problems?*
- *Was it just a singular event?*
- *When do the problems occur?*
- *What is the cause of the issue?*

During our research, three distinct means were derived: *performance snapshots*, *time series*, and *performance traces*. These are now explained in detail in the rest of this section.

3.2.1. Performance Snapshots

The first one we call *performance snapshots*. A snapshot is created periodically at a fixed interval like a couple of hours. It includes all monitors in the system in combination with detailed information on every one of them.

Among the detailed information required is the *total execution time*. This value is the sum of the duration of every single pass through the monitor. It serves as an indicator of where the application spends most of its time. Leveraging this information, optimization efforts can be focused on important areas. Tuning the performance of rarely used code parts which do not contribute to the overall response time in a high amount is just a waste of resources. This is easily explained with the following example [36]. Assume that 4

tasks contribute to an operation and their overall processing time is divided as shown in Table 3.1.

	Task 1	**Task 2**	**Task 3**	**Task 4**
Amount of processing time	60%	21%	13%	6%

Table 3.1.: Task contribution to operation

Increasing the performance of task 4 by 50% would only result in a benefit of 3% less execution time. On the other hand, reducing the processing time of task 1 by just 10% causes an increase in performance of 6%, so twice the one as compared to task 4. That is why important parts should be optimized first.

Furthermore, the *average execution time* must be included for every monitor. It is calculated as the *total execution time* divided by the number of *hits* the monitor had, i.e. how often it was triggered. The total alone has no direct implication on whether the application was slow or fast. A hundred executions of 10ms each lead to the same total of 1000ms as two executions with 500ms. Only by using the average value, one can draw conclusions concerning the performance. However, this requires a developer's knowledge. Someone who is not aware of the application's internals is not able to correctly judge the average execution time.

The last value a snapshot has to include for every monitor is the *maximum execution time*. The maximum, in contrast to the others, is able to capture singular events. If an operation is executed a lot of times but in only one case a special condition is fulfilled causing the task to take very long, the average and total value may seem normal. In that case the event is visible in a peek of the maximum. Of course, a high maximum does not always need to indicate a profound performance problem in the application. Nevertheless, especially this value can be used to quickly check for possible issues. If all maximums are very close or ideally equal to the average and the average is low, too, a problem is less likely. Instead of the average value it would be of advantage to use the some kind of mean value like an weighted arithmetic mean. Thus, single outliers would not have a potentially high effect as compared to the average value.

Performance snapshots are used to answer the first of the four questions: *Are there any performance problems?* With their help it is possible to get an overview of the application's performance over the last period of time very fast. Especially by looking at the *maximum execution time* of every monitor, problem indicators are found quickly. The monitor itself additionally serves as a first hint to where the problem may be located.

3.2.2. Time Series

Another mean is used to visualize the gathered data — *time series*. Using time series, a metric is plotted over a certain period of time. This allows the quick detection of developments or even relations to other data points.

Let us take a look at the example shown in Figure 3.2. Here, the process load is graphed over a time span of 1.5 days. It immediately meets the eye that the load increased drastically from a starting 10% to almost 50%. Since this is such a significant change, it

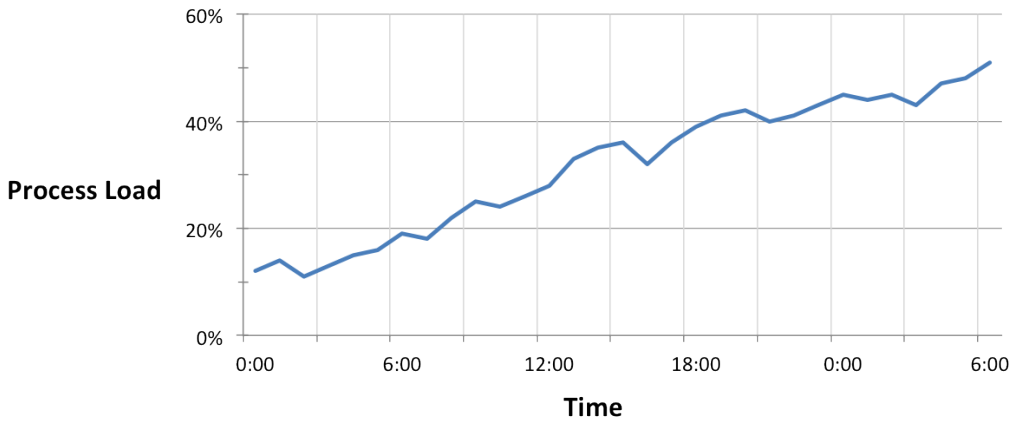


Figure 3.2.: Exemplary process load time series

requires attention and the cause has to be identified. Especially when a development takes place over a long time, single values in numerical form do not make it obvious. On the other hand, a graph can be scaled and the hidden process becomes visible soon.

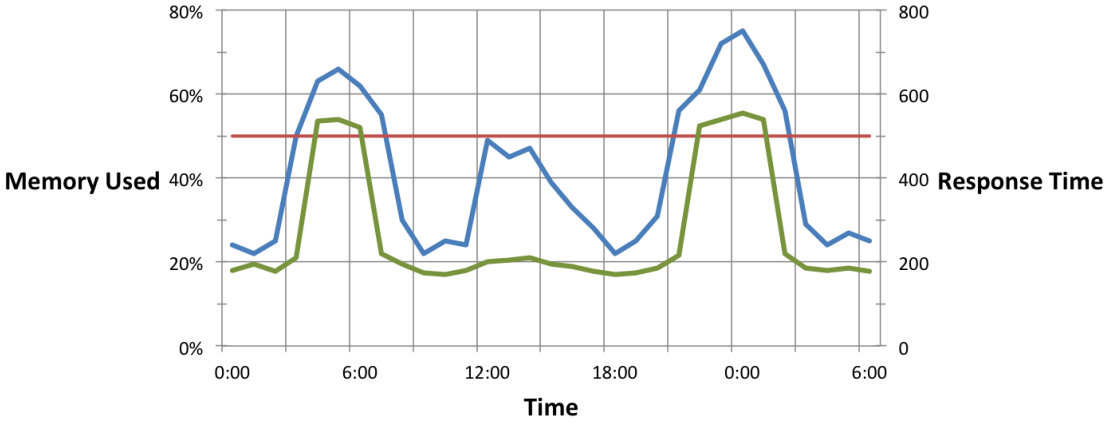


Figure 3.3.: Exemplary memory and response time relation

Figure 3.3 illustrates another aspect which is greatly simplified by time series. It shows two metrics, the amount of memory used (**blue line**) and the response time (**green line**). There is also a **red line** at the 50% mark of used memory. Obviously, as soon as the amount of memory used (**blue**) exceeds 50% (**red**), the response time (**green**) increases drastically. If the memory increases but stays below the threshold, though, there is almost no change to the response time (between *12:00* and *18:00*). Once again it would be a very tedious task to detect the relationship between those two metrics when analyzing only the numerical values. Time series enormously facilitate this process.

Concerning the four key questions, time series primarily help in answering the first three of them. First, it is possible to check if there are any problems at all. To do this, the gathered data is plotted over a wider range. If spikes occur in a graph then these are indicators for

a problem. Also, the second question if a problem was not just a singular event can be answered. Once a problem indicator has been discovered e.g. by a high maximum value in a snapshot, the metrics are plotted over the corresponding time interval. It is then possible to see if the issue persisted for a longer period. The last one, answering *when* a problem occurs, is also supported by time series. Due to their nature, especially relations to a certain time of day are detectable. This may be caused by a scheduled backup operation executed every day at 5:00 AM, for example. On the other hand, as stated above, time series are great to show connections in-between distinct metrics which is another important aspect concerning the question *when* an issue occurs.

3.2.3. Performance Traces

The last important question to answer is: *What is the cause of the issue?* In order to answer it, we introduce *performance traces*. A *stack trace* is well known among programmers — it is usually created in case of an exception. A stack trace contains the stack frames active before the error occurred, i.e. what method caused it and which other methods were called beforehand. A performance trace is quite similar. As soon as a performance problem is detected, e.g. by a defined threshold in the application, a trace is created. It contains all monitors which were triggered during the operation. For every monitor it also includes detailed information.

```
1 ------ long taking request: 5200 ms for /action/execute ------
2 webServer.request              : Hits:   1, Total:  5200.4, Avg:  5200.4
3 handler                        : Hits:   1, Total:  5000.0, Avg:  5000.0
4 es                             : Hits:  10, Total:   350.7, Avg:    35.1
5 db                             : Hits:  22, Total:    26.2, Avg:     1.2
6 utilities.parseUrl             : Hits:   5, Total:  4505.0, Avg:   901.0
7 ...
```

Listing 3.1: Examplary performance trace

How such a trace could look like is presented in Listing 3.1. In the first line, a description of the detected performance problem is given. In this example it is a request which was executed in a total of 5200ms for the URL */action/execute*. The following lines then contain the triggered monitors and for each their hit count as well as total and average execution times. Here, line 6 is of interest. The `parseUrl` task took a total of 4.5 seconds and 0.9 seconds on average. This is the absolute majority of processing time need for this request. It is now up to the developer to determine if that is causing the problem. Without having knowledge on the application's implementation it again cannot be judged appropriately.

Having a detailed log of the different tasks that took place during a slow operation is the key to finding the cause of the problem. A developer is able to close in on the issue and to possibly identify the one monitor tracking the problem. The next step is then to investigate log files and code to finally solve the problem.

3.2.4. Proposed Usage

Apart from identifying the three different means of evaluation — performance snapshots, time series, and performance traces — we also derived a proposal on how they can be

combined. This should serve as a guide for anyone interested in the process of finding and analyzing a performance problem.

First, it has to be determined whether a problem exists. One possibility is that someone using the application actively reports it. In any case, the applications performance should be checked on regularly. Two of the discovered means were deemed suitable for this task: performance snapshots and time series. Every now and then one of them should be used to see if there are any indicators for issues.

Once an indicator has been identified, it is necessary to validate it, i.e. to check if it was not just an exceptional event. If it cannot be verified, one has to keep an eye on the problem, of course. Immediate action is not needed, however, especially when the problem is not reproducible. Verification is done best by cross-checking. In case of the time series being used to find indicators, the snapshots should be checked for validation and vice versa.

Consequently, when an issue was identified and verified, its cause must be found. This is done best by the help of a performance trace. The trace contains the highest amount of information of all and serves best to narrow down where the problem originated.

3.3. Intermediate Results

In this chapter we were able to identify the key metrics that should be monitored. The following list sums up all the metrics proposed in Section 3.1:

- Response time
- Handler duration
- Requests rate
- Database operation execution time
- Database operations rate
- *Elasticsearch* operation execution time
- *Elasticsearch* operations rate
- System load
- Process load
- Memory consumption

Furthermore, Section 3.2 presented the appropriate means of evaluating the collected data. Three distinct ones were identified: *performance snapshots*, *time series*, and *performance traces*. Additionally, a proposal on how to combine them was made in Section 3.2.4. This can be aggregated into a three-step process. The corresponding means used in each step are given in parentheses:

1. Checking for indicators (performance snapshots, time series)
2. Verification (time series, performance snapshots)
3. Tracking down (performance trace)

The basic groundwork has therefore been done and we advance to consolidating aspects relevant to how *APM* can be employed in an existing application.

4. Relevant Tools and Standards

Having done a thorough analysis of the problem statement itself, we derived some important requirements and aspects concerning *Application Performance Monitoring*. The next step is to examine already existing solutions and tools out there as well as dig into relevant standards. The latter are especially relevant in the context of a cloud infrastructure, as stated in Section 2.4.

4.1. Relevant Tools

The set of relevant tools ranges from complete end-to-end solutions to small libraries utilized for data collection and aggregation. For each tool, its main purpose and typical usage is presented, as well as advantages and important drawbacks.

4.1.1. New Relic

Starting off with *New Relic* [27], we take a look at a complete end-to-end solution which is not just a "tool". *New Relic* offers application monitoring as so-called *SaaS — Software as a Service*.

Software as a Service describes a model where software including all necessary infrastructure is provided by the vendor [35]. Billing is either based on recurring subscriptions or on usage and customers can access the software globally, in most cases just using a browser or thin-clients. The greatest advantage is that all maintenance as well as monitoring and updates is done by the vendor itself. This means that the customer has almost no overhead when using some kind of *SaaS*. Widely known examples of *SaaS* software are Google Gmail [15] or Office 365 [23].

Overview

The main goal of *New Relic* is to provide easy-to-use application monitoring for a variety of platforms. On the one hand, they offer monitoring for web-applications based on e.g. *Ruby*, *Java* or *Node.js* for example, on the other hand they also provide solutions for monitoring mobile applications. Concerning web-applications, every typical layer is included, ranging from the database layer where single SQL statements are tracked to the end users usability when browser load times are measured.

The information is presented to the user via different and customizable dashboards — an example is shown in Figure 4.1. Most of the information is presented in graphs, where the time range can be configured to take a look at past behavior. However, the available time range to go back in the past depends on the data retention included in the subscribed plan. Metrics can be combined for display as preferred by the user resulting in a high

degree of flexibility. A custom overview of the monitored application's most important performance indicators is created with ease.

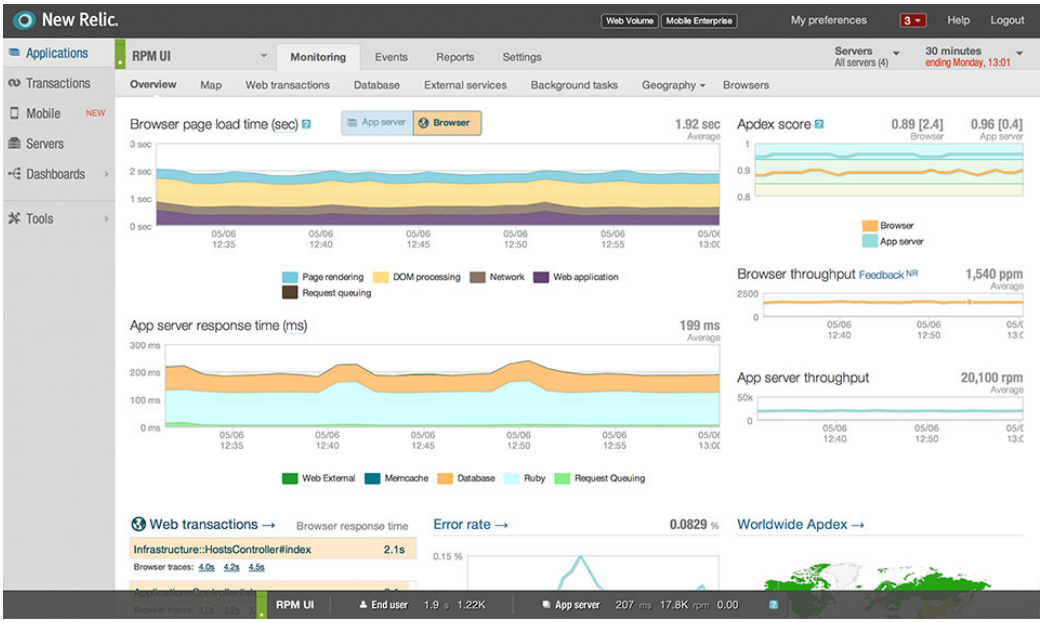


Figure 4.1.: New Relic dashboard

Apart from a very sophisticated presentation of the collected data, *New Relic* offers numerous other features. One is the triggering of alerts: a customer can create rules for specific metrics where the violation of such a rule results in the creation of an alert, which is then sent to the customer via email or push notification to his mobile phone. An example is that the monitored applications administrator wants to be notified as soon as the average applications response time exceeds $500ms$. Furthermore, transactions in web-applications can be traced even across different layers and nodes. This means that a request is dissected in a trace of function calls to show where highest amount of time was spent during the processing of the request. Such an analysis of a trace is one of the essential tools for *Application Performance Monitoring* as derived in Chapter 3.

Typical Usage

New Relic is employed in a wide range of applications. Since they offer free *lite* accounts with a very limited but sometimes sufficient set of features, small developers without funding make use of it. Apart from that, a lot of major companies like Nike and Groupon use *New Relic* as their tool for web-application monitoring or for mobile application monitoring, e.g. Nascar and RunKeeper.

Starting from late June, 2013, *New Relic* has also included an open platform. Customers and technology vendors can now create their own plugins to deliver data to the *New Relic* web-service. The custom data is then also accessible via all the offered features, resulting in the opportunity to be provided with everything — even highly application specific data — in one place.

Advantages and Drawbacks

The advantages are quite obvious — *New Relic* offers a huge set of very customizable features, what makes the most relevant information instantly visible to the customer. Taking into account the change towards an open platform, with the development of custom plugins to provide even more specific application data, it could evolve to the one single place to have all relevant performance data.

One of the most outstanding features and an incredible advantage however, is the ease of deploying and including the required libraries to get monitoring running. Concerning a standalone Java web-application, it is as easy as downloading a *JAR*-file and running the application with using the *New Relic JAR* as a *Java agent*. To get data collection running, no further configuration is required, resulting in an installation and therefore deployment time of just a few minutes.

Nevertheless, there exists one major drawback due to *New Relic* being a *SaaS*. There must be a constant internet connection available for the data to be transmitted to the servers. As this is not a problem in the majority of use cases, it is the crucial point to search for another solution when regarding highly secure cloud environments. Despite transmission being optionally encrypted using SSL, restrictions may prevent any outgoing traffic except regular HTTP traffic at all. In such rare cases, *New Relic* cannot be employed to do application monitoring. In addition to that, according to their terms and conditions, the collected data may be shared with third parties in a not completely obfuscated way. One scenario they explicitly name is providing an application's performance data to its hoster for him to be able to improve his service. Moreover, a free account is hardly sufficient, and pricing starts at $24 *per server* for a standard account if pre-paid annually or $49 on a monthly basis. Major features like transaction and SQL query analysis are only included in the pro edition, which in turn starts at $149. When the monitored application is running on multiple servers, e.g. because clients want their own installation, this soon becomes an important drawback.

Summary

All in all, *New Relic* provides a highly usable and valuable service, which is ideal to get an overview of all the data an application provides. Its ease of installation and almost no need for configuration enable customers to monitor their software in no time. On the other hand, the drawbacks mentioned before in some cases prevent the usage of *New Relic*. When the application is hosted only on a few servers or in a single, permissive cloud environment, *New Relic* is definitely a service worth paying attention to.

4.1.2. Kieker

The next existing tool we want to take a deeper look at is *Kieker* [40]. *Kieker* is a Java framework developed at the *Christian-Albrechts-University Kiel*, mainly by André van Hoorn, Jan Waller and Wilhelm Hasselbring.

Overview

Kieker's purpose is to provide an extensible system for monitoring and analyzing concurrent or distributed systems. The core framework offers initial features and components for instrumentation, logging and analysis / visualization. In this case, the term *logging* refers to writing the gathered data to some kind of storage or output. Due to its extensible nature, users of *Kieker* can extend the functionality according to their specific needs.

In *Kieker*, the collection of single metrics is done via so-called *monitoring probes*. They are used in code to do the measurement itself and to pre-process the data if necessary. An interesting piece of code which is to be monitored, has to be wrapped by a *probe* either manually or by the use of annotations also included in the framework. After a measurement has taken place, a *probe* typically creates a *monitoring record*. All records have a common superclass but the creation of custom subclasses is open to the user and encouraged. The records are then handed over to a *monitoring log writer* which is responsible for — as its name implies — writing or serializing a given record to the *monitoring log*.

On the analysis side of the framework, there are so-called *monitoring log readers* to read an existing *monitoring log*. A reader is used to extract single records out of the log and to create corresponding *records* again. These records in turn are fed to a *monitoring record consumer*, which can selectively discard unwanted records, e.g. depending on their type. The *consumer* is also responsible for evaluating the records and producing any type of visualizations.

The fact that *Kieker* is developed and maintained at a university is also reflected in the framework itself, since it has a lot of scientific-based features which exceed the sole purpose of performance monitoring. The visualization capabilities already included contain for example UML sequence diagrams, dependency graphs, and Markov chains. The analysis required to create such models is based on traces produced by the monitored application. These traces can also originate from distributed systems, i.e. systems running on different machines, whereas joining back all the different ones to a single trace is handled by *Kieker*.

Typical usage

Kieker has two main areas of application. The first one is typical *Application Performance Monitoring*, for which the base framework already includes the aforementioned probes to collect response times, sessions and traces, amongst others, but also system-level metrics like CPU load and memory usage. During the development, the team behind *Kieker* also focused on keeping the overhead introduced by employing the probes in code as low as possible. To prove their low footprint, the framework was used in several benchmarks as well as evaluated in several industry scenarios.

Another aspect is *Architecture Discovery*, i.e. the extraction of architectural information from data gathered by employing monitoring in an existing software system. The goal is to create a structural and behavioral model of the application and therefore identifying components, classes, etc. and interactions. As already mentioned, UML diagrams and dependency graphs fall into this domain of application and provide useful means of visualizing the results of the discovery process.

Moreover, one area of application the creators use *Kieker* for is *Application Performance Management* (see Chapter 1). In their special case, they want to automate the process of reacting to the current state of the software system, which in turn is detected by application monitoring. An example might be the following situation. An application fetches a lot of data from the database, due to a high amount of requests, however, all connections are handed out and new requests have to wait before being processed. The employed monitoring system detects the increasing execution time and automatically configures the connection pool to provide more connections. After the number of requests has dropped again, the connection pool's configuration is reverted to its original state. This runtime adaptation respectively re-configuration is a major point of focus of *Kieker*'s underlying research.

Advantages and Drawbacks

The low overhead and its very extensible structure are definitely one of *Kieker*'s main advantages. Since there is hardly any impact on the application's performance, monitoring probes can be put anywhere in the code, even in large numbers. This allows the collection of massive and widespread data to serve as a basis for an extensive analysis. Customization of both — collection and analysis — is further encouraged and supported by the extensible nature. Whatever metrics need to be observed and stored, by the creation of custom *monitoring records* there are no limitations of what can be monitored. Furthermore, users are able to implement their own *record writers* and *readers*, resulting in a free choice of data format and storage destination concerning the *monitoring log*.

The very advanced feature set — e.g. distributed traces, architecture discovery — provide powerful means of analyzing the monitored application, too. Especially when introduced afterwards in an already existing software system, these tools can offer a new insight in its runtime behavior and structure. When talking about a large system which evolved over a long period of time in particular, the discovery of its structure may reveal hidden dependencies introduced by adding components and expansion which the creators are not even aware of. Nevertheless, not everyone is in need of such advanced features but wants to stick to simple monitoring of an application.

As of the current standing in mid 2013 however, *Kieker* comes only with a limited set of *record writers* and *readers*. Those are one for the file system, relational databases and the *JMS* (*Java Messaging Service*). As a consequence, data storage has to be handled by the user himself. Enough disk or database space has to be provided and data has to be cleaned up manually, of course. Therefore, there is a certain overhead one must take into account when thinking about employing *Kieker* for monitoring. In addition, to use the framework in an existing application means a lot of effort since the monitoring probes have to be put into the code manually opposed to the automatic injection *New Relic* uses. The creation of time series, for example, is also not included in the framework itself. A user has to implement a custom *record consumer* to extract the necessary data out of the gathered records and then feed it into an appropriate tool for creating any graphs.

Summary

To come to a conclusion, *Kieker* offers a scientific based Java monitoring framework in a fairly early stage of development. It has some very advanced features in discovering the possible hidden behavior and structure of an application and is very extensible. The framework is also actively improved and new versions are released regularly. Despite its advantages, the high amount of introducing *Kieker* in an existing application and handling storage by oneself definitely represent drawbacks. Taking into account that the advanced features are not required by everyone, other solutions are worth a look when *Application Performance Monitoring* is to be employed in an application.

4.1.3. Java Simon

Another library suitable for the collection of data in a Java application is *Java Simon* [19]. It provides developers with utilities to easily place monitors in their code. The emphasis of the API is put on enabling *APM*.

Overview

First of all, *Java Simon* calls monitors *Simons*. Therefore, when we reference a *Simon* we mean essentially the same definition as given in Section 3.1.5. In *Java Simon*, all monitors implement the same interface `Simon`. Furthermore, every `Simon` is identified by a unique name. A manager can be queried for them by their respective name. Furthermore, `Simon`s are organized hierarchically in a tree according to their names. An example is shown in Figure 4.2.

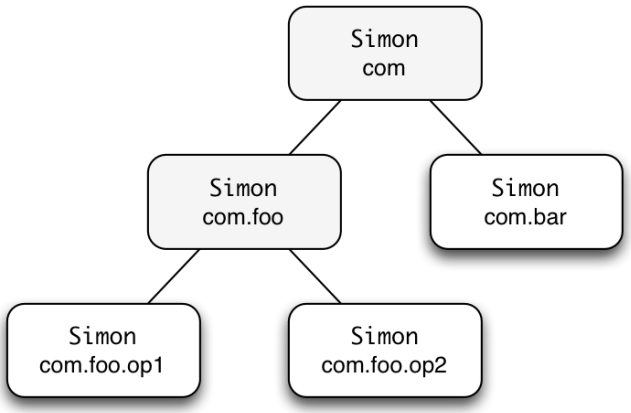


Figure 4.2.: Java Simon `Simon` hierarchy

Only the white `Simon`s were created in code, the gray ones were created automatically. Thus, the `Simon` named *com* can be asked for its children which would yield all other four: *com.foo*, *com.foo.op1*, *com.foo.op2*, and *com.bar*.

This especially comes into play concerning the two different types of monitors *Java Simon* includes: *Stopwatches* and *Counters*. The former is used to measure time spans in

code. A *Stopwatch* internally also calculates a variety of other values. These include the average execution time, hit count, or maximum execution time since its last reset. The values are also influenced by the imposed hierarchy. Let us take a look at the example from Figure 4.3. The total execution times are given in bold text. The common ancestor of *com.foo.op1* and *com.foo.op2* is *com.foo*. *Java Simon* then automatically sums up the execution times of the children, resulting in a value of *450ms* for *com.foo*. The same process is then applied to the parent *com* and its two children, *com.foo* and *com.bar*. The result thereby is *600ms*. This way it is very easy to track the execution times of single operations but automatically be provided with aggregated measurements.

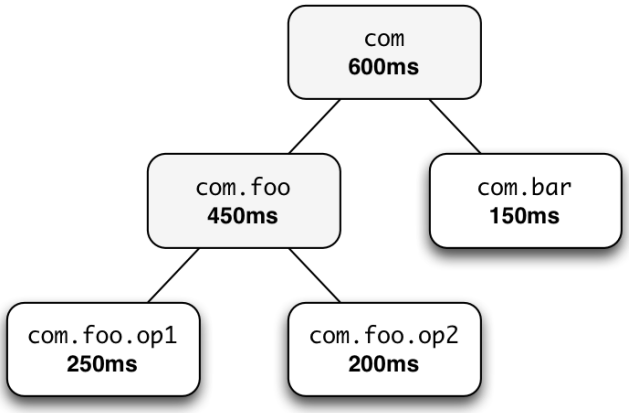


Figure 4.3.: Java Simon *Stopwatch* hierarchy

On the other hand, *Counters* are used to measure the occurrence of events. They basically contain a number which can be increased or decreased. As with *Stopwatches*, *Java Simon* provides additional metrics like the maximum or minimum value the *Counter* has reached since the last reset.

Typical Usage

Apart from manually placing the monitors in code, the API offers the ability to automatically track some operations. One example is a database connection usually created with *JDBC (Java Database Connectivity)*. *Java Simon* enables the developer to wrap such a connection resulting in the measuring of executed operations.

In essence, the purpose of the API is to be used for gathering massive data in code while keeping the introduced overhead at a minimum. The application itself is then free to process the collected information in any way. It can query the *Simon Manager* for all known `Simon`s and process the information further, for example. One possibility would be to create regular performance snapshots as proposed in Section 3.2.

Advantages and Drawbacks

The first advantage to name is that the *Java Simon* is especially designed to have a very low performance overhead. `Simon`s can be added anywhere in code without having

to fear a performance impact on the application. Thus, *Java Simon* is very suitable for gathering countless metrics. Moreover, it is very easy to use. The API is well designed, the name identification of a single `Simon` frees the developer from tracking instances. Another advantage is the hierarchy derived from the names. As a result, aggregated values are available without further configuration. Nevertheless, this does not cause any performance problems.

Apart from that, one important drawback exists. *Java Simon* includes only two different types of monitors: *Stopwatches* and *Counters*. Though these are sufficient for a lot of cases, *Counters* in particular have a disadvantage. *Java Simon* is not able to convert them in a rate, as for example, how much the counter is increased or used per second. This would be very useful considering the tracking of incoming requests in a web-application.

Summary

Coming to a conclusion, *Java Simon* is a very helpful and simple API for gathering basic performance data in an application. It provides a fast and efficient way to monitor especially the execution times of operations in the software. If the two types, *Stopwatches* and *Counters*, are sufficient for the majority of the tasks, *Java Simon* is the way to go.

4.1.4. RRDtool

Since the collected data has to be stored somewhere, we want to examine the *RRDTool* library [28]. *RRDTool* is an open source project providing the ability to store data with high performance.

Overview

RRDTool uses so-called *round robin* databases. This means that data is saved only for a fixed period of time in a fixed-size storage. Once the storage is full, the oldest entries are automatically overridden. Before being able to use *RRDTool*, an *rrd* file, i.e. a round robin database, must be created. This requires a complicated configuration. Since we leverage *RRDTool* later, the model behind an *rrd* file is explained.

First of all, single data values are named *Primary Data Points (PDP)* in the context of *RRDTool*. *RRDTool* must first be configured at which interval *PDPs* are collected. This is called the *stepSize* and is given in seconds. Therefore, the minimum time difference between two *PDPs* is 1 second. Moreover, hese data points are not stored directly. Storage is handled by so-called *archives*. An archive has four distinct options which must be configured: *Consolidation Function*, *Steps*, *Rows*, and *XFF*.

An archive always combines one or more *PDPs* into a *Consolidated Data Point (CDP)*. How this aggregation works is defined by the *consolidation function*. It is the mathematical function executed on the *PDPs* which are to be combined. One possible consolidation function is the maximum function *MAX*. The aggregation is also influenced by the next parameter, *steps*. *Steps* defines how many *PDPs* are used to calculate one *CDP*. If steps is equal to 5 and the consolidation function used is *MAX* for example, the the archive will wait until it has 5 *PDPs* and then store the maximum value of them. The ratio between stored data and collected data is thus equal to $1/steps$. The next parameter is *rows*. It just

states how many *CDPs* are stored in the archive. This essentially determines the covered time range data is available in. The total time span T_s contained in an archive can be calculated as follows:

$$T_s = (stepSize \cdot steps) \cdot rows$$

As an example, assume a *stepSize* of 5 seconds. The archive is configured with *steps* equal to 12, i.e. 12 *PDPs* are used to calculate 1 *CDP*. Furthermore, *rows* is set to 1440. Therefore the time span T_s is calculated as:

$$T_s = (stepSize \cdot steps) \cdot rows = (5s \cdot 12) \cdot 1440 = 60s \cdot 1440 = 86400s = 24h$$

Furthermore, the different data values themselves have to be specified. This is done by *datasources*. A *datasource* has a *name*, *type*, and type specific arguments. The *name* is the unique identifier of the source in the *rrd* file. Values can only be appended to an *rrd* file by specifying the corresponding *datasource* name. A *datasource* also has a *type*. This defines the source's data format. Two examples are *GAUGE* or *COUNTER*. A *GAUGE* is an arbitrary numerical value like the CPU load or free memory in bytes. On the other hand a *COUNTER* is assumed to never decrease. However, it can be increased by any value. *RRDTool* automatically handles an overflow.

Typical Usage

The author of *RRDTool* calls it *"an industry standard"* [28]. It is very commonly used and can be integrated into shell scripts or python, even Java libraries exist, to name a few. For the latter, there is the *rrd4j* API [4]. *RRDTool* is also used in various other monitoring solutions like *collectd* [12] or *Cacti* [37].

The core functionality of *RRDTool* is overall focused on an efficient and fast way to store data. Apart from that, some graphing capabilities are included. However, these are only accessible via command line, at least in the original *RRDTool* distribution. Most of the tools leveraging *RRDTool* internally access its command line interface in order to provide visualizations.

Advantages and Drawbacks

The key advantage is the *rrd* file's fixed size. As everything has to be configured before the database can be used, the file will always require a constant amount of disk space. Therefore, no special precautions need to be taken to provide enough storage during operation. Furthermore, the integrated *consolidation functions* specified for each archive are very helpful. An example is to specify two distinct archives, one with a a relatively high *step* value and the other with a small *step*. Also, both are assigned the average consolidation function. As a result, the former archive with high *step* value can be used to quickly examine a certain time period. If uncommon behavior is detected, the other archive with small *step* parameter provides the ability to zoom in on the time range the problem occurred in.

RRDTool's requirement of being configured beforehand is also one of its major drawbacks. It is a error-prone task to modify the archives or datasources after the *rrd* file has been created. This should be avoided as much as possible. Thus, dynamically adding new

sources is cumbersome. Additionally, all *types* of datasources in essence are stored as numerical values. No fully customized data can be stored. This is in contrast to the abilities of *Kieker* presented in Section 4.1.2.

Summary

All in all, *RRDTool* provides a highly attractive way of storing collected data. Even more so if the monitoring data is composed only of numerical values. It also gained a lot of popularity and attracted a high amount of attention due to its high performance and round robin fashion. This is also due to the fact of *RRDTool's* long existence. Version 1.0.0 was already published in *1999*. Over the years it turned into a very stable and reliable tool. On the other hand, in environments where datasources are added and removed dynamically, a careful decision has to be made whether one can take the risk of data corruption. The limited set of datasource *types* represents an additional aspect to consider.

4.2. Standards

Besides existing tools which can be used for *Application Performance Monitoring*, we also need to investigate what standards do exist in this area. It will soon be obvious that there are two major ones to be regarded and a lot of small protocols — not officially defined standards — which are employed frequently.

4.2.1. Java Management Extensions

The *Java Management Extensions* (*JMX*) [30, 11] standard was developed with three main questions in mind concerning the management of an application:

- Which solution is the best one?
- What standards should be followed?
- What is the amount of work required to make components manageable?

The outcome allows developers to make their system manageable with a low amount of effort but also provides very powerful ways of remotely interacting with the application. In the following, we shortly describe the architecture of *JMX*, as well as how it can be employed in an already existing Java application.

JMX Architecture

The architecture of *JMX* is divided into three different layers, each of them responsible for one of the following tasks:

- Making resources manageable
- Publishing managed resources
- Interacting with managed resources

The tasks can be mapped to their corresponding layers in the given order: *Instrumentation Level*, *Agent Level*, and *Distributed Services Level*. These are also illustrated among further details in Figure 4.4. We will now take a look one after another and explain its use.

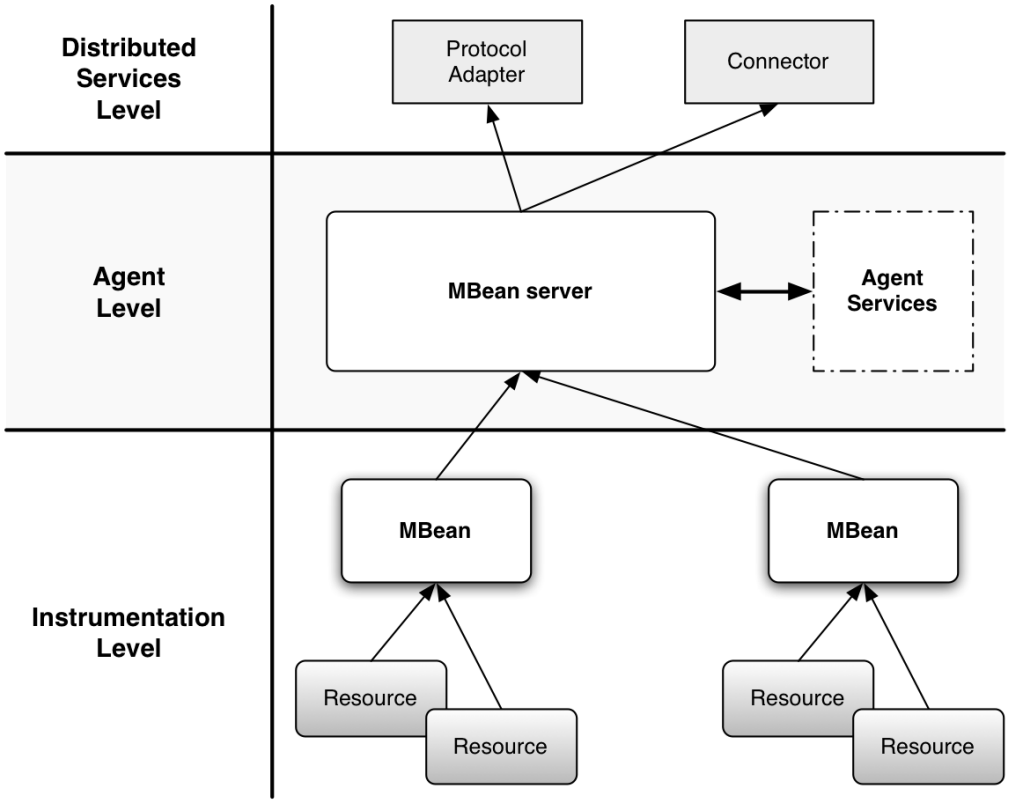


Figure 4.4.: JMX architecture overview (after [30])

The *Instrumentation Level* is the lowest of all. Here, so-called *MBeans*, short for *managed beans*, are created. They serve as the most basic entity in the *JMX* world and provide state information or operations which are to be published. In general, *MBeans* typically wrap up information of various application resources and export operations to be able to interact with those resources, e.g. to reconfigure them. There are different types of *MBeans*, ranging from *standard* ones defined via Java interfaces over *dynamic* beans, which can provide different attributes or operations at runtime to even *open* beans. *Open MBeans* serve the purpose of publishing very complex information by using custom defined types. Furthermore, *JMX* is able to handle *Notifications*. These are created on the *Instrumentation Level* and are then handed over to the *Agent Level*, namely the *MBean server* for broadcasting.

Once beans are created, they have to be published. This is the task of the core of *JMX* — the *MBean server*, located on the *Agent Level*. In order to later publish a bean, it has to be registered with an *Object Name*. This is an identifier unique to one specific *MBean* and is used to gain access to its information and operations by remote applications. The server contains an internal registry, in which a mapping of names to references of beans is stored. If someone wants to access an *MBean*, he has to query the server providing the

beans *Object Name*. In turn, the caller receives not the bean directly, but an object containing detailed information on the provided attributes and operations. This can then be used to retrieve values or execute the exposed functions. On the same level are also the so-called *Agent Services*. They are used by the *MBean server* and conversely registered as *MBeans* themselves so that they can be managed, too.

One example of such a service is the *Monitoring Service*. It provides *monitors* which can be configured to observe the attribute values of a specific *MBean* — the *observed object*. Values are collected at fixed intervals (*granularity period*) and based on them a *derived gauge* is calculated. The *JMX* specification defines three different types of monitors — *counters*, *gauges*, and *string monitors*. While the first one is self explanatory, *gauges* are used to surveil arbitrary numerical values. *String monitors* in turn constantly compare an attribute of type *String* with a given value. All of them can be configured to post the already mentioned *Notifications* if certain conditions are met, e.g. a counter reaches a threshold.

Finally, in order to make an *MBean server* accessible by remote applications, it has to be configured with *Connector* or *Protocol Adapter*. Though they are responsible for managing all access to the underlying server, their mode of operation is very different. A *connector* has two essential parts, a client proxy and a server stub. Their goal is to provide a common interface abstracting away from a concrete *MBean server* implementation as well as hiding details of where resources are located in a network, all in all ensuring the same access methods for local and remote beans. On the other hand, a *protocol adapter* is run on the same machine as the *MBean server* and exposes the servers functionality via another protocol — hence its name. An example for an adapter is the `HtmlAdaptorServer` which makes the whole *MBean server* available via *HTTP*.

Employing JMX

As we just presented a rough overview of the architecture of *JMX*, let us take a look at how it can be employed in a Java application. We start at the lowest level, the *Instrumentation Level*. The first task for a developer in order to use *JMX* is to create the specific *MBeans* which will be published. There are different ways to do this, as mentioned above, and we present an example with a *standard* bean here. For a *standard MBean* there must be an interface definition containing *getters* for attributes and / or additional operations. See Listing 4.1 for an example of such a Java interface offering two attributes, `Count` (line 3) and `Rate` (line 5), as well as an operation `reset` (line 7). One important aspect is the ending of the interface name which *must* be equal to `MBean`.

```
1 public interface CounterMBean {
2
3     int getCount();
4
5     double getRate();
6
7     void reset();
8
9 }
```

Listing 4.1: Standard MBean interface example

A class implementing the interface from Listing 4.1 can then be registered to an *MBean server*. The necessary steps are outlined in Listing 4.2. First, we retrieve a reference to the specific *MBean* (line 1). Next, a corresponding *Object Name* is constructed, which has to be unique in the server's registry (2). Now everything is set up and we obtain a reference to the platforms *MBean server* in line 4 to then register the bean with its name (line 5). Though the creation of an *Object Name* and the registration itself can throw exceptions, these are omitted here for clarity.

```
1 CounterMBean counterBean = ...;
2 ObjectName name = new ObjectName("com.foo.bar:type=Counter");
3
4 MBeanServer server = ManagementFactory.getPlatformMBeanServer();
5 server.registerMBean(counterBean, name);
```

Listing 4.2: Registering an MBean example

With only these few steps, it is possible to expose information and operations of an application via the use of *JMX*. Without further configuration, the bean server is accessible locally using tools like *jconsole* or *jvisualvm* which are bundled out of the box with a *JDK* installation.

If the bean server should be accessible from a remote location, too, it is necessary to start the application with additional *JVM* arguments. Extensive documentation exists on the various configuration options, especially on how to secure the access to the server, but a simple and sufficient example is given in Listing 4.3. The given arguments tell the *JVM* to start the *MBean server* on port `9999` and also disable any access checking. As a result, anyone knowing the address of the underlying machine can connect to the server and execute operations or gather information.

```
1 -Dcom.sun.management.jmxremote
2 -Dcom.sun.management.jmxremote.port=9999
3 -Dcom.sun.management.jmxremote.authenticate=false
4 -Dcom.sun.management.jmxremote.ssl=false
```

Listing 4.3: JVM arguments for JMX (insecure)

Summary

Having introduced the key aspects of the *JMX* standard, let us sum up the results. *JMX* provides a very powerful but easy-to-use way to make applications manageable — locally as well as remotely. It comes with an extensive feature set and hardly limits a developer to publish any kind of custom information. Besides that, it also serves as a mean to execute functions inside the application without knowing anything about the underlying code base or mode of operation. Therefore, when thinking about exposing data or methods in a Java application, *JMX* is definitely a very promising way of doing so.

4.2.2. Nagios®

The next one to examine is *Nagios®* [25]. It is a complete monitoring solution but has turned into an industry standard over the years. The main purpose of *Nagios®* is to provide *infrastructure monitoring*. This corresponds to the *availability monitoring* type presented in Section 1.1. The essential requirement is to determine whether systems, single components or applications are available.

Functionality

We first want to describe some of the core functionality of *Nagios®*. The core of *Nagios®* does not provide any means of collecting data or performing checks at all [26]. These tasks are delegated to *plugins*. Plugins are programs — compiled executables or scripts — that are run by the *Nagios®* core. They can operate in any way they want, only the returned values are then evaluated by the core. Therefore, the plugins serve as an abstraction layer away from the details of the monitored entity (Figure 4.5).

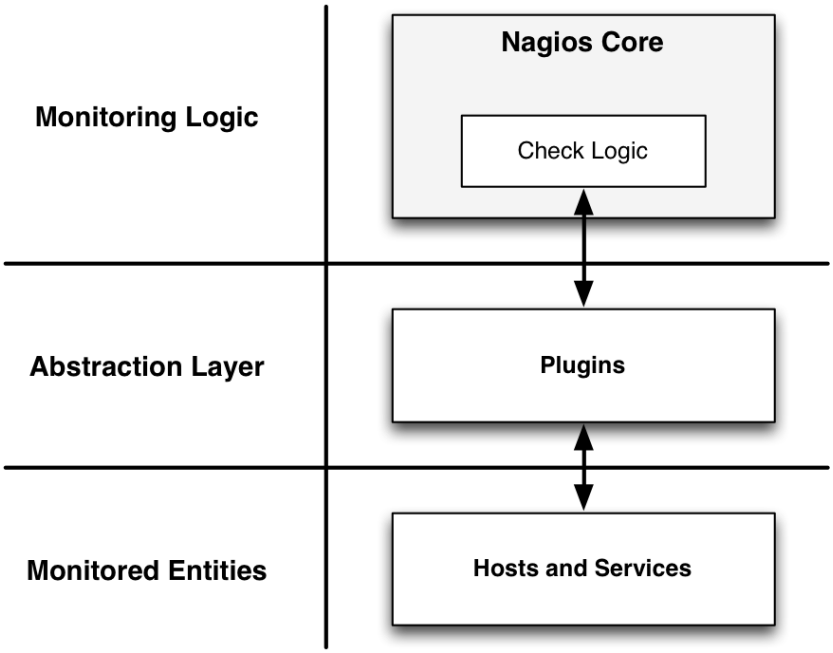


Figure 4.5.: Nagios® plugins abstraction (after [26])

The core only contains the *check logic*, i.e. the code needed to run the plugins and evaluate their results. The plugins in turn know exactly the *monitored entity* they are responsible for. They are designed to gather the values for one specific type of *host* or *service*. Thus the abstraction is created since *Nagios®* has no notion of what systems are monitored. These can range from the temperature of a fish tank to the speed of a CPU fan.

Concerning the monitored entities, *Nagios®* differentiates in *hosts* and *services* as shown in Figure 4.5. A host refers to a part of the infrastructure required to make a *service* available [1], i.e. hardware. Examples are a computer or a network switch. On the

other hand, *services* are most often applications which run on the infrastructure [1]. This could be an Apache webserver or a DNS server. In general, *services* depend on the underlying hardware in order to function properly. *Nagios*® offers the ability to model these dependencies in its configuration by assigning *parents* to entities [1].

To determine the state of an entity, so-called *checks* are performed. These are also separated into *host checks* and *service checks*. The state is then calculated out of the results of every plugin executed for the entity [26]. Four different results are possible: *OK (0)*, *WARNING (1)*, *CRITICAL (2)*, and *UNKNOWN (3)*. The *UNKNOWN* result is used to signal a problem during the execution of the plugin, e.g. due to wrong parameters. These return values are then mapped to a state. For *host checks*, three different states are available: *UP*, *DOWN*, and *UNREACHABLE*. The first one is used to express that the host is available. A host is *UNREACHABLE* if all of his parents are *DOWN*. If at least one parent is *UP*, the host is deemed *DOWN*. For *service checks*, the return value is directly used as state of the service.

Example

To better explain how everything works together, we look at the following example adapted from [1]. An overview of the monitored infrastructure is given in Figure 4.6. The *Nagios*® system is running on the computer labeled **Nagios**. Furthermore, there is a **server** with an **Apache** service running on it. The **Apache** service depends on data stored on the **Files** server. Consequently, **Apache** also depends on **switch_1** to be able to connect to **Files**. Therefore, **switch_1** is a *parent* of both, **server** as well as **Files**. Last, there is a **router** connected with the **server**.

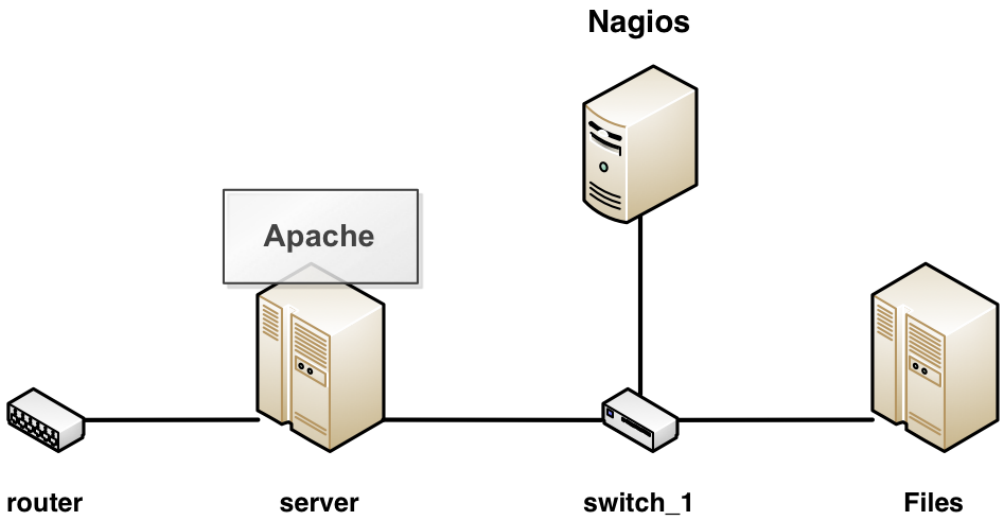


Figure 4.6.: Nagios® example infrastructure

We now describe how **Nagios** performs a service check on **Apache**. In our example, **switch_1** is currently not working.

1. The plugin to check **Apache** is executed. The return value *CRITICAL*, i.e. the service is not available. However, the reason why the service is not available is so far unknown.

2. *Nagios*® then tries to check if there is a hardware problem. Therefore, it checks the service's *host*, in our example **server**. Since **switch_1** is not up — what *Nagios*® does not know yet — the check will fail.

3. As the next step, the parent of **server** is checked: **switch_1**. Of course, the check fails. Due to the fact that *switch_1* has no more parent, its state is set to *DOWN*.

4. Now, the dependency structure is evaluated again. Knowing that **server** depends on **switch_1**, the state of **server** is set to *UNREACHABLE*. Even more so, **router** and **Files** also are set to *UNREACHABLE*, since they transitively depend on **switch_1**.

The end result of the process is illustrated in Figure 4.7. The numbers correspond to the order in which the steps were executed. Furthermore, for every entity its final state is given.

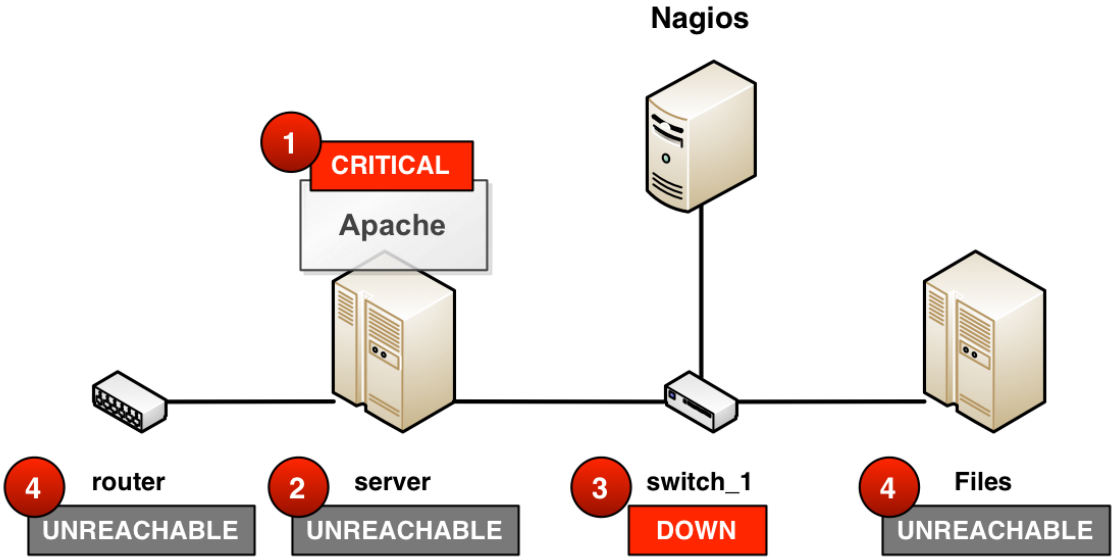


Figure 4.7.: Nagios® example result

Summary

During the examination of *Nagios*®, it becomes quite clear that the sole focus lies on monitoring if the infrastructure is *available*. Though *Nagios*® provides plugins with the ability to return additional performance data, this is not processed in any way by the core directly [26]. For further processing, various plugins have been developed requiring a specific format of the performance data output. One of them is *PerfParse* [6]. In the context of Java, two ways are possible to integrate an application in a monitoring system. The first

is to use one of the plugins for the JMX standard like *check_jmx* [24]. The second option would be to implement a custom plugin which then handles the checking of the target application. Overall, *Nagios*® is of limited use for employing thorough *APM*.

4.2.3. collectd, Cacti, and the like

Apart from the two major *standards* to consider there are countless solutions out there. Though most of them are tools, they all have different protocols and concepts concerning the collection of data from applications. That is why we decided to include them in this section. Two of them are *collectd* [12] and *Cacti* [37] which have already been mentioned in Section 4.1.4, another one is *Munin* [38]. We do not want to describe any of them in more detail here but focus on their common aspects.

A lot of them use a round robin fashioned way to store the collected data in the background and offer similar capabilities regarding the evaluation of information. All of the ones we looked at offered a plugin architecture making them extensible. Moreover, with respect to Java applications and thus the context of this thesis, all of them included plugins for *JMX*. These were either directly integrated in the solution or created by the community. Just as with *Nagios*®, a custom plugin could also be created.

All in all, some of those solutions are suitable to do *APM*. However, creating a custom plugin for one of them or, in the worst case, all of them is generally not an option. This would yield a very high development and maintenance effort to be required. But, as mentioned, all of them include the capability to monitor *JMX*-enabled applications. This is a substantial argument towards the decision on the used standard.

5. Interview

Before finally advancing to the practical part of this thesis, we wanted to gain an insight how others use *APM* for real applications. Therefore, we conducted an interview with Axel Wienberg of *CoreMedia* [7].

Axel Wienberg is *Product Owner Core / Search* at *CoreMedia*. He elaborated the area of monitoring for their product *CoreMedia 7 (CM7)*. The product is a very flexible *Web Content Management System (WCM)*. The main goal of *CM7* is to enable companies to deliver rich customer experiences by leveraging the creativity of online marketing teams. *CoreMedia* has a variety of very large and well-known customers from different industries. These include television companies like *ARD* and *ZDF* as well as telecommunication companies, e.g. *GMX* or *O2*.

The first question was whether *CM7* includes monitoring for important metrics. The answer was they employ monitoring at a lot of places but they do not store any types of monitoring data at all. The aggregation of data is not one of their key competencies. Nevertheless, they provide interfaces in their applications so others are able to acquire the collected information. Furthermore, *CoreMedia* tries to stick to architectural standards in order to facilitate the collection of data for other tools as far as possible.

We then wanted to know how the monitoring data is gathered in the application and afterwards exposed. According to Mr. Wienberg, a lot of metrics are gathered using the *Metrics* framework [16]. To then expose them for collection, primarily two different ways are used. One is the already introduced *JMX* standard, which is employed for the majority of metrics. Apart from that, a significant amount is written to log files in various data formats. A very important statement for us was that they plan to streamline their interface towards *JMX*.

In response to the question, how hosters or customers have access to the monitoring data, the core answer was: he must be flexible. This is due to the fact that the data is available over numerous different channels, e.g. but not only *JMX*, log files. However, that is not a problem. To get detailed information on hard drives, CPU, or *Apache* webserver data, the hoster/customer has to consolidate different sources of informationen anyway. Concerning *JMX*, a lot of monitoring tools they had contact with already include plugins for collecting data via *JMX*. A wide range of existing Java software like the application server *Tomcat* uses *JMX* to expose information. Among the monitoring tools *CoreMedia* has experience with were *Nagios*®, *Munin*, and *Cacti*.

The last aspect was how *CoreMedia* detects performance problems and how they react on problems reported by customers. *CoreMedia* uses a lot of *profiling* during the development process. The purpose of profiling has already been explained in Section 1.1.3. Especially during load testing they monitor the application extensively and collect data themselves. The data is then evaluated using tools like *Zabbix* or *Logstash*. If a customer reports issues, they ask him for the corresponding log files and request thread dumps. After that these are analyzed to find the cause of the problems.

Finally, Axel Wienberg had some recommendations for us. First, we should try to focus on a preventive solution instead of doing post mortem analysis. This means to detect problems before they lead to a failure of the application. Furthermore, he strictly advises to separate the analysis and collection of monitoring data from the monitored application. Otherwise the performance of the application can be influenced negatively or analysis might be impossible if the application fails.

By conducting the interview we received very viable input. Especially the aspects concerning *JMX* and the separation are considered in the development of *APM* in *Tricia*.

Part II.

Employing APM in Tricia

6. Design Decisions

We have now laid out the groundwork to proceed to employing *Application Performance Monitoring* in *Tricia*, an existing Java application. Now we take a look at the current state of performance monitoring in *Tricia*. Afterwards, it is necessary to identify what can be applied in our context before we present the main aspects of the planned monitoring solution as well as justify our decisions.

6.1. Application Performance Monitoring in Tricia

Currently, *Tricia* already includes some very basic performance monitoring utilities. The already presented framework *Java Simon* (see Section 4.1.3) is used to track the execution times of numerous tasks. Examples of places where monitoring takes place are the overall response time the system takes from receiving an incoming request to delivering the output as well as monitoring the time it takes to run a database statement. Due to its very small footprint, these *Simon* monitors are all over the application's code and are added by developers frequently.

Concretely, there are three different means of monitoring available in *Tricia*: *performance snapshots*, a *performance log*, and a *performance trace*. *Snapshots* are created regularly at an interval of two hours. They contain a list of all of the application's *Simon* monitors with their corresponding total, average, minimum and maximum execution time as well as their hit count, i.e. how often they were used. For every *snapshots* an *Excel* worksheet is created in a specific file. Using this solution, it is possible to roughly track where the application spent most of its time over the monitored range. Furthermore, the *performance log* contains the response time for every single request *Tricia* received in combination with the associated URL. At present, this file is evaluated by parsing it with *Excel* and serves as an indicator of how fast the application is reacting. It also provides a fast way of giving a customer feedback if he reports that *Tricia* is running slowly. The last one, *performance traces* are exactly the same as the ones presented in Chapter 3. If the execution of a request in *Tricia* takes longer as a predefined, static threshold, a trace is written to the main log file. It contains all *Simon* monitors which were triggered during the processing of the request together with their respective total time. Thus, a developer can find out the main reason for the experienced slowdown, i.e. the specific monitor whose wrapped task was responsible for the problem.

Despite the fact that there is a lot of information theoretically available, its evaluation proves very difficult. *Tricia* provides no easy way of accessing the data — apart from the *performance snapshots* which can be visually interpreted by leveraging *Excel*, a developer has to dig into log files to find the provided information. This makes it very difficult to get an overview of the system's state quickly. As a result, the gathered data is hardly used to solve problems or serve as a basis for analysis.

It would of course be best to continue using the already provided data and monitoring facilities as much as possible, adapting and extending them only where needed. Nevertheless, we take a look at the other solutions available.

6.2. Solution to be Developed

The first decision to make is whether to create a custom monitoring tool or to use an already existing solution. As presented in Chapter 4, the only end-to-end solution suitable for this task is *New Relic*. We already know one of its major drawbacks — it requires a constant connection to the *New Relic* servers where the gathered data is transferred to. However, this is a requirement which cannot be fulfilled in every cloud environment or company intranet. As an example, *Tricia* is deployed in the *Business Marketplace* of *Deutsche Telekom AG*, where applications are hosted in the cloud for customers to be available as *SaaS*. In this environment there are very strict security restrictions for the hosted applications especially concerning data security, and the transfer of lots of data would not be permitted. On the other hand, a *Tricia* license can be acquired by customers for local deployment on their infrastructure. In such a case it may be possible that the deployment target is inside a company intranet without further internet access. Nevertheless, it must be feasible to monitor the application — which could not be done using *New Relic*. This leads to the final conclusion that *New Relic* is not sufficient to do *Application Performance Monitoring* in combination with *Tricia*. As all other existing libraries and frameworks are no end-to-end solution, the development of a custom monitoring solution is required. This gives rise to more questions we need to consider.

6.2.1. Separate Monitoring Tool

Another question is the separation of a monitoring tool and the application itself; more precisely: should there be a monitoring tool outside of *Tricia* itself or should monitoring be integrated in *Tricia* completely? During the interview we conducted, it has already been strictly recommended to separate the monitoring from the monitored application (see Chapter 5). This becomes clear when considering the drawbacks a fully integrated solution has. Let us assume, monitoring is wholly done inside *Tricia*. If *Tricia* now fails completely and the server cannot deliver any further requests, it is impossible for support to connect to it, too. Therefore, no examination of the problem using the employed monitoring would be possible. Furthermore, the analysis of gathered data requires system resources. Doing the analysis on a live system could for that reason lead to even more performance problems, since resources are not available for other tasks. Clearly, the analysis of any information is to be done outside of the application.

6.2.2. Providing Data

Since collection is done outside of the application, *Tricia* has to either provide all performance data to the separate monitoring tool or *Tricia* only exposes single metrics and storage is handled by the tool itself. Once again, we want to bear in mind that the monitoring solution should be separated from the monitored application. This principle demands to

provide only single data points and let the monitoring tool do the collection as well as retention. There are also other advantages using this solution. One of the requirements is the integration into an existing monitoring solution of a cloud provider. We already came to the conclusion that this demands the implementation of standards. However, such a solution does data collection by itself, requiring applications to only expose single values, too. Therefore, if data storage was implemented inside *Tricia*, it would nevertheless be necessary to have an interface for that task.

That is why we decided to make *Tricia* provide discrete data points which are to be collected and stored outside of it. Furthermore, this interface is implemented using a specific standard we will determine later on. This way, the possibility to integrate the custom solution in an existing environment is ensured.

6.2.3. Data Evaluation

Apart form collection and storage it is of course necessary to be able to evaluate the monitoring data. Once again, the defined separation of *Tricia* and the tool demands the analysis part to be included in the latter. Furthermore, this setup is a big advantage. If *Tricia* and the analysis are independent, it can be done on totally different machines. As a consequence, a developer could initiate the collection of monitoring data in the target environment, then transfer the collected information to his own machine and start a thorough analysis without affecting the monitored system in any way.

6.3. Inconvenient Frameworks

Apart from *New Relic*, there are already some frameworks available which have the goal to aid in the employment of *Application Performance Monitoring*. Given the above results of the first design decisions, not all of them can be used in the context of *Tricia*. It would be very inconvenient to employ *Kieker* in our situation, which will be explained in this section.

Kieker is the flexible and extensible framework developed at the *Christian-Albrechts-University Kiel* (see Section 4.1.2). To use it in *Tricia*, the collection of performance data would have to be re-written from the ground up. Currently, *Java Simon* is already used in *Tricia* to do basic performance monitoring (see Section 6.1). As *Kieker* has its own custom data type for storing monitoring information, namely *Monitoring Records*, the existing *Simon* monitors must either be wrapped into these or completely replaced. These changes are not only very time consuming but also represent a high risk concerning the stability of the software. The code of *Tricia* has to be modified at a lot of different places yielding the possibility of creating new bugs and requiring a severe amount of additional testing and verification.

Moreover, we already explained that *Tricia* has to provide a public interface which conforms to a certain standard. At the moment, there is no standard dealing with custom *Monitoring Records* as required to use *Kieker*. Thus, all records have to be converted back to another format at the interface and the conversion must be implemented in *Tricia*, too. The data collection will also be done outside of *Tricia* leading to another drawback. To further use *Kieker* for the evaluation of the collected information, it must be stored as the already

mentioned records. This requires another conversion — *Tricia*'s interface exposes a simple data format as defined by a standard and now, to store everything, once again *Monitoring Records* need to be created. All in all, this would mean a total of at least *two* conversion steps resulting in a tedious and error-prone process.

In addition to this, many of *Kieker*'s advanced analysis features like the generation of UML sequence diagrams or dependency graphs is not primarily required by the *Tricia* developers. Their main goal is to be able to analyze a performance problem in the application and solve it.

Summing up the results, the introduction of *Kieker* inside *Tricia* would require a high amount of work and present a profound change with the likeliness of new bugs. Even more so, there need to be at least two different conversion steps between the measuring itself and the final storage of the collected data. Consequently, *Kieker* was deemed inconvenient in our context and is not part of the solution.

6.4. Employed Standard

We already postulated the need of an interface in *Tricia* to expose the gathered monitoring data so that it can be collected by the monitoring tool as well as already existing monitoring solutions, e.g. in a cloud environment. Especially the latter situation requires the usage of a common standard in order to *Tricia*'s information being accessible.

In Section 4.2, we presented some different but well-established standards which are heavily used. Apart from *JMX* (see Section 4.2.1), all of them are tools which have a specific protocol. One way to feed those tools with data from *Tricia* is to create an interface specifically using one of these protocols. Since the implementation of more than one interface would extend beyond the scope of this thesis, the separate monitoring tool consequently also has to use this. A major but easily recognizable drawback using this approach is that for every monitoring solution a custom interface must be provided. However, there is an alternative — the use of *JMX*. All considered solutions — *Nagios*®, *collectd*, etc. — are able to collect data exposed by *JMX* either directly or via some plugin. Therefore, *Tricia* only needs one interface for integration in various monitoring solutions to be possible at the same time. Another advantage of using *JMX* is the effect on the planned monitoring tool. As a result, it is able to not only collect data from *Tricia*, but — depending on its implementation — to collect information from any kind of application exposing its state via *JMX*.

Clearly, there are a lot of benefits if *JMX* is employed. In addition, its ease of integration in an existing application facilitates a corresponding implementation. As for this thesis we therefore decided to go with *JMX* as the interface standard.

6.5. Intermediate Results

Before advancing to developing the architecture, we want to consolidate the outcomes of the evaluation process so far. The key points can be listed as follows with their respective justification given above in this chapter:

- Development of a new, custom monitoring solution — no existing end-to-end solution is suitable in our context.
- Creation of a separate monitoring tool outside of *Tricia* responsible for three tasks: data collection, retention, and analysis.
- Implementation of an interface in *Tricia* to expose the gathered monitoring data via *JMX*.

The items are also illustrated in Figure 6.1. Based on this list of fundamental decisions, we now start to create the solutions architecture in the next chapter.

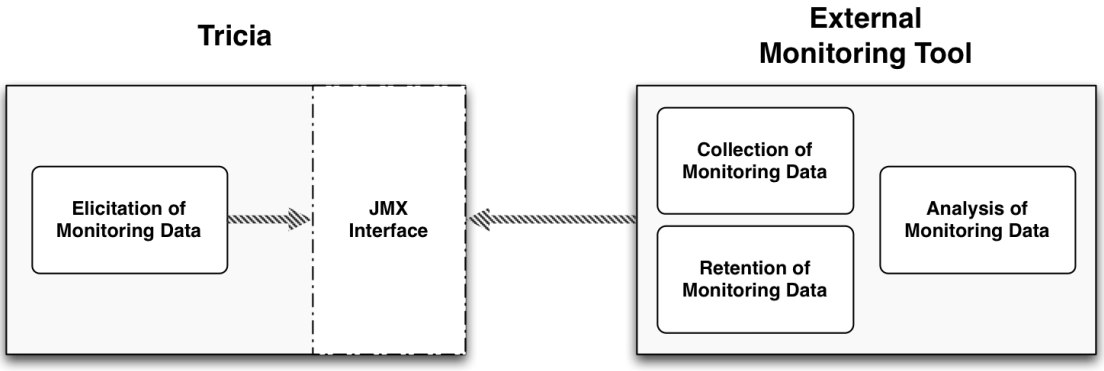


Figure 6.1.: Graphical Overview of Intermediate Results

7. Architecture

The key aspects of the monitoring solution's design have been worked out and it is now time to create its architecture. All main decisions were summed up in Section 6.5 and Figure 6.1 illustrated them graphically. In this section we first focus on developing the complete architecture for the monitoring tool and afterwards specify the elicitation process of monitoring data inside *Tricia*.

7.1. Monitoring Tool Architecture

In order to completely specify the separate monitoring tool's architecture, all three parts — *collection*, *retention*, and *analysis* — must be further dissected and defined. This is done in this section as well as the creation of the final architecture.

7.1.1. Data Collection

The first part to tackle is the collection of monitoring data using the *JMX* interface provided by *Tricia*. Since *Tricia* exposes only single values which will be updated on a to be defined basis, it is necessary to regularly collect and store them. This is also the same way the *JDK* included tools for managing applications via *JMX* work, namely *jconsole* and *jvisualvm* (see Section 4.2.1). Especially *jvisualvm* offers the capability to visualise numerical values in a small graph as shown in Figure 7.1. To create such a graph, the process polls new values every *5 seconds*.

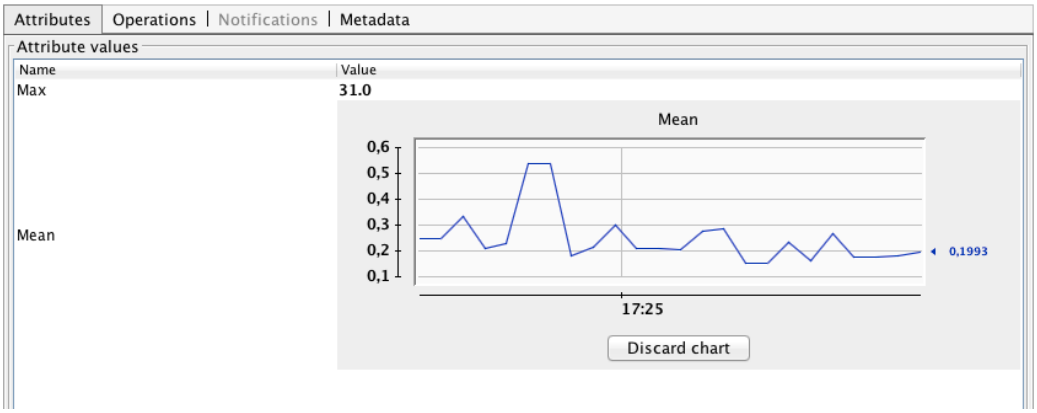


Figure 7.1.: Graphical presentation of values in *jvisualvm*

This is a reasonable interval to use since it is not too long to be useless for an analysis of a certain point in time, as *1 hour* would be. Additionally, it is not too short to produce a significant overhead and affect the system's performance. Keeping a low performance

overhead is one of the key requirements for the planned solution since *APM* is done in a non-intrusive manner (see Section 1.1).

The overall process of collecting monitoring information can thus be outlined as follows: Upon startup, the monitoring tool connects via *JMX* to *Tricia*. Then, a task is scheduled to regularly collect the exposed values every *5 seconds*.

7.1.2. Data Retention

When talking about the collection of information, we also have to decide on an appropriate retention strategy, i.e. how to handle the storage of the gathered data. This is also a very crucial point since monitoring an application can produce a very large amount of information over time.

One way would be to store everything incrementally in a database or file, for example. Thus, new values always lead to the creation of a new database entry or are appended at the end of the designated output file. Using such a mechanism, it is possible to make a very large time range available for analysis. However, this potentially requires a lot of storage space since more and more values have to be kept. In this case it would be necessary to manually implement some kind of clean-up task, which deletes old and no longer required entries. In such a scenario the database could be provided as a local file by *SQLite* for example, i.e. no *MySQL* or other server has to be available.

On the other hand, there are libraries to retain numerical data in a *round-robin* fashion like *RRDTool* (see Section 4.1.4). This means that one defines a fixed time range for which the information should be kept. The library then creates a database file with a fixed size and starts filling it with values. Once all free slots have been filled, the oldest entries are automatically overridden to ensure keeping the database's size fixed. One drawback is that the number of different metrics for which data points are collected has to be defined before creating the initial database. More details on this are described in Section 4.1.4.

Analysis and data storage should be handled outside of *Tricia* and therefore no direct access to its database is possible. Furthermore, it is hardly possible to provide access to a database server for use by an additional monitoring tool in every deployment scenario. Creating an additional database just to be able to store monitoring information represents a high amount of work and requires additional information. That is why only a file-based solution is qualified to be used for the planned tool. In the context of *Tricia*, only numerical values are to be collected, and as the amount of used memory should be as low as possible, using a *round-robin* approach with *RRDTool* is the most practical way. As a result, the process of cleaning up old data is handled by the respective library, too.

7.1.3. Data Evalutation

The last part to be defined is the evaluation of gathered data. This should be possible without the need of collecting information at the same time, i.e. in the ideal case all data to be analyzed is put at a specific place and then the evaluation process is triggered. Furthermore, this has to be very user friendly to facilitate the developer's task of finding problems and results.

Creating a Java application for visualizing the information is quite a tedious problem whereas the creation of a web-based solution using technologies like HTML, JavaScript

and CSS and a server to provide the required data is more feasible and more adaptive. There are especially very feature-rich JavaScript libraries available to help in visualizing data. Therefore, in favor of having a solution relying on very common technology as well as being very adaptive, the user interface will be web-based in combination with a web server to provide the needed information.

7.1.4. Combining Results

Combining all decisions and detailed considerations above, we are now able to create the monitoring tool's architecture. The result is shown in Figure 7.2. The resemblance to the outlined structure of the basic architecture as presented in Figure 6.1 is clearly visible. The two parts of *data collection* and *data retention* were composed in a single *Collector* component. This component is responsible for connecting to a *JMX*-enabled application and query the required data. The gathered information is then stored, which is presented in Figure 7.2 as a *data bundle*.

The second main component, the *analyzer*, is located in the tool's web server part. It is accessible via a browser over the server and operates on a just mentioned *data bundle*. The *analyzer* can extract the collected information from a bundle, pre-process it and then create corresponding visualizations for evaluation. Furthermore, a *live view* component is included — also accessible via a browser — to observe the current values delivered to the collector by the monitored application. The *live view* is also the only component in the web server part to be connected to the collector. Keeping a very loose coupling between those two parts enables the *analyzer* to be independent from any internal collection process.

7.2. Tricia Adaptation Architecture

Apart from designing the separate monitoring tool, adaptations have to be made for *Tricia* to fully gather the wanted monitoring data and expose it via *JMX*. We first discuss how the collection of data inside *Tricia* is handled and afterwards explain the use of *JMX* to finally create a standardized interface. An overview of the components in *Tricia* is given by Figure 7.3. The components are further explained in this section.

7.2.1. Sticking to Java Simon

In Section 6.1 the current state of *APM* in *Tricia* was presented including the use of *Java Simon*. As this framework provides us with the two required measuring tools, *stopwatches* and *counters*, but also creates a hierarchical structure automatically aggregating values (see 4.1.3), we continue to use it for gathering metrics in *Tricia*. This is shown in Figure 7.3 by the gray *Java Simon* component.

7.2.2. Monitor Handling

One requirement we already identified was the ability to create a performance *trace* (see 3.2), i.e. it should be possible to see for one request how long certain tasks took to complete. Thus, some kind of inventory is needed, where the execution time of all *Simon* monitors is registered and tracked for one single request (see *Monitor Inventory* in Figure 7.3).

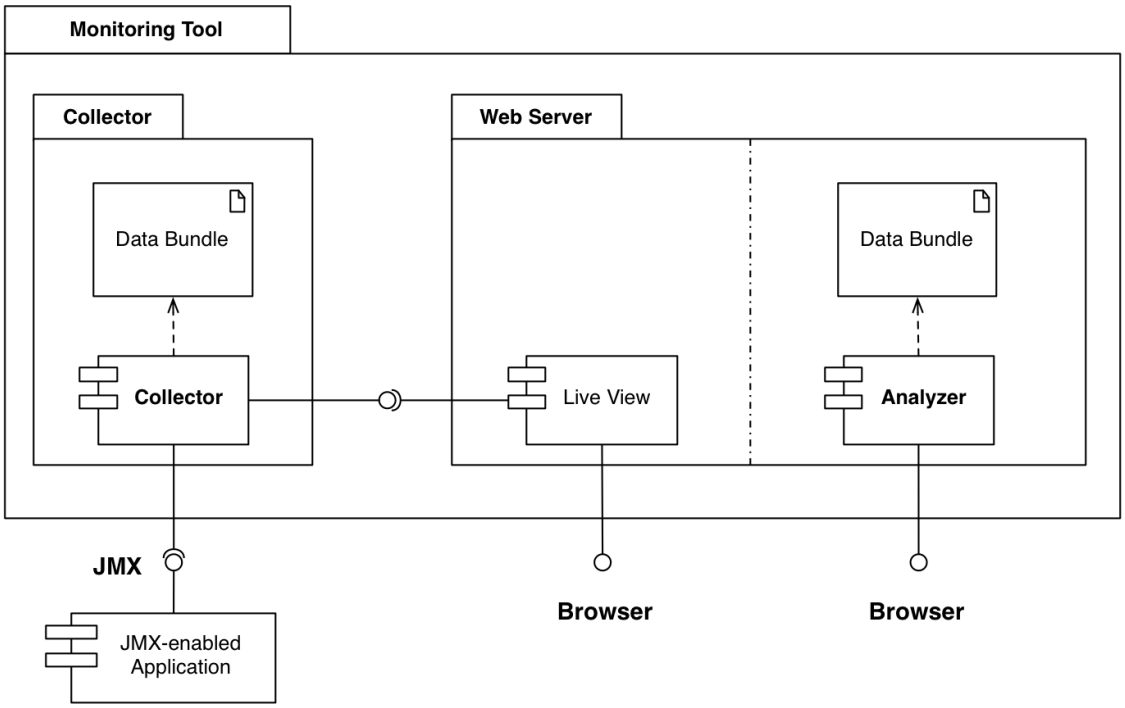


Figure 7.2.: Monitoring Tool Architecture

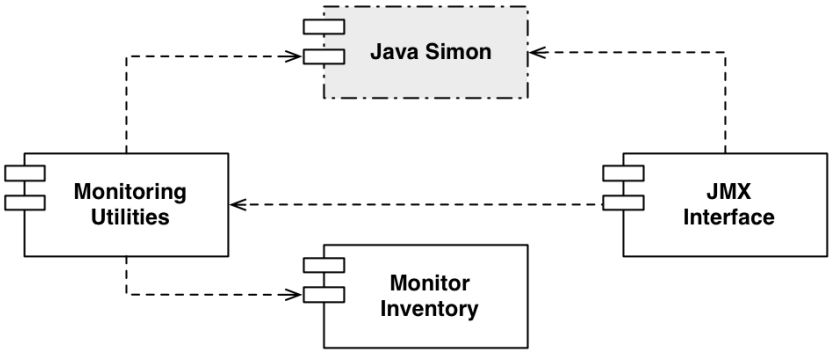


Figure 7.3.: Monitoring in Tricia

Since all triggered monitors are registered at this place, it serves the purpose of handling another feature: the detection of long running tasks. The goal is to capture single tasks whose execution times were over a predefined threshold. If a long running task occurs, a *trace* should be printed so that the source of the problem can be identified, e.g. which subtask was responsible for it. As all monitors are to be registered with the inventory, this is also the place where it is tracked whether one of them exceeded their threshold. When a request is then completed afterwards, the inventory stores the information of an exceeded threshold and *Tricia* can print out an appropriate *trace*.

To facilitate the handling of *Simon* monitors in *Tricia* there is a utility class for starting

and stopping *stopwatches* or triggering *counters*. This abstraction is needed in order to completely wrap the interaction with the inventory and make the use of *stopwatches* and *counters* as easy as possible. The utilities are represented in Figure 7.3 by *Monitoring Utilities*.

7.2.3. JMX Interface

The main aspect to keep in mind when designing the *JMX* interface (*JMX Interface* in Figure 7.3) is that *Tricia* exposes only single data values and does not store any history internally at all (see 6.5). Furthermore, as *JMX* never allows direct access to the underlying *MBeans* (4.2.1), it is necessary to refresh the *MBean* values at an regular interval. This is to be the same as the collection interval of the monitoring tool which we have already identified to be equal to 5 seconds. For gathering all available monitors in *Tricia*, the *JMX* interface takes advantage of the *Simon* hierarchy, i.e. the Java *Simon* framework keeps track of all used monitors and can be queried for them.

Another aspect to be considered is the life-cycle of *Simon* monitors. They are only created and registered upon their first use. Consequently, it is not possible to create *MBeans* to expose via *JMX* for all monitors in the system upon start-up. As they can be triggered more or less randomly, the exposed *MBeans* must be updated at a regular interval, too. However, the refresh interval can be longer than the interval to update single values, since after a few requests almost every monitor in *Tricia* has been used.

8. Implementation

The architecture has been constructed and it is now possible to implement both the monitoring tool as well as the required components inside *Tricia*. In this chapter we present key parts of the implementation, starting off with details on the monitoring tool. Afterwards some in-depth details on *Tricia*'s adaptation are given.

8.1. Monitoring Tool Details

Figure 7.2 in Section 7.1.4 contains the monitoring tool's overall architecture. At first, we talk about some configuration possibilities before advancing to the internal structure of the *collector*. The last part of this section then describes the *webserver* component of the tool.

8.1.1. Configuration

The collector has three purposes: collection, storage and analysis of data. These functions can be configured extensively. It is especially possible to disable not needed functionality. It is for example not required and not always possible to start a collection of information when one only wants to evaluate already gathered data. An overview of the configuration structure is given in Figure 8.1 (*getters* are omitted for clarity).

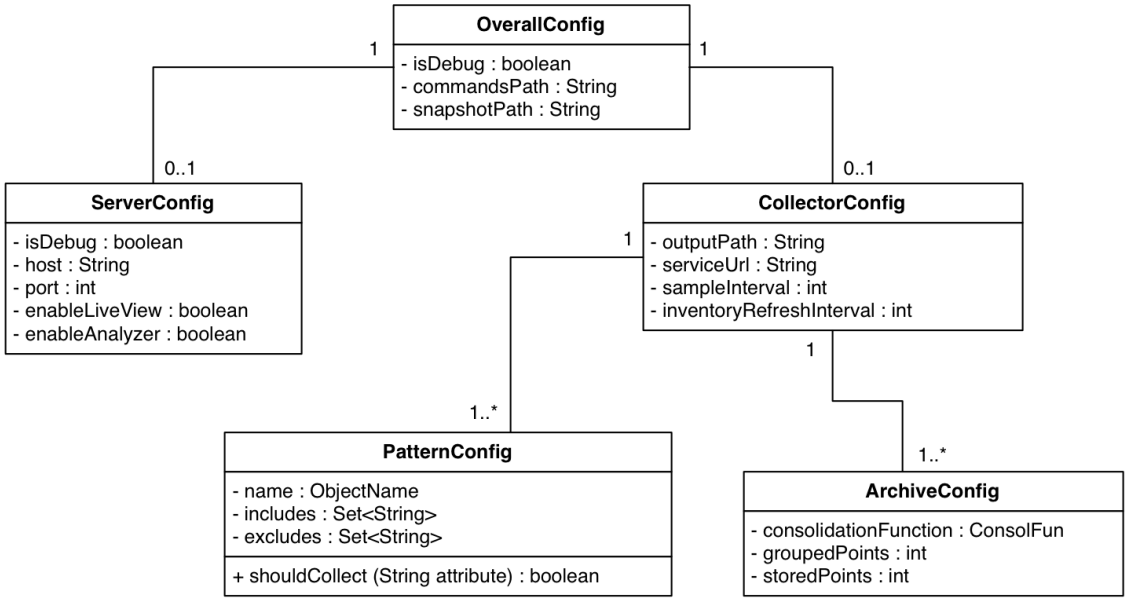


Figure 8.1.: Monitoring Tool Configuration

The main configuration object is represented by the `OverallConfig`. An aspect worth paying attention to is the multiplicity between the `OverallConfig` and the `ServerConfig`, respectively `CollectorConfig`. Both of them are `0..1`, meaning that they are not required. This serves the purpose of enabling only parts of the functionality. To support this, the `ServerConfig` additionally includes two options to enable the *live view* and *analyzer* separately, too (`enableLiveView` and `enableAnalyzer`). As their name implies, the webserver part is configured by the corresponding `ServerConfig` whereas collector configuration is handled by a `CollectorConfig`. Both `PatternConfig` and `ArchiveConfig` are explained in more detail below.

As the tool has to be as user friendly as possible, configuration must be too. Therefore, no complicated *XML* was chosen for its presentation but we decided to use *YAML* [10]. Using *YAML*, one can easily create advanced configurations including features like lists and mappings. A very minimal configuration for starting the monitoring tool is given for illustration purposes in Listing 8.1.

```
1  collector :
2    service : 'service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi'
3
4    archives :
5      - function : MAX
6        grouped : 1
7        stored : 120
8
9    patterns :
10     - 'de.infoasset*:*'
11
12 server :      # Minimal configuration to start the server
```

Listing 8.1: Minimal Example Configuration

The first part of the configuration in lines 1–10 enables the `collector`. The `service` option specifies the *JMX Service URI* of the application to be monitored. In this example, the *JMX* server of the application runs on `localhost` and port `9999`. Furthermore, at least one archive for *RRDTool* has to be specified. Here, an archive storing 120 single data points is created (for in-depth explanation see Section 8.1.2 below). Finally, a pattern is specified in line 10. This causes the collector to gather all beans whose *Object Name* matches `de.infoasset*:*`. In the last line of the configuration, only they key `server` is given without further options. The tool will therefore start its internal webserver on `localhost` and port `9000` with all features enabled.

8.1.2. Collector

The *collector* is used to retrieve data from a *JMX*-enabled application. The collected information is then stored as a *data bundle*. The use of a specific framework for handling storage requires a particular way of handling the *JMX* information which is explained right now. When not stated otherwise, all the work is done by the corresponding class `Collector`.

Data Storage

For data storage, a round-robin fashioned database is used, namely the framework *rrd4j*, which was already presented in Section 4.1.4. Thereby, a data set is identified by its name,

the *datasource*. It would be ideal if the *datasource* was equal to a value derived from the corresponding *MBean* and attribute. Every *MBean* has an *Object Name* (see Section 4.2.1) including a qualified domain, e.g. the containing package name. Assuming a situation as in Figure 8.2, naming the *datasource* `de.infoasset.monitoring.SampleMBean.count` would be sufficient.

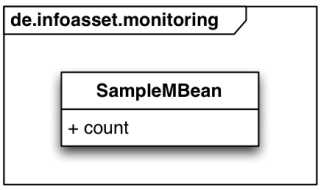


Figure 8.2.: Sample MBean in Package

However, this is not possible due to a limitation of *rrd4j*. *Datasources* can be no longer than 20 characters, whereas in our example we already have a total of *41*. Because of that the *datasources* are generated with a prefix and an incremental number. Since the association to the fully qualified name is important and may not be lost, a mapping between *datasource* and full name is also stored. As final output, a so called *data bundle* is created. It consists of an *rrd* file containing the data values as well as a text file with the mapping (see Figure 8.3).

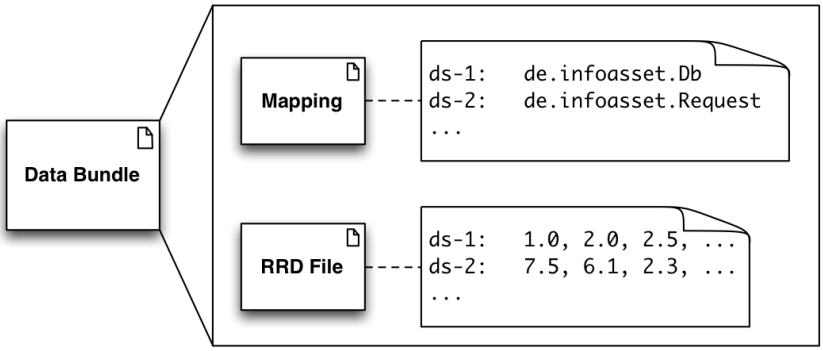


Figure 8.3.: Data Bundle

During operation, it is not recommended to just copy the currently used *rrd* and mapping file since they are actively modified. Copying them may result in data corruption. Therefore, in order to create a bundle representing the current state, the collection process has to be stopped first and restarted afterwards. This is the task of the `SnapshotCreator` class. It is responsible for coordinating and handling the following steps:

1. Stop the running collector.
2. Create a bundle consisting of *rrd* and mapping file by copying them.
3. Restart the collector.

The most integral part of the configuration in relation to data storage is represented by `ArchiveConfig` in Figure 8.1. At least one archive has to be present for the collector to work. As mentioned in Section 4.1.4 on *RRDTool*, there are a few options required for it to work. These are all contained in the collector's configuration and an `ArchiveConfig` maps to an *rrd* archive. Since this is a very complicated issue, we explain it again now using the monitoring tool's terms.

RRDTool requires first of all a *step size*, i.e. the smallest difference in seconds between two sequential updates. In our context this maps to the collector's `sampleInterval`. Additionally, *RRDTool* does not store every single data point right away but needs at least one *round robin archive*. We already know the four required configuration options for an archive: *consolidation function*, *XFF*, *steps*, and *rows* (see Section 4.1.4). As these names are not very expressive, some were renamed in `ArchiveConfig` to better resemble their meaning. *Steps* is equal to `groupedPoints` and *rows* is matched by `storedPoints`. `consolidationFunction` stayed the same and *XFF* was omitted and is set in code to a value of 0.5, i.e. more than half of the grouped points must be valid.

JMX Handling

Apart from data storage, the handling of the *JMX* connection is another one of the collector's tasks. On start-up, the collector automatically tries to establish a connection to the target application or fails and stops. Only once it has been started successfully, it will not abort operation during a disconnect but try to reconnect as soon as possible. This is to ensure a proper configuration concerning the *JMX* setup. Once running, there are two different tasks scheduled at regular intervals (`sampleInterval` and `inventoryRefresh Interval` in Figure 8.1).

One is to keep track of the different *MBeans* and attributes exposed by the monitored application. As we have seen in Section 7.2.3, not all beans are available in *Tricia* immediately. Furthermore, the collector must be configured with patterns of *Object Names* to be collected as represented by at least one `PatternConfig` in Figure 8.1. A pattern consists of an *Object Name* `name` and either a set with attributes to include or exclude (`includes / excludes`). An example for a `name` value would be its string presentation `de.infoasset.*:*` which would match all beans having a domain prefixed with "`de.infoasset.`" like `de.infoasset.monitoring.counter`. Furthermore, not all attributes of a bean are of interest. Therefore, it is possible to explicitly define what attributes should be collected using a set of `includes`, or on the opposite, what attributes not to collect using `excludes`.

The goal is to have a map with unique *Object Names* as keys and attribute names as values. Once this map is established, the collection task can then only iterate over this map to know what values to get. Thus, there is no need to increase the load by querying the monitored application every time. We call this map the collector's *JMX inventory*. The process of refreshing the inventory is executed as follows:

1. For each *Object Name* pattern in the configuration:
    a) Get all unique *Object Names* matching the pattern — a unique *Object Name* corresponds to exactly one *MBean*.
    b) For each unique *Object Name*:

i. Get detailed information on the bean from the monitored application using *JMX*.

ii. For each of the bean's attributes:

A. Check if the attribute name is allowed according to configuration and if it is of a compatible type.

B. If the check is successful, add the attribute to the mapping.

2. Calculate the difference to the old inventory, i.e. find out which beans or attributes were added.

3. Apply the difference to the *rrd* file, i.e. add new *datasources* to it.

The only critical aspect of this process is applying the difference to the *rrd* file. To do this, the collector has to be stopped first. Then all manipulations are executed and afterwards collection can be restarted again. During the manipulation it may occur that the *rrd* file is corrupted. In order to prevent a total data loss a backup is created before any edits are made to the data file.

The other Java task is the collection of the data values. The corresponding time interval should be equal to the one used in *Tricia* to refresh the exposed beans. Therefore, no data values are regularly omitted or collected twice. Using the *JMX inventory*, the collection task contains only the following steps:

1. Create an empty *rrd4j Sample* to later update the *rrd* file.

2. For each unique *Object Name* in the inventory:

a) For each attribute of this bean:

i. Retrieve its value via *JMX*.

ii. Convert the value to a `double` required by the *Sample*.

iii. Get the *datasource* name corresponding to this bean and attribute.

iv. Put the value as update for the *datasource* in the *Sample*.

3. If configured, collect the default system properties using the same procedure as for beans in the inventory.

4. Write the *Sample* to the *rrd* file.

To schedule both tasks, the Java `ScheduledExecutorService` is used [29]. It has one major peculiarity to be considered, however. If at any time a task does throw an exception which is not caught, the task will not be executed again in the future. Therefore it is extremely important to catch all exceptions in both, collection and inventory refreshing. This is done by using a `try-catch` clause surrounding all other code for those tasks.

8.1.3. Webserver Technology

Let us now take a deeper look at the webserver component of the monitoring tool. We first describe very briefly what server framework is used and then advance to explaining the user interface technology. At last the functionality of the analyzer is detailed.

Grizzly

The core webserver is provided by the *Project Grizzly* framework [31]. It is an easy-to-use library offering a rich set of features. In *Grizzly*, requests are mapped to so-called *resources* which are annotated Java classes. By using the provided annotations, a mapping of URLs to specific methods for handling them can be created without great effort. Take Listing 8.2 as an example containing the `OverviewResource`.

```
1  @Path("/overview")
2  public class OverviewResource {
3
4      @GET
5      @Produces(MediaType.APPLICATION_JSON)
6      public Scope getOverview() {
7          ...
8      }
9
10     @Path("/menu")
11     @GET
12     @Produces(MediaType.APPLICATION_JSON)
13     public Scope getMenu() {
14         ...
15     }
16
17 }
```

Listing 8.2: Grizzly Resource Example

By using the `@Path` annotation in line 1, the `OverviewResource` is mapped to the absolute path */overview*. An incoming *GET* request is then handled by the method `getOverview` (line 6). Additionally, the `@Provides` annotation added in line 5 specifies that the returned Java *POJO* is to be converted into a *JSON* representation. Moreover, the resource defines an additional route. As the method `getMenu` (line 13) is itself mapped to the path */menu* in line 10, *Grizzly* creates the overall route */overview/menu*. This is one example of the powerful annotations used with this framework to create a request mapping in very short time.

Two options taken from the `ServerConfig` in Figure 8.1 are important to *Grizzly*: `host` and `port`. They specify the address the webserver binds to, i.e. the base URL where all resources can be reached. Take *localhost* and *9999* as examples for those two options, then the base URL would be equal to *http://localhost:9999*.

User Interface Frameworks

To deliver a rich user interface, two major libraries for web front-end development are used: *Twitter Bootstrap* [39] and *AngularJS* [14]. *Bootstrap* includes a *CSS* framework used for layout and styling, as well as JavaScript components to implement advanced dynamic behavior. *AngularJS* is a complete JavaScript framework providing a *Model-View-Controller* like programming model for developing dynamic applications running in the browser. The key feature of *AngularJS* is its so-called *data binding*. By using specific structures in HTML code, these can be bound to JavaScript expressions. Thus, if a variable is changed in code, for example, the updated value is automatically detected and propagated to the view so the user can see the new state. Since more details are out of scope for this thesis, interested readers are recommended to read the documentation on [13].

8.1.4. Analyzer

Apart from collection, the monitoring tools other key component is the *analyzer*. It provides a visual presentation of the gathered performance data. In this section we explain how it is implemented and what features are provided to the user.

Providing Data

The first step to be completed before anything can be analyzed is to provide the corresponding data. The analyzer operates on a *data bundle* which has already been explained in detail in Section 8.1.2. In order to let the user select a specific bundle to analyze, the analyzer inspects the directory given by `snapshotPath` of an `OverallConfig` (see Figure 8.1). All subdirectories which contain a mapping and an *rrd* file are considered a valid bundle. The user is then represented a list where he can select one to evaluate as shown in Figure 8.4. If the collector is enabled and running, the user is offered the ability to create a bundle from the current data, too. It is then automatically placed in the right directory and shows up as an option in the list of bundles.

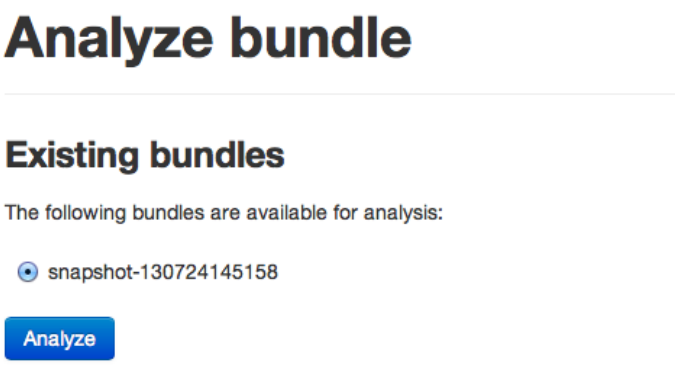


Figure 8.4.: Bundle Selection for Analysis

Creating Graphs

RRDTool and *rrd4j* use so-called *archives* to store their information as explained in Section 4.1.4 and Section 8.1.2. Before any graphs are created, one single archive has to be selected. This situation is shown in Figure 8.5.

For every archive found in the *rrd* file, its consolidation function (**1**), the interval between two values (**2**), and the archive's time range (**3**) are displayed. As soon as an archive has been selected, the user can further restrict the time range which should be visualized by modifying the slider (**4**). This is especially useful to reduce the analyzed data volume and increase performance when one has already identified a certain period wherein interesting events occurred.

The next step is to select the different datasources (Figure 8.6). Once again the *rrd* file is evaluated and with the help of the mapping file the stored sources are represented in a list (**1**). There is also the possibility to search for specific sources using the filter input (**2**).

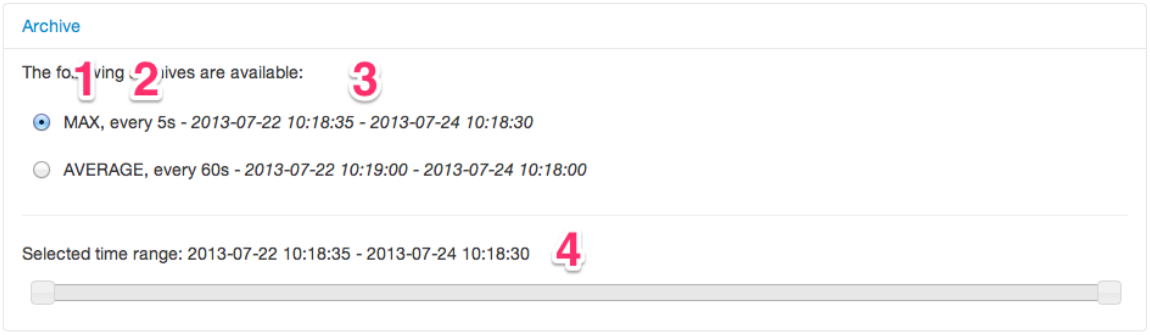


Figure 8.5.: Archive Selection

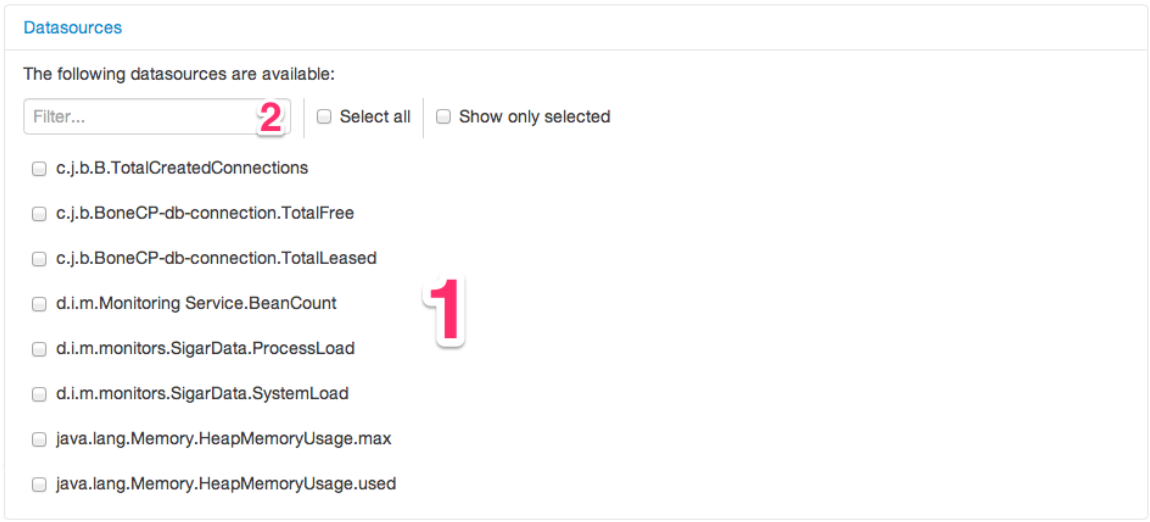


Figure 8.6.: Datasource Selection

When at least one is selected, the user can click a button to start the generation of a graph for each of them.

An AJAX request is sent to the server to retrieve the data to be visualized according to the user's selections. As soon as the results are returned, the values are fed into a *Rickshaw* graph [34]. *Rickshaw* is a JavaScript framework based on *D3* [5], another library for creating visualizations in the browser. Figure 8.7 shows an example with graphs for two different sources:

- `de.infoasset.monitoring.monitors.db.Mean` — The average execution time of a database operation in *Tricia*.
- `de.infoasset.monitoring.monitors.es.Mean` — The average execution time of an *Elasticsearch* operation in *Tricia*.

Since the collector and *rrd4j* have no notion of any units used for the metrics, this is knowledge only a developer has. In our example, both metrics are in milliseconds — therefore it can be seen that the *Elasticsearch* graph has its maximum well above 4 *seconds*.

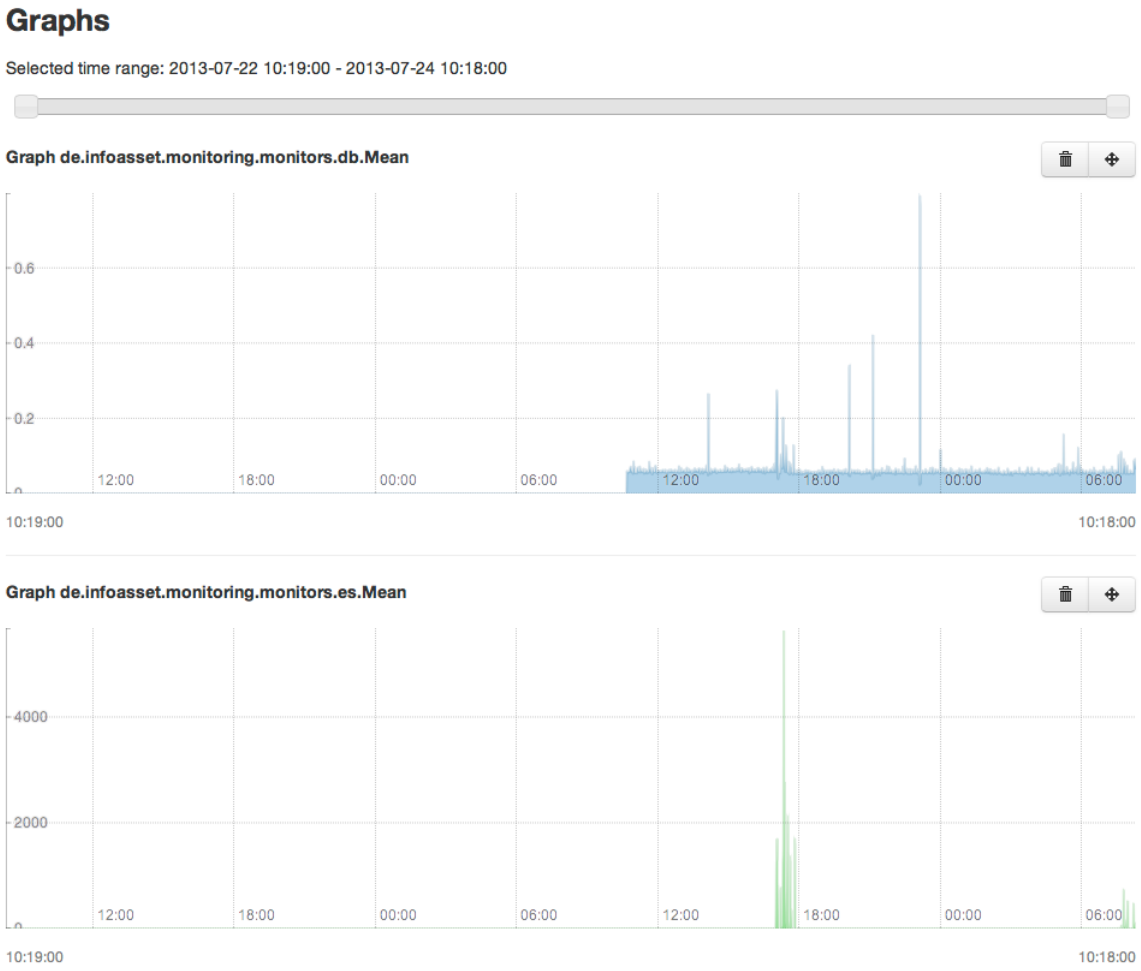


Figure 8.7.: Time Series Visualization

The user is also provided with additional actions. Located above all graphs is another slider which can be used to restrict the time range visible in the graphs. Therefore the user can zoom in on a period of time quickly. Additionally, he can rearrange the order of the graphs as well as delete them if not needed. Upon the deletion of a graph, the corresponding metric is also deselected in the list from Figure 8.6.

8.1.5. Summary

This section gave an overview of integral parts of the monitoring tool's implementation. The tool's main tasks are *collection*, *storage*, and *evaluation*. Evaluation is hereby done by creating graphical visualizations: time series. This fulfills the requirement posed in Section 3.2.4 as it provides an easy way to see if a certain problem occurred over time.

8.2. Adapting Tricia

Besides the development of a separate monitoring tool, there is also the need to adapt *Tricia* itself. *Tricia* has to expose certain metrics via *JMX*. Moreover, the generation of performance snapshots as well as traces — both were introduced in Section 3.2 — are also tasks *Tricia* is responsible for. In this section we now take a look at the implementation required to provides this functionality.

8.2.1. Monitor Handling

In Section 7.2 we proposed the creation of three different components: *Monitoring Utilities*, *Monitor Inventory*, and *JMX Interface*. In the following section, the first two are described in detail. Furthermore, we give examples on introducing a new monitor in *Tricia*.

To begin with, the *Monitor Inventory* is implemented in *Tricia* by an identically named class, `MonitorInventory`. Its class diagram is shown in Figure 8.8. The purpose of this class is to collect all monitors which were triggered during the processing of a request and their corresponding execution times. Since in *Tricia* a request is always handled by one thread, the `MonitorInventory` is designed to be a singleton for each one. This is ensured by the static `get()` method and an internal `ThreadLocal` [29] variable, `inventoryLocal` (see Figure 8.8). Monitors can be added by using the method `addMonitor(...)` and are subsequently stored in the `monitorMap`. It is possible to get all currently stored monitors by calling `getMonitors()`. The inventory also provides another method, `reset()`, which is used to clear the internal map and prepare it for the next request.

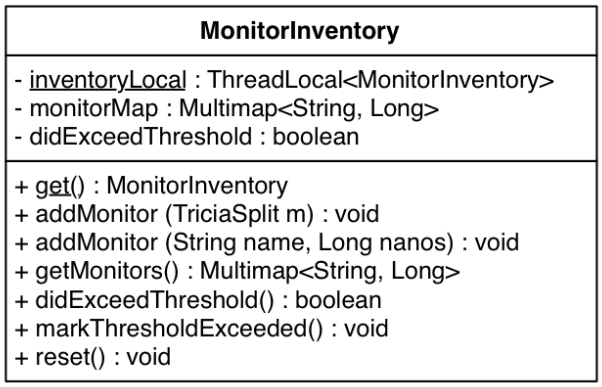


Figure 8.8.: Monitor Inventory

A developer only calls `didExceedThreshold` and `reset` on the `MonitorInventory`. Adding monitors and handling the threshold is all managed by the *Monitoring Utilities*. These are implemented in *Tricia* by the class `MonitoringUtilities`. The respective class diagram is presented in Figure 8.9. Mostly, only the first block of operations is used, i.e. the `start(...)`, `stop(...)`, and `tickCounter` methods. The `start` operations start a new *Simon Stopwatch* used for measuring execution time. It is identified — as all

Simons — by its unique name given as parameter. The overloaded variants provide the developer with additional options: `threshold` and an `expose` flag.

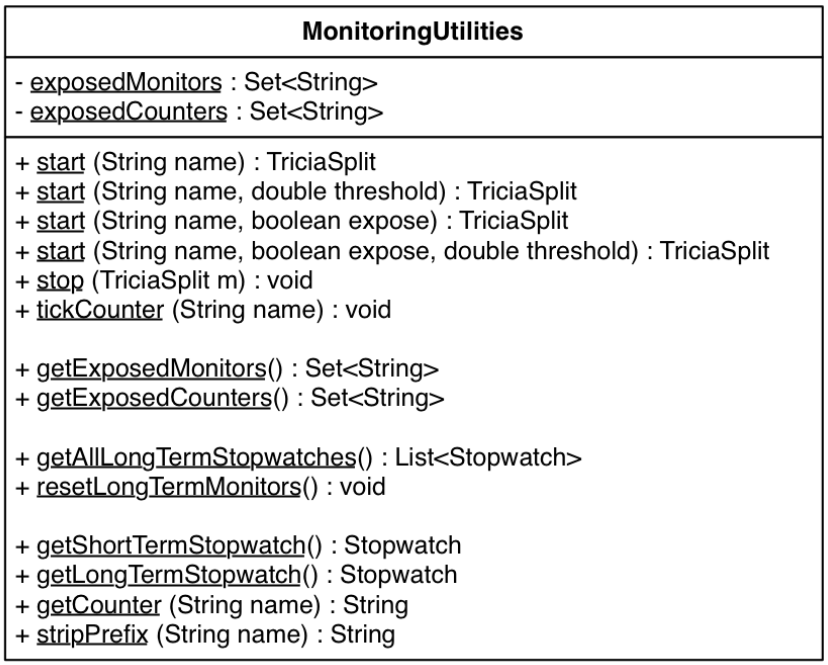


Figure 8.9.: Monitoring Utilities

The `threshold` defines an upper limit for the total execution timer of the monitor. It works in combination with the `MonitorInventory`. When a *TriciaSplit* is stopped by calling the utilities `stop` method, the elapsed time is compared with the threshold. If the threshold is exceeded, `MonitorInventory.markThresholdExceeded()` is called. This is used inside *Tricia* to detect operations which took longer than normal. Additionally, the `expose` flag defines whether a *Stopwatch* should be accessible via *JMX*. More details on how the creation of corresponding beans is handled are given below. If left out in the method calls, the `threshold` is set to *0*, i.e. it is not used , and `expose` is *false*.

Furthermore, `MonitoringUtilities` provide easy access to counters. *Counters* are also offered by the *Simon* framework and identified by their name. In *Tricia*, *Counters* are automatically set to be exposed by *JMX*. The corresponding method `tickCounter(name)` increments the *Counter* identified by the parameter `name`.

One important aspect is not directly visible but shows due to the existence of two helper methods, `getShortTermStopwatch` and `getLongTermStopwatch`. In Section 7.2.3 the conclusion was made that all *JMX* values must be refreshed at a regular interval. On the other hand, a *performance snapshot* is a required mean of detecting problems. A snapshot, however, has to gather values over a longer period of time than the *JMX* refresh interval. Therefore, when a monitor is started by calling the `start` method, two different *Stopwatches* are started. One is used to capture the data for exposure over *JMX* and the other is used in the generation of snapshots. This is necessary since the different *Stopwatches* have to be reset after the values are read from them for their purpose.

8.2.2. Performance Snapshots

Introduced in Section 3.2, *performance snapshots* are an important mean to detect problems in an application. In the context of *Tricia*, such snapshots are created at an interval of 2 hours. A snapshot contains data for all monitors which were used during this period of time. For every monitor, the following values are written to an output file:

- *Name* — Monitor name.
- *Hit count* — How often the monitor was used.
- *Avg* — Average execution time.
- *Total* — The summed up total time of all executions.
- *Min* — Minimum execution time.
- *Max* — Maximum execution time.

For every snapshot, a worksheet inside an *Excel* file is created. This enables developers to quickly check the data, e.g. by sorting or highlighting rows according to rules. In total, 24 worksheets are kept in the file resulting in a coverage of 2 days. As mentioned in the last paragraph of Section 8.2.1, to create snapshots the long term stopwatches are used. After a snapshot has been created, these are reset, i.e. all values are set to 0 again.

8.2.3. Performance Traces

The next important mean provided by *Tricia* are *traces* (see Section 3.2). A *trace* contains the execution time of all monitors which were triggered during a specific operation. Their goal is to enable the developer to identify slow tasks in the program. In *Tricia*, *traces* are generated with the help of two already mentioned features: *threshold* and the *exceeded* flag.

The *threshold* is passed as an argument to the `start` method of `MonitoringUtilities` (see Figure 8.9). This class also handles checking the limit when the corresponding monitor is stopped as described previously. After a request has been processed, *Tricia* currently checks if any monitor has exceeded its threshold by calling `MonitorInventory.didExceedThreshold()` (see Figure 8.8). If this is the case, a performance trace as shown in Listing 8.3 is logged. The trace contains first the request URL and the total execution time (line 2). In this example, the URL was `/requestLoginWithOpenId` and its execution time `1257 ms`. Afterwards, the monitors are listed with their names in combination with *hit count*, *total execution time*, and *average execution time* (line 3 – line 11). The monitors are additionally ordered according to their total execution time. The list of triggered monitors is retrieved from the `MonitorInventory` by calling its `getMonitors()` method.

```
1 2013-06-24 09:26:15,088 [http-connection-67] T: -
2     ------ long taking request: 1257 ms for /requestLoginWithOpenId------
3 webServer.dispatchHandler                    : Hits:  1, Total: 1257.5, Avg: 1257.5
4 webServer.sessionHandler                     : Hits:  1, Total: 1228.8, Avg: 1228.8
5 generic.request                              : Hits:  1, Total: 1228.6, Avg: 1228.6
6 handler                                      : Hits:  1, Total: 1200.1, Avg: 1200.1
```

```
handler.platform.handler.RequestLoginWithOpe...: Hits:  1, Total: 1199.8, Avg: 1199.8
response.platform.handler.RequestLoginWithOp...: Hits:  1, Total:    6.1, Avg:    6.1
response                                       : Hits:  1, Total:    6.1, Avg:    6.1
db                                             : Hits:  1, Total:    1.9, Avg:    1.9
db.getConnectionFromPool                       : Hits:  1, Total:    1.9, Avg:    1.9
```

Listing 8.3: Tricia Performance Trace

8.2.4. JMX Interface

At last we describe the *JMX* interface exposed by *Tricia*. It provides performance metrics for collection by external tools, e.g. the one also developed in this thesis. The first aspect is how *Tricia* knows what metrics to expose. This information is offered by the `MonitoringUtilities` class (see Figure 8.9). For that purpose it has two methods: `getExposedMonitors()` and `getExposedCounters()`.

Per default, a monitor in *Tricia* is not set to be exposed. As a consequence, it is not contained in the set returned by `getExposedMonitors()`. Only when a developer explicitly uses one of `start`'s overloaded variants and sets the `exposed` parameter to *true* (see Figure 8.9), *Tricia* will include it in the *JMX* interface. On the other hand, all counters used in *Tricia* are exposed per default. The set of counters can then be retrieved by calling `MonitoringUtilities.getExposedCounters()`.

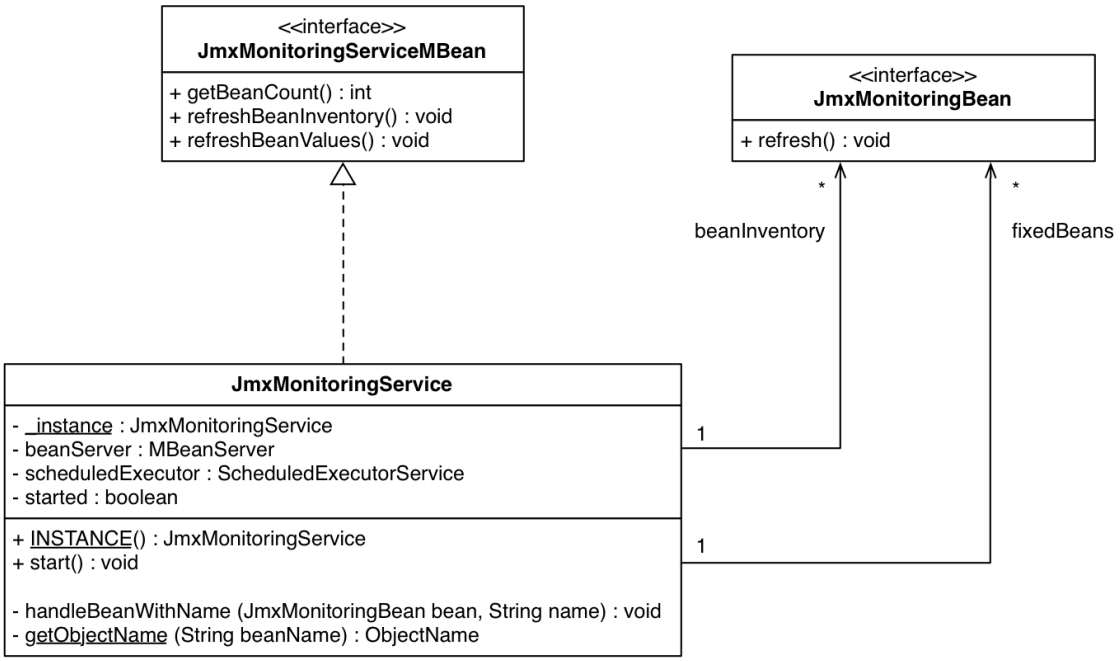


Figure 8.10.: JMX Interface Structure

In Figure 8.10 the basic structure of the *JMX* interface's implementation is shown. The core functionality is provided by `JmxMonitoringService` which will register itself as an *MBean*. To be available as bean, there needs to be an interface ending with `MBean`,

therefore the class implements `JmxMonitoringServiceMBean`. As a result, the interface's operations are available via *JMX*. The monitoring service itself offers the current bean count (`getBeanCount()` as well as two operations: `refreshBeanInventory()` and `refreshBeanValues()`. The former is used to force a reload of the exposed beans and the latter to refresh all beans' values. The service is implemented as a singleton, hence the private static `_instance` variable and the public static method `INSTANCE()` to retrieve it.

Furthermore, the monitoring service has two different sets of `JmxMonitoringBean`s. A `JmxMonitoringBean` is the parent interface for all beans to be exposed and offers a method `refresh` to trigger the reload of the single bean's values. The two sets refer to the different types of beans included in the *JMX* interface. First, there are beans which are exposed at all times. These are contained in the `fixedBeans` role. Once example for a fixed bean is the one providing CPU and memory data, explained in more detail later. Second, beans appear over time and are not available permanently. Those are added to the `beanInventory` role. As stated in Section 7.2.3, *Tricia* monitors and counters are not available from the start on but only when they have been used at least once. Therefore, they are part of the `beanInventory`.

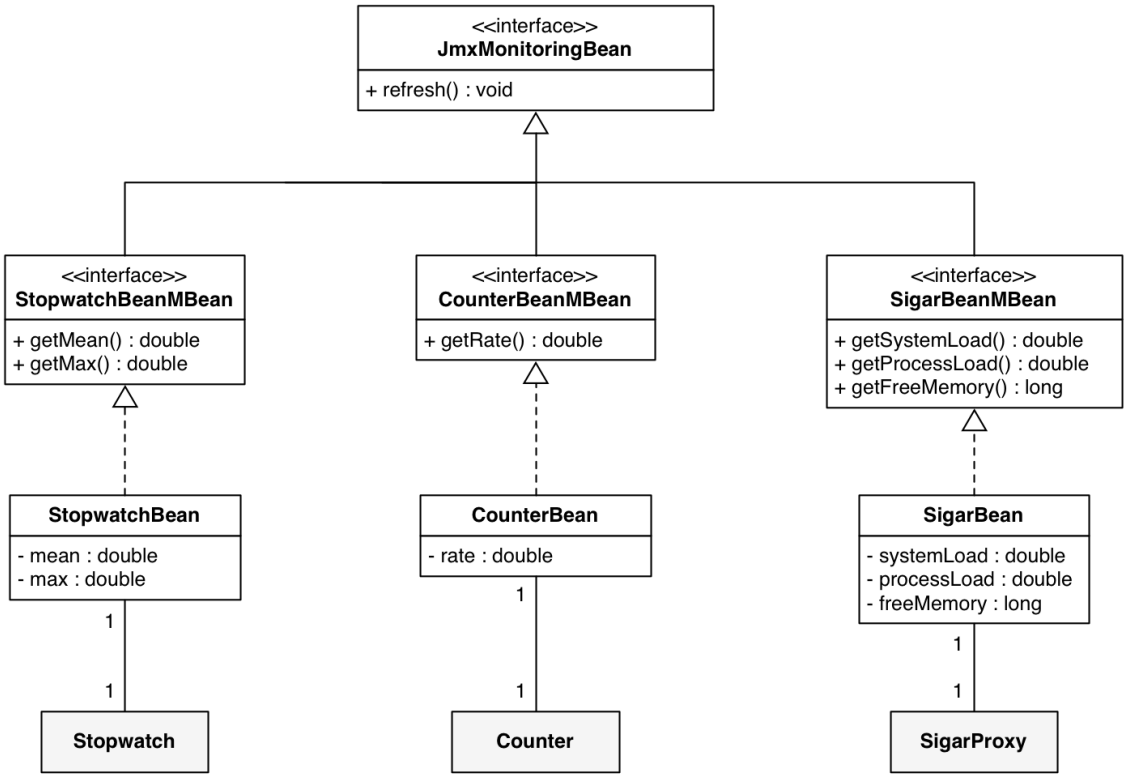


Figure 8.11.: JMX Bean Structure

For every exposed metric inside *Tricia* — monitors and counters — there is one bean. This structure is presented in the class diagram in Figure 8.11. A *Stopwatch*, *Counter*, and *SigarProxy* have corresponding classes and interfaces necessary for *JMX*. Each stopwatch

is wrapped by a `StopwatchBean` implementing the respective `StopwatchBeanMBean` interface. This contains two different fields: *mean* and *max*. Those refer to the average and maximum execution times. In the same way counters are exposed by the `CounterBean`. The respective `CounterBeanMBean` interface defines one value, the *rate*. In *Tricia*, counters are automatically converted into a rate, i.e. how often they were used per second over the last sample period. The last bean is the `SigarBean` with its `SigarBeanMBean` interface. This fixed bean is used to provide external tools with system level information, namely *system load*, *process load*, and *free memory*. These metrics cannot be collected by *JMX* on all platforms reliably, therefore the *SIGAR* library [18] is used inside *Tricia*.

In similar fashion as the collector of the monitoring tool, the `JmxMonitoringService` also has two tasks scheduled at regular intervals. One is to refresh the `beanInventory` (see Figure 8.9). At an configurable interval – per default 5 minutes — the `Monitoring Utilities` class is queried for exposed monitors and counters. If new ones are detected, a matching bean is created and registered with *Tricia's MBeanServer*. As a result, the bean is instantly visible over *JMX*. In the case of missing *Simons*, i.e. a monitor or counter disappeared, the corresponding *MBean* is unregistered from the *JMX* server. The second task is responsible for updating the beans' values. Every 5 seconds the monitoring service iterates over both sets, `fixedBeans` and `beanInventory` in Figure 8.10. For every instance it calls the `refresh()` method defined in `JmxMonitoringBean`. Thus, the provided values are updated and now visible on *JMX*. On every refresh, the bean's monitor or counter is reset. Concerning a monitor, the short term stopwatches are used here.

8.2.5. Code Examples

To round up the section about adapting *Tricia*, we present some code examples on how to employ a new monitor or counter. The easiest thing to use is a counter. Only one line of code is required as shown in Listing 8.4. This call automatically leads to the counter's exposure and its rate will be available via *JMX*. The generated *Object Name* will be equal to the prefix `de.infoasset.monitoring.` appended with the name given as parameter: `de.infoasset.monitoring.db.operations`.

```
MonitoringUtilities.tickCounter("db.operations");
```

Listing 8.4: Using a counter in *Tricia*

Next up are monitors to measure the execution time of code blocks. The pattern should always be the one visible in Listing 8.5. First, there is the method call to start the monitor. The code to be measured is surrounded by a `try-finally` block. The `finally` block then contains the call to stop the monitor again. Therefore, it is ensured that the monitor is stopped in any case, even if an unhandled exception occurs.

```
TriciaSplit m = MonitoringUtilities.start(...);
try {
    // Code to be measured
    ...
} finally {
    MonitoringUtilities.stop(m);
}
```

Listing 8.5: Pattern for using a monitor

The `start` call needs at least one parameter, the monitor's name. The name is used by the *Simon* library to provide another feature: nested monitors. Thus, the execution time of a child monitor is automatically added to its parent monitor. For example, one monitor has the name `db` and the other `db.commit`. As soon as the `db.commit` monitor is stopped, its execution time is also added to the `db` one. Listing 8.6 illustrates this behavior, see the comments for explanation.

```
// Assume both monitors have never been used before
// Total execution time of db:        0
// Total execution time of db.commit: 0

TriciaSplit dbCommit = MonitoringUtilities.start("db.commit");
try {
    // Operation taking 200ms
    ...
} finally {
    // No stop has occurred so far, still both are 0

    MonitoringUtilities.stop(m);

    // Total execution time of db.commit is now: 200ms
    // Total execution time of db is now:        200ms
}
```

Listing 8.6: Illustrating Nested Monitors

8.3. Implementation Summary

In this chapter, implementation details of both the monitoring tool as well as the adaptation of *Tricia* itself were shown. The monitoring tool is completely independent from *Tricia* and only requires a Java application using the *JMX* standard. Along with its powerful configuration options, the tool can therefore be used in a very general scenario. On the other hand, employing monitors and counters in *Tricia* to do performance measurements was kept as easy as possible. Additionally, given the class structure presented in Section 8.2, it is very simple to add new beans to be exposed via *JMX*. Finally, all the required means of detecting and tracing performance problems which were proposed in 3.2 were successfully implemented.

Part III.

Evaluation and Final Results

9. Practical Evaluation

As soon as the monitoring tool was developed and the adaptation of *Tricia* was completed, the solution was deployed by *infoAsset* . It did not take long for a performance problem to show up. *infoAsset* then used the new application performance monitoring to identify the problem. In this chapter the whole process starting from identifying the problem until finding its root cause is described. All steps are illustrated with the appropriate real data.

9.1. Evaluation Context

First of all, it is necessary to describe the relevant context. The problem occurred in the *Business Marketplace* by *DTAG*, one of the cloud environments (see Section 1.2.2). Let us briefly summarize the important facts concerning a cloud environment (see Figure 1.2 for a graphical representation):

- Multiple servers working together.
- One database shared by all *Tricia* instances.
- One *Tricia* as well as one *Elasticsearch* process per server.
- *Elasticsearch* running in a cluster configuration.

The problem itself can be described as follows: Various users of *Tricia* experienced extremely slow performance when creating a new workspace. It took several seconds until the request was completed. There was no initial clue of what was responsible for the problem.

9.2. Identification and Verification

The first step for the developers then was to look at the performance snapshots created by *Tricia*. Since the problem was reproducible and the times when the users tried to execute the operation were known, the corresponding snapshots were identified quickly. Table 9.1 shows the most important monitors along with their *hit count*, *average*, *total*, and *maximum* execution time. The execution times are given in milliseconds, decimal places are omitted.

The first entry, *handler*, represents the execution time of a *Tricia Handler* which is responsible for processing requests. Though the average value is acceptably low with 292ms, the maximum at roughly 86 **seconds** is a hint at a possible problem. Next is the *entity.persist* which measures how long it takes to write an entity to the back-end storage. This includes saving it in the database as well as updating the *Elasticsearch* search index. An average of 1.66s is already noticeably high but the maximum at staggering **154 seconds**, so almost

Name	Hits	Average	Total	Maximum
handler	608	292	177,920	85,877
entity.persist	1,541	1,660	2,559,488	154,076
es	1,886	580	1,094,083	85,545
es.searchables_ production_multitenancy.commit	196	1,161	227,713	84,974
db	221,487	1	317,457	60,022

Table 9.1.: Excerpt of a Performance Snapshot

2.5 minutes, clearly indicates a problem. As in a persist and therefore the creation of a workspace both *Elasticsearch* and the database are involved, it is worth taking a look at them, too. *Elasticsearch* operations are measured by the *es* monitor. Its average at 580ms seems OK but once again, almost **86 seconds** maximum are alarming. The same holds true for the next one, *es.searchables_production_multitenancy.commit*, tracking how long a commit operation for the index *searchables_production_multitenancy* takes. A commit operation should usually be very fast, since only a new document is added to the search index. Everything else, distributing it across the cluster and cleaning up scattered data, is done asynchronously. Therefore even the average with more than 1 second is too high — not to mention the maximum at almost **85 seconds**. The last involved monitor is wrapped around the execution of database operations, *db*. The average value of 1ms and a total of only 5 times the maximum show that there is not a real problem. Of all roughly 220.000 *db* hits, at most 5 have been slow, that is $2.2 \cdot 10^{-3}\%$. Nevertheless this was noted by *infoAsset* for further investigation.

Therefore, we now have the suspicion that *Elasticsearch* is the root of the problem due to its slow commit operations. The next step is to verify that the maximum execution times are not singular events but occur over a longer period of time. To do this, the data collected by the monitoring tool is downloaded from the webserver and analyzed locally. The resulting graph is shown in Figure 9.1. The mean value of the *es* monitor is plotted here.

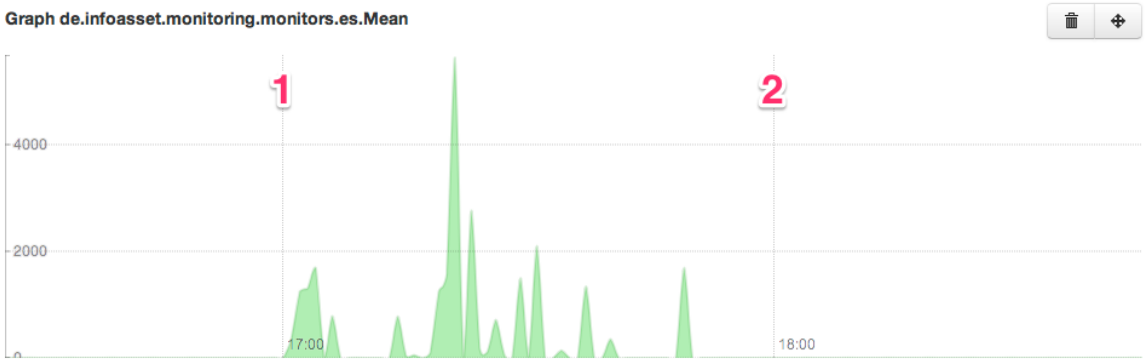


Figure 9.1.: Graph of *Elasticsearch* execution time

The y-axis on the left edge starts at 0 at the bottom whereas the top edge is at 6000ms or **6 seconds**. The two vertical lines delimit the timestamps, starting at 17:00 (**1**) and ending at 18:00 (**2**). It is clearly visible that over the period of **1 hour**, the average execution time of *Elasticsearch* operations increased drastically.

infoAsset then advanced to trying to reproduce the problem. Fortunately, the problem occurred *every time* a new workspace was created. Therefore, a request was triggered and as expected, it took such an amount of time long that *Tricia* logged a performance trace for it. The beginning of the trace is given in Listing 9.1. The values are all given in milliseconds.

```
1  2013-07-26 08:44:29,062 [http-connection-11966] T: -
2      ------ long taking request: 9607 ms for /space/submit------
3  webServer.handlerRequests                      : Hits:   1, Total:  9607.4, Avg:  9607.4
4  webServer.allRequests                          : Hits:   1, Total:  9606.7, Avg:  9606.7
5  generic.request                                : Hits:   1, Total:  9606.7, Avg:  9606.7
6  handler.platform.handler.space.SubmitHandler   : Hits:   1, Total:  9605.9, Avg:  9605.9
7  handler                                        : Hits:   1, Total:  9605.9, Avg:  9605.9
8  es                                             : Hits:  10, Total:  8788.7, Avg:   878.9
9  es.searchables_production_multitenancy         : Hits:   1, Total:  8686.3, Avg:  8686.3
10 es.searchables_production_multitenancy.commit  : Hits:   1, Total:  8686.3, Avg:  8686.3
11 entity.persist                                 : Hits:  10, Total:  2090.4, Avg:   209.0
12 entity.persist.pageSpace                       : Hits:   2, Total:  1144.9, Avg:   572.5
13 db                                             : Hits: 967, Total:   707.3, Avg:     0.7
```

Listing 9.1: Performance Trace for Workspace Creation

The request took **9.6 seconds** total (line 2). The monitors are sorted by their total execution time. In line 10 there is the commit operation. With almost **8.7 seconds** it took too long by far. As a result, the request was processed so slowly and the total values of the other monitors are affected.

The problem is therefore identified as being the commit operation executed on the *Elasticsearch* cluster. It was also verified that it is not a singular event but occurs frequently and has enormous impact.

9.3. Tracking Down

The next step is to see if a bad implementation in *Tricia* is responsible for the excess commit operation. The `EsClient` class in *Tricia* is used as abstraction to handle all *Elasticsearch* library calls. The method triggering the above identified commit is shown in Listing 9.2.

```
1  public static void commitIndex(final String indexName) {
2      EsMonitor.monitor(new Fun() {
3
4          @Override
5          public void fun() {
6              INSTANCE().getClient().admin()
7                      .indices()
8                      .refresh(Requests.refreshRequest(indexName))
9                      .actionGet();
10         }
11
12     }, indexName, "commit");
13 }
```

Listing 9.2: Code of commit Operation

The single parameter for this method is the name of the index to be committed. In our case the index is equal to `searchables_production_multitenancy`. The real method body is wrapped inside a call to `EsMonitor.monitor(...)` (line 2 – line 12). This is just a helper function for starting a corresponding monitor — here: `es.searchables_production_multitenancy.commit`. The interesting piece of code are lines 6–9. In essence, the *Elasticsearch* library is called and a refresh is triggered on the given `indexName`. A refresh causes all pending changes to be made available to the whole cluster [9]. The *Elasticsearch* API is written asynchronously. Since *Tricia* expects all results to be available after a commit, `actionGet()` is called in line 9. This forces the execution to wait until the refresh operation has completed.

However, even if the refresh is executed synchronously, it must not take as long as experienced. Consequently, the problem does not occur due to the implementation in *Tricia*. We then decided to investigate further on a very low level. The servers in the cloud environment were running Ubuntu Linux, thus we used the command line tool *strace* [2]. This tool tracks for example the system calls created by a process in combination with their arguments and respective return values. We used it to surveill the *Elasticsearch* process. To get reliable results, all other nodes were shut down so that only one single server was active. Once again, a request was triggered and *strace* captured the system calls. The output generated by *strace* contains a lot of information. Since we knew the exact time the request was started, we were able to identify roughly the start and end of the execution in the output. After some investigation, a pattern met the eye. The relevant parts from the *strace* data are shown in Listing 9.3.

```
1  ...
2  21169 08:44:20.436215 unlink("xxx/indices/searchables_production_multitenancy/0/index/
       _lm.tvx" <unfinished ...>
3  ...
4  21169 08:44:21.390968 <... unlink resumed> ) = 0
5  21169 08:44:21.391134 unlink("xxx/indices/searchables_production_multitenancy/0/index/
       _lm.fnm") = 0
6  21169 08:44:21.393180 unlink("xxx/indices/searchables_production_multitenancy/0/index/
       _lm.tvf" <unfinished ...>
7  ...
8  21169 08:44:22.347567 <... unlink resumed> ) = 0
9  21169 08:44:22.347654 unlink("xxx/indices/searchables_production_multitenancy/0/index/
       _lm.tvd" <unfinished ...>
10 ...
11 21169 08:44:23.303854 <... unlink resumed> ) = 0
12 ...
```

Listing 9.3: Excerpt of *strace* output

Though Listing 9.3 shows only 4 `unlink` operations, there is a total of 11 *unlinks* executed during the commit. An `unlink` system call in *Linux* is used to delete a file or symlink [20]. The files referenced in the executions from Listing 9.3 are very small. Nevertheless, the `unlink` call sometimes takes up to **1 second**, on average around **600 miliseconds**. On other systems where the same setup is used, we were able to extract the trace presented in Listing 9.4.

On the system used for comparion — remember: same technology as environment, same *Tricia* setup — the `unlink` operations took at most **a few milliseconds**. This is a drastic difference and causes the experienced problem.

```
1  ...
2  4516  09:06:02.430428 unlink("xxx/indices/searchables_production_multitenancy/0/index/
       _uf.tis" <unfinished ...>
3  ...
4  4516  09:06:02.431652 <... unlink resumed> ) = 0
5  ...
6  4516  09:06:02.432375 unlink("xxx/indices/searchables_production_multitenancy/0/index/
       _uf.tvf" <unfinished ...>
7  ...
8  4516  09:06:02.433206 <... unlink resumed> ) = 0
9  ...
10 4516  09:06:02.433280 unlink("xxx/indices/searchables_production_multitenancy/0/index/
       _uf.tvd" <unfinished ...>
11 ...
12 4516  09:06:02.434229 <... unlink resumed> ) = 0
13 ...
```

Listing 9.4: Comparison trace with same setup

9.4. Outcome

After all a bug in *Tricia* could be ruled out. A system configuration or basic I/O problem is likely to be the reason for this issue. The cloud provider *DTAG* was notified of the problem. It turned out that a hard drive error in one of their systems was responsible. After exchanging the respective component and an additional rebuild of the virtual machines used in the cloud environment, the problem was solved. All in all, the developed monitoring solution turned out to be very useful. It provided essential data to track down the issue.

10. Conclusion

Coming to a conclusion, let us take a step back and sum up the work covered in this thesis. We also want to identify remaining points of improvement. These can then be used as a basis for further research.

10.1. Results

In this thesis we took a thorough look at *APM* in general and how it can be applied in the context of an existing application. The results were worked out with respect to the requirements posed by the environment of a cloud infrastructure. Furthermore, the characteristics of a scalable and distributed application as well as specialities of *Tricia* had to be taken into account. The overall goal was on one hand to derive the essential aspects on what is needed in order to properly employ *APM*. On the other hand, it was important to describe a process of integrating *APM* in an already existing Java web-application.

Chapter 1 gave an overview of the theoretical background. It defined important terms required to understand this thesis as well as gave an introduction to *Tricia*. Following that, a short motivation was given before the problem itself was described in Chapter 2. We also put it in the relevant context and named the involved stakeholders. Afterwards, Chapter 3 contained a profound analysis. Key metrics necessary to do *APM* were identified successfully. Furthermore, the required means to evaluate gathered data have been presented. Section 3.3 sums up the findings of the analysis process. Before advancing to the practical part of this thesis, an examination of existing solutions and standards was done in Chapter 4. The focus was put on the characteristic advantages and drawbacks of every solution especially in the context of *Tricia*. Additionally, we conducted an interview with an employee of a large company who already has experience in *APM* for applications. The results thereof were given in Chapter 5 and proved to be very valuable.

The next part covered the practical work of employing *APM* in an existing application: *Tricia*. The first step was to decide on key aspects of the planned solution. This process was described in detail in Chapter 6. One of the main results was to separate the monitored application from the collection, retention, and analysis of monitoring data (Section 6.2.1). Thereafter, in Chapter 7 the architecture of the two components of the new solution, monitoring tool and *Tricia's* adaptation, has been worked out. Chapter 8 contains key implementation details of both parts of the solution. There, we also showed to employ monitoring in *Tricia* itself. Fortunately, we had the chance to evaluate the solution in cooperation with *infoAsset*. One problem was tackled by leveraging the developed *APM* and the corresponding results were presented in Chapter 9. The solution turned out to be very helpful and an application problem could finally be ruled out successfully.

It is also important to note that the created solution is fitted to be of special use for the developers. The data gathered cannot be judged efficiently by someone who has no

profound understanding of the application's internals. This is especially so concerning the detailed information contained in snapshots and traces. Practically only a developer can currently interpret the data included in them.

10.2. Points of Improvement

Though we worked very hard, the limited scope of this thesis did not allow for everything to be done perfectly. Therefore, some points of improvements do exist. They may be used in order to conduct further research.

First, it is possible to integrate an even higher degree of automation. Currently, apart from exposing data via *JMX*, *Tricia* also writes extensive output to log files. These also include the two means of evaluation: performance snapshots and performance traces. Upon the detection of a problem, a developer must then dig into them and extract the required information. This can sometimes be a tedious task. Therefore, one aspect is to introduce automatic log file analysis. A variety of flexible and powerful tools exist to do this job. For example *Graylog2* [21] or *Logstash* [22], to name two. Additionally, in case of performance problems, immediate alerts or notifications can be used to make the developers aware of them fast. In this thesis, we left them out completely in order to concentrate on creating a solid foundation. However, we see this as as an advanced feature which can be integrated later on. We therefore want to remark that *JMX* already provides some support for alerts and notifications.

The second big aspect is the amount of data exposed via *JMX*. As of the state of this thesis, only a variety of key metrics are available. Especially the above mentioned performance snapshots and performance traces are not included. Thus, one can investigate if including this information in the *JMX* interface is useful. We have in mind that the traces can be collected automatically, too. If a developer then evaluates the data by using the monitoring tool, it could also be possible for him to easily check existing traces referring to the discovered problem.

Bibliography

[1] W. BARTH, *Nagios system and network monitoring*, No Starch Press, San Francisco, 2008.

[2] S. BEST, *Linux debugging and performance tuning : tips and techniques*, Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, 2006.

[3] J. BLOCH, *Effective Java (2nd Edition)*, Addison-Wesley, 2008.

[4] M. BOGAERT, F. BACHELLA, et al., *rrd4j - a high performance data logging and graphing system for time series data*. `http://code.google.com/p/rrd4j/`. Accessed: 2013-07-30.

[5] M. BOSTOCK, *D3.js - Data-Driven Documents*. `http://d3js.org`. Accessed: 2013-07-25.

[6] B. CLEWETT, F. GLEIXNER, et al., *PerfParse Add On for Nagios*. `http://perfparse.sourceforge.net`. Accessed: 2013-08-02.

[7] COREMEDIA AG, *CoreMedia*. `http://www.coremedia.de/`. Accessed: 2013-07-15.

[8] G. COULOURIS, J. DOLLIMORE, T. KINDBERG, and G. BLAIR, *Distributed Systems: Concepts and Design (5th Edition)*, Addison-Wesley, 2011.

[9] ELASTICSEARCH, *Elasticsearch - Open Source Distributed Real-Time Search & Analytics*. `http://www.elasticsearch.org`. Accessed: 2013-07-25.

[10] C. C. EVANS, *YAML Ainť Markup Language*. `http://www.yaml.org`. Accessed: 2013-07-23.

[11] M. FLEURY, J. LINDFORS, and T. J. GROUP, *JMX: Managing J2EE with Java Management Extensions (Java (Sams))*, Sams, 2002.

[12] F. FORSTER et al., *collectd - The system statistics collection daemon*. `https://collectd.org/`. Accessed: 2013-07-30.

[13] GOOGLE, *AngularJS - Documentation*. `http://docs.angularjs.org`. Accessed: 2013-07-25.

[14] GOOGLE, *AngularJS - Superheroic JavaScript MVW Framework*. `http://angularjs.org`. Accessed: 2013-07-25.

[15] GOOGLE, *Google Gmail*. `http://mail.google.com/`. Accessed: 2013-07-07.

[16] C. HALE, *Metrics - Mind the Gap*. `http://metrics.codahale.com`. Accessed: 2013-07-30.

[17] C. HUNT and B. JOHN, *Java Performance*, Addison-Wesley Professional, 2011.

[18] HYPERIC, *SIGAR API - System Information Gatherer and Reporter*. `http://www.hyperic.com/products/sigar`. Accessed: 2013-07-26.

[19] JAVASIMON PROJECT, *Java Simon - Simple Monitoring API*. `https://code.google.com/p/javasimon/`. Accessed: 2013-08-03.

[20] M. KERRISK, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, No Starch Press, 2010.

[21] L. KOOPMANN, *Graylog2 - Free open source self-hosted log management and exception tracking*. `http://graylog2.org`. Accessed: 2013-07-31.

[22] LOGSTASH, *logstash - open source log management*. `http://www.logstash.net`. Accessed: 2013-07-31.

[23] MICROSOFT, *Office 365*. `http://office365.com`. Accessed: 2013-07-07.

[24] NAGIOS ENTERPRISES, LLC, *check_jmx Nagios plugin*. `http://exchange.nagios.org/directory/Plugins/Java-Applications-and-Servers/check_jmx/details`. Accessed: 2013-08-02.

[25] NAGIOS ENTERPRISES, LLC, *Nagios - The Industry Standard in IT Infrastructure Monitoring*. `http://www.nagios.org`. Accessed: 2013-08-02.

[26] NAGIOS ENTERPRISES, LLC, *Nagios Core Documentation*. `http://nagios.sourceforge.net/docs/nagioscore/3/en/`. Accessed: 2013-08-02.

[27] NEW RELIC, INC., *New Relic - Application Performance Management & Monitoring*. `http://www.newrelic.com/`. Accessed: 2013-07-31.

[28] T. OETIKER et al., *RRDTool - logging & graphing*. `http://oss.oetiker.ch/rrdtool/`. Accessed: 2013-07-30.

[29] ORACLE, *Java Platform Standard Edition 6 Documentation*. `http://docs.oracle.com/javase/6/docs/api/index.html`, 2011. Accessed: 2013-04-15.

[30] J. S. PERRY, *Java Management Extensions*, O'Reilly Media, 2002.

[31] PROJECT GRIZZLY, *Project Grizzly - NIO Event Development Simplified*. `http://grizzly.java.net/`. Accessed: 2013-07-25.

[32] A. REITBAUER and M. NOVAKOVIC, *Effizientes Performancemanagement*, Java Magazin, 11, 2009.

[33] A. SCHNEIDER, C. NEUBERT, and F. MATTHES, *Fostering Collaborative and Integrated Enterprise Architecture Modeling*, Journal of Enterprise Modelling and Information Systems Architectures, 2013.

[34] SHUTTERSTOCK IMAGES, *Rickshaw: A JavaScript toolkit for creating interactive time-series graphs*. `http://code.shutterstock.com/rickshaw/`. Accessed: 2013-07-25.

[35] B. SOSINSKY, *Cloud Computing Bible*, Wiley, 2011.

[36] S. SOUDERS, *Even faster web sites*, O'Reilly, Sebastopol, 2009.

[37] THE CACTI GROUP, INC., *Cacti - the complete rrdtool-based graphing solution*. `http://www.cacti.net`. Accessed: 2013-07-30.

[38] THE MUNIN PROJECT et al., *Munin*. `http://munin-monitoring.org`. Accessed: 2013-08-03.

[39] TWITTER, *Bootstrap - Sleek, Intuitive, and powerful front-end framework for faster and easier web development*. `http://twitter.github.io/bootstrap/`. Accessed: 2013-07-25.

[40] A. VAN HOORN, M. ROHR, W. HASSELBRING, J. WALLER, J. EHLERS, and S. FREY, *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. `http://eprints.uni-kiel.de/14459/`, 2009. Accessed: 2013-04-15.