UNIVERSITÄT
BERN

AppCheck

Monitoring of a JavaEE Server Application

Bachelor Thesis

Andreas Wälchli
from
Madiswil BE, Switzerland

Faculty of Science
University of Bern

September 8, 2017

Prof. Dr. Oscar Nierstrasz

Software Composition Group
Institute for Computer Science
University of Bern, Switzerland

Abstract

With ever growing numbers and complexity of server applications efficient monitoring is key to detect performance issues in production and prevent them during development. In this work we show how such a monitoring solution could be built for JavaEE to meet specific requirements.

We describe a software project for the *Informatik Service Center ISC-EJPD*[1] which is interested in integrating performance monitoring into an existing monitoring framework.

Our software provides a simple to use and highly customisable solution for basic performance monitoring in JavaEE applications.

[1] `https://www.isc-ejpd.admin.ch`

Acknowledgements

My special thanks to *Prof. Dr. Oscar Nierstrasz* for giving me the opportunity to do this software project as my bachelor thesis in the Software Composition Group of the University of Bern.

I would like to thank the *Informatik Service Center ISC-EJPD* for making this bachelor thesis possible.

Finally, I would like to extend a special thanks to *David Klaper* and *Rolf Scherer* as well as other developers at the *ISC* for their assistance and support during the whole project.

Contents

1 Introduction

Today good performance monitoring of server applications is key to a successful product. With the ever increasing size and complexity of applications good monitoring can help developers and operators in detecting and solving issues quickly and thoroughly.

AppCheck attempts to provide a good customised monitoring tool for Java EE applications to the *Informatik Service Center ISC-EJPD* that can integrated into preexisting internal monitoring services. It targets at covering and analysing many measurements taken directly from the Java virtual machine itself while leaving more application specific elements to be implemented by the developers of the individual applications. AppCheck tries to provide the developers with the tools necessary to quickly extend it to serve the needs of any given application. We also made sure that it is highly configurable to ensure good adaptability to any possible use case.

During the development we faced many challenges given the fact that this was our first project using JavaEE. We encountered countless issues from simple mistakes to special cases concerning the security elements of JavaEE that we sometimes had to work around.

In conclusion the project was a success. We were able to implement nearly all requirements and adjust those we could not to find a suitable alternative. We improved our knowledge of the architecture of Java server applications and in working within a relatively limiting set of development constraints.

2 Requirements

Based on a meeting with developers from the *ISC* and internal development guidelines, we defined the following two basic goals. These goals were kept rather abstract to allow for a general feasibility evaluation. After that evaluation these abstract goals crystallised into a set of more specific functional and non-functional requirements.

Provide a Basic Monitoring Solution

The *ISC* uses an internal standardised web service (Section 3.2) to provide on-demand health and performance information of a web application. The preexisting web service interface should be implemented to provide general performance data of an application. Such performance data may include garbage collection performance, database availability or memory usage. After gaining an overview over the possibilities we should choose the metrics we considered to be feasible for implementation.

Provide Monitoring for Timed operations

JavaEE supports the creation of EJB Timers (Subsection 3.1.2). These Timers specify a schedule on which a specified method will be called. This allows for easy management of timed task execution. However JavaEE does not provide any tools for directly monitoring

such timers. Therefore we have been asked to develop a way to list all the existing timers and check if all required timers are actually present. If possible we should also monitor the triggering of the method executions and check if they adhere to their schedules. The reference information should be provided externally through a configuration file. If considered possible, we could also implement a self recovery feature that recreates missing timers.

2.1 Functional Requirements

Execution of Application Checks On Demand

By issuing a web service request an authenticated user should trigger the execution of an application check. The results of that check should be returned to the user as a web service response. The *ISC* already provides an internal web service for this task.

Evaluation of Performance Metrics

An application check should check several performance metrics and decide for each one if the observed values are acceptable or not. Such performance metrics include:

- memory usage
- garbage collection performance
- database accessibility and performance
- existence of all required timers

Additionally some general metrics should be included that are not checked for system sanity but provide some general system information. These include:

- system uptime
- number of loaded classes

Monitoring Garbage Collection Performance

While the garbage collection performance is already listed as a performance metric, we believed it to be perhaps the most complex metric we would cover in this project. Proper monitoring of garbage collection can help in identifying both memory leaks and

excessive creation of short-lived objects. To allow long-term observation of the garbage collection we should create a long-term statistic from every single garbage collection cycle. This statistic would cover several garbage collection details including:

- garbage collection frequency
- garbage collection duration
- total memory usage after the garbage collection
- amount of memory cleared by the garbage collection
- memory usage after the garbage collection for each memory pool

Monitoring Timer Execution

As an optional requirement the execution of timed methods should be monitored. We should then check if the execution adheres to the required schedule for that timer. The information of which timers run correctly could also be introduced as a performance metric.

2.2 Non-Functional Requirements

Performance

The check executions and real-time monitoring of garbage collection and timer executions should have no measurable performance impact on the rest of the application. The memory footprint should also be minimised and known.

Reusability

The module should be applicable to essentially any JavaEE web application with minor modification.

Configurability

As many behavioural aspects as possible should be controllable through configuration files. This allows our solution to be adjusted to better suit the different needs of different

projects.

Extensibility

The module should be designed to allow both integration of new components into it and integration of the module in other pre-existing monitoring solutions, specifically the *ISC* internal web service.

Stability

Since web applications can be running for several months at a time all components should be as stable as possible and support theoretically unlimited system runtimes. This is especially important for the statistical data that should also remain numerically stable for as long as possible despite unavoidable rounding errors. Stability is very important since system crashes, negative impact on performance and even inaccurate results would quickly lead to our software not being trusted and therefore no longer being used.

Compliance

The implementation should wherever possible adhere to *ISC*-internal development guidelines and specifications. For us the relevant ones are:

- no Enterprise Java Bean may be stateful (Subsection 3.1.1)
- all checks must evaluate to a binary result indicating if a check passed or failed. If a check does not directly yield a binary result it must be mapped onto a binary result adequately.

3
Technical Background

In this chapter we will describe the main technologies we used in our project.

3.1 Java Enterprise Edition

We developed our software in Java Enterprise Edition (Java EE) Version 6 and Java Standard Edition (Java SE) Version 7. Java SE is the Java edition most people know as it is the edition Java desktop or command line applications run on. Java EE is an extension to the basic Java SE platform designed for web applications. Oracle explains Java EE as follows:

> The Java EE platform is built on top of the Java SE platform. The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications.[6]

Java EE provides countless features for simplifying design. Here we will only show the two that were relevant for our project.

3.1.1 Enterprise JavaBean

JSR 318 describes Enterprise JavaBeans (EJB) as an "*architecture for component-based transaction-oriented enterprise applications.*"[15] EJBs typically contain business logic that operates on business data. An EJB can represent a specific use-case or a set of transactions. EJBs can reference others to enable code reuse and promote a modular design.

There exist 3 basic types of EJBs: Stateful and Stateless Session Beans and Singleton Beans. Of the Session Beans there may exist any number of instances at any time but Singleton Beans are limited to a single instance. The key difference between Stateful and Stateless Session Beans is that Stateful Session Beans may have a state, *i.e.,* they can hold data outside of a single transaction. Stateless Session Beans may however not hold a state outside of transactions. We consider it to be a good practise to use Stateless Session Beans whenever possible as that removes global state. Global state is considered bad practise as it complicates understanding and testing[4][17].

3.1.2 EJB Timer

EJBs can register timers in a `TimerService` . Each EJB has its own associated `TimerService` . Whenever a timer is triggered a specific method of the EJB is invoked. This allows for repeated execution of business code on a fixed schedule *e.g.,* for transactions that have to be performed periodically. There exist different types of timer schedules. Timers can either be configured to trigger repeatedly with a given delay period (*e.g.,* every hour) or they can be configured to trigger at a specific date and time (*e.g.,* every first day of a month at 3am). The `TimerService` instances are specific to each EJB and cannot be accessed externally. Consequentially timers must usually be registered by the bean itself.

3.2 SOAP Web Service

To explain what a SOAP Web Service is we must take a step back from the name itself as it actually contains two misnomers[11]. A web service is defined by the *World Wide Web Consortium* (W3C) as follows:

> The World Wide Web is more and more used for application to application communication. The programmatic interfaces made available are referred to as Web services.

So in essence a web service is any internet protocol that allows the transfer of data between two or more programs.

SOAP was initially an acronym for *Simple Object Access Protocol* but with SOAP version 1.2 that meaning was removed since SOAP does not directly use objects[11]. SOAP web services are stateless, meaning that requests are isolated and do not influence each other. SOAP is uni-directional meaning that one application (the server) provides a SOAP web service and one or more applications (the clients) use that web service. A single SOAP transaction consists of a request being sent from a client to the server. The server then sends back a response. All messages in a SOAP transaction are XML data but the precise schema is specific to each individual web service.

3.3 Java Management Extensions

The Java Management Extensions (JMX) define an architecture, design patterns, APIs, and the services for application and network management and monitoring in Java[13]. We only used the monitoring features. JMX defines a framework of so called *MXBeans*. Using these *MXBeans* JMX provides access to a multitude of measurements for most aspects of the Java Virtual Machine including memory usage, garbage collectors, class loading, running threads and platform information.

The Java Runtime Environment defines base interfaces for all standard *MXBeans*. But these interfaces only provide the methods to extract the measurements that should be supported universally on all virtual machines. Since our project only targets the Oracle HotSpot Virtual Machine which is the default virtual machine for most people, we could get access to even more information by using the virtual machine dependent implementations of these Beans. Without these additional measurements our project would not have been possible.

3.4 Java Memory Management

Up to Java SE Version 7 the Oracle Java HotSpot Virtual Machine divides the heap memory into 6 memory pools with different roles[14]: *EdenSpace*, 2 *Survivor Spaces*, *OldGen*, *PermGen* and *CodeCache*. *EdenSpace* and the 2 *Survivor Spaces* together form the *Young Generation*. HotSpot uses 2 different garbage collector types: *minor* and *major* garbage collectors. The different memory pools and garbage collector types are best explained together:

New objects are instantiated in the *EdenSpace*. Whenever this memory pool starts to fill up, a *minor* garbage collection is performed. This garbage collector only works on the young generation memory pools. Any objects that survive the garbage collection are moved into one of the two *Survivor Spaces*. This is done in such a way that only one *Survivor Space* is used and with every garbage collector run the surviving objects are moved from one *Survivor Space* to the other compacting it. Objects that have survived several *minor* garbage collector runs are eventually moved into the *OldGen* memory pool. Whenever the *OldGen* starts to fill up a *major* garbage collection is performed. This garbage collector operates on both the young generation and the *OldGen* pool. This kind of garbage collection is very expensive and is only done as rarely as possible. The *PermGen* and *CodeCache* are used by the Java Virtual Machine itself and are not managed by any garbage collector.

With Java SE Version 8 the *PermGen* and *CodeCache* have been combined into a new *Metaspace*[8].

4 Implementation

For the implementation and testing during development our customer provided us a small internal example application. This application reflected the basic structure of their production code and the structure of their internal monitoring architecture and allowed us to quickly get to know the structure of JavaEE applications. In the following sections we will describe the implementation of the different elements.

4.1 Core Code

The core code of our code consists of a Singleton Bean (`RootCheckerBean`) that manages the initialisation of all checks. It also loads the configurations from a property file on disk and initialises the garbage collection monitoring which due to its nature requires some special treatment. A second core element is another Singleton Bean (`CheckerDataBean`) that manages continuous monitoring and provides the gathered data. The `RootCheckerBean` is designed to act both as the top-level component if our code were to be used as a stand-alone solution and as a common entry point for integration into existing monitoring solutions. Primarily considered for – but not limited to – stand-alone use, new custom `IChecker` implementations can be registered in the `RootCheckerBean` to allow for customisable extensions to our code. The `IChecker` interface serves as a common interface for all sort of checks and is rather simple. (Code 4.1) The method parameter allows for passing a unique identifier that

```java
package ch.awae.appcheck.api;
public interface IChecker {
    ICheckResponse doCheck(String uid);
}
```

Code 4.1: IChecker interface (Section 4.1)

```java
package ch.awae.appcheck.api;
public enum CheckResult {
    CHECK_OK, CHECK_NOK;
}
```

Code 4.2: CheckResult enum (Section 4.1)

should be used in logging to associate log entries with specific checks. The returned object of type `ICheckResponse` holds the results of a check. These include a title, a description, a binary check result, a list holding the results of all the sub-checks and optionally a message or an error message and a stack trace. The binary result is represented by a two-field enum. (Code 4.2) This `ICheckResponse` data structure can be used to provide the information over a WebService.

This structure allows us to adhere to the requirement that no Stateful Session Beans may be used by concentrating all global state into just 2 Singleton Beans.

4.1.1 Handling Non-Binary Check Results

One of the requirements was that all checks would need to evaluate to a binary result (Code 4.2). The problem is that real-world data often can not be directly interpreted in a binary way. With such fuzzy data the same value can sometimes indicate a problem but can at other times be perfectly normal depending on other external factors. We needed to find a solution that would allows us to handle fuzzy data in a sensible way in an unknown environment. We developed a general approach we used wherever such handling would be required:

Complex checks (*e.g.,* Section 4.4) often provide nested results with several values. We decided to first reduce the individual fuzzy values to binary values and then combine the individual binary results into a summarising binary result. To break the individual values

down to a binary result, we introduced an indicator function[9] with a threshold:

$$I_a(x) = \begin{cases} 1, x \leq a \\ 0, x > a \end{cases}$$

The thresholds can be tweaked individually through configuration parameters to tune the indicator function such that false positives and false negatives can be reduced. For determining the summarising result we introduced another parameter we call *strictness*. This strictness value s defines the smallest relative number of checks that need to pass for the summary check to be considered passed as well:

$$\sum_{i=0}^{n} \frac{I_{a_i}(x_i)}{n} \geq s$$

4.2 Basic Monitoring

To provide a general overview over the system we decided to include some basic information in the check results without giving a failure condition. This means that the checks that provide this information will only ever fail if the data can not be provided.

We included the uptime of the virtual machine, *i.e.,* the time since the virtual machine was started. This information is extracted from the `RuntimeMXBean` . Through a configuration parameter it is possible to choose between an exact representation where the duration is given in milliseconds and a representation with better readability where the duration is given in a `d hh:mm:ss.uuu` format[1].

We also included some class loading information extracted from the `ClassLoadingMXBean` :

- the number of currently loaded classes
- the total number of classes loaded since the virtual machine was launched
- the total number of classes unloaded since the virtual machine was launched

While these measurements in most cases only have little value since the system uptime can be known through other means and the number of classes in a system can usually be known at build time it was simple to include them in the output just in case. The class

[1] `uuu` represents the milliseconds within the current second

loading information may be useful in some systems with dynamic class loading through custom class loaders where classes can be loaded and unloaded again[5].

4.3 Thread Monitoring

For a second simple monitoring option we looked at the data we could gather efficiently from basic measurements about the running threads that were provided by the `ThreadMXBean` . It provides the number of currently running threads as well as the peak number of simultaneously active threads. We provide both these numbers with the check result. Additionally the `ThreadMXBean` enables us to access more detailed information about the currently running threads. It also provides troubleshooting features such as the ability to find deadlocked threads, *i.e.,* two or more threads that all wait on a resource another one currently accesses but can only release after it get access to another currently occupied resource. This can however be a very expensive operation since a complex analysis of the current system state has to be performed. Therefore we decided not to use this option instead we focussed on the information we could obtain from the threads themselves. The most notable included data are the thread name and the thread CPU time. The thread CPU time is the time the thread has spent executing since it started. Combined with the total time the thread has been alive and the number of active cores in the CPU this information could be combined to determine an average CPU load for the thread using the following formula:

$$\%\text{CPU} = \frac{\text{Thread CPU Time}}{\text{Thread Life Time} * \text{Number of CPU Cores}}$$

However there is no way to determine the life time of a thread through the `ThreadMXBean` . Nonetheless we still decided to include the name and CPU time for each thread in the check result. This can be enabled or disabled through configuration parameters. While the lack of CPU load information reduced the usefulness of our thread monitoring, we expect server operators to be able to monitor the CPU usage through the application server or the operating system itself.

4.4 Garbage Collection Monitoring

Monitoring of the garbage collectors is probably one of the best ways to quickly gain an overview over the performance of a system. Observing the frequency of garbage collection runs alone can be enough to detect both memory leaks (frequent major garbage

collection) and a flooding with many short-lived objects (frequent minor garbage collection).

For a first attempt we implemented a simple checker analogous to other simple metrics. JMX provides direct access to summaries for the last garbage collection run for each garbage collector type (minor & major) over a `GarbageCollectorMXBean`. Therefore we started off by simply analysing these garbage collection runs. It quickly became obvious that this monitoring technique would not be sufficient since the last garbage collection runs are not necessarily representative of the long-term system performance. Especially major garbage collection runs can be far in the past and therefore provide no current information. The time elapsed since the last garbage collection run does not provide any useful information either. The only thing that can reliably be determined by observing the last garbage collection run are some memory leak scenarios that result in the major garbage collector not collecting much while most of the available memory is used. For a better analysis we needed access to the full garbage collection history.

4.4.1 Sliding Average

For the implementation of our second approach we needed access to said garbage collection history. Here JMX helped us by providing another useful feature: It is possible to register a notification function that will be called whenever a garbage collection run was completed. Using that function our solution would record each garbage collection run and create statistics for relevant garbage collection performance values:

- the type of garbage collector
- memory usage before and after the garbage collection for each memory pool
- the duration of the garbage collection run

From these primary measurements we can then extract secondary values such as the total time spent in garbage collection or the frequency garbage collection runs. We could also create statistics over different time periods (*e.g.,* system lifetime, last hour, last day, last week). The most obvious approach for creating such statistics would be a sliding average, that only considers values that have not yet exceeded a certain age.

But for an implementation of such a sliding average all values would have to be kept in memory at all time. This would lead to growing memory usage with increasing runtime. Especially if the system has memory issues that increase the garbage collection frequency this approach would increase its memory requirement with an accelerated pace eventually itself causing a memory overflow.

Since the monitoring should not influence the system performance such a scenario would

not be acceptable as the monitoring and the frequent garbage collection runs could create a positive feedback loop that negatively impacts the system performance. Another issue could be the deteriorating performance of the monitoring itself since with increasing data volume the calculation of the statistical values will slow down. If possible the memory usage should be kept minimal and constant.

4.4.2 Discretised Sliding Average

At this point we decided to compromise and drop all statistical values that require full knowledge of all past measurements. In our case this would be the median and variance. We would only record the number of data points, the arithmetic mean, the minimum and the maximum. These values can be calculated without knowledge of past measurements as long as their previous values are known. These values can be updated using a few rather simple formulae. For a data series v with n recorded measurements, update the statistics values with the new measurement $v(n+1)$ as:

- maximum: $\uparrow v_{n+1} = \max(\uparrow v_n, v_{n+1})$
- arithmetic mean: $\bar{v}_{n+1} = \frac{\bar{v}_n \times n + v_{n+1}}{n+1}$
- minimum: $\downarrow v_{n+1} = \min(\downarrow v_n, v_{n+1})$

This way only the 4 values must be stored. This keeps the memory usage constant and known. However this does not allow for statistics over different time periods and forces the use of lifetime statistics since for finite constant time periods old values must be removed, but they are no longer known.

We were able to circumvent this limitation by re-introducing a discretised adaptation of a sliding average: We would split each time period into a fixed number of sections of equal length. Each of these sections holds the aggregated values[2] representing the data points within its time period. When the age of the youngest section has exceeded its length, a new section is created that holds the newer measurements. Whenever a new section is created, the oldest one is discarded. Whenever the statistics over the full time period are required the sub-statistics of all sections can be combined in constant time (since the number of sections is fixed). This solution approximates a continuous sliding average by not discarding old values individually but rather keeping them around and discarding them all at once as soon as all values in a section have exceeded their age limit.

This solution obviously sacrifices accuracy in the time resolution since the actual timespan of the statistics does oscillate by the length of a single section. Using more sections this oscillation can be reduced by increasing the temporal resolution and sacrificing memory.

[2] minimum, average, maximum and number of data points

Using fewer sections reduces the memory footprint by sacrificing temporal resolution. However over long periods of time this oscillation does not matter too much especially since the measurements themselves are quite variable as in a normal server application they are influenced by interactions with other applications and users. For example for a period of one day that is split into 24 sections of one hour each the actual statistics period oscillates between 24 and 25 hours. This is a deviation of up to approximately 4.2%. We believe that for GC monitoring this solution provides a good compromise as it allows for statistics over multiple independent time periods with a constant and known memory footprint. Statistics over the full application lifetime can still be done easily by only using one section and never discarding anything.

Since the Java HotSpot Virtual Machine divides the memory into multiple memory pools[14](Section 3.4) we decided to create this statistic for each of them. Through multiple configuration parameters memory thresholds[3](Subsection 4.1.1) can be defined for each memory pool individually. The way the memory sizes are represented can also be selected through configuration parameters: Either the precise number of bytes (*e.g.,* 465383B) can be provided or the size can be rounded to the largest multiple of 1024 smaller than the size itself (*e.g.,* 465KiB[4]). The number of statistics and the time periods these statistics cover as well as the number of segments the time periods should be divided into can also be customised through configuration parameters. There is no upper limit to the number of statistics but we recommend to only use a few since with each additional statistic the memory footprint and the amount of processing necessary increases.

4.4.3 Accuracy of Floating-Point Representation

After some experimentation it became apparent that in floating point arithmetics the order of operations can noticeably influence the accuracy of the result. We encountered this in the calculation of the arithmetic mean:

$$\bar{v}_{n+1} = \frac{\bar{v}_n \times n + v_{n+1}}{n+1} = \bar{v}_n \left(\frac{n}{n+1}\right) + v_{n+1}\left(\frac{1}{n+1}\right)$$

In our first implementation the first term was calculated by multiplying the previous mean by the factor $\frac{n}{n+1}$. Since for large values of n this fraction approaches 1, the number of usable digits, and therefore the accuracy decreases. If we change the order of operations this loss in accuracy can be avoided. For the new and more precise formula we used the full fraction directly, as this minimises the number of operations and performs them in

[3] the percentage of used memory above which the check should fail

[4] KiB denotes the *Kibibyte*. 1KiB $= 1024$B

the optimal order:

$$\bar{v}_{n+1} = \frac{\bar{v}_n \times n + v_{n+1}}{n+1}$$

The reason for the difference in accuracy comes down to how floating-point numbers are represented: A floating point number is represented in binary by a rational number in the range $[01_2, 10_2[$ with a fixed number of digits and an exponent. For example the decimal number 12.3456_{10} would be represented as the binary double-precision value $1.1000101100001111001001111011011001011111111011000101_2 \times 10_2^{11_2}$

Since $\lim_{n\to\infty} \frac{n}{n+1} = 1$, [5] the binary representation will have a lot of leading 1's that have to be physically stored due to the representation and therefore occupy significant digits. Since $\lim_{n\to\infty} \frac{1}{n} = 0$ [16], the binary representation will have a lot of leading 0's that however do not have to be physically stored as significant digits as they are already represented by the exponent[3]. For that reason $\frac{1}{n+1}$ can be calculated more precisely than $\frac{n}{n+1}$. This is the reason why we decided to use the second formula instead of the first one since the second solution never explicitly calculates the ratio $\frac{n}{n+1}$ but only uses $\frac{1}{n+1}$. The graph in Figure 4.1 shows the measured error in the formula:

$$\left(1 - \left(\frac{n}{n+1}\right)\right) - \left(\frac{1}{n+1}\right) = 0$$

[5] $\lim_{n\to\infty} \frac{n}{n+1} = \lim_{n\to\infty} (1 - \frac{1}{n+1}) = 1 - \lim_{n\to\infty} \frac{1}{n+1} = 1 - 0 = 1$ [16]

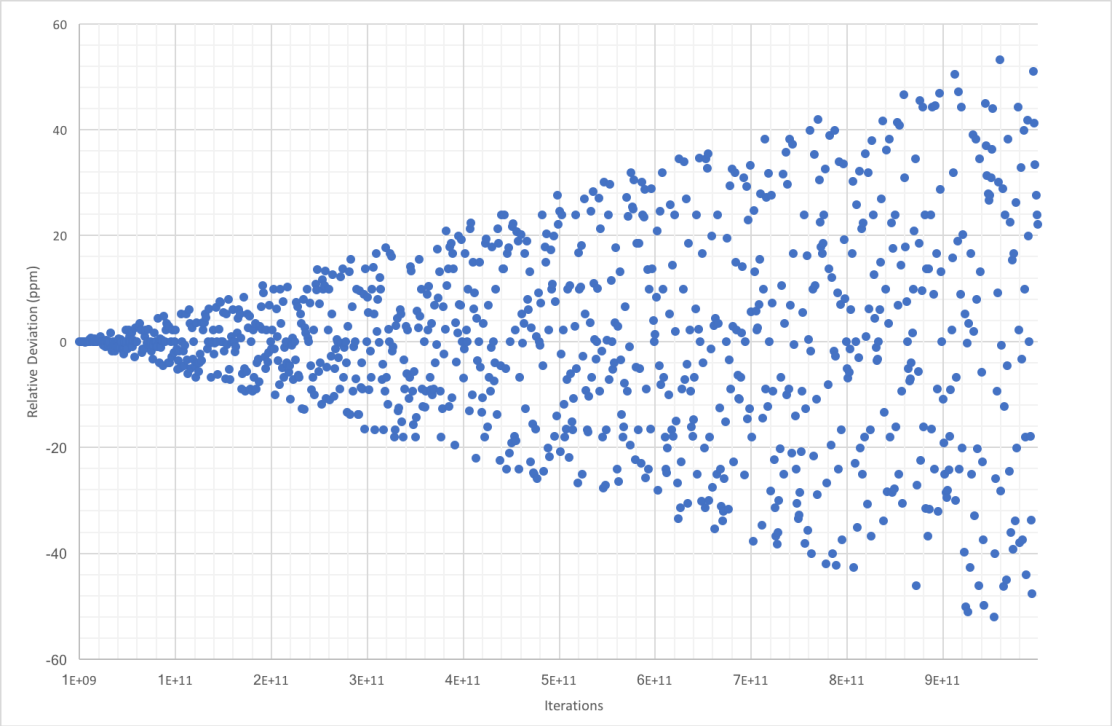

Figure 4.1: Relative Deviation of Conversion Factor Values Depending Order of Operations

4.4.4 Long-Term Stability

There is no upper limit to the number of GC runs that can be recorded other than the limitation of long integers ($2^{63} - 1 \approx 9.2234 \times 10^{18}$). Even assuming a more than unrealistic 100 GC runs per second, an overflow would only occur after approximately 2.9 billion years. It is very probable that rounding errors in the statistics would make the results unusable much earlier than that. We expect however due to our consistent use of double-precision operations and optimisation of the operations that the lifetime statistics remain usable for a long time. Since the number of significant digits in double-precision floating-point numbers is limited to a fixed 53 binary digits[3], increasing rounding errors cannot be avoided. Therefore at some point new values are so small relative to the current arithmetic mean that they become insignificant. We consider any changes in the last 15 binary digits to be insignificant. Therefore after around $2^{(53-15)} = 2^{38} \approx 2.75 \times 10^{11}$ measurements any new value becomes insignificant. Assuming once again a more than unrealistic 100 GC runs per second new measurements would become insignificant after approximately 87 years. We are therefore convinced that this technique provides usable and sensible results for runtimes of at least 1 year, very probably even for significantly

```java
package ch.awae.appcheck.api;
import java.util.Collection;
import javax.ejb.Timer;
public interface IMonitoredTimer {
   Collection<Timer> getTimers();
}
```

Code 4.3: IMonitoredTimer interface (Subsection 4.5.1)

longer. The statistics over fixed timespans do not suffer from increasing rounding errors since a finite number of operations are performed on any given value before being discarded. They will remain accurate over potentially infinite runtimes since old values are discarded and the number of GC runs in the time period will remain relatively small.

4.5 Timer Monitoring

After the garbage collection monitoring the timer monitoring was the second most challenging part of our project. The two main objectives were to check if a given list of timers actually exist at runtime and to check if the timers adhere to their defined schedule. The following subsections shall document our approaches to both objectives.

4.5.1 Timer Existence Check

To confirm the existence of timers we required a list of all timers currently running in the application. However there exists no way to get such a list directly. Next we tried to get a list of all timers associated with a given bean assuming we got access to that bean. But once again this did not work because the JavaEE `TimerService` actively denies access from anywhere but the associated bean. So in the end we decided to use the following solution:

Each EJB with timers that should be monitored must implement the `IMonitoredTimer` interface (Code 4.3) with an implementation like the one in Code 4.4.

On the class path there would be a text file listing all monitored EJBs and their timers by name. Our code would get hold of the EJB instance using the *Java Naming and Directory Interface* (*JNDI*). The EJB itself would then provide a list of its timers through

```java
public class Bean implements IMonitoredTimer {

    @Resource
    private TimerService timerService;

    @Override
    public Collection<Timer> getTimers() {
        return timerService.getTimers();
    }

    /* ... */
}
```

Code 4.4: example IMonitoredTimer implemetation (Subsection 4.5.1)

the `IMonitoredTimer` interface. *JNDI* allows looking up resources including EJBs by name[12]. That way we can dynamically access any EJB and get its timers as long as it implements the `IMonitoredTimer` interface. Finally our code checks if all required timers do exist and lists them all in the check result. Additionally for each of them the date and time of the next timeout can also be provided. This is controlled through configuration parameters.

4.5.2 Timer Execution Monitoring

As noted in Subsection 3.1.2 the `TimerService` can not be accessed externally. Additionally it does not provide any information about past invocations. To be able to monitor the individual timer based method invocations we needed a way to get notified whenever such an invocation occurs. As it is already provided by JavaEE itself we turned towards interceptors. Interceptors allow for adding cross-cutting concerns, *i.e.,* functionalities that are separate from the main functionality of a method but that are required for correct operation of the application as a whole, while retaining a good separation of concerns. A good example for a cross-cutting concern is that of authorisation checks. These checks may be necessary for ensuring the security of an application but are not directly related to the actual functionality of a method. When an interceptor is registered on a method each invocation of that method will be intercepted and redirected to a method of the interceptor. There the interceptor can inspect the origin of the invocation, the passed parameters and can decide on whether and how the actual target method is invoked. After that invocation returns the interceptor can inspect the return value. Getting that control of the method invocation would allow us to collect a lot of different information such as the time of the invocation, the duration of the execution, whether an

exception was thrown by the method, just to name a few.

There are 3 ways an interceptor can be registered on a class or method: It can be registered through the addition of an annotation in the target class directly. This solution is the most direct one but requires modification of the source code. Alternatively interceptors can be registered through an entry in a configuration xml file. This is still quite explicit but removes the source code change and collects all interceptor configuration in a single location. This also improves the separation of concern in the system since ensuring that cross-cutting concerns are covered is a requirement for the system and not for a specific class or method. The third way is to register interceptors dynamically while the application is being loaded. While loading the application the class loading of EJBs can be intercepted which allows additional annotations to the be added to the classes through reflection.

Since we wanted to be able to remove the necessity for source code changes during integration of our code we chose the third option. The idea was that the a configuration file would be read during class loading. Then we would determine the classes that required an interceptor to be added to. These interceptors would then be added through reflection.

We however quickly realised that this option brings with it a lot of complications: While multiple interceptors can be added to a single class the order they appear in influences the behaviour. Adding an annotation during class loading is in principle a code modification. Therefore we would need to make sure that that a new interceptor would not influence the behaviour of any application. Also our interceptor would need to be designed in such a way that it never fails. If it were to ever malfunction it could due to its invasiveness dramatically change the behaviour of an application.

Considering the risks of a failure and the fact that we could never guarantee that our solution would not impact any application we finally decided after a meeting with the customer that we would not pursue this approach any further. The customer also confirmed that they had never observed existing timers to skip executions but that the main issue lies in potentially disappearing timers and in timers that may not be started correctly. Therefore the existence checks (Subsection 4.5.1) already cover timer monitoring sufficiently.

4.6 Interoperability

From the beginning of the project it was clear that we wanted to be able to release our code publicly. Initially we utilised customer internal code. At some point, as it became clear that we would not be able to release the customer code this was no longer possible

and we needed to find an alternative. In search for a solution we decided to try to fully separate the two codebases in such a way that a release of only our code would contain all essential features and resources necessary for adaption for new or existing projects. We designed our own data structures to replace all dependencies on the customer codebase. We then wrote an interoperability package that contained everything necessary to provide fully bidirectional compatibility between the two codebases.

We expected our code to be integrated into existing code on their side. Therefore we needed to be able to convert our data types into the corresponding customer types. We also expected customer checkers to be added to our main checker. Therefore we also needed to provide conversion of their data types into ours.

To make the customer data structures and checkers usable in our code we used the adapter pattern. This converts the interface of a class into another interface to allow two classes to work together that usually could not due to incompatible interfaces.[10] In essence an adapter class is used to wrap one data type in a class that allows access to it as if it were supporting another interface by redirecting calls to that interface to the wrapped class and if necessary adjust parameters and return data. This solution provides full compatibility without needing to actually convert our data structures into theirs. The conversion only takes place implicitly whenever object is accessed. Additionally if a wrapped version of the customer type needed to be handed back to customer code we could save the reconversion and simply unwrap the original object. To make our checkers compatible with the customer's we also used the adapter pattern. To make our data structures compatible with the customer types we decided not to use the adapter pattern again since here a reconversion is not expected. Also by converting our data into the customer data structures we could ensure that customer code would never have to handle wrapped versions of our code, eliminating any risk of issues on their side.

5
Testing & Validation

For the isolated testing of our arithmetic methods we used unit testing to validate their mathematical and behavioural correctness. Unit testing was however not applicable for testing the output processing code and the code that extracted JMX data. All the data our code needed to work with was extracted from JMX. Since the Java Virtual Machine specific methods were not always very well documented we needed to experimentally determine the behaviour of these methods. While it would have been possible to mock the MXBeans this would not have eliminated the need for experimental testing just to ensure that the MXBean bindings would work correctly. Therefore we decided to drop mock testing completely and do all of our testing experimentally.

For basic experimental testing of our JMX related code we used small applications in Java SE. This facilitated development by eliminating long build and deployment times for each test. We felt comfortable with using Java SE since it can be seen as a subset of JavaEE and therefore code tested with Java SE should in theory also work identically in JavaEE[6]. Once we were happy with the code we would move it into the main project and test it in the JavaEE environment. These preliminary tests mainly used direct console logging. When the compatibility was confirmed we would integrate the code into the main project. The output processing code was created in a similar way by taking the result data from the processing code and converting it into the output data structure. Here console logging was again our main method for testing. At this point however we were unable to use the more convenient Java SE. We would often implement the full output processing code and then test all the steps at once. Once we were sufficiently confident

in our solution we would remove most of the console outputs and rework the remaining ones for production use.

The nature of this project made functionality testing rather difficult: Since the behaviour of our code is highly dependent on the current state of the JVM and the rest of the application we needed to create a controlled environment to test different scenarios. This was mainly done by deploying our code with a small dummy application. We considered this to be the most stable testing environment we could create without unnecessarily increasing testing complexity. For different scenarios we also included small EJBs in the deployment that were designed to create a specific JVM state. One of the most heavily used components ran a simple infinite loop continuously instantiating large numbers of small objects and discarding half of those immediately while keeping the rest indefinitely. This had two effects on the VM: A lot of new objects were created filling up *EdenSpace*(Section 3.4) resulting in a high frequency of minor garbage collection runs. The objects that were kept indefinitely would effectively form a memory leak slowly using up the *OldGen* memory pool. Over time this would lead to an increasing frequency of major garbage collection runs whilst being unable to collect any objects. This allowed for good testing of our garbage collection monitoring in both an inactive application and one with a memory leak scenario. We did notice that this testing method did not always create reproducible results.

In one situation we had an issue where sometimes the garbage collection monitoring would return a lot of `NaN` values. Sometimes this issue would resolve itself after a few moments once again providing apparently correct values. But sometimes it would persist until the application was restarted. What complicated the debugging a lot was the fact that some aspects of the monitoring are influenced by the WebService requests themselves since the output processing code runs once per request. After an extensive search we were finally able to find the underlying reason for this issue: Apparently an uncaught edge case in the formula combining several time periods of the garbage collector statistics, that was not discovered during unit testing lead to `NaN` values whenever at least one of the time periods in the statistics did not contain any data points. This explained the sudden appearance of `NaN` s in the check results when no garbage collection runs were performed since the last time period was created. These `NaN` s would disappear as soon as a garbage collection run was recorded. If however the still empty time period was finished and a new one was created, the `NaN` values would not disappear until the empty time period was considered too old and discarded.

Whilst we expected that behaviour we never observed this as our tests usually used statistics over the last 20 minutes divided into 20 one-minute time periods. Due to most tests having the synthetic memory leak active our tests rarely lasted that long since after only a few minutes the major garbage collection would run nearly continuously, resulting in the application freezing. At that point we could not get any sensible data

over the WebService as it was also subject to the same freeze. Sometimes we even needed to forcibly kill the application due to the server no longer responding to any commands.

Whilst not being a very scientific testing method we are still content with the result as it allowed us to design a desired state and compare the received results with expected ones. And most importantly it uncovered several sometimes very small but relevant errors in our code helping us improving the quality of our product. The issues with the freezing test server demonstrated the importance of a constant memory footprint very well: What we notice here in the testing environment could also happen to production servers if we did not use a constant amount of memory or accidentally create a memory leak.

6 Conclusion & Future Work

This chapter reflects on the development process and the results of the project by summarising the goals we have so far achieved and by proposing future work.

6.1 Conclusion

In short we have achieved most goals we set at the beginning of the project (Chapter 2). We were able to provide the customer with useful metrics for application and server performance. The most notable of these is probably the garbage collection monitoring (Section 4.4). Our solution for the timer monitoring is not the most elegant one as it requires source code changes on the customer side but with those adaptions it serves its purpose quite well (Section 4.5). Monitoring of the actual timer invocations is still not possible but once the existence of a timer is confirmed JavaEE should take care of guaranteeing its execution[7].

The configurability of nearly every component allows for easy customisation of the included checks. We believe that the compromises we chose for handling fuzzy data (Subsection 4.1.1) solve the difficulties with the binary results in a way that is both easy to understand and minimises false positives. Any inadequacies of the threshold based binary results should be mitigated by the fact that we also provide the raw values for every measurement. The '*strictness*' also reduces the chance of false positives by

allowing a certain number of measurements to indicate a failure before the whole check fails. An example for such a situation would be a very full *EdenSpace* just before a garbage collection run(Section 3.4), where that one full memory pool does not indicate memory issues as long as the other pools are relatively empty. Here the strictness would ignore the full *EdenSpace* and thereby preventing a false positive as the check would still be considered passed.

We have laid a solid foundation for future customisation and expansion and while we already provide several checks there still room for improvement (Section 6.2).

At the time of publication our module has not yet been integrated into the customer's production code. We tried to provide everything needed for a successful integration but the long-term stability and usability in a production environment still has to be determined. While we have not tested our code with real production code we believe that there will be no major issues in production as our dummy application followed the same architecture as *ISC* production code and also included representations for most of the relevant components.

6.2 Future Work

The first open item we see potential for is the timer execution monitoring as discussed in Subsection 4.5.2. While we did not pursue this interceptor based approach we see its potential for providing more information about the timers, such as the last time the timer was executed, potential exceptions thrown during execution or the duration of the invocation. A future expansion of our project could therefore see the addition of this feature – even if in some projects it may not be used due to its invasiveness.

Another monitoring option would be real-time memory monitoring where precise information about each memory pool at the time of the check could be provided. We did not include this in our solution since the garbage collection monitoring already provides very detailed memory information (Section 4.4). Especially in case of memory issues – where memory information is the most relevant – it is kept up to date by the frequently running garbage collectors.

In the current state anyone who wishes to integrate our code into their projects has to implement a compatible WebService themselves. Therefore we should probably provide one ourselves so that a basic integration does not have to worry about that. Only if a custom WebService is desired ours would have to be disabled in its favour. This has not been a priority for us since the *ISC* will be integrating our code into their existing internal monitoring services (Section 4.6). But providing our own WebService would clearly increase the value of our project as a ready to use package.

Our implementation of the garbage collection monitoring (Section 4.4) directly accesses information of the different memory pools(Section 3.4). Due to the changes in HotSpot for Java 8 our code is only compatible with Java 7. To ensure proper behaviour in future we recommend to either provide a Java 8 version or alternatively generalise the memory monitoring to ensure compatibility with Java 7 and Java 8.

Lastly we propose to extend our solution to recognise failure patterns and to indicate probable causes. This could help reducing debugging times by already pointing developers in the right direction. One approach might be the use of pattern recognition using a classification algorithm[1].

Bibliography

[1] Geoff Dougherty. *Pattern Recognition and Classification*. Springer Science & Business Media, October 2012.

[2] David Flanagan. *Java in a Nutshell*. O'Reilly, 5th edition, March 2006.

[3] S. K. Ghoshal. *Understanding the IEEE 754 floating point number system*. Supercomputer Education & Research Centre, Indian Institute of Science, http://140.129.20.249/%7Ejmchen/NM/refs/ieee754.introduction.pdf, March 1997. [Online; accessed 3-September-2017].

[4] Harry-The Anonymous Hacktivist. Harry. H. Chaudhary. *Cracking The Java Programming Interview*. Oracle Press, July 2014.

[5] Sheng Liang and Gilad Bracha. *Dynamic Class Loading in the Java Virtual Machine*. Sun Microsystems, Inc., October 1998.

[6] Oracle. *Your First Cup: An Introduction to the Java EE Platform*. Oracle Corporation, http://docs.oracle.com/javaee/6/firstcup/doc/firstcup.pdf, 2012. [Online; accessed 3-September-2017].

[7] Oracle. *The Java EE 6 Tutorial*. Oracle Corporation, http://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf, January 2013. [Online; accessed 3-September-2017].

[8] Oracle. *HotSpot Virtual Machine Garbage Collection Tuning Guide*. Oracle Corporation, http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/title.html, March 2015. [Online; accessed 3-September-2017].

[9] David Pollard. *A User's Guide to Measure Theoretic Probability*. Cambridge University Press, 2002.

[10] Vaskaran Sarcar. *Java Design Patterns*. Apress, 1st edition, 2016.

[11] Brian Suda. Soap web services. Master's thesis, School of Informatics, University of Edinburgh, http://suda.co.uk/publications/MSc/brian.suda.thesis.pdf, 2003. [Online; accessed 3-September-2017].

[12] Sun. *Java Naming and Directory Interface Application Programming Interface*. Sun Microsystems, Inc., http://www.oracle.com/technetwork/java/jndi-150206.pdf, July 1999. [Online; accessed 3-September-2017].

[13] Sun. *Java Management Extensions (JMX) Specification, version 1.4*. Sun Microsystems, Inc., https://docs.oracle.com/javase/8/docs/technotes/guides/jmx/JMX_1_4_-specification.pdf, November 2006. [Online; accessed 3-September-2017].

[14] Sun. *Memory Management in the Java HotSpot Virtual Machine*. Sun Microsystems, Inc., http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf, April 2006. [Online; accessed 3-September-2017].

[15] Sun. *JSR 318: Enterprise JavaBeans, Version 3.1*. Sun Microsystems, Inc., December 2009.

[16] Christiane Tretter. *Analysis I*. Springer Basel AG, 2013.

[17] Bart Baesens; Aimee Backiel; Seppe vanden Broucke. *Beginning Java Programming*. John Wiley & Sons, February 2015.

A

Anleitung zu wissenschaftlichen Arbeiten

With this guide we aim to explain the installation, configuration and customisation of AppCheck for use in a JavaEE project. We assume basic knowledge of JavaEE and will not explain how to set up a development environment or how to start with JavaEE. There are several good guides available online. For a very detailed tutorial we recommend "*Java Platform, Enterprise Edition: JavaEE Tutorial*" by Oracle available at `https://docs.oracle.com/javaee/7/JEETT.pdf`.

Please note that at the time of writing the source code has not yet been released. The public release is expected to be completed by the end of September 2017. The information in this document is based on experience in our own development environment and the referenced JavaEE Tutorial. For the sake of simplicity we will for the remainder of this document treat our code as if it were already available.

A.1 Installation

Here we cover the basic set up of AppCheck for use in a JavaEE application. AppCheck can be integrated by either adding it as a Maven dependency or by simply copying the AppCheck source code into your project.

A.1.1 Maven Dependency

AppCheck can be added to a JavaEE application as a Maven dependency. This is the simplest way to start with AppCheck. Add the following dependency into your `pom.xml` file:

```
<dependency>
    <groupId>ch.awae</groupId>
    <artifactId>appcheck</artifactId>
    <version>LATEST</version>
</dependency>
```

A.1.2 Source Integration

AppCheck can also be added to a project by directly copying the complete source code. While this is a bit more complicated it facilitates changes in the provided checks. Download and extract the source code from `http://www.awae.ch/appcheck`. Once it is extracted, copy the package `ch.awae.appcheck` and all subpackages into your project source. Make sure to also adjust the `enterprise-beans` section of your `ejb-jar.xml` file to ensure the correct EJB bindings:

```
<session>
      <ejb-name>CheckerDataEJB</ejb-name>
      <ejb-class>ch.awae.appcheck.data.CheckerDataBean</ejb-class>
      <session-type>Singleton</session-type>
      <transaction-type>Container</transaction-type>
</session>
<session>
      <ejb-name>RootCheckerEJB</ejb-name>
      <ejb-class>ch.awae.appcheck.RootCheckerBean</ejb-class>
      <session-type>Singleton</session-type>
      <transaction-type>Container</transaction-type>
</session>
```

A.2 Using AppCheck

AppCheck has a single entry point in the form of the method `doCheck(String uid)` in the Singleton EJB

`ch.awae.appcheck.RootCheckerBean` . The `String` parameter specifies a universal ID that will be mentioned in any log message from the checkers. This allows these log messages to be correctly associated with each other.

A.3 Configuration

AppCheck depends on a few configuration files for correct operation. An example configuration file is available at `http://www.awae.ch/appcheck`. Make sure to include that file as a resource on the classpath – it can also be renamed. If Timer monitoring is desired (Section 4.5), make sure to also add a simple text file for the timer configuration.

A.3.1 Runtime Parameters

For providing maximal flexibility the names of the configuration files are not predetermined and may be chosen freely. The names of the configuration files are read from the system properties `ch.awae.appcheck.props` and `ch.awae.appcheck.timers` . Therefore the names must be passed as VM options like `-Dch.awae.appcheck.props=appcheck.properties` and `-Dch.awae.appcheck.timers=timers.txt` . Without these two parameters AppCheck will load correctly.

A.3.2 Base Configuration

AppCheck is configured through the `*.properties` file described above. Given the length of the configuration file and the number of explanatory comments we have included we will not go through all of them, but rather give an overview over the mechanics and features. We also cannot give a definitive guide to finding the optimal configuration for any given project. The provided example file is designed to work relatively well in most cases.

Enabling and Disabling Checks

At the top of the section for each check you will find a parameter like `check.gc.enabled=true` . This parameter controls if the associated check is actually performed. Changing it to `false` will disable the check.

Strictness

The strictness parameters control the summary result mapping as described in Subsection 4.1.1. All strictness values are number between 0 and 1 and indicates the relative number of subchecks that must pass for a check to pass as well. A strictness of 0.8 for example specifies that at least 80% of the subchecks must pass for a check to pass.

The value `check.common.strictness` defines a *default* strictness.

The value `check.root.strictness` defines the *root* strictness, *i.e.,* the strictness for the `RootCheckerBean` . Since all checks are combined into a single *root* check that *root* check must have a binary result as well. The *root* strictness specifies the relative number of checks that must pass for the *root* check to pass as well. In addition to a numeric value the parameter can also be set to `default` . In that case the *default* strictness will be used.

Garbage Collection Strictness

The top-level strictness is controlled by the `check.gc.root_strictness` . This specifies the strictness that is used for combining the results from each of the two garbage collectors. We recommend to use a value of 1 to ensure that a failed garbage collector check fails the complete check as well. A value of `default` is also allowed.

The second-level strictness is controlled by `check.gc.strictness` . It controls the strictness to be used when combining the subchecks that make up the check of a single garbage collector. It can also be set to `default` .

The third-level strictness is controlled by `check.gc.minor.strictness` and `check.gc.major.strictness` . These 2 parameters control the strictness with which the results for the different measurements[1] for the 2 garbage collector types. They can both be set to `default` to use the *default* value or to `inherit` to use the value of `check.gc.strictness` .

The fourth-level strictness is controlled by `check.gc.minor.innerStrictness` and `check.gc.major.innerStrictness` . These 2 parameters control the strictness for combining the individual checks for the values in the different statistics into a single value per measurement. They can both be set to `default` to use the *default* value or to `inherit` to use the value of `check.gc.minor.strictness` or `check.gc.major.strictness` respectively.

[1] including garbage collection frequency, memory pool usage, garbage collection duration

GC Thresholds

The memory thresholds for each memory pool[14] are controlled by the parameters named `check.gc.?.max?` . They define the limit for each memory pool. A value of 0.8 for example specifies that the corresponding memory pool may only be used to up to 80% of its capacity.

The `maxFrequency` parameter specifies the maximum number of garbage collector runs per second. The `maxDuration` parameter specifies the maximum duration for a single garbage collector run in milliseconds.

GC Statistics

The garbage collection statistic are controlled by the parameter `check.gc.stats` . This parameter holds a semicolon separated list of configurations. Each entry defines a statistics instance to be created. The structure of an entry is:

```
title,duration,frames;
```

The `title` is provided in the result, the `duration` defines the duration in milliseconds for the statistics and the `frames` value defines the number of frames to be used. A `duration` of -1 leads to an infinite – or lifetime – statistics. For example an entry of `last day,86400000,24;` defines a statistics over 24 hours divided into 24 frames.

exactSize and exactTime

At multiple points there exist parameters named `exactSize` and `exactTime` . These control the formatting of time and data size values. If set to `true` the exact times in milliseconds and data sizes in bytes are provided, if set to `false` the times are provided in a `d hh:mm:ss.uuu` format and data sizes rounded to the largest smaller multiple of 1024. We recommend to set them to `false` to improve readability and to only set them to `true` for debugging.

Thread Modes

The parameter `check.thread.cpuTime.mode` specifies the amount of details provided in the result. It has 3 possible values:

- `none` – only the thread counts are provided

- `summary` – only the thread counts and the sum of all the CPU time measurements from all threads are provided
- `all` – the thread counts, the sum of all CPU times and a list of all threads with their CPU times and thread information is provided.

A.3.3 Timer Configuration

The timer configuration consist of a simple text file containing a list of all timers that need to be monitored. Each line provides an entry. Each entry consists of a JNDI reference to an EJB and the name of a timer. For example the entry `java:module/TestTimerEJB;myTimer` references the timer named `myTimer` in the EJB with the name `TestTimerEJB` in the current module. For a guide to using JNDI please consult the JavaEE Tutorial referenced in to beginning of this document.

A.4 Adding Custom Checks

To add custom checks please implement the interface `ch.awae.appcheck.api.IChecker` . This way compatibility is ensured. To add the check simply call the method `addChecker(IChecker checker)` in the Singleton EJB `ch.awae.appcheck.RootCheckerBean` . Since this class is an EJB it can be accessed by Resource Injection.

”Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.”