

# Profiling, Monitoring and Tracing in SAP Web Application Server

Johannes Passing

*Seminar System Modeling 2005*

*Hasso-Plattner-Institute for Software Systems Engineering*

## Abstract

*After shortly describing and distinguishing the commonly used terms Monitoring, Tracing, Logging and Profiling, the paper will describe their role in large SAP installations.*

*Finally, some of the monitoring and tracing features provided by the SAP Web Application Server (Web AS) for Java will be presented.*

**Keywords:** Alert Monitor, CCMS, DSR, GRMG, JARM, JMX

## 1. Introduction

The ability of a software system to both offer insight into the state of the system and to allow administrators as well as developers to easily spot and solve problems is an important aspect of a *supportable* system (see [JPW]).

Features and tools offered for this purpose may be classified into two categories – problem detection and problem analysis. Whereas the former includes techniques to identify or even anticipate problems, the latter deals with finding the reasons behind the problems identified (see [SDN0], [SDN1]).

## 2. Profiling, Monitoring and Tracing

This section will describe the concepts of tracing, logging, profiling and monitoring and will show the differences between these concepts.

### 2.1. Tracing versus Logging

Though logging and tracing are similar in implementation, they are used for different purposes (see [SDN2]).

Logging is the most common technique used for reporting the state of a running system. Programs or whole systems report normal and especially exceptional events to a sequentially organized persistent storage – often a flat file.

The log can both act as history – as often applied in case of Web servers and, if checked regularly, as a source for determining misconfiguration of the system. As such, logging is mainly considered a technique for problem detection. Though the verbosity of logging can be adjusted, logging is rarely turned off – logs are written by staging systems as well as production systems.

In contrast to logging, which is mainly targeted at system administrators, tracing is a means for developers to track down misbehaviour of running systems – thus being a technique for problem analysis. Rather than just reporting the occurrence of certain events, traces serve to reveal the program flow of a certain component by reporting each action performed. In the case of multi-threaded systems, these records have to contain information about the caller-thread. Based on this information, the developer may examine the actions performed by each thread separately.

If tracing records contain timing information – such as the time spent by a specific action, traces may also serve for performance analysis. In this case, however, the distinction between tracing and profiling is blurred.

Since tracing may generate immense amounts of data, it is mostly performed on an application or component basis rather than on a system basis. Further more, tracing can pose significant additional load on the system. To avoid this overhead, tracing is – in contrast to logging – switched off during normal operation and is only activated on demand.

## 2.2. Profiling

Profiling deals with analyzing the performance of certain components or complete systems. During development, profiling is used to identify portions of code that consume the most time. As a common rule of thumb, 80 percent of the execution time of a program is spent in 20% of its code.

To identify these parts, most profiling tools observe a program by measuring both the time consumed by a procedure and the number of times the procedure has been called. Using this technique, 'hot' procedures may be easily identified. Identifying these 'bottlenecks' may well reveal weaknesses of the program such as poorly performing algorithms or bad locking strategies.

In production mode, profiling may be used to determine the overall performance of a system – this includes measuring response times of certain components or web pages as well as database queries. This information can be especially useful if gathered regularly to compare the results over certain time intervals or to contrast response time during high-load hours and low-load hours.

## 2.3. Monitoring

The intention of monitoring is mainly to identify the 'health' of a running system. To accomplish that, data is to be retrieved from different sources. This may include analyzing log files for the frequency of severe errors or using data provided by performance counters. Further more, applications may be instrumented to provide monitoring data themselves.

Though the exact usage of the term differs, monitoring is not a replacement for logging and tracing, but rather embraces these techniques.

As such, it is greatly supported by tools used to aggregate and analyze the data provided as well as presenting the user an overview of the overall state of the monitored system. Moreover, these tools often include the ability to generate reports and statistics.

Depending on the information contained, these reports may not only be targeted at system administrators, but may also serve as a valuable resource for management.

## 3. Monitoring infrastructure in SAP installations

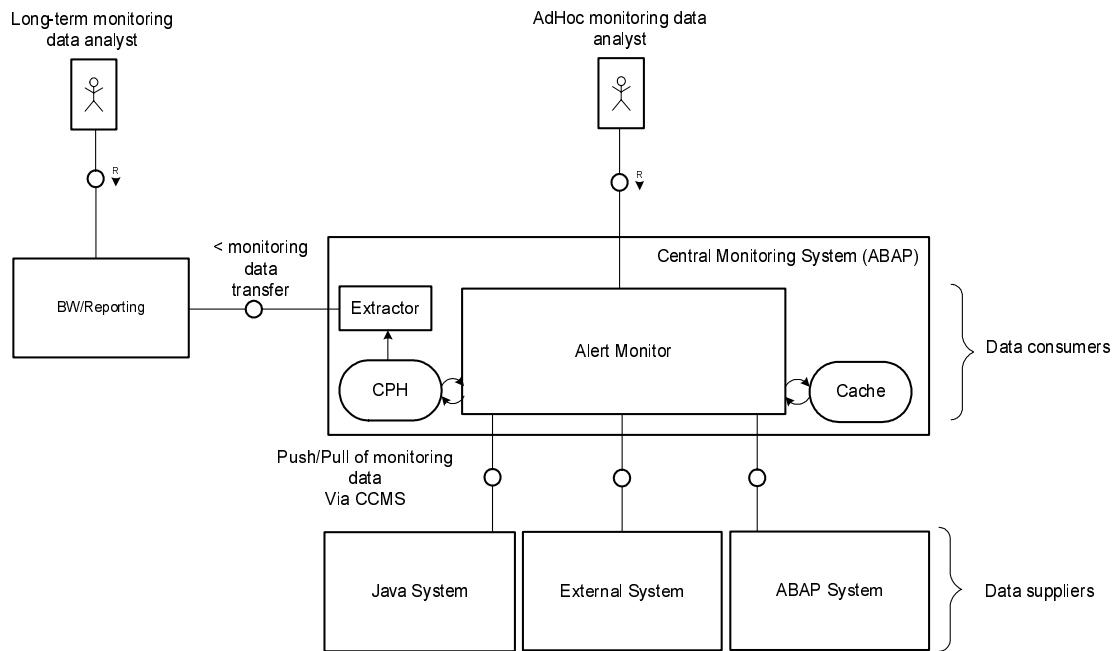
A system landscape – especially in the SAP environment – seldom consists of only a single server (see [SDN3]). However, maintaining and monitoring multiple servers, including clusters, different operating systems and different software puts greater demands on a monitoring system.

### 3.1. Computing Center Management System (CCMS)

The SAP Computing Center Management System (CCMS) accomplishes the task of monitoring large installations by providing an infrastructure that allows monitoring of both different SAP systems as well as external systems such as a database server.

The components of a CCMS system may be split in three roles (see [SAPL], [SDN3])

- *Data suppliers*  
The system to be monitored is considered a data supplier (shown at the bottom of fig. 1) – during its normal operation, the system collects different sorts of monitoring data which may be used by Data Consumers. The data is made accessible by being stored in a machine-wide known shared memory segment used by all Data suppliers on the same machine.
- *Data consumers*  
Data consumers (shown in the middle of fig. 1) gather monitoring data provided by data suppliers for the purpose of analysis – this data is either obtained by reading from the local monitoring shared memory segment or from CCMS agents. One of the most commonly used data consumer is the *Alert Monitor*.
- *CCMS agents*  
Whereas the monitoring data provided by suppliers is only accessible on the local machine, CCMS agents (not shown in fig. 1), which are independent processes running on the monitored systems, provide access to this data to remote machines. The usage of CCMS agents allows running suppliers and consumers on different machines and thus makes central monitoring of distributed systems feasible.



**Figure 1: CCMS architecture (own illustration)**

### 3.2. CCMS Data Consumers

In order to centralize monitoring facilities and not to put additional load on production machines, one ABAP-based system may be declared as the *Central Monitoring System (CEN)* (see [SAPL], [SDN1], [SDN3]). The CEN is dedicated to only contain monitoring-related components which mainly act as data consumers.

Essential part of a CEN and the most commonly used data consumer is the Alert Monitor.

Besides accessing the local shared memory segment, the Alert Monitor queries CCMS agents running on remote machines via *Remote Function Calls (RFC)* for monitoring data. In recent releases, CCMS agents may also 'push' their data collected to the CEN in certain time intervals. To improve performance, all data obtained is held in a local cache on the CEN. Though data may be held in the cache for up to 24 hours, the Alert Monitor's responsibility is only to provide an ad-hoc view of the system.

In order to store monitoring data to build up a history, the *Central Performance History (CPH)* may be used. The CPH is a database that stores monitoring records over longer periods of time. Based on this data, reorganization and aggregation may be performed, as well as creating reports showing the evolution of data. To

allow even more thorough analysis, *Extractors* allow CPH data to be transferred to the Business Warehouse.

Just as the Alert Monitor, the CPH is located on the CEN.

#### 3.2.1. Alert Monitor

Whereas the actually monitored entities as well as the data they provide may differ, the CCMS provides a common model to which data is mapped (see [SAPL]).

A monitored system is considered to consist of several monitored entities - referred to as *monitoring objects*. While a monitoring object identifies such an entity, it does not provide any data - instead it acts as container for at least one *monitoring attribute*. Each attribute is named and yields the value for one specific trait of the object.

A common example for a monitoring object is a Central Processing Unit (CPU) of a server containing an attribute describing the current load.

A *CCMS monitor* is a tree-structured set of *monitoring tree entries (MTE)*. While monitoring objects and monitoring attributes constitute the two bottom-most layers of this tree, virtual nodes may be created to further structure the tree. A special kind of virtual nodes are the summary monitoring tree entries - these aggregate information provided by their child nodes.

To each attribute, threshold-values may be assigned. The graphical user interface (GUI) of the Alert Manager will show attributes in different colors, depending on whether its value has exceeded its threshold or not. In case of a critical value, the attribute as well as all its parent nodes will be highlighted in red color. The user of the graphical user interface will thus immediately notice the existence of the critical value and can easily locate the corresponding attribute by navigating down the tree along the highlighted nodes.

Further more, *methods* may be assigned to monitoring tree entries, which will be triggered if certain alerts - such as the exceed of a certain threshold - occur.

In conjunction with *SAP Central Alert Management (ALM)* framework, the administrator may be automatically notified of the occurrence of such events in a variety of ways - this includes sending e-mails, fax or SMS messages (see [SDN3]).

### 3.3. CCMS Agents

Monitoring data provided by data suppliers is - as mentioned above - stored in a shared memory segment which is well known on the local machine.

However, this technique has two implications:

- Data may only be accessed by consumers executing on the same machine
- Programs have to be implemented to make use of the shared memory segment - this implies that by default, third-party programs cannot be monitored this way.

CCMS agents address both these issues. The agent is an independent process executing on the same machine as the system to be monitored. It attaches itself to - or creates in the lack of existence - the local shared memory segment (see [SAPL], [SDN3]).

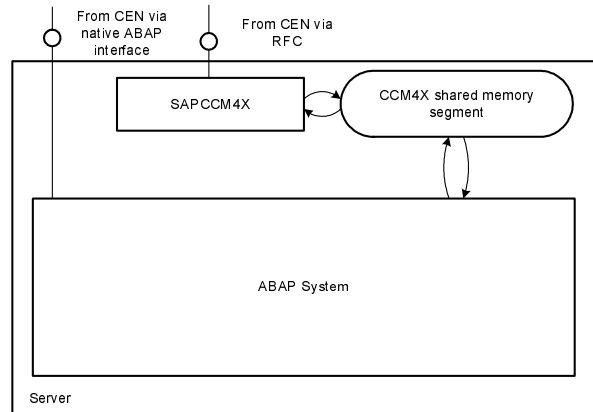
By acting as a RFC server, consumers such as the Alert Monitor may actively request data - this scenario is referred to as 'pulling' monitoring data.

By acting as a RFC client, recent releases may 'push' data by periodically sending data - again, the Alert Monitor uses this technique.

Depending on its usage, three different agents may be used - namely SAPCM3X, SAPCCM4X and SAPCCMSR.

#### 3.3.1. Monitoring an ABAP 3.x or 4.x system with SAPCM3X or SAPCCM4X

As these systems are implemented to use the CCMS architecture, monitoring data is directly written to the shared memory segment (see fig. 2). Thus, the main responsibility of the CCMS Agent is providing access to this data for remote consumers.



**Figure 2: Monitoring an ABAP system using SAPCM3X/SAPCCM4X (own illustration)**

Though not discussed in greater detail, these systems offer the ability to provide access to monitoring data via a native ABAP interface as well. However, to improve robustness and performance, using CCMS agents to accomplish this is now considered the preferred way.

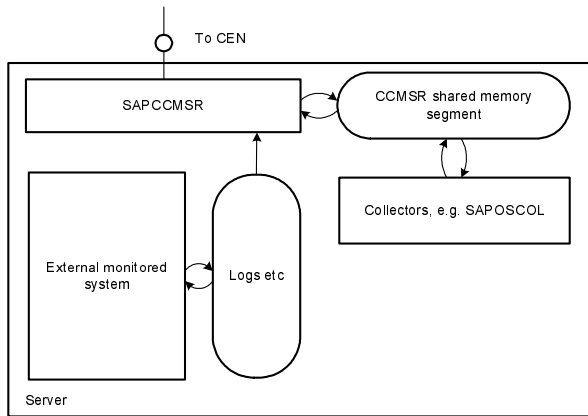
#### 3.3.2. Monitoring an external system using SAPCCMSR

An external system to be monitored may be a database server. However, these systems cannot make use of the services provided by CCMS.

Thus, these systems report their state in their usual way - for example, by writing log files (see fig. 3). The SAPCCMSR agent provides the ability to analyze these log files and write the results to the local shared memory segment. Another option is to set up the agent to receive *Simple Network Management Protocol (SNMP)* traps issued by the system to be monitored.

Just as the SAPCM3X and SAPCCM4X agents, this data is then made accessible via an RFC interface.

Consequently, SAPCCMSR agents solve the second issue raised above and facilitate the transparent integration of external systems in the CCMS monitoring architecture.



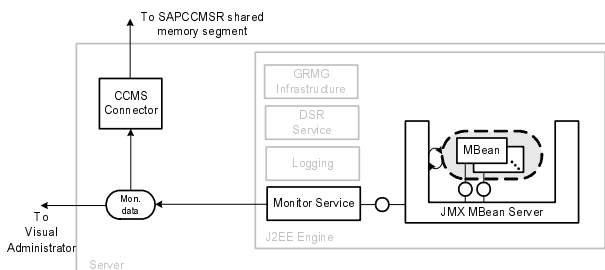
**Figure 3: Monitoring an external system using SAPCCMSR (own illustration)**

Beside the features provided by the agent itself, further *collectors* may be used – a collector is another independent process collecting monitoring data and writing the results to the memory segment. The most commonly used collector is the *SAP Operating System Collector (SAPOSCOL)*, which gathers data from the underlying operating system (shown at the right of fig. 3).

## 4. Web AS Java

Being a J2EE Application Server, Web AS Java not only offers features known from ABAP systems, but also introduces standardized technologies like *Java Management Extensions (JMX)* to the SAP monitoring infrastructure.

This section will present some of these technologies and will show how they integrate with the CCMS architecture.



**Figure 4: Usage of JMX within the J2EE engine (own illustration)**

## 4.1. Monitoring a Web AS Java system

### 4.1.1. Java Management Extensions

J2EE engines, of which one or more may coexist on a single machine, use the JMX infrastructure to provide standardized interfaces both for administration and monitoring (see [JPW], [MW], [SAPL]).

In a nutshell, JMX provides a central authority (the MBeanServer), which MBeans may be registered to (see fig. 4). An MBean wraps a specific resource to be managed. It consists of a Java class implementing one of a set of JMX interfaces and, through the attributes and methods defined herein, allows querying the state as well as adjusting settings of this resource. The MBeanServer in turn allows clients to indirectly interact with these resources by acting as a broker delegating requests to the corresponding MBeans. Further details on JMX may be obtained in [JMX].

While administrative usage both reads and writes attributes of MBeans as well as invokes its methods, monitoring is mainly based on querying data exposed via attributes.

As a result of being based on JMX, tools may access and manage MBeans by directly connecting to the MBeanServer. For remote access, adapters such as the SAP RMI/P4 adapter may be used.

In the case of a solely J2EE-based landscape, exclusively using these tools may be an attractive and lightweight alternative to using CCMS.

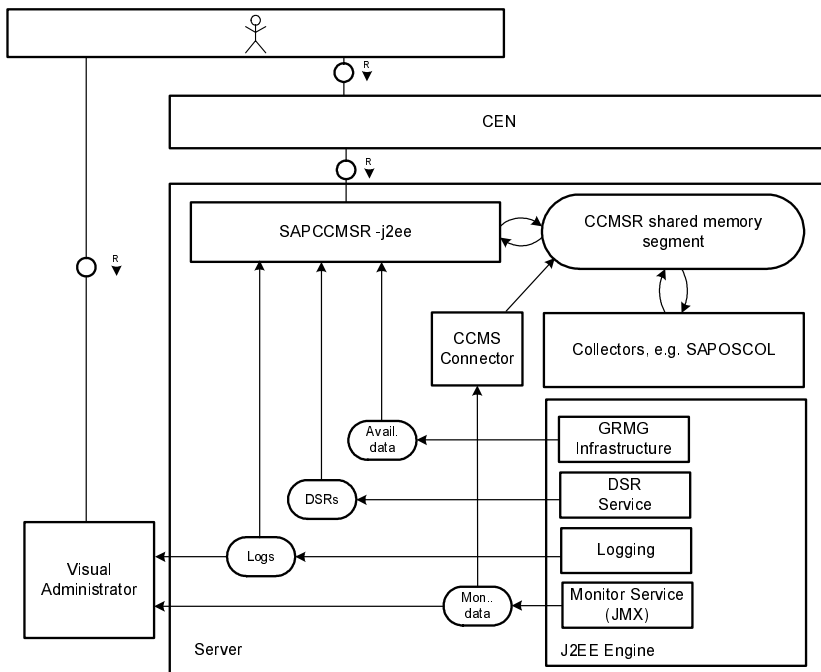
Further more, being a widely adopted industry standard, JMX greatly lowers the burden for third-party component developers to make use of the monitoring infrastructure.

Though JMX offers the ability of monitoring a J2EE system such as SAP Web AS without the need for a CCMS infrastructure, there are still strong reasons for integrating JMX technology with the CCMS – the most obvious and compelling reason being the need to integrate J2EE systems with an existing, non-J2EE SAP system landscape.

### 4.1.2. CCMS integration using SAPCCMSR

Besides offering the ability to monitor an external system, the SAPCCMSR agent provides a *-jee* switch, activating further features specific to monitoring a J2EE system.

One of these features is the integration of the JMX-based monitoring infrastructure used by



**Figure 5: Monitoring a Web AS Java system using SAPCCMSR -j2ee (own illustration)**

J2EE systems with the mentioned CCMS-based infrastructure.

Though the implementations of these two concepts differ, both share the concept of monitors providing data through attributes. Using the CCMS Connector (see fig. 5), monitoring data obtained from the JMX infrastructure is transformed and transferred to the CCMS shared memory segment - being available in this segment, different machines and tools may access this data using the features of CCMS agents already discussed.

## 4.2. Problem detection in Web AS Java

### 4.2.1. Logging

During normal operation, the Web AS server writes logs. On the one hand, these logs originate from several components of the server itself, on the other hand, developers may use the *SAP Logging API* to include logging features in their applications (see [JPW], [SAPL]).

By setting the verbosity level, the administrator may adjust the granularity and amount of information written.

These logs may be either viewed using the *Log Viewer* or - if CCMS is used - may be analyzed by the SAPCCMSR agent.

### 4.2.2. Distributed Statistics Records

On ABAP systems, a concept called *Statistics Records* is used to record information about the workload generated and the resources used by specific actions (see [SAPL], [SDN2], [SDN3]). Based on this, information about the overall state of a system may be derived. Further more, the root causes of problems such as performance bottlenecks may be determined by analyzing these records.

*Distributed Statistics Records (DSR)* are an enhancement of Statistics Records that may be used on both ABAP and J2EE systems.

To make use of DSR in case of a system spanning multiple machines, correlating records collected on different machines must be identifiable to be grouped together - in case of a *Logical Unit of Work (LUW)*, in which components on different machines are involved, all records created on the affected machines must be combined to determine the workload generated by the LUW as a whole.

To facilitate this, a *passport* is sent along all communications such as RFC calls. Along further information such as the user ID of the user triggering the LUW, the passport contains a *Globally Unique Identifier (GUID)*, which uniquely identifies the LUW. Based on these passports, correlating records generated on different

machines may be identified.

Besides the *Certificate Subrecord*, which corresponds to the passport, a Distributed Statistics Record contains a *Main Record* as well as any number of *Call Subrecords*. Whereas the Main Record contains the actual statistical information about actions performed, Call Subrecords contain information about other components called.

The *DSR Service*, which generates the DSR, stores all records in files, which in turn can be read by the SAPCCMSR agent to be made available to the CEN.

Using the *Global Workload Monitor*, the data collected by all machines may be aggregated and analyzed on the CEN.

#### 4.2.3. Generic Request and Message Generator

Another important aspect of maintaining a production system is to guarantee availability. As a web server, Web AS has to face the challenge of handling significant amounts of concurrent requests during peak hours – this not only implies that the server as a whole must be able to serve all these requests, but also that every single component used by any of these requests must be able to respond within the allotted time.

Though most requests may be handled with only short delay, it is possible to have certain components not responding in such a situation due to problems like lock contention.

To identify these components and problems, the *Generic Message and Request Generator (GRMG)* may be used (see [SAPL], [SDN1]).

The idea behind GRMG is to create a *GRMG application* (shown at the top right of fig. 6), mostly implemented as a JSP page or Servlet, that may run different *scenarios* – that is, the component determines the availability of a single or a small number of components.

GRMG applications are called via HTTP, using an XML-based format for both request and response. Whereas the request contains the name and additional information about which of the scenarios implemented to run, the response contains a result code along with further messages generated during the measurement process. To simplify handling the XML formats, SAP provides an API for reading and encoding GRMG messages.

The *GRMG infrastructure* (shown at the left of fig. 6), which may also be located on the CEN, calls these applications and thus invokes their

measurement process (see fig. 6). As the communication is based on HTTP, these applications may be hosted on the same machine as well as on different machines. The results received are aggregated and may be made available to the Alert Monitor.

If performed periodically, data obtained by GRMG applications provide a good overview about both the availability and the responsiveness of the system during different levels of load.

### 4.3. Problem Analysis in Web AS Java

#### 4.3.1. Tracing

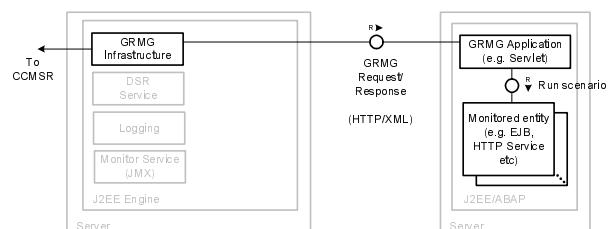
Though plenty of techniques exist for finding and eliminating failures in software during development, it is hardly inevitable to find failures in an application when it has already been deployed to a production machine.

As the Java Virtual Machine does not support the attachment of a debugger unless it is launched in debug mode, using a debugger for troubleshooting applications deployed on a productive Web AS server ‘on the fly’ is not possible.

To compensate for this shortcoming, Web AS offers various tracing techniques. Providing thorough information about the execution of a component, these techniques aim at providing the developer enough information to spot the failure and thus eliminating the need to restart the machine in debug mode. Further more, tracing is extensively used for performance measurement.

#### 4.3.2. SQL Tracing

The *Open SQL Engine* acts as an abstraction layer above the actual database drivers used. As such, it compensates for the differing implementation and SQL dialects used among drivers of different vendors (see [BS], [JPW]).



**Figure 6: Monitoring availability using GRMG (own illustration)**

As all SQL queries pass through the Open SQL Engine, detailed statistics about the frequency and the load generated by queries may be gathered. This information is made available through a separate web-based user interface.

Complex and inefficient SQL queries may consume a significant amount of time and can thus slow down the overall performance of an application - analyzing the information provided may therefore be of great use when spotting performance bottlenecks.

Just as other tracing techniques, SQL tracing may be enabled or disabled at any time.

#### 4.3.3. Java Application Response Time Measurement

Components may be manually instrumented for performance-tracing by using *Java Application Response Time Measurement (JARM)* (see [SAPL], [SDN1]). The concept of this API is to have the component first acquire a named *monitor* and to enclose each processing part of a request or method invocation by calls to `IMonitor.startComponent()` and `IMonitor.endComponent()`. Calls to these methods may also be nested if, for example, further methods are involved.

As the time elapsed between each two corresponding calls is recorded, an application instrumented with JARM provides detailed information about how much execution time has been spent in which part of the component.

Using tools like the Visual Administrator, this information may be aggregated and analyzed - this includes calculating the average time consumed by a request or displaying those requests having consumed most execution time.

#### 4.3.4. Single Activity Trace

If components have been instrumented with JARM, the *Single Activity Trace (SAT)* can be used for cross-component tracing (see [JMI], [JPW], [SAPL]). Similar to the concept of DSR, SAT allows the inspection of a specific request involving several components by analyzing the information provided by JARM.

Using SAT is thus especially useful if a certain workflow - such as a specific HTTP-Request - has shown to be especially expensive.

In this case, the administrator may choose to enable SAT for this request to have all JARM data collected. Assumed that each component

involved has been instrumented using JARM, the component consuming most execution time and thus causing weak performance may be located. Using techniques like Application Tracing or debugging, this part of code may then be further analyzed.

#### 4.3.5. Application Trace

Using the *Application Trace*, a developer may observe a component at the level of method calls (see [JPW], [SAPL]). However, the component itself does not have to be manually instrumented for tracing purposes as in the case of JARM - instead, the required code is injected automatically by the J2EE engine on demand.

To enable tracing, the server unloads the selected component and searches for the corresponding byte code. A copy of the byte code is then instrumented by being added tracing statements to each method contained. After the byte code enhancement has been performed, the application is started again.

While performing its normal task again, the application now writes tracing records. Beside the names of a method being called, these records also contain the time spent within the method.

When the desired information has been gathered, tracing can be disabled again - the server will then reload the application using the original code.

Based on the information collected, tools like Visual Administrator may reconstruct the flow of execution of the component - the developer is now able to browse the tree of nested method invocations that were performed on each single thread.

In combination with a byte code decompiler, this may well serve as an alternative for an interactive debugger in the case of troubleshooting a production system.

Besides giving thorough insight into a component, the Application Trace is also a valuable means for locating performance bottlenecks. By revealing the number of times a method has been called as well as its time consumption, the Application Trace greatly simplifies pinpointing those methods that are called extremely frequently as well as poorly-performing methods.

## 5. Summary

Through its numerous logging, tracing and monitoring features, the SAP Web Application Server offers a variety of techniques supporting both administrators and developers in developing and maintaining a stable and responsive system.

Monitoring, tracing and profiling techniques offered by Web AS as well as those techniques provided by other SAP and external systems may be integrated using the proven CCMS architecture, which may greatly simplify the maintenance of large, distributed and heterogeneous installations.

## References

[BS] Bernd Schäufele, J2EE persistence mechanisms of the Java Web AS, HPI Seminar System Modelling 2005, Hasso-Plattner-Institute for Software Systems Engineering, 2005.

[JMI] Astrid Tschense, Java-Monitoring Infrastruktur im SAP NetWeaver 04 (Chapter 3), SAP Press, 2005.

[JMX] Java™ Management Extensions Specification <http://www.jcp.org/en/jsr/detail?id=3>, 2005 (retrieved 20.05.2005).

[JPW] Karl Kessler, Peter Tillert, Panayot Dobrikov, Java Programmierung mit dem SAP Web Application Server (Chapter 12), SAP Press, 2005.

[MW] Martin Wolf, Administration of the SAP Web Application Server, HPI Seminar System Modelling 2005, Hasso-Plattner-Institute for Software Systems Engineering, 2005.

[SAPL] SAP Library, <http://help.sap.com/>, 2005 (retrieved 11.05.2005).

[SDN0] Astrid Tschense, LCM204: Troubleshooting for SAP Web AS Java (Presentation), SAP Developer Network, <https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/library/webas/STU/Troubleshooting%20for%20SAP%20Web%20AS%20Java.pdf>, 2005 (retrieved 18.05.2005).

[SDN1] Julia Levedag, Monitoring in Web AS 6.40 (Presentation), SAP Developer Network, <https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/documents/a1-8-4/Systems%20and%20Application%20Monitoring%20WebAS%20640%20Java.pdf>, 2005 (retrieved 18.05.2005).

[SDN2] Scott Braker, Tips and Tricks to fully leverage SAP Web AS for Java developers (Presentation), SAP Developer Network, <http://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/library/uuid/c59736a7-0301-0010-c3bf-9f9f07bc6bc9>, 2005 (retrieved 19.06.2005).

[SDN2] Christoph Nake, Session ID: LCM202 System Management for SAP Solutions (Presentation), SAP Developer Network, <https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/documents/a1-8-4/System%20Management%20for%20SAP%20Solutions.pdf>, 2005 (retrieved 30.06.2005).